Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris*

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis

MPI-SWS, Saarbrücken and Kaiserslautern, Germany†
{janno,haidang,dreyer,orilahav,viktor}@mpi-sws.org

Abstract

The field of *concurrent separation logics* (CSLs) has recently undergone two exciting developments: (1) the *Iris framework* for encoding and unifying advanced higher-order CSLs and formalizing them in Coq, and (2) the adaptation of CSLs to account for *weak memory models*, notably C11's release-acquire (RA) consistency. Unfortunately, these developments are seemingly incompatible, since Iris only applies to languages with an operational interleaving semantics, while C11 is defined by a declarative (axiomatic) semantics. In this paper, we show that, on the contrary, it is not only feasible but useful to marry these developments together. Our first step is to provide a novel operational characterization of RA+NA, the fragment of C11 containing RA accesses and "non-atomic" (normal data) accesses. Instantiating Iris with this semantics, we then derive higher-order variants of two prominent RA+NA logics, GPS and RSL. Finally, we deploy these derived logics in order to perform the first mechanical verifications (in Coq) of several interesting case studies of RA+NA programming. In a nutshell, we provide the first foundationally verified framework for proving programs correct under C11's weak-memory semantics.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs; F.3.2 Semantics of Programming Languages

Keywords and phrases Weak memory models, release-acquire, concurrency, separation logic

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.88

1 Introduction

Separation logic [25] is a refinement of Hoare logic with an intrinsic notion of *ownership*: whereas an assertion in Hoare logic denotes a fact about the global machine state, an assertion in separation logic denotes ownership of (and knowledge about) a *piece* of that state, and the *separating conjunction* $P * Q$ denotes that the assertions P and Q own disjoint pieces of state. This ownership reading of assertions is useful for giving "local" (or "small-footprint") specifications for primitive commands, which are much easier to compose soundly into specifications for larger programs. Moreover, as O'Hearn was the first to observe [24], separation logic is also eminently suitable for *concurrent* programs. In particular, ownership provides a direct and convenient way of explaining how synchronization mechanisms serve to transfer control of shared state between threads. Although O'Hearn's original concurrent version of separation logic was geared toward reasoning about coarse-grained synchronization via semaphores, the subsequent decade of research into *concurrent separation logics* (CSLs) has shown that ownership and separation are just as useful for reasoning about more fine-grained

* An extended version of this paper with a technical appendix can be found at [1].
† Saarland Informatics Campus.

© Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis;
licensed under Creative Commons License CC-BY

and low-level synchronization mechanisms, such as those employed in the implementations of non-blocking data structures [35, 11, 7, 32, 30, 23, 6].

In this paper, we consider two of the most recent, boundary-pushing developments in concurrent separation logics: (1) the *Iris framework* for encoding and unifying advanced higher-order CSLs and formalizing them in Coq [14, 13, 16, 17], and (2) the adaptation of CSLs to account for *weak memory models*, notably C11's *release-acquire* (RA) consistency [34, 33, 8, 20]. Although these developments have thus far (for reasons explained below) appeared to be incompatible, we show that in fact they are not! Quite the contrary: we demonstrate that it is not only feasible but useful to marry them together, and in so doing, provide the first foundationally verified framework for proving programs correct under C11's weak memory semantics.

1.1 Iris: A Unifying Framework for Concurrent Separation Logics

After O'Hearn's original CSL, there came a steady stream of "new and improved" CSLs appearing on at least a yearly basis. Unfortunately, as these new CSLs grew ever more expressive, they also grew increasingly complex, baking in increasingly sophisticated proof rules as primitive, with the relationships and compatibility between different proof rules (*e.g.,* whether they could be soundly combined in one logic) remaining unclear.

The central source of complexity in most existing CSLs lies in their mechanisms for controlling *interference* between threads accessing shared state, which have evolved from Jones's *rely-guarantee* [12] to the much more sophisticated and elaborate *protocol* mechanisms appearing in logics like CaReSL [32], iCAP [30], and TaDA [6]. In an attempt to consolidate the field, Jung *et al.* developed **Iris** [14, 13, 16], a logic with the express goal of showing that even the fanciest of these interference-control mechanisms could be encoded via a combination of two orthogonal "off-the-shelf" ingredients: (1) *partial commutative monoids* (PCMs) for formalizing protocols on shared state, and (2) *invariants* for enforcing them. Invariants are an old and ubiquitous concept in program verification, and PCMs have been used in a number of prior logics to represent different kinds of *ghost* (or *auxiliary*) state *i.e.,* logical state that is manipulated as part of the proof of a program but is not manipulated directly by the program itself. Jung *et al.*'s observation was that in fact these two simple mechanisms are all you need: using just PCMs and invariants, one can *derive* a variety of powerful forms of protocol-based reasoning from prior CSLs *within* Iris, and by virtue of working in a unified framework, these derived mechanisms are automatically compatible (different mechanisms can be used soundly to verify different modules in a program). Iris also goes beyond most prior CSLs by supporting higher-order quantification and *impredicative invariants*—invariants that can talk recursively about the existence of (other) invariants—which are crucial for reasoning about languages with higher-order state (*e.g.,* Rust).

In the past, the complexity of CSLs was further exacerbated by the fact that (until very recently [27]) they only supported manual and error-prone "pencil-and-paper" proofs. The initial version of Iris [14] was no exception: the *soundness* of the core logic was verified in Coq, but the Coq development provided no support for *using* the logic (either to encode other logics or to verify programs interactively). However, in the past year, Krebbers *et al.* [17] have developed IPM, an interactive proof mode geared toward using Iris as a proof development environment for verifying concurrent programs within Coq. With IPM, Iris has begun the transition to a more practically useful proof tool, and is already being deployed effectively for larger verification efforts, *e.g.,* in the RustBelt project [10].

1.2 Separation Logics for Release-Acquire Consistency

Iris is a "generic" logical framework in that it is parameterized over the programming language in question—it merely requires, like the vast majority of prior work on concurrent program verification, that the language have an *operational, interleaving semantics*, typically known as a *sequentially consistent* (SC) semantics [21]. Under SC, threads take turns accessing the shared memory, and updates to memory are immediately visible to all other threads.

SC semantics has the benefit that it is easy to define and manipulate formally, but it is also woefully unrealistic: no serious language guarantees a fully SC semantics, because of the significant performance costs associated with maintaining the fiction of a single, globally consistent view of memory on modern multi-core architectures. One of the reasons for this discrepancy between the theory and the reality of concurrent programming is that, until relatively recently, formal accounts of more realistic—so-called *weak* (or *relaxed*)—memory models for concurrent programming languages were not available. However, in the past decade, great progress has been made on formalizing weak memory models, with a notable high point being the formalization of the C/C++11 memory model (hereafter, C11) [4].

In response to this development, a number of verification researchers have followed suit by building new verification tools—program logics, model checkers, testing frameworks, etc.—that account for these more realistic memory models. In particular, Vafeiadis and collaborators have thus far developed several different separation logics for C11, including RSL [34] and GPS [33]. The main focus of these logics is on RA+NA, an important fragment of C11 consisting of *release-acquire* (RA) accesses and *non-atomic* (NA) accesses. RA accesses are useful because they support a common idiom of message-passing synchronization at low cost compared to SC. NA accesses are intended for "normal" data accesses and are even more efficiently implementable than RA accesses, with the proviso that they are not permitted to race (*i.e.,* races on non-atomics cause the entire program to have undefined behavior).

A major challenge that Vafeiadis *et al.* had to overcome was the fact that C11 is defined using a radically different semantics than SC. Specifically, it is defined by a *declarative* (or *axiomatic*) semantics, in which the allowed behaviors of a program are defined by enumerating candidate executions (represented as "event graphs") and then restricting attention to the executions that obey various coherence axioms. In building separation logics for C11, Vafeiadis *et al.* were thus not able to use the standard model of Hoare-style program specifications from prior separation logics because notions like "the machine states before and after executing a command C" do not have a clear meaning in C11's declarative semantics.

To account for this radically different type of semantics, they were instead forced to essentially throw away the "separation-logic textbook" and come up with an entirely new, non-standard model of separation logic in terms of predicates on event graphs. While groundbreaking, this approach has had several downsides. Firstly, certain essential mechanisms of SC-based separation logic (such as ghost state), which are easy to justify in standard models, became very difficult to justify in the new event-graph-based models of RA+NA logics. Secondly, the complexity of these new models has made them challenging to adapt and extend, and their non-standard nature has posed a major accessibility hurdle for researchers accustomed to traditional models of separation logic. Last but not least, although the soundness of these logics has been verified formally in Coq, there has thus far been no tool support for *using* the logics to prove programs correct under RA+NA semantics.

1.3 Our Contributions

Given our above description, it may seem that the Iris framework's reliance on interleaving semantics renders it fundamentally inapplicable to reasoning about C11's weak-memory semantics. In this paper, we show that this is not the case at all—not only is it possible to derive RA+NA logics like GPS and RSL within Iris, but there are several tangible benefits to doing so. Deriving such logics within Iris:

- Lets us take advantage of the rich features of the Iris host logic (*e.g.,* separation, invariants) when proving soundness of the derived logics, thereby significantly lifting the abstraction level at which those soundness proofs are carried out (compared to prior work).
- Allows us to support some very useful features in our derived logics by directly importing them from Iris. Such features include PCM-based ghost state, higher-order impredicative quantification, and Iris's interactive proof mode in Coq. By virtue of being encoded in Iris, our derived logics inherit these features for free.
- Makes it easy to experiment with the derived logics and quickly develop new and useful extensions (*e.g.,* single-writer protocols, see below).
- Makes it possible to soundly compose proofs from different derived logics, since they are all carried out in the uniform framework of Iris.

Our first step (§2) is to avoid the essential complicating factor—C11's declarative account of RA+NA—and instead work with an operational account. Building closely on Lahav *et al.*'s recently proposed "strong release-acquire" (SRA) semantics [19], we define a novel, operational, interleaving semantics for RA+NA. Our operational account of the RA fragment of the language is very similar to Lahav *et al.*'s operational account of SRA in that it models writing and reading of memory via the sending and receiving of timestamped messages; the main difference is that the RA rule for assigning timestamps is slightly more liberal. Our account of NA is new, though; it uses timestamps to model races on non-atomics as stuck (unsafe) machine states. We have proven that, under the reasonable restriction that programs do not mix RMWs (atomic updates) and non-atomic reads at the same location, our semantics is equivalent to the standard declarative semantics of the RA+NA fragment of C11.

Next, since our new semantics for RA+NA is an interleaving semantics, we can instantiate Iris with it. In §3, we review the basic reasoning mechanisms of Iris, and show how to use them to derive small-footprint proof rules for reasoning about RA+NA programs. We apply these rules to verify a simple message-passing example of RA+NA programming in Iris. However, as will become clear, reasoning directly with the Iris primitive mechanisms is rather too low-level and a more abstract logic is needed.

In §4, we present iGPS, a higher-order variant of Turon *et al.*'s GPS logic [33], which supports much higher-level reasoning about RA+NA programs. Unlike the original GPS, iGPS is derived within Iris on top of the small-footprint proof rules from §3. It also extends GPS with *single-writer protocols*, an extremely useful feature that simplifies proofs of RA+NA programs in the common case where there are no write-write races on atomic accesses.

In §5, we briefly describe some other contributions, including iRSL, a higher-order variant of RSL [34] derived within Iris, and several case studies that we have verified using iGPS and iRSL in Coq. These examples showcase one of the major advantages of working in the Iris framework: our ability to verify weak-memory programs, foundationally and mechanically, with the same degree of ease that was previously only possible for SC programs.

Finally, in §6, we conclude with related work.

2 Release-Acquire and Non-Atomics

In this section, we introduce our operational semantics for RA+NA, which we then use as the machine for our working language λ_{RN}. Subsequent sections will show how to build a logic for λ_{RN} using Iris.

C11 provides several memory access modes, each ensuring a different degree of consistency. In this paper we focus on RA+NA, the fragment of C11 consisting only of release-acquire (RA) and non-atomic (NA) accesses. Non-atomic accesses (which we denote with "[na]") are the default type of memory accesses, intended to be used for normal data rather than for synchronization. Thus, C11 forbids any data races on non-atomic accesses, and programs that may have such races are considered buggy (they have undefined semantics). In contrast, RA accesses (which we denote with "[at]" for *atomic*) are permitted to race, but provide just enough consistency guarantees to enable the well-known *message passing* (MP) idiom:

$$\begin{array}{c} x_{[\mathrm{na}]} := 0;\, y_{[\mathrm{na}]} := 0; \\ \begin{array}{r||l} x_{[\mathrm{na}]} := 37; & \textbf{repeat } y_{[\mathrm{at}]}; \\ y_{[\mathrm{at}]} := 1 & x_{[\mathrm{na}]} \end{array} \end{array}$$

Initially, both variables x and y are set to 0. The first thread will initialize x to 37 (non-atomically) and then set the variable y to 1 (via a release write) as a way of sending a message to the second thread that x has been properly initialized and is ready for consumption. The second thread will repeatedly read y (via an acquire read) until it observes $y \neq 0$, at which point—thanks to release-acquire semantics—it will know that it can safely access x. Summing up, the use of RA here ensures that the non-atomic write to x in the first thread "happens before" the non-atomic read of x in the second thread—*i.e.,* that they do not race—and furthermore that the read of x will return 37.

The formal semantics of RA+NA is "declarative", formulated as a set of constraints on execution graphs. We will instead now present an alternative *operational* semantics of RA+NA. Our operational semantics is not completely coherent with C11's for programs that mix atomic and non-atomic accesses to the same location (although the semantics of such programs is already known to be problematic [3]—see §6 for further discussion of this point). However, for the large class of programs that do not mix atomic updates (like CAS) and non-atomic reads at the same location, our semantics is provably equivalent to C11's declarative semantics. This class of programs includes all C11 programs considered (and verified) in this paper. (For formal details of the correspondence between our semantics and C11's, see our technical appendix [1].) We will first start with the pure RA fragment, and then add a "race detector" for non-atomic accesses.

2.1 Release-Acquire

Our operational semantics for RA starts from the observation that in RA—in contrast to a standard heap language—different threads have a different view of what the state is. Accordingly, we need to keep track of past write events as they might still be relevant for some subset of threads. Moreover, we need to keep writes to the same location in a total order enforced in C11 under the name *modification order* (*mo* for short). Finally, we also need to keep track of each thread's "progress" in terms of which writes are visible to it, as this determines what a thread may read and where its writes may end up.

For the *mo* order, the RA machine manages for each location a totally ordered set of timestamps $t \in \mathrm{Time} \triangleq \mathbb{N}$. Each write of some value v to a location ℓ gets assigned a

$$\textsc{Thread-Read}\quad \frac{(\ell, v, t, V) \in M \qquad T(\pi)(\ell) \leq t}{(M, T) \xrightarrow{\langle \text{Read}, \ell, v\rangle}{}^{\pi} (M, T[\pi \mapsto T(\pi) \sqcup V])}$$

$$\textsc{Thread-Write}\quad \frac{\neg\exists v', V.\ (\ell, v', t, V) \in M \qquad T(\pi)(\ell) < t \qquad V' = T(\pi)[\ell \mapsto t]}{(M, T) \xrightarrow{\langle \text{Write}, \ell, v\rangle}{}^{\pi} (M \cup \{(\ell, v, t, V')\}, T\big[\pi \mapsto V'\big])}$$

$$\textsc{Thread-Update}\quad \frac{\begin{array}{cc}(\ell, v_o, t, V) \in M & T(\pi)(\ell) \leq t \\ \neg\exists v, V.\ (\ell, v, t+1, V) \in M & V' = T(\pi)[\ell \mapsto t+1] \sqcup V\end{array}}{(M, T) \xrightarrow{\langle \text{Update}, \ell, v_o, v_n\rangle}{}^{\pi} (M \cup \{(\ell, v_n, t+1, V')\}, T\big[\pi \mapsto V'\big])}$$

$$\textsc{Thread-Fork}\quad \frac{\rho \notin \text{dom}(T)}{(M, T) \xrightarrow{\langle \text{Fork}, \rho\rangle}{}^{\pi} (M, T[\rho \mapsto T(\pi)])}$$

$$\textsc{Thread-Uninitialized}\quad \frac{\begin{array}{c}\varepsilon \in \{\langle \text{Read}, \ell, v\rangle, \langle \text{Update}, \ell, v_o, v_n\rangle\} \\ T(\pi)(\ell) = \bot\end{array}}{(M, T) \xrightarrow{\varepsilon}{}^{\pi} \bot_{\text{uninit}}}$$

Figure 1 Per-thread reductions for RA without NA.

timestamp (that is unique for ℓ), resulting in a write event $\omega \in \text{Event} \triangleq \text{Loc} \times \text{Val} \times \text{Time}$, where $v \in \text{Val} \triangleq \mathbb{Z}$.[1] Using timestamps, the thread's "progress" is represented by a *view*, $V \in \text{View} \triangleq \text{Loc} \xrightarrow{\text{fin}} \text{Time}$, which records the timestamp of the most recent write event observed by the thread for every location. To enable communication between threads, every write event is augmented with the writing thread's view, yielding a message $m \in \text{Msg} \triangleq \text{Event} \times \text{View}$. The machine state σ comprises a message pool (called *memory*) and a view for every thread.

▶ **Definition 1** (Simplified Physical State)**.** Let $\sigma \in \Sigma \triangleq \Big(\mathcal{P}(\text{Msg}) \times (\text{ThreadId} \xrightarrow{\text{fin}} \text{View})\Big) \uplus \{\bot_{\text{uninit}}\}$ represent physical machine states, where $\bot_{\text{uninit}}$ represents an error state. We write M and T to denote the two components of a non-error state.

The λ_{RN} language's reductions are factored into *expression reductions*, concerned with the evaluation of the language's expressions, and *machine reductions*, concerned with how the execution of an expression affects the machine state. We will define the expression reductions later when we formally define λ_{RN}. Here we focus on the machine reductions.

We define event labels $\varepsilon \in \mathcal{E} \triangleq \{\langle \text{Read}, \ell, v\rangle, \langle \text{Write}, \ell, v\rangle, \langle \text{Update}, \ell, v_o, v_n\rangle, \langle \text{Fork}, \rho\rangle\}$, representing reads, writes, atomic updates (RMW's), and forks (with ρ being the newly created thread id), respectively. The reductions are defined by a set of local, per-thread reductions $\xrightarrow{\varepsilon}{}^{\pi} \subseteq \Sigma \times \Sigma$ given in Figure 1, where π represents the current thread's id.

A write (Thread-Write) picks an *unused* timestamp t for location ℓ that is greater than the thread's view of ℓ, updates the thread's view to the new view V' that includes t, and adds a corresponding message to the memory. A read (Thread-Read) incorporates the view V of the message that it reads into the thread's own view.[2] Note that the message being read

[1] The full semantics also supports allocation, which induces an allocation value A. We do not mention it here for the sake of simplicity.

[2] Here, we use the join operator $\sqcup$ on views: $(V_1 \sqcup V_2)(\ell) = \max\{V_1(\ell), V_2(\ell)\}$ if $\ell \in \text{dom}(V_1) \cap \text{dom}(V_2)$; $(V_1 \sqcup V_2)(\ell) = V_i(\ell)$ if $\ell \in \text{dom}(V_i) \setminus \text{dom}(V_j)$; and $(V_1 \sqcup V_2)(\ell)$ is undefined if $\ell \notin \text{dom}(V_1) \cup \text{dom}(V_2)$.

is required to have a timestamp that is not smaller than the thread's view of the relevant location. Updates (Thread-Update) combine reading and writing in one step. In addition, updates must "pick" $t+1$ as a timestamp for the new message, where t is the timestamp of the read message. This implies, in particular, that two different updates cannot read the same message, and corresponds to C11's atomicity condition, which requires every update to read from its *mo*-immediate predecessor. Thread-Fork adds a new thread whose view is copied from its parent. Finally, Thread-Uninitialized detects reads from uninitialized locations, and moves to the error state $\bot_{\text{uninit}}$.

Functional correctness of MP

With the operational semantics of RA, we can now sketch why MP (assuming for now that all its accesses are RA) is functionally correct, *i.e.,* why the read of x by the second thread will return 37 when the program terminates. The write of 37 to x is recorded at a view V_{37}, which is then included in the view V_1 of the write of 1 to y by the first thread. When the second thread reads 1 from y, its local view is updated to incorporate V_1 (and thus also V_{37}). A read from x is now guaranteed to read from the message setting x to 37 or from a more recent one, but no more recent one exists. Consequently, the return value will be 37.

2.2 Non-Atomics

Formally, C11 defines a data race as two memory accesses to the same location—of which at least one is a write and at least one is non-atomic—that are not ordered by "happens-before." A program that exhibits data races in *some* of its execution graphs is called racy, and its behavior is considered undefined. We now show how to account for non-atomics and data races in the context of our operational semantics.

Let us first extend the set of physical states by another error state $\bot_{\text{race}}$, whose intent is captured by the following correspondence: a program is racy if and only if at least one of its machine executions can reach $\bot_{\text{race}}$ (stated and proved formally in our appendix [1]).

To detect data races during the execution of a program, we add an additional component to the physical state: the *non-atomic view* N, which tracks the timestamp of the most recent non-atomic write to every location. Then, we place the following restrictions on all atomic and non-atomic operations (if violated, the program will enter the $\bot_{\text{race}}$ state):

- To perform any access (atomic or non-atomic) to a location ℓ, a thread π must have observed the most recent non-atomic write to ℓ, *i.e.,* $N(\ell) \leq T(\pi)(\ell)$.
- A thread π can only perform a non-atomic read from a location ℓ if it has observed the most recent (atomic or non-atomic) write to ℓ, *i.e.,* $\nexists t, (\ell, _, t, _) \in M.\ T(\pi)(\ell) < t$.

In addition to these restrictions, we require non-atomic writes to pick timestamps greater than all existing timestamps of messages of the same location. Intuitively, these restrictions enforce that each non-atomic write to ℓ starts a new "era" in ℓ's timestamps, after which any attempt to access writes from a previous era (or to write with a timestamp from a previous era) constitutes a race. Note that there is an asymmetry between non-atomic reads and writes: non-atomic writes to ℓ are allowed even when the thread has not observed the most recent write to ℓ, as it is only required to observe the most recent *non-atomic* write to ℓ. One might fear that this fails to detect the case when a non-atomic write is racing with a concurrent atomic write (and the atomic write happens first); but in this case the race will be detected in a different execution where the non-atomic write happens first (and the atomic write enters the $\bot_{\text{race}}$ state), so the program will nevertheless be declared racy.

Read
$$\frac{(\ell, v, t, V) \in M \qquad T(\pi)(\ell) \leq t \qquad \alpha = \text{na} \Rightarrow \forall v', t', V'.(\ell, v', t', V') \in M \Rightarrow t' \leq T(\pi)(\ell)}{(M, T, N) \xrightarrow{\langle \text{Read}_\alpha, \ell, v\rangle}{}^{\pi} (M, T[\pi \mapsto T(\pi) \sqcup V], N)}$$

Write-at
$$\frac{\neg\exists v', V.\ (\ell, v', t, V) \in M \qquad N(\ell) \leq T(\pi)(\ell) < t \qquad V' = T(\pi)[\ell \mapsto t]}{(M, T, N) \xrightarrow{\langle \text{Write}_{\text{at}}, \ell, v\rangle}{}^{\pi} (M \cup \{(\ell, v, t, V')\}, T[\pi \mapsto V'], N)}$$

Write-na
$$\frac{\neg\exists v', V.\ (\ell, v', t, V) \in M \qquad N(\ell) \leq T(\pi)(\ell) \qquad V' = T(\pi)[\ell \mapsto t] \qquad \forall v', t', V.\ (\ell, v', t', V) \in M \Rightarrow t' < t}{(M, T, N) \xrightarrow{\langle \text{Write}_{\text{na}}, \ell, v\rangle}{}^{\pi} (M \cup \{(\ell, v, t, V')\}, T[\pi \mapsto V'], N[\ell \mapsto t])}$$

Update
$$\frac{(\ell, v_o, t, V) \in M \qquad N(\ell) \leq T(\pi)(\ell) \leq t \qquad \neg\exists v, V.\ (\ell, v, t+1, V) \in M \qquad V' = T(\pi)[\ell \mapsto t+1] \sqcup V}{(M, T, N) \xrightarrow{\langle \text{Update}, \ell, v_o, v_n\rangle}{}^{\pi} (M \cup \{(\ell, v_n, t+1, V')\}, T[\pi \mapsto V'], N)}$$

Fork
$$\frac{\rho \notin \text{dom}(T)}{(M, T, N) \xrightarrow{\langle \text{Fork}, \rho\rangle}{}^{\pi} (M, T[\rho \mapsto T(\pi)], N)}$$

Race-I
$$\frac{\varepsilon \in \{\langle \text{Read}_\alpha, \ell, v\rangle, \langle \text{Write}_\alpha, \ell, v\rangle, \langle \text{Update}, \ell, v_o, v_n\rangle\} \qquad T(\pi)(\ell) < N(\ell)}{(M, T, N) \xrightarrow{\varepsilon}{}^{\pi} \bot_{\text{race}}}$$

Race-II
$$\frac{\exists v', t', V'.(\ell, v', t', V') \in M \wedge T(\pi)(\ell) < t'}{(M, T, N) \xrightarrow{\langle \text{Read}_{\text{na}}, \ell, v\rangle}{}^{\pi} \bot_{\text{race}}}$$

Uninitialized
$$\frac{\varepsilon \in \{\langle \text{Read}_\alpha, \ell, v\rangle, \langle \text{Update}, \ell, v_o, v_n\rangle\} \qquad T(\pi)(\ell) = \bot}{(M, T, N) \xrightarrow{\varepsilon}{}^{\pi} \bot_{\text{uninit}}}$$

Figure 2 Per-thread reductions for the RA+NA machine.

Revisiting MP, we note that it is safe to have non-atomic accesses to x: the write is performed while the left thread is necessarily aware of the most recent non-atomic write to x (the initialization); and the read is performed while the right thread is necessarily aware of the most recent write to x, whose timestamp was incorporated into the right thread's view when it read $y = 1$.

Figure 2 presents the full operational semantics. It is based on the following definition of a physical state.

▶ **Definition 2** (Physical State).
Let $\sigma \in \Sigma \triangleq \left(\mathcal{P}(\text{Msg}) \times (\text{ThreadId} \xrightarrow{\text{fin}} \text{View}) \times \text{View}\right) \uplus \{\bot_{\text{race}}, \bot_{\text{uninit}}\}$ represent physical machine states. We write M, T, and N to denote the components of a non-error state. The initial physical state, denoted σ_{init}, is given by $(\emptyset, [0 \mapsto \emptyset], \emptyset)$.

2.3 The λ_{RN} language

λ_{RN} is a standard lambda calculus with recursive functions, forks, and references with atomic and non-atomic accesses. The **repeat** construct that we have used in MP can be defined in terms of recursive functions. The interesting part of the language and its expression reductions is given in Figure 3. The expression reduction relation ($e \xrightarrow{\varepsilon} e', \overline{e}_f$) has four components: the original expression e, an (optional) machine memory event ε, the resulting

$$v \in \text{Val} ::= () \mid z \in \mathbb{Z} \mid \ell \in \text{Loc} \mid \textbf{fix}\ (f, x).\ e$$
$$\alpha \in \text{Access} ::= \text{at} \mid \text{na}$$
$$e \in \text{Expr} ::= v \mid e_1\ e_2 \mid \textbf{if}\ z\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \mid \textbf{fork}\ e \mid \ell_{[\alpha]} \mid \ell_{[\alpha]} := v \mid \textbf{cas}(\ell, v, v) \mid \ldots$$

$$\begin{array}{rcll}
\ell_{[\alpha]} & \xrightarrow{\langle \text{Read}_\alpha, \ell, v\rangle} & v, \mathsf{nil} & \\
\ell_{[\alpha]} := v & \xrightarrow{\langle \text{Write}_\alpha, \ell, v\rangle} & (), \mathsf{nil} & \\
\textbf{cas}(\ell, v_o, v_n) & \xrightarrow{\langle \text{Update}, \ell, v_o, v_n\rangle} & 1, \mathsf{nil} & \\
\textbf{cas}(\ell, v_o, v_n) & \xrightarrow{\langle \text{Read}_{\text{at}}, \ell, v\rangle} & 0, \mathsf{nil} & \text{if } v \neq v_o \\
\textbf{fork}\ e & \xrightarrow{\langle \text{Fork}, \rho\rangle} & (), [e] & \\
(\textbf{fix}\ (f, x).\ e)\ v & \rightarrow & e[(\textbf{fix}\ (f, x).\ e)/f][v/x], \mathsf{nil} & \\
\textbf{if}\ z\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & \rightarrow & e_1, \mathsf{nil} & \text{if } z \neq 0 \\
\textbf{if}\ z\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & \rightarrow & e_2, \mathsf{nil} & \text{if } z = 0 \\
& \ldots & &
\end{array}$$

Figure 3 Main λ_{RN} expressions and expression reductions.

$$\textsc{Combined-Pure}\ \frac{e \rightarrow e', \mathsf{nil}}{\sigma; e \rightarrow^{\pi} \sigma; e', \mathsf{nil}} \qquad \textsc{Combined-Mem}\ \frac{\varepsilon \neq \langle \text{Fork}, _\rangle \quad e \xrightarrow{\varepsilon} e', \mathsf{nil} \qquad \sigma \xrightarrow{\varepsilon}^{\pi} \sigma'}{\sigma; e \xrightarrow{\varepsilon}^{\pi} \sigma'; e', \mathsf{nil}}$$

$$\textsc{Combined-Fork}\ \frac{e \xrightarrow{\langle \text{Fork}, \rho\rangle} e', [e_f] \qquad \sigma \xrightarrow{\langle \text{Fork}, \rho\rangle}^{\pi} \sigma'}{\sigma; e \xrightarrow{\langle \text{Fork}, \rho\rangle}^{\pi} \sigma'; e', [e_f]}$$

$$\textsc{Threadpool-Red-Pure}\ \frac{\sigma; \mathcal{TS}(\pi) \rightarrow^{\pi} \sigma'; e', \mathsf{nil}}{\sigma; \mathcal{TS} \rightarrow^{\pi} \sigma'; \mathcal{TS}\left[\pi \mapsto e'\right]} \qquad \textsc{Threadpool-Red-Mem}\ \frac{\sigma; \mathcal{TS}(\pi) \xrightarrow{\varepsilon}^{\pi} \sigma'; e', \mathsf{nil}}{\sigma; \mathcal{TS} \xrightarrow{\varepsilon}^{\pi} \sigma'; \mathcal{TS}\left[\pi \mapsto e'\right]}$$

$$\textsc{Threadpool-Red-Fork}\ \frac{\sigma; \mathcal{TS}(\pi) \xrightarrow{\langle \text{Fork}, \rho\rangle}^{\pi} \sigma'; e', [e_f]}{\sigma; \mathcal{TS} \xrightarrow{\langle \text{Fork}, \rho\rangle}^{\pi} \sigma'; \mathcal{TS}\left[\pi \mapsto e'\right] \uplus [\rho \mapsto e_f]}$$

Figure 4 Threadpool reductions.

expression e', and a list of newly created threads $\overline{e}_f$. Only the rule for **fork** e creates a new thread (*i.e.*, a singleton list $[e]$), while all other reductions produce an empty list (*i.e.*, nil).

The *per-thread language reductions* $(\sigma; e \xrightarrow{\varepsilon}^{\pi} \sigma'; e', \overline{e}_f)$ are then the combination of the expression reductions and the machine reductions, given by the Combined-* rules in Figure 4. Non-stateful reductions (Combined-Pure) simply defer to the expression reductions, while stateful reductions (Combined-Mem and Combined-Fork) use the event label ε and the thread id π to tie the expression and machine reductions together correctly. These per-thread reductions then are lifted in a straightforward manner to the full (threadpool) reductions.

3 Iris

Iris is a generic framework for constructing concurrent separation logics. One can instantiate the framework with any language that has an operational interleaving semantics, and then easily derive time-tested reasoning principles for one's target logic, including various "protocol" mechanisms for controlling interference. Figure 5 provides an excerpt of Iris syntax.

$$P ::= \mathsf{False} \mid \mathsf{True} \mid P \Rightarrow Q \mid P \wedge Q \mid P \vee Q \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \exists x.\ P \mid \forall x.\ P \mid \mu x.\ P$$
$$\mid \mathsf{Phys}(\sigma) \mid \boxed{a}^{\gamma} \mid \triangleright P \mid \Box P \mid \boxed{P}^{\mathcal{N}} \mid P \;{}^{\mathcal{N}_1}\!\!\Rrightarrow^{\mathcal{N}_2} Q \mid \{P\}\ e\ \{x.\ Q\}_{\mathcal{N}} \mid \ldots$$

Figure 5 An excerpt of Iris syntax.

Iris supports the common connectives ($\mathsf{False}, \mathsf{True}, \Rightarrow, \wedge, \vee, *, \mathrel{-\!\!*}, \exists, \forall, \mu$) and proof rules standard in higher-order separation logics. Iris's extended set of constructs includes physical state ownership $\mathsf{Phys}(\sigma)$, ghost state ownership $\boxed{a}^{\gamma}$, the later $\triangleright$ and always $\Box$ modalities, invariants $\boxed{P}^{\mathcal{N}}$, view shifts $P \;{}^{\mathcal{N}_1}\!\!\Rrightarrow^{\mathcal{N}_2} Q$, and Hoare triples $\{P\}\ e\ \{x.\ Q\}_{\mathcal{N}}$. We will first explain these constructs via a running example, in which we use Iris to verify the MP example in a simple, sequentially consistent language called λ_{SC}. This will not only illustrate how one can derive within Iris a target logic for a language defined by an operational semantics, but will also serve as a warm-up for our subsequent explanation of how we can instantiate Iris to reason about weak memory.

Road map

The process of instantiating Iris to derive new logics follows a simple pattern, which is worth articulating up front:

1. When we first instantiate Iris, the only primitive assertion we get about the state of the program is the *physical state ownership* assertion $\mathsf{Phys}(\sigma)$, which asserts that σ is the current global state of the machine. Together with this assertion we also get for free a bunch of *large-footprint* specifications for the primitive commands of the language, based directly on their operational semantics. For example, the primitive specification we get for updating a location in λ_{SC} will be $\{\mathsf{Phys}(\sigma)\}\ \ell := v\ \{\mathsf{Phys}(\sigma[\ell \mapsto v])\}$.
2. Of course, one of the main points of separation logic is to be able to reason modularly using *local* assertions about the machine state, such as the *points-to* assertion, $\ell \hookrightarrow v$, and correspondingly give *small-footprint* specifications of the primitive commands, such as $\{\ell \hookrightarrow w\}\ \ell := v\ \{\ell \hookrightarrow v\}$. In Iris, such local assertions are not baked into the logic, but rather are encodable using *ghost state ownership* assertions, and the user of the logic has a great deal of flexibility concerning how these assertions are defined. In the case of the points-to assertion, $\ell \hookrightarrow v$, we will define this assertion so as to represent the *knowledge* that ℓ currently points to v and the *rights* to read and write ℓ.
3. On its own, a local, user-defined ghost state assertion like the points-to assertion is merely a *representation* of knowledge and rights. In order to give meaning to such a ghost state assertion—*i.e.,* to make sure it is in sync with the primitive physical state assertion—we establish an *invariant* tying the assertions together. In the case of the points-to assertion, this invariant will enforce that when a thread owns the ghost state assertion $\ell \hookrightarrow v$, its “knowledge” that ℓ currently points to v in the physical machine state is actually correct.

In short, *ghost state assertions represent* local knowledge and rights concerning the machine state, and *invariants enforce* that ghost state assertions mean what they say they mean.

3.1 Iris by Example

Our example programming language λ_{SC} is a standard lambda calculus with references. It is basically the same as λ_{RN}, except that all accesses are sequentially consistent, and races are permitted (they do not induce stuckness). The language's physical state is a heap σ, which is a finite map from allocated locations to values. The main heap-related reductions of λ_{SC} are given in Figure 6. When we instantiate Iris with λ_{SC}'s operational semantics,

$$\begin{array}{rcll} (!\ell, \sigma) & \to & (v, \sigma) & \text{if } \sigma(\ell) = v \\ (\ell := v, \sigma) & \to & ((), \sigma[\ell \mapsto v]) & \text{if } \ell \in \text{dom}(\sigma) \\ & \dots & & \end{array}$$

Figure 6 Main heap-related reductions of the λ_{SC} language.

(as explained in the above road map) what we get automatically from Iris are the following *large-footprint* Hoare triples concerning the physical state ownership assertion $\mathsf{Phys}(\sigma)$:

Phys-Heap-Read
$\{\mathsf{Phys}(\sigma) * \sigma(\ell) = v\}\ !\ell\ \{z.\ z = v * \mathsf{Phys}(\sigma)\}$

Phys-Heap-Write
$\{\mathsf{Phys}(\sigma)\}\ \ell := v\ \{\mathsf{Phys}(\sigma[\ell \mapsto v])\}$

Note that z in the first triple binds the return value of the expression $!\ell$. In the second triple, the expression returns the unit value, so we elide the binder.

3.1.1 Encoding Separation Logic for λ_{SC}

We would now like to encode these *small-footprint* Hoare triples for λ_{SC}:

Heap-Read
$\{\ell \hookrightarrow v\}\ !\ell\ \{z.\ z = v * \ell \hookrightarrow v\}$

Heap-Write
$\{\ell \hookrightarrow w\}\ \ell := v\ \{\ell \hookrightarrow v\}$

The first step is to define the points-to assertion, $\ell \hookrightarrow v$, using Iris's *ghost state*.

Ghost state and partial commutative monoids

Ghost state is non-physical state that is only used as part of a program verification but is not itself part of the machine state. In Iris, ghost state is formalized using partial commutative monoids (PCMs).[3] The assertion $\boxed{a : M}^{\gamma}$ asserts the ownership of the ghost resource a for an instance γ of the PCM M. Separating conjunction for ghost state assertions simply lifts the PCM composition operation to the assertion level: $\boxed{a : M}^{\gamma} * \boxed{b : M}^{\gamma} \iff \boxed{a \cdot_M b : M}^{\gamma}$. If two PCM fragments are not compatible (i.e. their composition is not defined), then it is not possible to own both of them at the same time, *i.e.*, if $a \cdot b = \bot$ then $\boxed{a}^{\gamma} * \boxed{b}^{\gamma} \Rightarrow \mathsf{False}$.[4] In order to maintain consistency of the logic, therefore, changes to ghost state are restricted to *frame-preserving* updates, in which a PCM fragment a can only be updated to b if b preserves compatibility with any other fragments in the environment (the *frame*):

Ghost-Update
$$\frac{\forall a_f.\ a \cdot a_f \neq \bot \Rightarrow b \cdot a_f \neq \bot}{\boxed{a}^{\gamma} \Rrightarrow \boxed{b}^{\gamma}}$$

Ghost updates belong to the set of *logical computations*, or in Iris terminology, *view shifts*. A view shift $P \Rrightarrow Q$ represents the capability of transforming a resource satisfying P into a resource satisfying Q without changing the physical state.

[3] Actually, ghost state in Iris is based on the more general mechanism of "cameras" (aka step-indexed resource algebras), which can support a more general form of higher-order ghost state [13].

[4] In the rest of the paper we also suppress the PCM M in $\boxed{a : M}^{\gamma}$ when it can be inferred in context.

A PCM for heaps

As a step towards defining $\ell \hookrightarrow v$, let us now construct a PCM called HEAP that has the same basic structure as the physical heap, but allows splitting and recomposition. (We will ultimately need a slightly more sophisticated PCM to define $\ell \hookrightarrow v$, but HEAP is an important part of the construction.) HEAP is a finite partial map from locations to values, with the empty heap as its unit element, and the composition on heaps is defined as disjoint union (*i.e.,* union if the heaps have disjoint domain, and undefined otherwise). The composition implies that the singleton heap $[\ell := v]$ does not combine with itself, so it can only be uniquely owned, and it also represents the permission required to update ℓ:

GHOST-HEAP-EXCLUSIVE
$$\boxed{[\ell := v]}^{\gamma} * \boxed{[\ell := w]}^{\gamma} \vdash \mathsf{False}$$

GHOST-HEAP-UPDATE
$$\boxed{[\ell := w]}^{\gamma} \Rrightarrow \boxed{[\ell := v]}^{\gamma}$$

The singleton heap $[\ell := v]$ therefore has the desired properties for defining the local assertion $\ell \hookrightarrow v$, but unfortunately it is still not quite enough: we also need some way to tie this ghost state assertion to the underlying physical state of the program. Toward this end, we employ Iris's *invariants*.

Invariants

Invariants in Iris can be thought of as assertions that hold of *some* shared resource at all times, although the choice of which shared resource satisfies them is allowed to vary over time. The Iris invariant assertion $\boxed{P}^{\mathcal{N}}$ stipulates that P is an invariant. The resource that satisfies it is shared with all threads, and thus any thread can access it freely in a single physical step[5]: it can *open* the invariant and gain local ownership of the resource for the duration of the operation, so long as it can *close* the invariant by relinquishing ownership of some (potentially different) resource satisfying P at the end of the operation. For bookkeeping purposes—specifically, to ensure that we do not unsoundly open the same invariant more than once in a nested fashion—invariants in Iris are named, and the $\mathcal{N}$ in the above invariant assertion is a *namespace* (set of names) from which the name of the invariant must come.

Invariants belong to the set of *persistent* assertions, denoted with the always modality $\Box$. The assertion $\Box P$ establishes the *knowledge* that P holds without any ownership, and therefore holds forever after. Putting resources into an invariant is thus a common way to share or transfer ownership through the use of freely distributable knowledge.

Meanwhile, the actions of opening and closing invariants belong to the set of logical computations, or view shifts. To account for invariants, view shifts are extended with namespaces as well: $P \;{}^{\mathcal{N}_1}\!\!\Rrightarrow^{\mathcal{N}_2} Q$ asserts that, assuming the invariants named in $\mathcal{N}_1$ hold before the view shift, then the invariants named in $\mathcal{N}_2$ hold after the view shift. Opening and closing of invariants are then formalized as follows:

INV-OPEN
$$\boxed{P}^{\mathcal{N}} \vdash \mathsf{True} \;{}^{\mathcal{N}}\!\!\Rrightarrow^{\emptyset} P$$

INV-CLOSE
$$\boxed{P}^{\mathcal{N}} \vdash P \;{}^{\emptyset}\!\!\Rrightarrow^{\mathcal{N}} \mathsf{True}$$

INV-OPEN allows a thread to open the invariant $\boxed{P}^{\mathcal{N}}$ and gain ownership of P but prevents it from doing so more than once. Only after applying INV-CLOSE and re-establishing the invariant will the thread be able to open it again.

[5] In Iris terminology, a resource in an invariant can be accessed within an *atomic* operation, which is an operation that takes only a single physical step of execution. We do not use the term here to avoid confusion with C11 atomics.

Note: The Inv-Open rule as stated here is only sound if P talks about ownership (of physical or ghost state) and not about the existence of other invariants. In general, however, Iris makes no such restriction; rather, it supports *impredicative invariants*, meaning that P can be an arbitrary assertion. In order to avoid paradoxes caused by impredicative circularities (like the one described in [16]), the fully general version of this rule in Iris requires that P be guarded by the step-indexed *later* modality ($\triangleright$). Fortunately, in most cases the $\triangleright$'s can be stripped away automatically (the Iris proof mode in Coq provides support for doing this) and do not play an interesting role in proofs. To focus the presentation of this paper, we will therefore suppress further discussion of the $\triangleright$ modality.

Hoare triples in Iris are also annotated with invariant namespaces, since Hoare triples combine both physical and logical computations. A Hoare triple $\{P\}\ e\ \{x.\ Q\}_{\mathcal{N}}$ with a namespace $\mathcal{N}$ implies that if the invariants in $\mathcal{N}$ hold before the expression's execution, then they will be preserved between every step and also after its execution. Consequently, when verifying any single physical step of computation, we are free to open the invariants in $\mathcal{N}$ so long as we immediately close them. This reasoning is encapsulated in the following "atomic rule of consequence":

ACONSQ
$$\frac{P\ {}^{\mathcal{N}\uplus\mathcal{N}'}\!\Rrightarrow^{\mathcal{N}}\ P' \qquad \{P'\}\ e\ \{v.\ Q'\}_{\mathcal{N}} \qquad \forall v.\ Q'\ {}^{\mathcal{N}}\!\Rrightarrow^{\mathcal{N}\uplus\mathcal{N}'}\ Q \qquad e \text{ takes 1 physical step}}{\{P\}\ e\ \{v.\ Q\}_{\mathcal{N}\uplus\mathcal{N}'}}$$

Since bookkeeping of namespaces is largely a tedious detail (and one which Coq will force us to get right), we will for the remainder of the paper suppress namespaces from definitions and proofs. We will always use disjoint namespaces to ensure correctness in opening invariants.

Linking physical and ghost state using invariants and the "authoritative" PCM

Now, returning to our example, the key idea is to use an invariant to tie the physical state assertion together with local ghost state assertions, thereby giving them meaning. To achieve this, we will employ an extremely useful construction called the *authoritative* PCM [14].

Given a base PCM $\mathcal{M}$, the authoritative PCM $\textsc{Auth}(\mathcal{M})$ has two types of elements: authoritative $\bullet\, a$ and non-authoritative $\circ\, a$ (for $a \in \mathcal{M}$). For any instance γ, $\boxed{\bullet\, a}^{\gamma}$ is *exclusive* (*i.e.,* $\boxed{\bullet\, a}^{\gamma} * \boxed{\bullet\, a}^{\gamma} \vdash \mathsf{False}$), and is the main point of reference for all the non-authoritative fragments, in the sense that any ownable fragment $\boxed{\circ\, b}^{\gamma}$ must have b *included* in a, that is $\exists c.\ a = b \cdot c$. The PCM's update therefore requires more: if one wants to update b to b', it has not only to ensure b' is compatible with c, but also has to update a to $a' = b' \cdot c$. These properties are summarized in the following two rules:

AUTH-AGREE
$$\boxed{\bullet\, a}^{\gamma} * \boxed{\circ\, b}^{\gamma} \vdash \exists c.\ a = b \cdot c$$

AUTH-UPDATE
$$\frac{b' \cdot c \neq \bot}{\boxed{\bullet\, b \cdot c}^{\gamma} * \boxed{\circ\, b}^{\gamma} \Rrightarrow \boxed{\bullet\, b' \cdot c}^{\gamma} * \boxed{\circ\, b'}^{\gamma}}$$

With these two rules in hand, we can derive the following rules for operations on $\textsc{Auth}(\textsc{Heap})$:

AGHOST-HEAP-EXCLUSIVE
$$\boxed{\circ\ [\ell := v]}^{\gamma} * \boxed{\circ\ [\ell := w]}^{\gamma} \vdash \mathsf{False}$$

AGHOST-HEAP-AGREE
$$\boxed{\bullet\, \sigma}^{\gamma} * \boxed{\circ\ [\ell := v]}^{\gamma} \vdash \sigma(\ell) = v$$

AGHOST-HEAP-UPDATE
$$\boxed{\bullet\, \sigma}^{\gamma} * \boxed{\circ\ [\ell := w]}^{\gamma} \Rrightarrow \boxed{\bullet\, \sigma[\ell \mapsto v]}^{\gamma} * \boxed{\circ\ [\ell := v]}^{\gamma}$$

We are now ready to establish the invariant $\boxed{\exists \sigma.\ \mathsf{Phys}(\sigma) * \boxed{\bullet\, \sigma}^{\gamma}}$, which binds together the physical state ownership and the authoritative ghost heap ownership. With the invariant

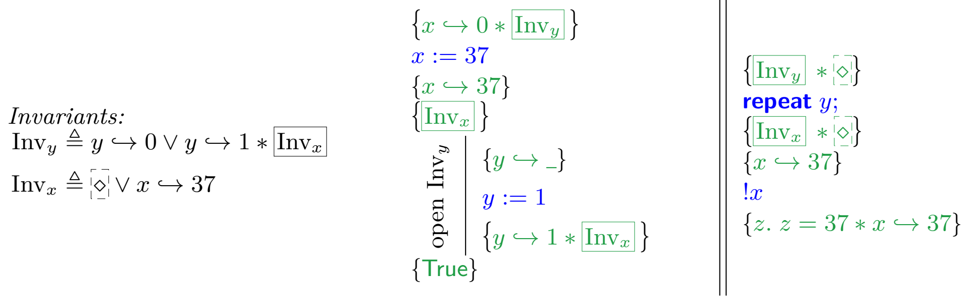

Figure 7 Verification of MP in λ_{SC}.

in place, AGhost-Heap-Agree implies that if a thread owns the singleton ghost heap $\circ [\ell := v]$ locally, then, in combination with the invariant, it is guaranteed that ℓ currently has value v in the physical heap. AGhost-Heap-Exclusive and AGhost-Heap-Update ensure that only the one thread who owns $\circ [\ell := v]^{\gamma}$ can make updates to the contents of ℓ.

The points-to assertion is then defined as $\ell \hookrightarrow v \triangleq \circ [\ell := v]^{\gamma}$, and we can easily prove the small-footprint triples from the beginning of this section by combining the above rules for authoritative ghost heaps with those for opening and closing invariants.

3.1.2 Verifying MP in λ_{SC}

Using the small-footprint triples, we are ready to verify MP in λ_{SC}. We discuss the proof in a bit of detail here, since later on we will show how to adapt this proof to verify MP under weak-memory semantics.

The proof of MP is given in Figure 7. As a proof convention, we only mention persistent assertions (like invariants) once and use them freely later, since they are always true after being established. The proof works essentially as follows.

First of all, both threads want to operate on y simultaneously, so we need to put ownership of y into an invariant $\boxed{\mathrm{Inv}_y}$. This invariant says that y is in one of two states—0 or 1. We can establish the invariant right after the initialization of y (the write of 0 to y), because y is in state 0 at that moment. The first thread is responsible for setting y to state 1. When the second thread observes that y is in state 1, it will expect to be able to gain ownership of $x \hookrightarrow 37$. To achieve this, in state 1, Inv_y asserts the existence of another invariant $\boxed{\mathrm{Inv}_x}$ concerning x, and it is this latter invariant that we use to transfer ownership of location x from the first thread to the second thread.

To understand Inv_x, it helps to have seen the film *Raiders of the Lost Ark*, or at least the first few minutes of it, in which Indiana Jones (played by Harrison Ford) attempts to steal a precious golden idol from an ancient Peruvian temple—without setting off booby traps—by swapping it for a similarly weighted bag of sand. Unfortunately for him, the temple detects his ruse and tries to kill him. But we can play a similar trick, and Iris will be perfectly happy! In our case, the “golden idol” is $x \hookrightarrow 37$, which is transferred into the invariant $\boxed{\mathrm{Inv}_x}$ when it is established by the first thread. The “bag of sand” is a “token” $\diamond$ (a uniquely ownable piece of ghost state) that is given to the second thread at the beginning of its execution. Inv_x simply asserts that it owns either the golden idol or the bag of sand. Thus, when the second thread learns of the existence of Inv_x, it can safely use the invariant opening and closing rules to swap the bag of sand in its possession for the golden idol owned by Inv_x, and thereafter claim local ownership of $x \hookrightarrow 37$.

$$\mathrm{Hist}(\ell, h) \triangleq \boxed{\circ\, [\ell := h]}^{\gamma_1}$$
$$\mathrm{Seen}(\pi, V) \triangleq \boxed{\circ\, [\pi := V]}^{\gamma_2}$$
$$\mathtt{PSInv} \triangleq \exists \sigma.\ \exists H \in \mathrm{Loc} \xrightarrow{\mathrm{fin}} \mathcal{P}(\mathrm{Val} \times \mathrm{Time} \times \mathrm{View}).\ \mathsf{Phys}(\sigma) * \boxed{\bullet\, H}^{\gamma_1} * \boxed{\bullet\, \sigma.T}^{\gamma_2} * \mathtt{HInv}(\sigma, H)$$
$$\mathtt{HInv}(\sigma, H) \triangleq \mathrm{dom}(H) = \{m.\ell \mid m \in \sigma.M\} \wedge (\forall m \in \sigma.M.\ m.t = m.V(m.\ell))$$
$$\wedge\, \forall \ell \in \mathrm{dom}(H).\ H(\ell) = \{(m.v, m.t, m.V) \mid m \in \sigma.M \wedge m.\ell = \ell \wedge m.t \geq \sigma.N(\ell)\}$$

Figure 8 Ghost state and invariant setup for λ_{RN}.

3.2 Instantiating Iris with λ_{RN}

We now consider an instantiation of Iris with our λ_{RN} language from §2.3. A key difference between λ_{RN} and λ_{SC} is that the expression reductions of λ_{SC} do not depend on which thread is executing the expression, since every thread has the same global view of the memory, whereas the reductions of λ_{RN} depend on the current thread's subjective view of the memory. Thus, we need to be able to talk about thread ids in our logic as well. To this end, we pair up expressions from λ_{RN} with thread ids, making them visible in our specifications. Eventually, in §4, we will see how we can reason about λ_{RN} without talking explicitly about thread ids.

3.2.1 Encoding Separation Logic for λ_{RN}

After instantiating Iris with λ_{RN}, as in the case of λ_{SC}, Iris provides us with large-footprint specifications of the primitive commands for free, which concern the physical state assertion and mirror the rules of λ_{RN}'s operational semantics. Recall that in λ_{RN} the physical state is a tuple (M, T, N) of the message pool M, the current view map T, and the non-atomic timestamp map N. As before, we aim to develop "local" assertions using ghost state, establish an invariant that connects those local assertions to the physical state assertion, and then derive small-footprint specifications of the primitive commands for use in modular verification. But what kind of "local" assertions do we want?

For λ_{SC}, we had the points-to assertion $\ell \hookrightarrow v$, but in λ_{RN} we no longer have a simple mapping from locations to values. Rather, associated with each location ℓ is a set of messages corresponding to writes to ℓ. We will represent that associated information as a *history* h, consisting of a set of triples (v, t, V), where v is a value written to ℓ, t is the timestamp at which that value was written, and V is the view of the writing thread at the time the write occurred. Note that V incorporates the new timestamp, *i.e.,* $V(\ell) = t$. To reflect the per-location history, we define our first local assertion: The *history ownership* assertion $\mathrm{Hist}(\ell, h)$ asserts ownership of ℓ and knowledge of its write history h.

Since the ability to read or write values in λ_{RN} depends on threads' local views of memory, we would also like to support an assertion of *thread-view ownership*, $\mathrm{Seen}(\pi, V)$, which asserts ownership of the current view V of the thread π and is required to update π's view. Since any operation by a thread π on a location ℓ relies on both π's current view and ℓ's history, the pair of history and thread-view ownership assertions comprise the general ghost ownership required for accessing the location.

To tie these local assertions to the global physical state, we use a construction very similar to the one for λ_{SC}. The local assertions are defined as before by wrapping finite-map PCMs with the authoritative PCM construction, and the invariant enforces that these ghost maps are coherent with the physical state. The definitions of the invariant `PSInv` and the two local assertions are given in Figure 8. The $\mathrm{Hist}(\ell, h)$ and $\mathrm{Seen}(\pi, V)$ assertions also have the exact same rules for exclusiveness, agreement, and updating as the $\ell \hookrightarrow v$ assertion of λ_{SC}.

Base-NA-Read

$$\boxed{\texttt{PSInv}} \vdash \{\mathrm{Seen}(\pi, V) * \mathrm{Hist}(\ell, h) * \mathrm{init}(h, V) * \mathrm{na}(h, V)\}$$
$$\ell_{[\mathrm{na}]}, \pi$$
$$\{v.\ \mathrm{Seen}(\pi, V) * \mathrm{Hist}(\ell, h) * \mathrm{na}(h, V) * \max(h).v = v\}$$

Base-NA-Write

$$\boxed{\texttt{PSInv}} \vdash \{\mathrm{Seen}(\pi, V) * \mathrm{Hist}(\ell, h) * \mathrm{alloc}(h, V)\}$$
$$\ell_{[\mathrm{na}]} := v, \pi$$
$$\{\exists V' \sqsupseteq V, t, h' = \{(v, t, V')\}.\ \mathrm{Seen}(\pi, V') * \mathrm{Hist}(\ell, h') * \mathrm{na}(h', V') * \mathrm{init}(h', V')\}$$

Base-AT-Read

$$\boxed{\texttt{PSInv}} \vdash \{\mathrm{Seen}(\pi, V) * \mathrm{Hist}(\ell, h) * \mathrm{init}(h, V)\}$$
$$\ell_{[\mathrm{at}]}, \pi$$
$$\{v.\ \exists t_1, V_1, V' \sqsupseteq V \sqcup V_1.\ \mathrm{Seen}(\pi, V') * \mathrm{Hist}(\ell, h) * (v, t_1, V_1) \in h * t_1 \geq V(\ell)\}$$

Base-AT-Write

$$\boxed{\texttt{PSInv}} \vdash \{\mathrm{Seen}(\pi, V) * \mathrm{Hist}(\ell, h) * \mathrm{alloc}(h, V)\}$$
$$\ell_{[\mathrm{at}]} := v, \pi$$
$$\{\exists V' \sqsupseteq V, t, h' = h \uplus \{(v, t, V')\}.\ \mathrm{Seen}(\pi, V') * \mathrm{Hist}(\ell, h') * \mathrm{init}(h', V')\}$$

Figure 9 Selected Hoare triples of λ_{RN} base logic.

It is important to note that the history ownership assertion $\mathrm{Hist}(\ell, h)$ does not record the full history of ℓ, but only write events of the *current era* (see §2.2), *i.e.,* only events as recent as or more recent than the last non-atomic write to ℓ. This is reflected in the last condition of `HInv`: $m.t \geq \sigma.N(\ell)$. Defining $\mathrm{Hist}(\ell, h)$ this way helps to simplify the job of the user of the logic in establishing the absence of races: by construction, it is impossible to even attempt to read (racily) from write events before the current era. In order to preserve this property of $\mathrm{Hist}(\ell, h)$, a non-atomic write $(v_{\mathrm{na}}, t_{\mathrm{na}}, V_{\mathrm{na}})$ must completely remove all write events currently in h (which would be racy to access now), and replace it with a new history h' that contains only the newly created non-atomic write event: $h' = \{(v_{\mathrm{na}}, t_{\mathrm{na}}, V_{\mathrm{na}})\}$.

With the physical state shared and its ghost counterpart splittable, we are ready to derive the small-footprint Hoare triples, which constitute a base logic that is powerful enough to verify programs in λ_{RN}. A selected set of these triples is given in Figure 9. The general pattern of these rules is that a thread π needs to own the history h of a location ℓ and its own thread view V in order to operate on ℓ. Additionally, π needs to show certain relations between h and V in order to guarantee the safety of its operations. These relations are represented by the alloc, init, and na predicates. The $\mathrm{alloc}(h, V)$ predicate (resp. $\mathrm{init}(h, V)$) asserts that V has observed an event in h which ensures that ℓ is allocated (resp. initialized). The $\mathrm{na}(h, V)$ predicate asserts that V has observed the *mo*-latest write event of h, which is needed to do a non-atomic read. Note that all of these predicates require that V has seen *some* event from h—*i.e.,* an event from the current era of ℓ—which, as discussed above, is a prerequisite for non-raciness.

These base logic rules provide a very concise explanation of λ_{RN}'s operational semantics. Base-AT-Read, for example, requires the current view's knowledge of ℓ being initialized and ensures that the new updated view V' is at least the join of the local view V and the view V_1 of the write event that the thread reads from. This event must be from h and not *mo*-earlier

Invariants:

$$\mathrm{Inv}_y(V_0) \triangleq \exists h.\ \mathrm{Hist}(y,h) * (0,_,V_0) \in h * \forall V_1, v_1 \neq 0.\ (v_1,_,V_1) \in h \Rightarrow \exists V_{37} \sqsubseteq V_1.\ \boxed{\mathrm{Inv}_x(V_{37})}$$

$$\mathrm{Inv}_x(V_{37}) \triangleq \boxed{\diamond} \vee \mathrm{Hist}(x,[(37,_,V_{37})])$$

Thread 1 proof outline:

$\{\mathrm{Seen}(\pi,V_0) * \mathrm{Hist}(x,[(0,_,V_x)]) * V_x \sqsubseteq V_0 * \boxed{\mathrm{Inv}_y(V_0)}\}$
$x_{[\mathrm{na}]} := 37$
$\{\exists V_{37} \sqsupseteq V_0.\ \mathrm{Seen}(\pi,V_{37}) * \mathrm{Hist}(x,[(37,_,V_{37})])\}$
$\{\mathrm{Seen}(\pi,V_{37}) * \boxed{\mathrm{Inv}_x(V_{37})}\}$

open Inv_y:
$\{\mathrm{Seen}(\pi,V_{37}) * \exists h.\ \mathrm{Hist}(y,h) * ...\}$
$y_{[\mathrm{at}]} := 1$
$\{\exists V_1 \sqsupseteq V_{37}.\ \mathrm{Seen}(\pi,V_1) * \mathrm{Hist}(y, h \uplus [(1,_,V_1)]) * \boxed{\mathrm{Inv}_x(V_{37})}\}$

$\{\mathrm{Seen}(\pi,V_1) * \boxed{\mathrm{Inv}_y(V_0)}\}$

Thread 2 proof outline:

$\{\mathrm{Seen}(\pi,V_0) * \boxed{\mathrm{Inv}_y(V_0)} * \boxed{\diamond}\}$
repeat $y_{[\mathrm{at}]}$;
$\{\exists V_1, V_{37}, V_2.\ V_2 \sqsupseteq V_1 \sqsupseteq V_{37} * \mathrm{Seen}(\pi,V_2) * \boxed{\mathrm{Inv}_x(V_{37})} * \boxed{\diamond}\}$
$\{\mathrm{Seen}(\pi,V_2) * V_{37} \sqsubseteq V_2 * \mathrm{Hist}(x,[(37,_,V_{37})])\}$
$x_{[\mathrm{na}]}$
$\{z.\ \mathrm{Seen}(\pi,V_2) * z = 37 * \mathrm{Hist}(x,[(37,_,V_{37})])\}$

Figure 10 Verification of MP in λ_{RN} base logic.

than the write event observed by the thread previously. Base-NA-Read is similar, except that it requires that the current view must have observed the *mo*-latest write event to ℓ, and therefore reads from that write event, which we denote by $\max(h)$. Base-NA-Write preserves the `HInv` invariant by proactively dropping from the history all the old write events, which are *mo*-before this non-atomic write. Notice that, unlike Base-NA-Read, Base-NA-Write does not require $\mathrm{na}(h,V)$, as explained in §2.2.

3.2.2 MP in λ_{RN}

We show that the base logic is enough to verify MP in λ_{RN}. The invariants and the proof, given in Figure 10, follow closely those used for MP in λ_{SC}. The singleton heap ownership in λ_{SC} is replaced with the history ownership, and extra conditions on view extension are added to reflect the view updates inherent in λ_{RN}. More specifically, in Inv_y, we enforce that any write of a non-zero value[6] to y be at a view V_1 that extends V_{37}, which is the view at which the write of 37 to x is made by the first thread. Consequently, when the second thread observes V_1 (by reading y to be non-zero), it must have also observed V_{37}, and thus can read $x = 37$, using the Indiana Jones invariant Inv_x. The extra conditions on V_0 ensure that y is initialized with 0 at V_0, so that the second thread (having observed V_0) can safely read y.

We have shown that the base logic is powerful enough to verify MP in λ_{RN}, and in principle it is powerful enough to verify many other realistic weak memory programs that

[6] In the MP example this value is always 1.

are expressible in λ_{RN} as well. However, it is also clear that the base logic is not abstract enough: one has to burden oneself with keeping track of the low-level details of changes to locations' histories and threads' views. What we really want is a way of abstracting away from those low-level details and finding simple high-level reasoning principles for λ_{RN}, the type of reasoning principles supported by RA+NA logics like GPS and RSL. We will now see how such high-level principles can in fact be derived on top of our low-level base logic.

4 iGPS

Vafeiadis *et al.* have introduced several logics for C11, and two in particular that were focused on RA+NA: GPS [33] and RSL [34]. The key difference between these logics and the RA+NA base logic presented in §3.2.1 is that, in GPS and RSL, the user does not reason explicitly about views—instead, the assertions of these logics are *implicitly* predicated over the view of the thread asserting them. This helps to hide much tedious reasoning about views, and leads naturally to a model of assertions as predicates over views (§4.3).

We have encoded iGPS and iRSL, variants of both GPS and RSL, in Iris, and we will focus here on iGPS, since it is the more sophisticated of the two logics. (We briefly describe iRSL in §5.) GPS introduced several useful abstractions that were not present in RSL: PCM-based ghost state, single-location protocols, and escrows. In encoding GPS in Iris, as noted in the introduction, we get PCM-based ghost state completely for free, just by working in Iris. Below, we will describe the other features, and how we account for them in Iris.

Note that our goal with iGPS is not to slavishly imitate all details of GPS, but to provide proof rules very similar to GPS's that are both strong enough to support all the examples from the GPS papers [33, 31] *and* significantly easier to prove sound within Iris. To this end, we slightly restrict select rules, but only in a way that does not impact their utility on known case studies. We discuss the differences from the original rules in detail in §6. Furthermore, in §4.2, we show how iGPS supports an additional feature—*single-writer protocols*—that was not supported by the original GPS and that significantly simplifies proofs.

4.1 Key Features of GPS

In this section, we explain the key features of GPS and how they are formalized in iGPS (besides PCM-based ghost state, which is directly imported into iGPS from Iris). A selected set of iGPS proof rules is given in Figure 11.

Non-atomics

Since non-atomic locations may not be raced on, GPS (and iGPS) reason about them much in the same way that locations are reasoned about in standard separation logic: using the points-to assertion, $\ell \hookrightarrow v$. Note that the proof rules for reading (iGPS-NA-Read) and writing (iGPS-NA-Write) and the exclusivity property (iGPS-NA-Exclusive) are equivalent to those from the logic for λ_{SC} (§3.1.1). Additionally, GPS (and iGPS) support fractional ownership of non-atomics to allow such locations to be read (but not written) by multiple threads at once. We omit the rules of fractional ownership for brevity.

Protocols

To reason about RA atomics, we need a mechanism for controlling interference on such accesses. Toward this end, CSLs for SC have supported a variety of *protocol* mechanisms, which control how shared state may evolve over time, and several of the more recent logics [32, 30, 23]

iGPS-NA-Read
$\{\ell \hookrightarrow v\}\ \ell_{[\mathrm{na}]}\ \{w.\ w = v * \ell \hookrightarrow v\}$

iGPS-NA-Write
$\{\ell \hookrightarrow _\}\ \ell_{[\mathrm{na}]} := v\ \{\ell \hookrightarrow v\}$

iGPS-NA-Exclusive
$\ell \hookrightarrow v * \ell \hookrightarrow w \Rrightarrow \bot$

iGPS-Read
$$\frac{\forall s' \sqsupseteq s, v.\ P * \tau_{\mathrm{read}}(s', v) \Rrightarrow Q}{\left\{\boxed{\ell : s \mid \tau} * P\right\}\ \ell_{[\mathrm{at}]}\ \left\{v.\ \exists s' \sqsupseteq s.\ \boxed{\ell : s' \mid \tau} * Q\right\}}$$

iGPS-Write
$$\frac{(\forall s''.\ s' \sqsupseteq s'') \qquad P \Rrightarrow \tau_{\mathrm{full}}(s', v) * Q}{\left\{\boxed{\ell : s \mid \tau} * P\right\}\ \ell_{[\mathrm{at}]} := v\ \left\{\boxed{\ell : s' \mid \tau} * Q\right\}}$$

iGPS-CAS
$$\frac{\forall s' \sqsupseteq s.\ P * \tau_{\mathrm{full}}(s', v_o) \Rrightarrow \exists s'' \sqsupseteq s'.\ \tau_{\mathrm{full}}(s'', v_n) * Q \qquad \forall s' \sqsupseteq s.\ P * \tau_{\mathrm{read}}(s', v_o) \Rrightarrow R}{\left\{\boxed{\ell : s \mid \tau} * P\right\}\ \mathsf{cas}(\ell, v_o, v_n)\ \left\{v.\ \exists s'' \sqsupseteq s.\ \boxed{\ell : s'' \mid \tau} * ((v = 1 \land Q) \lor (v = 0 \land R))\right\}}$$

iGPS-Persistent
$\boxed{\ell : s \mid \tau} \Rrightarrow \Box\boxed{\ell : s \mid \tau}$

iGPS-Agree
$\boxed{\ell : s_1 \mid \tau} * \boxed{\ell : s_2 \mid \tau} \Rrightarrow s_1 \sqsubseteq s_2 \lor s_2 \sqsubseteq s_1$

iGPS-Escrow-Intro
$Q \Rrightarrow [P \rightsquigarrow Q]$

iGPS-Escrow-Elim
$P \land [P \rightsquigarrow Q] \Rrightarrow Q$

iGPS-Escrow-Persistent
$[P \rightsquigarrow Q] \Rrightarrow \Box [P \rightsquigarrow Q]$

Figure 11 iGPS proof rules for non-atomics, protocols, and escrows.

employ *state transition systems* (STSs) to formalize such protocols. Crucially, protocols enforce *irreversibility*: the state of an STS protocol can only make forward progress over the course of a proof. For example, in §3.1.2, a protocol could enforce that the variable y could only progress from 0 to 1 but not back again. (We did not need to enforce that property to verify the MP example, but it is useful to be able to in general.) In Iris, protocols are encoded using a combination of invariants and ghost state.

Under weak memory, invariants and protocols are unsound in general because they require a single coherent history of updates to all locations. GPS showed how to partially restore protocol reasoning for weak memory with *single-location protocols*: protocols which restrict the evolution of a single shared location. Intuitively, single-location protocols are sound due to the per-location coherence property of C11 (often called "SC per location"): the writes to any single location are totally ordered (by *mo*). In particular, they maintain the invariant that the order of protocol states associated with writes is consistent with their timestamp (*mo*) order. If write event x to location ℓ with associated protocol state s_x is *mo*-before write event y (to the same location) with protocol state s_y, then s_x will be before s_y in protocol order. Thus, once a thread has observed that the protocol on ℓ has reached state s_x, it can from that point on only observe the protocol on ℓ to be in states that are accessible from s_x. This fulfills the expectation that protocol transitions are irreversible.

GPS protocols come equipped with an interpretation predicate which specifies the resources held by the protocol depending on the protocol's state and the location's value. The primitive operations on an atomic location serve to transfer resources in and out of its protocol:

- Writes may transfer resources into the protocol, but may not transfer anything out.
- Reads may not transfer any resources into the protocol, but they may transfer "knowledge" (*i.e.,* duplicable resources) out of it. They are restricted to transferring out duplicable resources because there may be many reads of the same write event.
- Updates (RMWs), by virtue of the physical synchronization they provide, may transfer resources both in and out of the protocol.

In iGPS, we represent protocols in a slightly different way, using *two* interpretations: a "full" interpretation, and a duplicable "read" interpretation that is implied by the full

interpretation. The intuition is that the read interpretation is used for reads (since they may only transfer duplicable resources out of the protocol) and the full interpretation is used for the other operations.

We will now present the formal definition of protocols and the corresponding proof rules.

▶ **Definition 3** (Protocols). A protocol τ comprises a non-empty state set $\mathbb{S}$, a reflexive, transitive transition relation $\sqsubseteq \subseteq \mathbb{S} \times \mathbb{S}$, and two interpretation predicates $\tau_m(\cdot, \cdot) \in \mathbb{S} \times \mathrm{Val} \to \mathsf{Prop}$ with $m \in \{\text{read}, \text{full}\}$ representing read and full interpretations, respectively. The interpretation predicate has to fulfill the following two laws:

$$\tau_{\text{full}}(s, v) \Rrightarrow \tau_{\text{full}}(s, v) * \tau_{\text{read}}(s, v) \qquad\qquad \tau_{\text{read}}(s, v) \Rrightarrow \tau_{\text{read}}(s, v) * \tau_{\text{read}}(s, v)$$

We write $\boxed{\ell : s \mid \tau}$ (as in GPS) to denote the persistent assertion that ℓ is governed by protocol τ and that the protocol has been observed in state s.

The first rule, iGPS-Agree, represents the guarantee that every protocol is always in a state consistent with all observations, *i.e.,* that all observed states can be linearly ordered w.r.t. to the transition relation $\sqsubseteq$.

iGPS-Read enables reading from a location governed by protocol τ—allowing the user to observe a future state s' of whatever state s it has previously observed, and providing the associated read interpretation τ_{read}.

Writes to the location are subject to iGPS-Write, which allows the user to move the protocol to a "final state" s'—*i.e.,* a state accessible from every state in the protocol—so long as they can provide τ_{full} for s'. This rule may seem very weak, since it forces the protocol into a final state, but this weakness derives from the need to handle the general case where there can be write-write races. Write-write races allow only very limited reasoning: both writes have to prove that their protocol state is later in protocol order to the other one. The write rule presented here solves this problem in a very simple way (see §6 for a comparison with the original GPS rule). In §4.2, we present a much stronger write rule that is optimized for the common case where there are no write-write races on the location.

Finally, iGPS-CAS governs updates. Its two premises represent the success and failure case, respectively. If the operation succeeds, the value read, v_o, is guaranteed to belong to a future state s'. The user picks the new state s'' depending on s' and establishes $\tau_{\text{full}}(s'', v_n)$, making use of $\tau_{\text{full}}(s', v_o)$. In case of a failure, the rule degenerates to iGPS-Read.

Escrows

A limitation of GPS protocols is that they offer no way to transfer ownership of (non-duplicable) resources from one thread to another unless the receiving thread performs physical synchronization via an update operation. For example, in our MP example, there is no update operation, and yet we want to transfer ownership of the non-atomic location x from the first thread to the second. For such an example, an additional mechanism for ownership transfer is required.

This motivates *escrows*, a mechanism for *logical* synchronization which, unlike protocols, is not tied to physical locations.

▶ **Definition 4** (Escrows). An escrow $[P \rightsquigarrow Q]$ consists of a *guard* resource P and a *payload* resource Q to be transferred. The guard resource P must be exclusive, i.e. $P * P \Rightarrow \bot$. The escrow assertion itself is persistent knowledge (freely duplicable).

The idea of escrows is really just a slight generalization of the "Indiana Jones invariant" Inv_x that we used in the proof of the MP example from §3.1.2. Following the explanation

$$\mathbf{XE}(x) \triangleq \left[\boxed{\diamond} \rightsquigarrow x \hookrightarrow 37\right]$$
$$\mathbf{YP}(x)(0, v) \triangleq v = 0$$
$$\mathbf{YP}(x)(1, v) \triangleq v = 1 * \mathbf{XE}(x)$$

$\left\{x \hookrightarrow 0 * \boxed{y : 0 \mid \mathbf{YP}(x)}\right\}$
$x_{[\mathrm{na}]} := 37$
$\{x \hookrightarrow 37\}$
$\left\{\mathbf{XE}(x) * \boxed{y : 0 \mid \mathbf{YP}(x)}\right\}$
$y_{[\mathrm{at}]} := 1$
$\left\{\boxed{y : 1 \mid \mathbf{YP}(x)}\right\}$

$\left\{\boxed{y : 0 \mid \mathbf{YP}(x)} * \boxed{\diamond}\right\}$
repeat $y_{[\mathrm{at}]}$;
$\left\{\boxed{y : 1 \mid \mathbf{YP}(x)} * \mathbf{XE}(x) * \boxed{\diamond}\right\}$
$\{x \hookrightarrow 37\}$
$x_{[\mathrm{na}]}$
$\{z.\ z = 37 * x \hookrightarrow 37\}$

Figure 12 Verification of MP with iGPS.

there, the payload resource Q is the "golden idol", the guard resource P is the "bag of sand", and the escrow allows one to swap P for Q. The restriction on exclusivity of P ensures that this swap can only be performed once.

The proof rules for escrows follow the above intuition. iGPS-Escrow-Intro places the payload resource Q in escrow. Any thread that learns of the existence of this escrow can then use iGPS-Escrow-Elim to trade ownership of the guard resource P for Q.[7]

Message passing in iGPS

The verification of MP using iGPS is given in Figure 12. Although the verification is sound for λ_{RN}, it is much simpler than the proof we carried out in the base logic of Iris, and is in fact very close in structure to the SC verification of MP in λ_{SC}. In particular, the Indiana Jones invariant Inv_x has now become an escrow **XE**, and the invariant Inv_y has now become a (2-state) iGPS protocol **YP**, but otherwise the steps are almost the same. The abstraction of iGPS has relieved us from the burden of reasoning with history and view updates explicitly.

4.2 Single-Writer Protocols

As we observed above, the iGPS protocol write rule suffers from a restriction: the user has to transition to a final state. This restriction is not present in CSLs for SC, which let the user pick the future state depending on the current one, much as iGPS-CAS does. Fortunately, in the common case when there are no write-write races to the location, this restriction can be lifted by *single-writer protocols*, a novel invention of iGPS.

A single-writer protocol splits the protocol assertion into two parts: an exclusive writer assertion $\boxed{\ell : s \mid \tau}_W$ and a persistent reader assertion $\boxed{\ell : s \mid \tau}_R$. Owning the writer assertion provides both the permission to change the state as well as the guarantee that no one else can change it. Owning the reader assertion only allows reads. Thus, the reader assertion represents a lower bound on the protocol state whereas the state contained in the writer assertion is exactly the most recent state of the protocol. The full proof rules for single-writer protocols are given in Figure 13, and of these, the write rule iGPS-SW-Write is the most important. The writer knows exactly what the current state is, and is free to pick the next state accordingly.

[7] The rule given for elimination is only sound if Q is "timeless", meaning that it only describes ownership of (ghost) state, not knowledge about protocols or escrows, as is the case in our message passing example. A more general rule, which returns Q under the later modality, is given in the appendix [1].

iGPS-SW-Exclusive-Writer
$$\boxed{\ell : s_1 \mid \tau}_W * \boxed{\ell : s_2 \mid \tau}_W \Rrightarrow \bot$$

iGPS-SW-Agree
$$\boxed{\ell : s_1 \mid \tau}_R * \boxed{\ell : s_2 \mid \tau}_R \Rrightarrow s_1 \sqsubseteq s_2 \vee s_2 \sqsubseteq s_1$$

iGPS-SW-Max
$$\boxed{\ell : s_1 \mid \tau}_W * \boxed{\ell : s_2 \mid \tau}_R \Rrightarrow s_1 \sqsupseteq s_2$$

iGPS-SW-Read-Exclusive
$$\left\{\boxed{\ell : s \mid \tau}_W\right\} \ell_{[\mathrm{at}]} \left\{v.\ \boxed{\ell : s \mid \tau}_W * \tau_{\mathrm{read}}(s, v)\right\}$$

iGPS-SW-Read
$$\frac{\forall s' \sqsupseteq s, v.\ P * \tau_{\mathrm{read}}(s', v) \Rrightarrow Q}{\left\{\boxed{\ell : s \mid \tau}_R * P\right\} \ell_{[\mathrm{at}]} \left\{v.\ \exists s' \sqsupseteq s.\ \boxed{\ell : s' \mid \tau}_R * Q\right\}}$$

iGPS-SW-Write
$$\frac{P * \boxed{\ell : s'' \mid \tau}_W * \tau_{\mathrm{full}}(s, _) \Rrightarrow \tau_{\mathrm{full}}(s'', v) * Q \qquad s'' \sqsupseteq s}{\left\{\boxed{\ell : s \mid \tau}_W * P\right\} \ell_{[\mathrm{at}]} := v \left\{\boxed{\ell : s'' \mid \tau}_R * Q\right\}}$$

Figure 13 A selection of single-writer proof rules.

Applications of single-writer protocols

Single-writer protocols provide more explicitly intuitive and concise proofs over normal protocols when there are no *write-write* races, *i.e.,* only one thread is writing to the location. This may mean that there is exactly one writer in the whole program, or (perhaps more typically) that the programmer is using sufficient synchronization to ensure that there is exactly one active writer at a time. Such is the case for several headlining examples verified in GPS, including *circular buffer*, *bounded ticket lock*, and *read-copy update* [33, 31].

In the original GPS proofs for these examples, the lack of single-writer protocols meant that the proofs had to employ a significant amount of tedious ghost state (mostly in the form of so-called "protocol tokens") to formalize the fact that the writing thread knew exactly which state the protocol had to be in at any given time. Using single-writer protocols, this reasoning is immediate from the iGPS-SW-Write rule. By removing the need for such boilerplate ghost state, single-writer protocols simplify and clarify the proofs of these examples.

An intriguing feature of the iGPS-SW-Write rule is that it is possible for the writer to relinquish ownership of the exclusive writer permission *while doing the write itself.* (This is why the writer permission appears in the precondition of the premise.) This extra flexibility is particularly useful when reasoning about RA implementations of locks (such as the bounded ticket lock). When the lock holder releases the lock (typically with a release write), this feature allows them to also give up their permission to do further release writes to the lock, so that it can be transferred to the next thread that acquires the lock.

4.3 The Model of iGPS

We now briefly describe the model of iGPS assertions. Figure 14 contains an excerpt of the encoding of the standard assertions and connectives of CSL as well as non-atomics and escrows. The somewhat more involved model of protocols is detailed in the appendix [1].

We model iGPS assertions as monotone predicates over views. The view parameter represents the current view of the thread making the assertion. The monotonicity requirement is motivated by the observation that the view of a thread only grows over the execution of a program. To ensure properties like the frame rule, it is therefore crucial that simply adding information to a view does not invalidate previously held (*e.g.,* framed) assertions. As a consequence of this requirement, we explicitly monotonize the encoding when necessary.

$$\boxed{a}^{\gamma} \triangleq \lambda_.\ \boxed{a}^{\gamma}$$
$$[\![\Box P]\!] \triangleq \lambda V.\ \Box[\![P]\!](V)$$
$$[\![P * Q]\!] \triangleq \lambda V.\ [\![P]\!](V) * [\![Q]\!](V)$$
$$[\![P \Rightarrow Q]\!] \triangleq \lambda V.\ \forall V' \sqsupseteq V.\ [\![P]\!](V') \Rightarrow [\![Q]\!](V')$$
$$[\![\ell \hookrightarrow v]\!] \triangleq \lambda V.\ \exists V_{\text{na}} \sqsubseteq V.\ \text{Hist}(\ell, \{(v, _, V_{\text{na}})\})$$
$$[\![[P \rightsquigarrow Q]]\!] \triangleq \lambda V.\ \exists V_0 \sqsubseteq V.\ \boxed{[\![P]\!](V_0) \vee [\![Q]\!](V_0)}$$

$$[\![\{P\}\ e\ \{v.\ Q\}]\!] \triangleq \lambda V.\ \forall V' \sqsupseteq V, \pi.\ \left\{\boxed{\texttt{PSInv}} * \text{Seen}(\pi, V') * [\![P]\!](V')\right\}\ (e, \pi)\ \{v.\ \exists V'' \sqsupseteq V'.\ \text{Seen}(\pi, V'') * [\![Q]\!](V'')\}$$

Figure 14 Definition of iGPS assertions.

We benefit greatly from the support offered by the surrounding logic. As a result, the model is extremely simple, with the lion's share of connectives being translated in a purely structural way and the remaining ones making direct use of ambient Iris connectives. The most interesting encodings are those of Hoare triples, non-atomics, escrows, and protocols. We now discuss these in more detail.

The model of Hoare triples embodies the intuition behind our encoding of iGPS assertions as view predicates: the view at which we operate is that of the local thread. In the encoding, we achieve this by quantifying over a view V' and tying V' to the physical view of the thread π via the $\text{Seen}(\pi, V')$ assertion and to the original pre-condition P, which is required to hold at V'. As the thread's physical view may evolve during the execution of the expression e, the triple returns an extended view $V'' \sqsupseteq V'$ and the corresponding $\text{Seen}(\pi, V'')$ assertion, together with the post-condition Q, which is guaranteed to be valid at V''.

The encoding of non-atomics is particularly simple due to the properties of the Hist assertion. As all writes in the history have to be *mo*-after the most recent non-atomic write, the history becomes a singleton when the location is used non-atomically. To tie the local view V to the view of the non-atomic write V_{na}, we simply demand that V extend V_{na}.

We encode escrows with a single, simple invariant, which holds either the guard resource P or the payload resource Q. The view V_0 at which the invariant owns one of these resources is the view used to initialize the escrow. Knowing an escrow at a local view V thus reduces to knowing that $V_0 \sqsubseteq V$.

Protocols

The model of iGPS protocols consists of two parts: a protocol invariant, and local protocol assertions given out to clients. The protocol invariant owns the location's history as well as the *logical history* Δ, which tracks the transition history of protocol states and is always kept in agreement with the location's history. Additionally, the invariant owns τ_{read} for all writes in the history and τ_{full} for select ones, depending on the protocol's type. For example, in normal protocols, τ_{full} is only stored for CAS-*able* write events, justifying that only an update can access the full interpretation of the write event that it reads from (see iGPS-CAS). Meanwhile, local protocol assertions hold knowledge about the logical history, which gives effectively lower bounds on the current state of the protocol, and by the protocol invariant, indirectly implies knowledge about the location's history. In the case of single-writer protocols, the writer assertion also owns the exclusive right to change the state. This construction also relies on the authoritative PCM (see §3.1.1). More details are given in the appendix [1].

Soundness of iGPS

The soundness of iGPS is expressed by the following theorem.

▶ **Theorem 1** (Adequacy). *For any expression e, physical state σ', and meta-level predicate on values Φ, we have*

$$(\vdash \{\top\}\ e\ \{v.\ \Phi(v)\}) \Rightarrow \forall \mathcal{TS}.\ \sigma_{init}; [0 \mapsto e] \to^{*} \sigma'; \mathcal{TS} \Rightarrow$$
$$(\mathcal{TS}(0) \in \mathit{Val} \Rightarrow \Phi(\mathcal{TS}(0)))$$
$$\wedge\, \forall \rho \in \mathit{dom}(\mathcal{TS}).\ \mathcal{TS}(\rho) \in \mathit{Val} \vee \exists \sigma'', e'', e''_f.\ \mathcal{TS}(\rho) \to^{\rho} \sigma'', e'', e''_f$$

The theorem connects iGPS program specifications ($\vdash \{\top\}\ e\ \{v.\ \Phi(v)\}$) and the program's possible executions, and provides two guarantees:

1. If the original thread with id 0 terminates with a value v, we know that $\Phi(v)$ holds.
2. For any pair of a state σ' and a threadpool $\mathcal{TS}$ reachable from the initial state σ_{init} (see Definition 2) and the initial threadpool $[0 \mapsto e]$, we know that any thread ρ in $\mathcal{TS}$ either has terminated with a value or can still be reduced in the state σ'.

The proof follows from the adequacy theorem of Iris.

5 Other Contributions

In our Coq development accompanying this paper, we make several additional contributions that we briefly summarize here.

An RSL encoding

Using the same base logic from §3.2.1, we have mechanized iRSL, a higher-order variant of RSL [34]. RSL focuses on the message-passing style transferring of resources through release-write/acquire-read pairs. The two main assertions of RSL are $\mathrm{Rel}(\ell, Q)$ and $\mathrm{Acq}(\ell, Q)$, representing the permission to write to and read from ℓ, respectively. The resource $Q(v)$ is released by writing v to ℓ, and then acquired by reading v from ℓ. Consequently, to support this MP-like mechanism, the encoding's model shares a great deal with the model of iGPS, namely the full vs. read interpretation construction for protocols.

One particular feature of RSL, however, demands special attention. Although simpler than GPS, RSL has the extra ability to split the receiver predicate Q into *smaller* predicates, so that different acquire reads of the same value v can acquire different parts of the transferred resource: $\mathrm{Acq}(\ell, \lambda v.\ Q_1(v) * Q_2(v)) \Rrightarrow \mathrm{Acq}(\ell, Q_1) * \mathrm{Acq}(\ell, Q_2)$. It is not obvious how to prove this sound when the splitting is completely arbitrary. Fortunately, a similar pattern, called the *barrier* pattern, has been addressed by Jung *et al.* [13], who propose the mechanism of "higher-order ghost state" to support such splitting. Our iRSL model basically extends Jung *et al.*'s barrier proof with a more complex (Iris) protocol to carefully manage resource splitting.

The encoding in Iris also provides us with useful extensions to the logic. Without extra work, iRSL naturally supports PCM-based ghost state and higher-order assertions, which were not available in the original RSL. The encoding shows that our approach has the right, reusable foundations to construct different logics for RA+NA.

In our RSL encoding, assertions are encoded as view predicates and proof rules are proven sound with respect to the base logic—in the same way as our GPS encoding. This allows us to soundly combine RSL and GPS reasoning principles in the same proof *at no additional cost.* It is even possible to design iGPS protocols whose state interpretation mentions iRSL assertions and vice versa. Of course, at a single point in time a location can only be governed by either iGPS or iRSL, as they represent incompatible modes of ownership transfer.

Allocation and deallocation

We have also incorporated support for memory allocation and deallocation into our RA+NA operational semantics. Since C11 is not clear about the semantics of allocation and deallocation, we take the liberty of defining them as reasonably as possible: in short, allocation and deallocation behave as non-atomic writes with special values A and D, respectively.

Fractional protocols

So far, all protocols presented are permanent: once the protocols are established, they govern their locations forever. This poses two interesting questions: 1) Can we change the protocol which governs a location? and 2) How can we deallocate a location governed by a protocol? To support these features, we derive, with little modification to the iGPS model, *fractional protocols*, whose protocol assertions also assert the permission to even *use* the protocol. Initially, a protocol τ for a location ℓ will be established with the full fraction, and then it will be distributed to those who want to use τ. Later, when the full fraction is recollected, one can *disable* the protocol (since no one else can use it), regain the *raw* ownership of ℓ, and then deallocate ℓ—or more interestingly, establish a new protocol for ℓ! These *recollectable* protocols open up possibilities for verifying programs that do custom memory reclamation, *e.g.,* RCU (see below). In the current Coq development, we have created fractional versions of both normal and single-writer protocols.

Mechanization and Case Studies

Our Coq mechanization employs a shallow embedding of iGPS (and iRSL), making critical use of the Iris Proof Mode [17]. In its current form, the proof mode is specific to the algebra of Iris and offers no additional support for embedded logics like our encoding of iGPS assertions. There are two consequences to this: 1) Unlike in the paper presentation of iGPS, where thread views are completely hidden in the (Iris) model of the logic, in our Coq proofs thread views are visible. However, they are also unobtrusive: all assertions in a given proof context always hold at the current thread's local view, and the view only changes when the thread takes a step. Thus, while the views are visible, they are always manipulated and kept in sync in a very straightforward way, which we mostly automate with a set of simple tactics. 2) iGPS assertions cannot always be manipulated directly by the proof mode. We sometimes have to unfold our embedding of iGPS assertions—but not in the statements of our lemmas and theorems—to make explicit the underlying Iris assertions so that the proof mode can operate on them. As our embedding is very simple, this has little additional overhead. In particular, all lemmas and theorems stated at the iGPS level remain easily applicable even to the unfolded definitions at the Iris level.

We have verified all of the standard examples that have been proven in previous work. The simplest of these is the *spin-lock* example, proven in iRSL. More interestingly, using iGPS, we have also mechanized the *message passing*, *circular buffer*, *bounded ticket lock*, and *Michael-Scott lock-free queue* examples, which were only verified by hand in the original GPS paper. We have also verified a variant of the *read-copy update* (RCU) technique employed in the Linux kernel, following the proof of Tassarotti *et al.* [31]. The RCU proof is the most substantial example in iGPS, which simplifies the original proof in GPS by using fractional single-writer protocols that allow garbage collection. To our knowledge, our development provides the very first mechanized proofs of the circular buffer, bounded ticket lock, Michael-Scott queue, and RCU examples in a weak-memory setting.

6 Related Work

This paper demonstrates one of the first major applications of the Iris framework. Other recent applications include Krebbers *et al.* [17], who developed the interactive Coq proof mode for Iris that we rely on heavily in this paper, and Krogh-Jespersen *et al.* [18], who use Iris to encode a logical-relations model of a relational model of a type-and-effect system for a higher-order, concurrent programming language. Neither of those papers considers weak memory.

There are a number of program logics for weak memory models [28, 5, 2, 20], some of which have mechanized soundness proofs [34, 8, 26], but none of which provide real support for mechanized proofs of weak-memory programs in the way that we do.

FSL++ [9], an extension of FSL [8] (with ghost state) and RSL (with relaxed accesses), was used to mechanize a proof of an implementation of atomic reference counters based on the one in Rust's `Arc` library. The proof is done by applying the basic laws of separation logic, resulting in painful manual work. Our approach alleviates a great deal of such tedium using the Iris proof mode. As of now, however, iGPS cannot reason about relaxed accesses.

More recently, RSL, FSL, and FSL++ have been encoded in Viper [22] to provide an automated verification approach to weak memory programs [29]. The encodings, however, axiomatize all proof rules without providing soundness guarantees, and are specific to the style of RSL and its FSL descendants. Our base logic, in contrast, is not tied to any specific surface logic and can be used to develop and prove sound different surface logics. It remains to be seen if the more expressive GPS protocols can be encoded in Viper.

iGPS is based on the GPS logic [33] and supports all reasoning mechanisms of GPS. However, the exact rules of iGPS differ in various small ways from the original ones in GPS. These differences stem from pragmatic choices made to simplify the soundness proof of iGPS. A particularly noteworthy difference from GPS appears in the premises of the iGPS-Read and iGPS-Write rules: the split of the protocol interpretations τ into τ_{read} and τ_{full}. We show the original GPS rules below for comparison.

GPS-Read

$$\frac{\forall s' \sqsupseteq s.\ P * \tau(s', v) \Rightarrow \Box Q}{\left\{\boxed{\ell : s \mid \tau} * P\right\}\ \ell_{[\text{at}]}\ \left\{v.\ \boxed{\ell : s' \mid \tau} * Q\right\}}$$

GPS-Write

$$\frac{\forall s' \sqsupseteq s.\ P * \tau(s', v) \Rightarrow s' \sqsubseteq s'' \qquad P \Rrightarrow \tau(s'', w) * Q}{\left\{\boxed{\ell : s \mid \tau} * P\right\}\ \ell_{[\text{at}]} := w\ \left\{\boxed{\ell : s'' \mid \tau} * Q\right\}}$$

The interpretation $\tau(s, v)$ in these two GPS rules is equivalent to $\tau_{\text{full}}(s, v)$ in iGPS. In GPS, the user can gain access to the full interpretation of the "current" protocol state (s'), but only to obtain some *knowledge*, not to consume (*i.e.,* transfer out of the protocol) any non-duplicable resources owned by that interpretation. This is enforced in GPS-Read by guarding Q with an always modality $\Box$, and in GPS-Write by requiring the user to establish the interpretation of the new write using only the local resource P. In contrast, iGPS does not provide the user with the full interpretation, but only the read interpretation τ_{read}. This weakens, for example, the iGPS-Write rule in comparison with GPS-Write, because the user cannot use τ_{full} to show $s' \sqsubseteq s''$.

The reason for this discrepancy between GPS and iGPS is that the soundness proof of GPS reasons about an entire program execution graph at once. With its bird's-eye view of the entire execution, GPS can, for the duration of a step, assemble resources that have been transferred elsewhere in the graph to re-construct τ_{full} for the user. The soundness of iGPS, on the other hand, is established in a simpler and more local manner, without involving reasoning about the full execution of the program. We avoid the global soundness argument of GPS and instead provide a pragmatic solution which—judging from our success in porting

GPS examples—is effectively as strong as GPS and, at the same time, makes for a very simple soundness proof: we maintain the duplicable τ_{read} for all (past) events and can thus easily provide it to the client of iGPS-Read.

Essentially, the reason our rules are as effective (if not more so) than those of GPS is that GPS provides one-size-fits-all rules, which are applicable to both programs with write-write races and those without, whereas we provide special support for the common case where there are no write-write races. For programs without those races, the rules provided by GPS are quite cumbersome to use and often require additional ghost state for bookkeeping. iGPS instead supports an optimized write rule for the common case in which there are no such races, via single-writer protocols. For the remaining cases, the rather simple-minded rule iGPS-Write appears to suffice in all the examples we have considered thus far.

Our operational semantics for RA draws heavily on Lahav *et al.*'s semantics for SRA [19]—a stronger variant of RA, which is equivalent to it in the absence of write-write races. SRA was developed to provide an intuitive operational characterization which is as efficiently implementable as RA. However, as we observe here, moving to an operational characterization does not in fact require any strengthening of the RA semantics (even the slight strengthening of SRA). The main difference between the operational semantics of SRA and the one we give for RA is that writes in SRA always take a globally maximal timestamp, whereas in RA they need not do so. The canonical example demonstrating this difference is the 2+2W example (see Lahav *et al.* [19] for more details).

Going beyond Lahav *et al.*, we offer the first operational account of the interaction of RA and non-atomic accesses. Our semantics corresponds to C11's for programs that do not mix atomic RMW operations and non-atomic reads at the same location. We feel this is a reasonable restriction, given that C11's treatment of programs mixing atomic and non-atomic accesses is already known to be problematic [3]. Our semantics does not correspond to C11's for arbitrary programs, as evidenced by the following example:

$$\mathbf{cas}(x, 0, 1) \;\Bigg\|\; \begin{array}{l} x_{[\text{at}]} := 2; \\ a := x_{[\text{na}]} \end{array}$$

C11 considers this program racy because, if the CAS succeeds, the first thread's update of x to 1 and the second thread's non-atomic read of x are not in a happens-before relation. In contrast, our semantics does not consider this a race because the non-atomic read is always guaranteed to read from the previous write with value 2. We find the C11 semantics for this program to be rather unintuitive, but leave a more thorough investigation of the issue to future work.

Our RA semantics may also be considered a close derivative of Kang *et al.*'s "promising" semantics [15], which is geared toward solving a broader problem with the full C11 model (the so-called "out-of-thin-air" problem). We look forward to using Iris to construct program logics for this promising semantics.

Acknowledgements

We would like to thank Mohit Vyas for spotting a mistake in our original proof of correspondence to C11, and Mark Batty for helpful conversations.

This research was supported in part by a European Research Council (ERC) Consolidator Grant for the project "RustBelt", funded under the European Union's Horizon 2020 Framework Programme (grant agreement no. 683289).

References

1. Technical appendix and Coq development accompanying this paper, available at the following URL: http://plv.mpi-sws.org/igps/.
2. Tatsuya Abe and Toshiyuki Maeda. Observation-based concurrent program logic for relaxed memory consistency models. In *APLAS*, pages 63–84, 2016.
3. Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. The problem of programming language concurrency semantics. In *ESOP*, 2015.
4. Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *POPL*, pages 55–66, 2011.
5. Richard Bornat, Jade Alglave, and Matthew J. Parkinson. New lace and arsenic: adventures in weak memory with a program logic. *CoRR*, abs/1512.01416, 2015.
6. Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. TaDA: A logic for time and data abstraction. In *ECOOP*, 2014.
7. T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP 2010*, volume 6183 of *LNCS*, pages 504–528. Springer, 2010.
8. Marko Doko and Viktor Vafeiadis. A program logic for C11 memory fences. In *VMCAI*, volume 9583 of *Lecture Notes in Computer Science*, pages 413–430. Springer, 2016.
9. Marko Doko and Viktor Vafeiadis. Tackling real-life relaxed concurrency with FSL++. In *ESOP*, 2017.
10. Derek Dreyer. The RustBelt project. http://plv.mpi-sws.org/rustbelt/.
11. Xinyu Feng. Local rely-guarantee reasoning. In *POPL*, pages 315–327, 2009.
12. C. B. Jones. Tentative steps toward a development method for interfering programs. *TOPLAS*, 5(4):596–619, 1983.
13. Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In *ICFP*, pages 256–269, 2016.
14. Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*, pages 637–650, 2015.
15. Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. In *POPL*, pages 175–189, 2017.
16. Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. The essence of higher-order concurrent separation logic. In *ESOP*, 2017.
17. Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In *POPL*, pages 205–217, 2017.
18. Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. A relational model of types-and-effects in higher-order concurrent separation logic. In *POPL*, pages 218–231, 2017.
19. Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. Taming release-acquire consistency. In *POPL*, POPL 2016, pages 649–662. ACM, 2016.
20. Ori Lahav and Viktor Vafeiadis. Owicki-Gries reasoning for weak memory models. In *Automata, Languages, and Programming, ICALP 2015*, volume 9135 of *LNCS*, pages 311–323. Springer, 2015.
21. Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.
22. Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *VMCAI*, pages 41–62, 2016.
23. Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *ESOP*, 2014.
24. Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

25 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

26 Tom Ridge. A rely-guarantee proof system for x86-TSO. In *VSTTE 2010*, volume 6217 of *LNCS*, pages 55–70. Springer, 2010.

27 Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized verification of fine-grained concurrent programs. In *PLDI*, pages 77–87, 2015.

28 Filip Sieczkowski, Kasper Svendsen, Lars Birkedal, and Jean Pichon-Pharabod. A separation logic for fictional sequential consistency. In *ESOP 2015*, volume 9032 of *LNCS*, pages 736–761. Springer, 2015.

29 Alexander Summers. Personal communication, 2017.

30 Kasper Svendsen and Lars Birkedal. Impredicative concurrent abstract predicates. In *ESOP*, 2014.

31 Joseph Tassarotti, Derek Dreyer, and Viktor Vafeiadis. Verifying read-copy-update in a logic for weak memory. In *36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 110–120. ACM, 2015.

32 Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*. ACM, 2013.

33 Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. In *OOPSLA*, OOPSLA 2014, pages 691–707. ACM, 2014.

34 Viktor Vafeiadis and Chinmay Narayan. Relaxed separation logic: A program logic for C11 concurrency. In *OOPSLA 2013*, pages 867–884. ACM, 2013.

35 Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR 2007*, volume 4703 of *LNCS*, pages 256–271. Springer, 2007.