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

Max Planck Institute for Software Systems (MPI-SWS)
{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 useful and feasible 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.

1 Introduction

Separation logic [22] 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 $P \ast Q$ 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 [21], 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 and low-level synchronization mechanisms, such as those employed in the implementations of non-blocking data structures [31, 9, 5, 28, 26, 20, 4].

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 [12, 11, 14, 15], and (2) the adaptation of CSLs to account for weak memory models, notably C11’s release-acquire (RA) consistency [30, 29, 6, 18]. 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 useful and feasible 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 [10] to the much more sophisticated and elaborate protocol mechanisms appearing in logics like CaReSL [28], iCAP [26], and TaDA [4]. In an attempt to consolidate the field, Jung et al. developed Iris [12, 11, 14], 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: just using 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 [24]) they only supported manual and error-prone “pencil-and-paper” proofs. The initial version of Iris [12] 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. [15] 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 [8].

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 [19]. 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 multicore 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) [2].

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 [30] and GPS [29]. 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 ground-
breaking, 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 [17], 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 completely new, and shows how to use timestamps to model races on non-atomics as stuck (unsafe) machine states. We have proven that 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 [29], 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 [30] 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 $\lambda_{RN}$. Subsequent sections will show how to build a logic for $\lambda_{RN}$ 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:

\[
x_{[na]} := 0; y_{[na]} := 0; \quad x_{[na]} := 37; y_{[at]} := 1 \quad \text{repeat } y_{[at]}; \quad x_{[na]}
\]
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 that is provably equivalent to C11’s declarative semantics (see Appendix A for formal definitions, statements and proof outlines). We will 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 $t \in \text{Time} \triangleq \mathbb{N}$. Every write to a location $\ell$ gets assigned a timestamp (that is unique for $\ell$), resulting in write events $\omega \in \text{Event} \triangleq \text{Loc} \times \text{Val} \times \text{Time}$, where $v \in \text{Val} \triangleq \mathbb{Z}$.

Using timestamps, the thread’s “progress” is represented by a view $V \in \text{View} \triangleq \text{Loc} \Downarrow \text{fin} \Rightarrow \text{Time}$, 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 messages $m \in \text{Msg} \triangleq \text{Event} \times \text{View}$. The machine state $\sigma$ comprises a message pool (called memory) and a view for every thread.

The $\lambda_{RN}$ 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 $\lambda_{RN}$. Here we focus on the machine reductions.

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

A write (Thread-Write) picks an unused timestamp $t$ for location $\ell$ that is greater than the thread’s view of $\ell$, updates the thread’s view to the new view $V'$ that includes $t$, and

\footnote{The full semantics also supports allocation, which induces an allocation value $A$. We do not mention it here for the sake of simplicity.}
With the operational semantics of RA, we can now sketch why MP (assuming for now that
a program that exhibits data races in
which is then included in the view
will return
Here, we use join operator $\sqcup$ on views:
$(V_1 \sqcup V_2)(\ell) = \max\{V_1(\ell), V_2(\ell)\}$ if $\ell \in \text{dom}(V_1) \cap \text{dom}(V_2)$;
$(V_1 \sqcup V_2)(\ell) = V_1(\ell)$ if $\ell \in \text{dom}(V_1) \setminus \text{dom}(V_2)$; and $(V_1 \sqcup V_2)(\ell)$ is undefined if $\ell \notin \text{dom}(V_1) \cup \text{dom}(V_2)$.

### 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

---

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

---

**Figure 1** Per-thread reductions for RA without NA.

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.\footnote{Here, we use join operator $\sqcup$ on views: $(V_1 \sqcup V_2)(\ell) = \max\{V_1(\ell), V_2(\ell)\}$ if $\ell \in \text{dom}(V_1) \cap \text{dom}(V_2)$; $(V_1 \sqcup V_2)(\ell) = V_1(\ell)$ if $\ell \in \text{dom}(V_1) \setminus \text{dom}(V_2)$; and $(V_1 \sqcup V_2)(\ell)$ is undefined if $\ell \notin \text{dom}(V_1) \cup \text{dom}(V_2)$.} Note that the message being read
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 n-o-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 an 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 $V_{37}$. A read
from $x$ is now guaranteed to read from that message or from a more recent one, but no more
recent one exists. Consequently, the return value will be 37.

---

2 Here, we use join operator $\sqcup$ on views: $(V_1 \sqcup V_2)(\ell) = \max\{V_1(\ell), V_2(\ell)\}$ if $\ell \in \text{dom}(V_1) \cap \text{dom}(V_2)$; $(V_1 \sqcup V_2)(\ell) = V_1(\ell)$ if $\ell \in \text{dom}(V_1) \setminus \text{dom}(V_2)$; and $(V_1 \sqcup V_2)(\ell)$ is undefined if $\ell \notin \text{dom}(V_1) \cup \text{dom}(V_2)$.
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 \( \bot_{\text{race}} \), the intent of which is captured by the following correspondence: a program is racy if and only if at least one of its machine executions can reach \( \bot_{\text{race}} \) (formally stated and proved in Appendix A).

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 \( \ell \), a thread \( \pi \) must have observed the most recent non-atomic write to \( \ell \), i.e., \( N(\ell) \leq T(\pi)(\ell) \).
- A thread \( \pi \) can only perform a non-atomic read from a location \( \ell \) if it has observed the most recent (atomic or non-atomic) write to \( \ell \), i.e., \( \exists t \in M \; (\ell, t, t_{-\bot}) \in M \) \( T(\pi)(\ell) < t \).

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 \( \ell \) starts a new “era” in \( \ell \)’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 \( \ell \) are allowed even when the thread has not observed the most
\( \nu \in \text{Val} ::= () \mid z \in \mathbb{Z} \mid \ell \in \text{Loc} \mid \text{fix}(f, x) \cdot e \)

\( \alpha \in \text{Access} ::= \text{at} \mid \text{na} \)

\( e \in \text{Expr} ::= \nu \mid e_1 e_2 \mid \text{if } z \text{ then } e_1 \text{ else } e_2 \mid \text{fork } e \mid \ell[\alpha] := \nu \mid \text{cas}(\ell, \nu, \nu) \mid \ldots \)

\[\begin{align*}
\ell[\alpha] & \xrightarrow{\text{Read}_{\alpha}, \ell, v} v, \text{nil} \\
\ell[\alpha] & \xrightarrow{\text{Write}_{\alpha}, \ell, v} () \text{, nil} \\
\text{cas}(\ell, \nu_0, \nu_n) & \xrightarrow{\text{Update}_{\alpha}, \ell, \nu_0, \nu_n} 1, \text{nil} \\
\text{cas}(\ell, \nu_0, \nu_n) & \xrightarrow{\text{Read}_{\alpha}, \ell, \nu_0} 0, \text{nil} \quad \text{if } v \neq \nu_0 \\
\text{fork } e & \xrightarrow{\text{Fork}_{\ell}} () \text{, } e \\
\text{if } z \text{ then } e_1 \text{ else } e_2 & \rightarrow e_1, \text{nil} \quad \text{if } z \neq 0 \\
\text{if } z \text{ then } e_1 \text{ else } e_2 & \rightarrow e_2, \text{nil} \quad \text{if } z = 0
\end{align*}\]

Figure 3: Main \( \lambda_{RN} \) expressions and expression reductions.

recent write to \( \ell \), as it is only required to observe the most recent non-atomic write to \( \ell \).
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 \( \bot \) race state), so the program will nevertheless be declared racy.

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.

\textbf{Definition 2 (Physical State).}

We define \( \sigma \in \Sigma \triangleq \{ \mathcal{P}(\text{Msg}) \times (\text{ThreadId} \ \widetilde{\times} \ \text{View}) \times \text{View} \} \uplus \{ \bot \text{-race}, \bot \text{-uninit} \} \), and denote the components \( M \), \( T \), and \( N \), respectively, if applicable. The initial physical state, denoted \( \sigma_0 \), is given by \( (\emptyset, 0 \mapsto \emptyset) \).

2.3 The \( \lambda_{RN} \) language

\( \lambda_{RN} \) is a standard lambda calculus with recursive functions, forks, and references with atomic and non-atomic accesses. The \texttt{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. Stateful reductions share memory event labels with the machine reductions, while non-stateful (pure) reductions are not labeled. The per-thread language reductions are then the combination of the expression reductions and the machine reductions, axiomatized with the \texttt{Combined-*} rules in Figure 4. Non-stateful reductions (\texttt{Combined-Pure}) simply defer to the expression reductions, while stateful reductions (\texttt{Combined-Mem} and \texttt{Combined-Fork}) use the event label \( \epsilon \) and the thread id \( \pi \) to tie the expression and machine reductions together correctly. These per-thread reductions then are lifted in a straightforward manner to the full machine (threadpool) reductions.
When we first instantiate Iris, the only primitive assertion we get about the state of the
1. machine is the physical state ownership assertion Phys(\varsigma), which asserts that \varsigma is the
2. current global state of the machine. Together with this assertion we also get for free a
3. 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
4. for updating a location in \lambda_SC will be \{\text{Phys}(\varsigma)\} \ell := v \{\text{Phys}(\varsigma[\ell \mapsto v])\}.
5. 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 \mapsto v,
and correspondingly give small-footprint specifications of the primitive commands, such as \{ \ell \mapsto w \} \ell := v \{ \ell \mapsto 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 \mapsto v, we will define this assertion so as to represent the knowledge that \ell currently points to v and the rights to read and write \ell.

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 \mapsto v, its “knowledge” that \ell 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 \( \lambda_{SC} \) is a standard lambda calculus with references. It is basically the same as \( \lambda_{RN} \) without atomic accesses (and therefore without a race detector). The two main reference expressions are \! \ell (a read from the location \ell) and \ell := v (a write of \( v \) to \ell), which are similar to non-atomic accesses in \( \lambda_{RN} \). The language’s physical state is a heap \( \varsigma \), which is a finite map from allocated locations to values. The main heap-related reductions of \( \lambda_{SC} \) are given in Figure 6. When we instantiate Iris with \( \lambda_{SC} \)’s operational semantics, (as explained in the above roadmap) what we get automatically from Iris are the following large-footprint Hoare triples concerning the physical state ownership assertion \( \text{Phys}(\varsigma) \):

\[
\begin{align*}
\text{Phys-Heap-Read} & \quad \{ \text{Phys}(\varsigma) \ast \varsigma(\ell) = v \} \ell \{ z, z = v \ast \text{Phys}(\varsigma) \} \\
\text{Phys-Heap-Write} & \quad \{ \text{Phys}(\varsigma) \} \ell := v \{ \text{Phys}(\varsigma[\ell \mapsto v]) \}
\end{align*}
\]

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 \( \lambda_{SC} \)

We would now like to encode these small-footprint Hoare triples for \( \lambda_{SC} \):

\[
\begin{align*}
\text{Heap-Read} & \quad \{ \ell \mapsto v \} \ell \{ z, z = v \ast \ell \mapsto v \} \\
\text{Heap-Write} & \quad \{ \ell \mapsto w \} \ell := v \{ \ell \mapsto v \}
\end{align*}
\]

The first step is to define the points-to assertion, \ell \mapsto 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 \(a : M_\gamma\) asserts the ownership of the ghost resource \(a\) for
an instance \(\gamma\) of the PCM \(M\). Separating conjunction for ghost state assertions simply lifts
the PCM composition operation to the assertion level: \(a : M_\gamma \otimes b : M_\gamma \iff \langle a, b \rangle : M_\gamma\).
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 \(a : M_\gamma \otimes b : M_\gamma \Rightarrow 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):
\[
\begin{align*}
\text{Ghost-Update} \quad 
\forall a_f, a \cdot a_f \neq \bot \Rightarrow b \cdot a_f \neq \bot
\end{align*}
\]
Ghost updates belong to the set of logical computations, or in Iris terminology, view shifts.
A view shift \(P \Rightarrow Q\) represents the capability of transforming a resource satisfying \(P\) into a
resource satisfying \(Q\) without changing the physical state.

A PCM for heaps

Let us now construct a PCM called \(\text{Heap}\) that has the same basic structure as the physical
heap, but allows splitting and recomposing. \(\text{Heap}\) 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 \(\ell\):
\[
\begin{align*}
\text{Ghost-Heap-Exclusive} \quad [\ell := v] \otimes [\ell := v] \not\vdash False
\end{align*}
\]
The singleton heap \([\ell := v]\) therefore has the desired properties for defining the local assertion
\(\ell \mapsto v\), although as we will see shortly, we need a slightly more sophisticated PCM for the
final definition of \(\ell \mapsto v\). Before we get there, let us turn attention to the key mechanism
that we will need to enforce our intended meaning of \(\ell \mapsto v\), namely 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 \(P^N\) 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

---

\(^3\) Actually, ghost state in Iris is based on the more general mechanism of CMRAs (aka step-indexed
resource algebras), which can support a more general form of higher-order ghost state \([11]\).

\(^4\) In the rest of the paper we also suppress the PCM \(M\) in \(a : M_\gamma\) when it can be inferred in context.

\(^5\) 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.
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 invariant masks (which are also namespaces), written \( P^{\mathcal{N} \rightarrow \mathcal{N}'} \), to assert that, assuming the invariants named in \( \mathcal{N} \) hold before the view shift, then the invariants named in \( \mathcal{N}' \) hold after the view shift. Opening and closing of invariants are then formalized as follows:

\[
\begin{align*}
\text{Inv-Open} & : P^{\mathcal{N}} \vdash \text{True}^{\mathcal{N} \rightarrow \emptyset} P \\
\text{Inv-Close} & : P^{\emptyset} \vdash P^{\emptyset} \rightarrow^{\mathcal{N}} \text{True}
\end{align*}
\]

\text{Inv-Open} allows a thread to gain ownership of \( P \) but prevents it from doing so more than once. Only after applying \text{Inv-Close} and re-establishing the invariant can the thread have the right mask (\( \mathcal{N} \)) needed to open it again.

\textbf{Note}: The \text{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 [14]), 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 masks, since Hoare triples combine both physical and logical computations. A Hoare triple \( \{ P \} e \{ Q \} \mathcal{N} \) with a mask \( \mathcal{N} \) 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”:

\[
\begin{align*}
\text{AConsq} & : P^{\mathcal{N} \cup \mathcal{N}'} \rightarrow^{\mathcal{N}} P' \\
& \{ P' \} e \{ v. Q' \}_{\mathcal{N}} \forall v. Q' \rightarrow^{\mathcal{N} \cup \mathcal{N}'} Q \\
& e \text{ takes 1 physical step}
\end{align*}
\]

Due to the fact that bookkeeping of namespaces is just a tedious detail, we will from now on suppress namespaces from definitions and proofs (Coq forces us to get these details right). We will always use disjoint namespaces to ensure correctness in opening invariants.

\textbf{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 need to employ an extremely useful construction called the authoritative PCM [12].

Given a base PCM \( \mathcal{M} \), the authoritative PCM \( \text{Auth}(\mathcal{M}) \) has two types of elements: authoritative \( \bullet a \) and non-authoritative \( \circ a \) (for \( a \in \mathcal{M} \)). For any instance \( \gamma, [a]_{\mathcal{M}} \) is exclusive (i.e., \( [a]_{\mathcal{M}} \bullet [a]_{\mathcal{M}} \vdash \text{False} \)), and is the main point of reference for all the non-authoritative fragments, in the sense that any ownable fragment \( [b]_{\mathcal{M}} \) 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**

$\begin{align*}
\text{\textbullet} x \in y \land \text{\textbullet} b', x \in y' & \implies \exists c. \ a = b \cdot c \\
\text{\textbullet} b', x \in y' & \implies b' \cdot c = b \cdot c
\end{align*}$

**AUTH-UPDATE**

$\begin{align*}
\exists c. \ a = b \cdot c \\
\implies b' \cdot c = b \cdot c
\end{align*}$

With these two rules in hand, we can derive the following rules for operations on AUTH(Heap):

**AGHOST-HEAP-EXCLUSIVE**

$\begin{align*}
\text{\textbullet} \mathbf{f} \in y & \land \text{\textbullet} \mathbf{f} \in y' \land \text{\textbullet} \mathbf{x} \in y \land \text{\textbullet} \mathbf{x} \in y', y, y' \models \text{False} \\
\text{\textbullet} \mathbf{f} \in y & \implies \mathbf{f} \in y', y \models \text{False}
\end{align*}$

**AGHOST-HEAP-UPDATE**

$\begin{align*}
\text{\textbullet} \mathbf{x} \\
\text{\textbullet} \mathbf{f} \in y & \implies \mathbf{f} \in y', y \models \text{False}
\end{align*}$

3.1.2 Verifying MP in $\lambda_{SC}$

Using the small-footprint triples, we are ready to verify MP in $\lambda_{SC}$. 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 after initialization, so we need to put ownership of $y$ into an invariant $\text{Inv}_y$. This invariant says that $y$ is in one of two states—0 or 1. When the second thread observes that it is in the 1 state, it will expect to be able to gain ownership of $x \leftarrow 37$. To achieve this, in the 1 state, $\text{Inv}_y$ asserts the existence of another invariant $\text{Inv}_x$ 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 $\text{Inv}_x$, 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 $x \leftarrow 37$, which is transferred into the invariant $\text{Inv}_x$ when it is established by the first thread. The “bag of sand” is a “token” $\mathbf{f}_0^0$ (a uniquely ownable piece of ghost state) that is given to the second thread at the beginning of its execution. $\text{Inv}_x$ simply asserts that it owns either the golden idol or the bag of sand. Thus, when the second thread learns of the existence of $\text{Inv}_x$, 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 $\text{Inv}_x$, and thereby claim local ownership of $x \leftarrow 37$. 
3.2 Instantiating Iris with \(\lambda_{RN}\)

We now consider an instantiation of Iris with our \(\lambda_{RN}\) language from §2.3. A key difference between \(\lambda_{RN}\) and \(\lambda_{SC}\) is that the expression reductions of \(\lambda_{SC}\) 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 \(\lambda_{RN}\) 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 \(\lambda_{RN}\) with thread ids, making them visible in our specifications. Eventually, in §4, we will see how we can reason about \(\lambda_{RN}\) without talking explicitly about thread ids.

3.2.1 Encoding Separation Logic for \(\lambda_{RN}\)

After instantiating Iris with \(\lambda_{RN}\), as in the case of \(\lambda_{SC}\), Iris provides us with large-footprint specifications of the primitive commands for free, which concern the physical state assertion and mirror the rules of \(\lambda_{RN}\)’s operational semantics. Recall that in \(\lambda_{RN}\) 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 \(\lambda_{SC}\), we had the points-to assertion \(\ell \rightarrow v\), but in \(\lambda_{RN}\) we no longer have a simple mapping from locations to values. Rather, associated with each location \(\ell\) is a set of messages corresponding to writes to \(\ell\). 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 \(\ell\), \(t\) is the timestamp at which that value was written, and \(V\) is the view of the writing thread at the time the write occurred. Correspondingly, one of the local assertions we want to define is history ownership, \(\text{Hist}(\ell, h)\), which asserts ownership of \(\ell\) and knowledge of its write history \(h\).

Since the ability to read or write values in \(\lambda_{RN}\) depends on threads’ local views of memory, we would also like to support an assertion of thread-view ownership, \(\text{Seen}(\pi, V)\), which asserts ownership of the current view \(V\) of the thread \(\pi\) and is required to update \(\pi\)’s view. Since any operation by a thread \(\pi\) on a location \(\ell\) relies on both \(\pi\)’s current view and \(\ell\)’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 \(\lambda_{SC}\). 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 \(\text{PSInv}\) and the two local
The base logic is enough to verify MP in \( \lambda_{RN} \). The invariant and the proof, which are given in Figure 10, follow closely those used for MP in \( \lambda_{SC} \). The singleton heap ownership in \( \lambda_{SC} \) is replaced with the history ownership, and extra conditions on view extension are added to reflect the view updates inherent in \( \lambda_{RN} \). More specifically, in Inv\(_y\)(\( V_0 \)), we enforce that the write of 1 to \( y \) is at a view \( V_1 \) that extends \( V_0 \), so that the second thread knows it can read the write of 1 when its local view is \( V_0 \), i.e., it can move.
from knowing $y = 0$ to knowing $y = 1$. Additionally, the write of 37 to $x$ must be at a view $V_{37} \sqsubseteq V_1$, so that when the second thread observes $V_1$ (by reading $y$ to be 1), it must also have observed $V_{37}$, and thus can read $x = 37$.

We have shown that the base logic is powerful enough to verify MP in $\lambda_{RN}$, and in principle it is powerful enough to verify many other realistic weak memory programs that are expressible in $\lambda_{RN}$ 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 $\lambda_{RN}$, 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 [29] and RSL [30]. 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
Invariants:

\[ \text{Inv}_y(V_0) \triangleq \text{Hist}(y, [(0, V_0)]) \lor \exists V_1 \sqsupseteq V_0. h. \text{Hist}(y, h \uplus [(1, V_1)]) \land \exists V_{37} \sqsubseteq V_1. \text{Inv}_x(V_{37}) \]

\[ \text{Inv}_x(V_{37}) \triangleq \exists z \lor \text{Hist}(x, [(37, V_{37})]) \]

Thread 1 proof outline:

\[
\begin{align*}
\{ & \text{Seen}(\pi, V_0) \land \text{Hist}(x, [(0, V_x)]) \land V_x \sqsubseteq V_0 \land \text{Inv}_y(V_0) \} \\
& \{ \text{yz} := 37 \} \\
& \{ \exists V_{37} \sqsupseteq V_0. \text{Seen}(\pi, V_{37}) \land \text{Hist}(x, [(37, V_{37})]) \} \\
& \{ \text{Seen}(\pi, V_{37}) \land \exists h. \text{Hist}(y, h) \} \\
& \{ \text{yz} := 1 \} \\
& \{ \exists V_1 \sqsupseteq V_{37} \sqsubseteq V_0. \text{Seen}(\pi, V_1) \land \text{Hist}(y, h \uplus [(1, V_1)]) \land \text{Inv}_x(V_{37}) \} \\
& \{ \text{Seen}(\pi, V_1) \land \text{Inv}_y(V_0) \} 
\end{align*}
\]

Thread 2 proof outline:

\[
\begin{align*}
\{ & \text{Seen}(\pi, V_0) \land \text{Inv}_y(V_0) \land \exists z \} \\
& \{ \text{yz} := 37 \} \\
& \text{repeat } y_{\text{at}} := 1; \\
& \{ \exists V_2 \sqsupseteq V_0. \exists V_1 \sqsubseteq V_2. \text{Seen}(\pi, V_2) \land \text{Inv}_x(V_1) \land \exists z \} \\
& \{ \text{Seen}(\pi, V_2) \land \exists V_{37} \sqsubseteq V_1 \sqsubseteq V_2. \text{Hist}(x, [(37, V_{37})]) \} \\
& \{ \text{yz} := 1 \} \\
& \{ \exists z. \text{Seen}(\pi, V_2) \land z = 37 \land \text{Hist}(x, [(37, V_{37})]) \} 
\end{align*}
\]

Figure 10 Verification of MP in \( \lambda_{\text{RN}} \).

Proof rules very similar to GPS’s that are both strong enough to support all the examples from the GPS papers [29, 27] 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, \( \ell \rightarrow v \). Note that the proof rules for reading (\( \text{GPS-NA-Read} \)) and writing (\( \text{GPS-NA-Write} \)) and the exclusivity property (\( \text{GPS-NA-Exclusive} \)) are equivalent to those from the logic for \( \lambda_{\text{SC}} \) (§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 [28, 26, 20] 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 \( \tau \)). 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 \( \ell \) 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 \( \ell \) has reached state \( s_x \), it can from that point on only observe the protocol on \( \ell \) 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.

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

\[
\begin{align*}
\tau_{\text{full}}(s, v) & \Rightarrow \tau_{\text{full}}(s, v) \circledast \tau_{\text{read}}(s, v) \\
\tau_{\text{read}}(s, v) & \Rightarrow \tau_{\text{read}}(s, v) \circledast \tau_{\text{read}}(s, v)
\end{align*}
\]

We write \(\ell : s \vdash \tau\) (as in GPS) to denote the persistent assertion that \(\ell\) is governed by protocol \(\tau\) and that the protocol has been observed in state \(s\).

The first rule, \(\text{iGPS-Agree}\), 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 \(\sqsubseteq\).

\(\text{iGPS-Read}\) enables reading from a location governed by protocol \(\tau\) – allowing the user to observe a future state \(s'\) of whatever state \(s\) it has previously observed, and providing the associated read interpretation \(\tau_{\text{read}}\).

Writes to the location are subject to \(\text{iGPS-Write}\), 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 \(\tau_{\text{full}}\) 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 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, \(\text{iGPS-CAS}\) 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 \(\text{iGPS-Read}\).

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.
Figure 12 Verification of MP with iGPS.

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

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

The idea of escrows is really just a slight generalization of the “Indiana Jones invariant” \(\text{Inv}_x\) that we used in the proof of the MP example from §3.1.2. Following the explanation 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 \(Q\).

Message passing in iGPS

The verification of MP using iGPS is given in Figure 12. Although the verification is sound for \(\lambda_{RN}\), 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 \(\lambda_{SC}\). In particular, the Indiana Jones invariant \(\text{Inv}_x\) has now become an escrow \(\text{XE}\), and the invariant \(\text{Inv}_y\) has now become a (2-state) iGPS protocol \(\text{YP}\), 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 \(\ell : s W\) and a persistent reader assertion \(\ell : s R\). 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.

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 two of the headlining examples in the original GPS paper, the circular buffer and bounded ticket lock.

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
The somewhat more involved model of protocols is detailed in the appendix.

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.

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 greater 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 $\pi$ 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 $V'' \supseteq V'$ and the corresponding $\text{Seen}(\pi, V'') \ast Q(V'')$ assertion, together with the original post-condition $Q$, which is now guaranteed to be valid at $V''$.

The encoding of non-atomics is particularly simple due to the properties of the $\text{Hist}$ assertion. As all writes in the history have to be $\text{mo}$-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 next turn to escrows, which we encode with a single, simple invariant. The invariant 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 $V \supseteq V_0$.

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 an authoritative fragment of the logical history $\Delta$—an augmented history that pairs write events with protocol states. Additionally, it owns $\tau_{\text{read}}$ for all writes and $\tau_{\text{full}}$ for select ones. Protocol assertions hold non-authoritative fragments of the logical history, which are effectively lower bounds on the current state of the protocol, as well as the exclusive right to change the state in the case of single-writer protocols. The full construction is explained in §F.
Soundness of iGPS

The soundness of iGPS is expressed by the following theorem.

**Theorem 1** (Adequacy). For all expressions $e$, threads $\pi$, physical states $\sigma'$, meta-level predicates on values $\Phi$ we have

\[
\begin{align*}
(\vdash \{ \top \} e \{ v. \Phi(v) \}) & \Rightarrow \forall TS. \sigma_0; [\pi \mapsto e] \rightarrow^* \sigma'; TS \\
& \Rightarrow (TS(\pi) \in Val \Rightarrow \Phi(TS(\pi))) \\
& \land \left( \forall \rho \in \text{dom}(TS). \ TS(\rho) \in Val \lor \exists \sigma'', e'', e''' f. TS(\rho) \rightarrow^\rho \sigma'', e'', e'''ight)
\end{align*}
\]

The theorem connects iGPS program specifications and the program’s possible executions and provides two guarantees:

1. If the original thread terminates with a value $v$, we know that $\Phi(v)$ holds.
2. For any pair of a state $\sigma'$ and a thread-pool $TS$ reachable from the initial state $\sigma$ and the initial thread-pool $[\pi \mapsto e]$, we know that any thread $\rho$ in the $TS$ will either have terminated with a value or can still be reduced in the state $\sigma'$.

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 a variant of RSL [30], which is a simpler predecessor of GPS. RSL focuses on the message-passing style transferring of resources through release-write/acquire-read pairs. The two main assertions of RSL are $\text{Rel}(\ell,Q)$ and $\text{Acq}(\ell,Q)$, representing the permission to write to and read from $\ell$, respectively. The resource $Q(v)$ is released by writing $v$ to $\ell$, and then acquired by reading $v$ from $\ell$. 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: $\text{Acq}(\ell, \lambda v. Q_1(v) \ast Q_2(v)) \Rightarrow \text{Acq}(\ell, Q_1) \ast \text{Acq}(\ell, Q_2)$. 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. [11], 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, which was not available in the original RSL, and higher-order assertions, which were simulated in the original RSL by using equivalence classes of syntactic assertions. The encoding shows that our approach has the right, reusable foundations to construct different logics for RA+NA. It also answers the open question (affirmatively) about whether proofs from RSL and GPS can be soundly composed.
Allocation and deallocation

We have also incorporated allocations and deallocations 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

In the context of deallocation, it is natural to ask if we can have recollectable protocols. Currently the protocols we have are permanent: once the protocols are established, they govern their locations forever. However, with little modification to iGPS’s model, we can derive fractional protocols, in which each protocol assertion also asserts the permission to even use the protocol. Initially, a protocol $\tau$ for a location $\ell$ will be established with the full fraction, and then it will be distributed to those who want to use $\tau$. Later, when the full fraction is recollected, one can disable the protocol (since no one else can use it), regain the raw permission of $\ell$, then deallocate $\ell$, or more interestingly, establish a new protocol for $\ell$! This type of recollectable protocols opens up possibilities for verifying programs that do explicit garbage collection. In the current Coq development, we have created fractional versions of both normal and single-writer protocols.

Mechanized examples

We have verified in Coq (using Iris) several of the standard examples that have been proven in previous work. In particular, we have mechanized the spin-lock example in iRSL. More interestingly, using iGPS, we have also mechanized the message passing, circular buffer, and bounded ticket lock examples, which were only verified by hand in the original GPS paper. To our knowledge, our development provides the very first mechanized proofs of the circular buffer and bounded ticket lock 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. [15], who developed the interactive Coq proof mode for Iris that we rely on heavily in this paper, and Krogh-Jespersen et al. [16], 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 [25, 3, 1], some of which have mechanized soundness proofs [18, 30, 6, 23]. However, no existing logics support mechanized proofs of programs under C11’s weak memory semantics with the same ease afforded to mechanized proofs of SC programs.

FSL++ [7], an extension of FSL [6], was used to mechanize a proof of an implementation of atomic reference counters based on the one in Rust. The proof is done by applying the basic laws of separation logic, resulting in painful manual work. By using the Iris proof mode, we can automate a great deal of such tedium.

iGPS is based on the GPS logic [29] 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. There are two particularly striking differences from GPS: the split of the protocol
interpretations $\tau$ into $\tau_{\text{read}}$ and $\tau_{\text{full}}$ and the premise of the iGPS-Read, and the premise of iGPS-Write. We show the original GPS rules below for comparison.

\[
\begin{align*}
\text{GPS-Read} & \quad \forall s' \sqsupseteq s. P \cdot \tau(s', v) \Rightarrow \Box Q \\
\{ \ell : s \cdot \tau \} & \quad \{ v. \ell : s' \cdot \tau \} \\
\text{GPS-Write} & \quad \forall s' \sqsupseteq s. P \cdot \tau(s', v) \Rightarrow s'' \sqsupseteq s' \quad P \Rightarrow \tau(s'') \cdot Q \\
\{ \ell : s \cdot \tau \} & \quad \{ \ell : s'' \cdot \tau \} \\
\end{align*}
\]

The reason for these differences stems from the different approaches to proving soundness: the soundness proof of GPS reasons about an entire execution graph at once, while iGPS proofs are done locally without additional assumptions about the full execution of the program. With its bird’s-eye view of the entire execution, GPS can, for the duration of a step, re-assemble resources that have been transferred out of memory events throughout the execution graph. Thus, GPS-Read is thus able to provide $\tau_{\text{full}}$ to the user. However, since the resources have to be returned to their current place, it does not allow the extraction of ownership.

In iGPS, we avoid these global arguments 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 $\tau_{\text{read}}$ for every event and can thus provide it to the client of iGPS-Read. Essentially, the reason our rules are as effective (if not more so) than GPS’s 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 states for bookkeeping. iGPS provides an optimized write rule for the common case in which there are no such races, via single-writer protocols. The iGPS rule for the case of write-write races seems to be good enough in practice.

Our operational semantics for RA draws heavily on that for SRA [17]—a stronger variant of RA, which is equivalent to it in the absence of write-write races. In SRA, writes 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. [17] for more details).

Our RA semantics may also be considered an instance of Kang et al.’s “promising” semantics [13], which is geared toward solving a broader problem (the “out of thin air” problem) with the full C11 model. We look forward to using Iris to construct program logics for this promising semantics.

Our treatment of non-atomics, however, is entirely novel.

References


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


### A.1 Declarative Semantics

We define threadpool reduction that generates traces:

<table>
<thead>
<tr>
<th>Trace-Red-Silent</th>
<th>Trace-Red-Mem</th>
<th>Trace-Red-Fork</th>
</tr>
</thead>
<tbody>
<tr>
<td>$TS(\pi) \rightarrow e, \text{nil}$</td>
<td>$TS(\pi) \rightarrow e, \text{nil}$</td>
<td>$TS(Fork,\rho) \rightarrow e, e_f$</td>
</tr>
<tr>
<td>$\Rightarrow \pi , TS \Rightarrow \pi , TS[\pi \rightarrow e]$</td>
<td>$\Rightarrow \pi , TS \Rightarrow \pi , TS[\pi \rightarrow e]$</td>
<td>$\Rightarrow \pi , TS \Rightarrow \pi , TS[\pi \rightarrow e] \cup [\rho \rightarrow e_f]$</td>
</tr>
</tbody>
</table>

We write $TS \Rightarrow \pi \, TS'$ if $TS \Rightarrow \pi \, TS'$ for some transition label $\pi$: and $TS \Rightarrow \pi \, TS'$ if $TS \Rightarrow \pi \, TS'$ for some transition label $\pi$ and thread identifier $\pi$. A threadpool is called final if $TS(\pi) \in \text{Val}$ for every $\pi \in \text{dom}(TS)$.

**Definition 2.** A trace is a sequence of pairs $\langle \varepsilon_1, \pi_1 \rangle, ..., \langle \varepsilon_n, \pi_n \rangle$. We say that $tr = \langle \varepsilon_1, \pi_1 \rangle, ..., \langle \varepsilon_n, \pi_n \rangle$ is a trace of an expression $e$ if

$$[0 \Rightarrow e] \xrightarrow{\varepsilon_1} \xrightarrow{\pi_1} \xrightarrow{\varepsilon_2} \xrightarrow{\pi_2} \cdots \xrightarrow{\varepsilon_n} \xrightarrow{\pi_n} \xrightarrow{\text{Final}} TS$$

for some thread $\pi$ and threadpool $TS$. When $TS$ is final, we call $tr$ a full trace.

**Lemma 3 (Receptiveness).** Let $\langle \varepsilon_1, \pi_1 \rangle, ..., \langle \varepsilon_n, \pi_n \rangle$ be a trace of an expression $e$.

- If $\varepsilon_n = \langle \text{Read}_a, \ell, v \rangle$, then for every value $v'$, $\langle \varepsilon_1, \pi_1 \rangle, ..., \langle \text{Read}_a, \ell, v' \rangle, \langle \varepsilon_n, \pi_n \rangle$ is a trace of $e$.
- If $\varepsilon_n = \langle \text{Update}, \ell, v_o, v_n \rangle$, then for every value $v'_o$, either $\langle \varepsilon_1, \pi_1 \rangle, ..., \langle \text{Read}_a, \ell, v'_o \rangle, \langle \varepsilon_n, \pi_n \rangle$ is a trace of $e$, or there exists a value $v'_n$ such that $\langle \varepsilon_1, \pi_1 \rangle, ..., \langle \text{Update}, \ell, v'_o, v'_n \rangle, \langle \varepsilon_n, \pi_n \rangle$ is a trace of $e$.

**Definition 4.** A trace $tr = \langle \varepsilon_1, \pi_1 \rangle, ..., \langle \varepsilon_n, \pi_n \rangle$ induces a partial order $\text{sb}(tr)$ (sequence-before) on $\{1, ..., n\}$ that is given by:

$$k_1 < k_2 \quad \pi_{k_1} = \pi_{k_2}$$

$$(k_1, k_2) \in \text{sb}(tr) \quad \text{if} \quad \varepsilon_{k_1} = \langle \text{Fork}, \pi_{k_2} \rangle$$

$$(k_1, k_2) \in \text{sb}(tr) \quad \text{if} \quad \pi_{k_1} \in \text{sb}(tr)$$

$$(k_1, k_2) \in \text{sb}(tr) \quad \text{if} \quad \pi_{k_2} \in \text{sb}(tr)$$

**Lemma 5.** Let $tr$ be a trace of an expression $e$. Then:

- Any prefix of $tr$ is a trace of $e$.
- Any permutation of $tr$ that respects $\text{sb}(tr)$ is a trace of $e$.

**Definition 6.** An execution graph $G$ consists of:

1. a finite set of nodes $E \subseteq N$ (called events).
2. a function $\text{tid}$ assigning a thread identifier to every event in $E$. We denote by $E^\pi$ the set of events in $a \in E$ with $\text{tid}(a) = \pi$.
3. a function $\text{lab}$ assigning a label to every event in $E$. Labels take one of the following forms: $\langle \text{Read}_a, \ell, v \rangle$, $\langle \text{Write}_a, \ell, v \rangle$, $\langle \text{Update}, \ell, v_o, v_n \rangle$, or $\langle \text{Fork}, \pi \rangle$, where $a \in \{\text{at, na}\}$ is the access mode, $\ell$ is the location accessed, and $v, v_o, v_n$ are read/written values, and $\pi$ is a thread identifier. The function $\text{lab}$ naturally induces functions $\text{typ}, \text{mod}, \text{loc}, \text{val}_l$, and $\text{val}_u$ that return (when applicable) the type ($\text{FORK}, R, W$ or $U$), access mode (in particular, $\text{mod}(a) = \text{at}$ if $\text{typ}(a) = U$), location, and read/written values of an event. For $T \in \{S, R, W, U\}$, $T$ denotes the set $\{a \in E \mid \text{typ}(a) = T\}$. We also concatenate the event sets notations, use subscripts to denote the accessed location, and superscripts for access modes (e.g., $W_1^U$ denotes all events $a \in W \cup U$ with $\text{loc}(a) = \ell$ and $\text{mod}(a) = \text{at}$).
4. a strict partial order \( sb \subseteq E \times E \), called \textit{sequenced-before}, (a.k.a. \textit{program order}).

5. a binary relation \( rf \subseteq E \times E \) called \textit{reads-from}, satisfying: (i) \( a \in \mathbb{WU} \) and \( b \in \mathbb{RU} \) for every \( (a, b) \in rf \); (ii) \( \text{loc}(a) = \text{loc}(b) \) and \( \text{val}_a(a) = \text{val}_b(b) \) for every \( (a, b) \in rf \); and (iii) \( a_1 = a_2 \) whenever \( (a_1, b), (a_2, b) \in rf \).

6. a strict partial order \( mo \) on \( \mathbb{WU} \), called \textit{modification order}, which is a disjoint union of relations \( \{ mo_\ell \}_{\ell \in \text{Loc}} \), such that each \( mo_\ell \) is a strict total order on \( \mathbb{WU}_\ell \).

In what follows, to resolve ambiguities, we may include a prefix “\( G \)” to refer to the components of an execution graph \( G \).

\begin{definition}
An execution graph \( G \) \textit{follows} a trace \( tr = \langle \varepsilon_1, \pi_1 \rangle, \ldots, \langle \varepsilon_n, \pi_n \rangle \) if the following hold:
\begin{itemize}
  \item \( E \) consists of \( n \) events \( a_1, \ldots, a_n \) with \( \text{lab}(a_k) = \varepsilon_k \) and \( \text{tid}(a_k) = \pi_k \) for every \( 1 \leq k \leq n \).
  \item \( sb = \{ \langle a_{k_1}, a_{k_2} \rangle \mid (k_1, k_2) \in sb(tr) \} \).
\end{itemize}
We call \( G \) an \textit{execution graph of an expression} \( e \) if \( G \) follows some trace of \( e \).
\end{definition}

Note that \( rf \) and \( mo \) are not restricted.

\begin{definition}
Given an execution graph \( G \), the relations \( sw \) (\textit{synchronization}) and \( hb \) (\textit{happens-before}) are given by:
\[ sw = \{ \langle a, b \rangle \in rf \mid \text{mod}(a) = \text{mod}(b) = \text{at} \} \quad \text{and} \quad hb = (sb \cup sw)^+ \]
\end{definition}

\begin{definition}[Consistency]
An execution graph \( G \) is \textit{consistent} if the following hold:
\begin{itemize}
  \item For every \( b \in \mathbb{RU} \), if there exists some \( a \in \mathbb{WU}\text{loc}(b) \) such that \( \langle a, b \rangle \in hb^+; sb \), then there exists some \( c \in E \).
  \item \( sb \cup rf \) is acyclic.
  \item \( rf; hb \) is irreflexive.
  \item \( rf; [U] \cap (mo; mo) = \emptyset \).
\end{itemize}
\end{definition}

\begin{definition}
A trace \( tr \) is \textit{allowed} for an expression \( e \) if \( tr \) is a trace of \( e \) and there exists a consistent execution graph \( G \) that follows \( tr \).
\end{definition}

\begin{definition}[Races]
Two events \( a \) and \( b \) are called \textit{conflicting} in an execution graph \( G \) if \( a, b \in E \), \( \{ \text{typ}(a), \text{typ}(b) \} \cap \{ \mathbb{W}, \mathbb{U} \} \neq \emptyset \), \( \neq b \), and \( 1\text{oc}(a) = 1\text{oc}(b) \). A pair \( \langle a, b \rangle \) is called a \textit{race} in \( G \) if \( a \) and \( b \) are conflicting events in \( G \), and \( \langle a, b \rangle \not\in hb \cup hb^{-1} \). An execution graph \( G \) is called \textit{racy} if there is some race \( \langle a, b \rangle \) in \( G \) with \( na \in \{ \text{mod}(a), \text{mod}(b) \} \).
\end{definition}

\begin{definition}[Initialization]
A location \( \ell \) is called \textit{uninitialized} in an execution graph \( G \) if there exists some \( b \in \mathbb{RU}_\ell \) such that \( \langle a, b \rangle \not\in hb \cup hb^{-1} \); \( sb \) for every \( a \in \mathbb{RU}_\ell \). An execution graph \( G \) is called \textit{uninitialized} if some location is uninitialized in \( G \).
\end{definition}

\begin{definition}[Buggy expression]
An expression \( e \) is called \textit{buggy} if some consistent execution graph of \( e \) is either racy or uninitialized.
\end{definition}

The following definition and lemma are useful in the sequel.

\begin{definition}
Given an execution graph \( G \), we denote by \( mo^{na} \) the relation \( mo \cap ((\mathbb{W}^{na} \times E) \cup (E \times \mathbb{W}^{na})) \).
\end{definition}

\begin{lemma}
Let \( G \) be a non-racy consistent execution graph. Then, \( hb = (sb \cup rf \cup mo^{na})^+ \).
\end{lemma}

\begin{proof}
Since \( G \) is not racy, we have \( rf \cup mo^{na} \subseteq hb \cup hb^{-1} \). Since \( G \) is consistent, \( rf; hb \) and \( mo^{na}; hb \) are irreflexive, and so \( rf \cup mo^{na} \subseteq hb \). It easily follows that \( hb = (sb \cup rf \cup mo^{na})^+ \).
\end{proof}
A.1.1 Alternative declarative semantics

Next, we present an alternative declarative semantics in which $hb$ is substituted by $(sb \cup rf)^+$. This semantics simplifies our proof below of the equivalence to the operational semantics.

Definition 16 (Strong consistency). Strong consistency is defined exactly as consistency with $(sb \cup rf)^+$ instead of $hb$.

Observing that $hb \subseteq (sb \cup rf)^+$ in every execution graph, it is easy to see that strong consistency implies consistency. In addition, strong consistency can be simplified as follows:

Lemma 17. An execution graph $G$ is strongly consistent if the following hold:

- For every $b \in RU$, if there exists some $a \in WU_{loc(b)}$ such that $(a, b) \in (sb \cup rf)^+; sb$, then $(c, b) \in rf$ for some $c \in E$.
- $(sb \cup rf)$ is acyclic.
- $mo; (sb \cup rf)$ is irreflexive.
- $mo; (sb \cup rf)^+; rf^{-1}$ is irreflexive.
- $\{ \rf; \emptyset \} \cap (mo; mo) = \emptyset$.

Definition 18 (Strong races). A strong race is defined exactly as a race with $(sb \cup rf)^+$ instead of $hb$. We say that $G$ is strongly racy if there is some strong race $(a, b)$ in $G$ with $na \in \{ mod(a), mod(b) \}$.

Definition 19 (Strong initialization). A location $\ell$ is called strongly uninitialized in an execution graph $G$ if there exists some $b \in RU$ such that $(a, b) \not\in (sb \cup rf)^+; sb$ for every $a \in WU$. An execution graph $G$ is called strongly uninitialized if some location is strongly uninitialized in $G$.

Notation 1. Let $G$ be an execution graph. For a set $E \subseteq E$, we denote by $G \cap E$ the execution graph $G'$ given by: $G'.E = E, G'.tid = G.tid|_E, G'.lab = G.lab|_E, G'.sb = G.sb \cap (E \times E), G'.rf = G.rf \cap (E \times E)$, and $G'.mo = G.mo \cap (E \times E)$. Given $a \in E$, we write $G \setminus \{ a \}$ for $G \cap (E \setminus \{ a \})$. Given $R \subseteq E \times E$, by $R$-prefix of $G$ we mean any execution graph $G'$ such that $G' = G \cap E$ for some set $E$ satisfying $a \in E$ whenever $b \in E$ and $(a, b) \in R$.

Lemma 20 (Adding writes). Let $G$ be an execution graph and $a \in W$ be an $(sb \cup rf)^+$-maximal event in $E$. Suppose that $G \setminus \{ a \}$ is strongly consistent. Let $G'$ be the execution graph that is obtained from $G$ by making $a$ an $mo$-maximal event. Then, $G'$ is strongly consistent.

Lemma 21 (Adding reads). Let $G$ be an execution graph and $a \in R$ be an $(sb \cup rf)^+$-maximal event in $E$. Suppose that $G \setminus \{ a \}$ is strongly consistent, and that $(b, a) \in (sb \cup rf)^+; sb$ for some $b \in WU_{loc(a)}$. Then, there exists a strongly consistent execution graph $G'$ that satisfies the following:

- $G'.E = G.E, G'.tid = G.tid, G'.sb = G.sb, \text{ and } G'.mo = G.mo$.
- $G'.lab(b) = G.lab(b)$ for every $b \in G.E \setminus \{ a \}$ and $G'.lab(a) = \langle \text{Read}_{G,mod(a),G.loc(a)}, v \rangle$ for some value $v$.
- $G'.rf \cap (G.E \times (G.E \setminus \{ a \})) = G.rf \cap (G.E \times (G.E \setminus \{ a \}))$.
- $G'.rf \cap (G.E \times \{ a \}) \subseteq (G'.sb \cup G'.rf)^+; G'.sb$.

**Proof.** Let $B = \{ b \in WU_{loc(a)} \mid (b, a) \in (sb \cup rf)^+; sb \}$, and let $b = \max_{sb} B$. Let $G'$ be the execution graph obtained from $G$ by replacing the value read at $a$ by $val_b(b)$, and adding the $rf$-edge $(b, a)$ (instead of the $rf$-edge entering $a$ in $G$, if such exists). It is straightforward to see that $G'$ is a strongly consistent. □
Lemma 22 (Adding updates). Let $G$ be an execution graph and $a \in U$ be an $(ab \cup rf)^+$-maximal event in $E$. Suppose that $G \setminus \{a\}$ is strongly consistent, and that $(b, a) \in (ab \cup rf)^*$; $ab$ for some $b \in W_{loc(a)}$. Then, there exists a strongly consistent execution graph $G'$ that satisfies the following:

2. $G'.lab(b) = G.lab(b)$ for every $b \in G.E \setminus \{a\}$ and $G'.lab(a) = \langle \text{Read}_{G',mod(a)}, G.loc(a), v \rangle$ for some value $v$.
3. $G'.rf \cap (G.E \times (G.E \setminus \{a\})) = G.rf \cap (G.E \times (G.E \setminus \{a\}))$.
4. $G'.rf \cap (G.E \times \{a\}) \subseteq ((G'.ab \cup G'.rf)^+ \cap G'.sb) \cup (G.W_{loc} \times \{a\})$.
5. $G'.mo \cap ((G.E \setminus \{a\}) \times (G.E \setminus \{a\})) = G.mo \cap ((G.E \setminus \{a\}) \times (G.E \setminus \{a\}))$.

Proof. Let $B = \{W_{eta}^k \setminus \{a\}\} \cup \{b \in W_{eta}^k \mid (b, a) \in (ab \cup rf)^*; sb\}$, where $\ell = loc(a)$. Let $b = \max B$, and let $G'$ be the execution graph obtained from $G$ by (i) replacing the value read at $a$ by $val_b(b)$; (ii) adding the $rf$-edge $(b, a)$ (instead of the $rf$-edge entering $a$ in $G$, if such exists); and (iii) making $a$ to be the $mo$-successor of $b$. It is straightforward to see that $G'$ is a strongly consistent.

Lemma 23 (Modifying updates). Let $G$ be a strongly consistent execution graph, and let $a \in U$ be an $(ab \cup rf)^+$-maximal event in $E$. Then, any execution graph $G'$ that is obtained from $G$ by changing the value written at $a$, or by making $a$ a read event, is strongly consistent.

Lemma 24. Let $G$ be a consistent execution graph of an expression $e$. Suppose that $hb \neq (ab \cup rf)^+$. Then, some strongly consistent execution graph of $e$ is either strongly racy or strongly uninitialized.

Proof. Clearly, we have $rf \not\subseteq hb$. Let $a_1, \ldots, a_n$ be an enumeration of $E$ that respects $ab \cup rf$, and let $k$ be the minimal index such that $rf \cap \{(a_1, \ldots, a_k) \times \{a_1, \ldots, a_k\}\} \not\subseteq hb$. Let $j < k$ such that $(a_j, a_k) \in rf \setminus hb$. Let $E = \{a \in E \mid (a, a_k) \in (ab \cup rf)^+\}$, and let $G_0 = G \cap E$. Note that the minimality of $k$ ensures that $a_j$ has no outgoing $ab$-edges in $G_0$, and its only outgoing $rf$-edge is to $a_k$. Then, it is easy to see that $G_0$ is a strongly consistent execution graph of $e$. If $G_0$ is strongly uninitialized, then we are done. Otherwise, $(b, a_k) \in (G_0.ab \cup G_0.rf)^+; G_0.ab$ for some $b \in W_{loc(a_k)}$. Consider two cases:

1. $a_k$ is a read event: Let $G'$ be a strongly consistent execution graph that satisfies the properties mentioned in Lemma 21 (applied on $G_0$ and $(a_k)$. The receptiveness property ensures that $G'$ is an execution graph of $e$, and we have that $(a_j, a_k)$ is a strong race in $G'$ with $na \in \{G'.mod(a_j), G'.mod(a_k)\}$. Hence, $G'$ is a strongly racy strongly consistent execution graph of $e$.

2. $a_k$ is an update event: Let $G'$ be a strongly consistent execution graph that satisfies the properties mentioned in Lemma 22 (applied on $G_0$ and $(a_k)$. The receptiveness property ensures that some execution $G''$, obtained from $G'$ by either changing the value written in $a_k$ or by making $a_k$ a read event, is an execution graph of $e$. By Lemma 23, $G''$ is strongly consistent as well. In addition, we have that $(a_j, a_k)$ is a strong race in $G''$ and $G''\mod(a_j) = na$. Hence, $G''$ is a strongly racy strongly consistent execution graph of $e$.

Corollary 25. Let $e$ be an expression. Then:

1. If $e$ is not buggy, then a trace $tr$ is allowed for $e$ iff $tr$ is a trace of $e$ and there exists a strongly consistent execution graph $G$ that follows $tr$.

2. $e$ is buggy iff some strongly consistent execution graph of $e$ is either strongly racy or strongly uninitialized.

Proof. Easily follows from our definitions and Lemma 24.
A.2 First operational semantics

We present an operational semantics, equivalent to the declarative RA+NA semantics, in the form of an abstract machine whose states are execution graphs, incrementally growing during the run. The initial execution graph, denoted $G_0$, is the empty execution graph.

**Notation 2.** For two execution graphs $G$ and $G'$, thread identifiers $\pi, \rho$, and a label $\epsilon$, we write $G' \in \text{Add}(G, \pi, \rho, \epsilon)$ if there exists an event $a$ such that the following hold:

- $G'.E = G.E \cup \{a\}$, $G'.tid = G.tid \cup \{a \mapsto \rho\}$, and $G'.lab = G.lab \cup \{a \mapsto \epsilon\}$.
- $G'.sb = (G.sb \cup (G.E^* \times \{a\})).$.
- $G'.rf \supseteq G.rf$.
- $G'.mo \supseteq G.mo$ and if $\epsilon = (\text{Write}_{na}, -,-)$, then $a$ is $G'.mo$-maximal.

The machine steps are based on execution graph steps:

$$\epsilon \in \{\text{Read}_a, \ell, v\}, \langle \text{Write}_a, \ell, v\rangle, \langle \text{Update}_a, \ell, v, o, v_n\rangle\}
\forall a \in W^\pi_a, \exists b \in E^*. \langle a, b \rangle \in (sb \cup rf)^*$$

$$\epsilon = \langle \text{Read}_a, \ell, v\rangle \Rightarrow \forall a \in W^\pi_a, \exists b \in E^*. \langle a, b \rangle \in (sb \cup rf)^*$$

$$\epsilon \in \{\text{Read}_a, \ell, v\}, \langle \text{Update}_a, \ell, v, o, v_n\rangle\} \Rightarrow \exists a \in W^\pi_a, b \in E^*. \langle a, b \rangle \in (sb \cup rf)^*$$

$G' \in \text{Add}(G, \pi, \rho, \epsilon) \quad G' \text{ is strongly consistent}$

$$G \xrightarrow{\text{Race-I}} G'$$

$$G' \in \text{Add}(G, \pi, \rho, (\text{Fork}, \rho)) \quad G \xrightarrow{(\text{Fork}, \rho), \pi} G'$$

$$\forall a \in W^\pi_a, \exists b \in E^*. \langle a, b \rangle \in (sb \cup rf)^*$$

$$\forall a \in W^\pi_a, \exists b \in E^*. \langle a, b \rangle \in (sb \cup rf)^*$$

$$G \xrightarrow{\text{Race-II}} G'$$

$$\neg \forall a \in W^\pi_a, \exists b \in E^*. \langle a, b \rangle \in (sb \cup rf)^*$$

$$\neg \exists a \in W^\pi_a, b \in E^*. \langle a, b \rangle \in (sb \cup rf)^*$$

$$G \xrightarrow{\text{Uninitialized}} G'$$

The machine steps are combined with expression reductions and lifted to thread-pool reductions exactly as in Figure 4 (using execution graphs $G$ instead of physical states $\sigma$), defining reductions of the form $G; TS \xrightarrow{\pi} G'; TS'$.

For every $x$, we write $G; TS \xrightarrow{\pi} G'$ if $G; TS \xrightarrow{\pi} G'; TS'$ for some transition label $x$; and $G; TS \xrightarrow{\pi} G'$ if $G; TS \xrightarrow{\pi} G'; TS'$ for some thread identifier $\pi$; and $G; TS \xrightarrow{\pi} G'$ if $G; TS \xrightarrow{\pi} G'; TS'$ for some transition label $x$ and thread identifier $\pi$.

**Lemma 26.** Suppose that $G_0 \xrightarrow{\pi_1} \ldots \xrightarrow{\pi_n} G$ for some trace $tr = \langle \epsilon_1, \pi_1\rangle, \ldots, \langle \epsilon_n, \pi_n\rangle$. Then $G$ is a strongly consistent execution graph that follows $tr$.

**Proof.** Easily follows from our definitions (note that strong consistency is trivially preserved in fork steps). \hfill $\blacksquare$

**Lemma 27.** Let $G$ be a strongly consistent execution graph of an expression $e$. Suppose that $G$ is not strongly racy or strongly uninitialized. Then, there exists a trace $\langle \epsilon_1, \pi_1\rangle, \ldots, \langle \epsilon_n, \pi_n\rangle$ of $e$, such that $G_0 \xrightarrow{\pi_1} \ldots \xrightarrow{\pi_n} G$.

**Proof.** Since $G$ is consistent and not strongly racy, $G.sb \cup G.rf \cup G.mo^{na}$ is acyclic. Let $\alpha_1, \ldots, \alpha_n$ be an enumeration of $G'.E$ that respects $(G'.sb \cup G'.rf \cup G'.mo^{na})^+$. For every $1 \leq i \leq n$, let $\tau_i = \text{tid}(a_i)$ and $\epsilon_i = \text{lab}(a_i)$. Then, $\langle \epsilon_1, \pi_1\rangle, \ldots, \langle \epsilon_n, \pi_n\rangle$ is a trace of $e$, and it is easy to see that by following this enumeration, i.e., adding $a_i$ in step $i$ for every $1 \leq i \leq n$, $G \xrightarrow{\tau_i} G'$. \hfill $\blacksquare$
we obtain that $G_0 \xrightarrow{\xi_1 \pi_1} \ldots \xrightarrow{\xi_n \pi_n} G'$. In particular, note that the fact that $G$ is strongly consistent entails that any $(G.\text{sb} \cup G.\text{rf})^+$-prefix of $G$ is also strongly consistent. 

---

**Theorem 28.** Let $e$ be a non-buggy expression. Then, a trace $tr = \langle \epsilon_1, \pi_1 \rangle, \ldots, \langle \epsilon_n, \pi_n \rangle$ is allowed for $e$ iif

$$G_0; [0 \mapsto e] \xrightarrow{\ *= \xi_1 \pi_1} \ldots \xrightarrow{\ *= \xi_n \pi_n} \xrightarrow{\ *= \xi_n \pi_n} G; TS$$

for some execution graph $G$ and threadpool $TS$.

**Proof.** Easily follows from our definitions Cor. 25, Lemma 26 and Lemma 27.

---

**Theorem 29.** An expression $e$ is buggy iif $G_0; [0 \mapsto e] \not\rightarrow^* \bot_{\text{race}}$ or $G_0; [0 \mapsto e] \not\rightarrow^* \bot_{\text{uninit}}$.

**Proof.** ($\Rightarrow$) Suppose that $e$ is buggy. By Cor. 25, some strongly consistent execution graph $G$ of $e$ is strongly racy or strongly uninitialized. Let $a_1, \ldots, a_n$ be an enumeration of $G.\epsilon$ that respects $\text{sb} \cup \text{rf}$. Let $k$ be the minimal index such that $G \cap \{a_1, \ldots, a_k\}$ is strongly racy or strongly uninitialized. Consider the two cases:

- $G \cap \{a_1, \ldots, a_k\}$ is strongly racy: Let $j$ be the minimal index such that $\langle a_j, a_k \rangle$ is a strong race in $G$ and $na \in \{\text{mod}(a_j), \text{mod}(a_k)\}$. Consider the following cases:
  - Either $a_j \in W^n_a$ or $a_k \in W^n_a$. Suppose, w.l.o.g., that the former holds. Let $B = \{a \in E | \langle a, a_k \rangle \in (G.\text{sb} \cup G.\text{rf})^+ \lor \langle a, a_j \rangle \in (G.\text{sb} \cup G.\text{rf})^*\}$, and let $G' = G \cap B$. Then, $G'$ is strongly consistent, not strongly racy, and not strongly uninitialized, and by Lemma 27, $G_0 \xrightarrow{\xi_1 \pi_1} \ldots \xrightarrow{\xi_n \pi_n} G'$ for some trace $\langle \epsilon_1, \pi_1 \rangle, \ldots, \langle \epsilon_n, \pi_n \rangle$ of $e$. Since $\langle a_j, a_k \rangle \not\in (G.\text{sb} \cup G.\text{rf})^+$, we have that $\langle a_j, b \rangle \not\in (G'.\text{sb} \cup G'.\text{rf})^+$ for every $b \in G'.G'\epsilon^{\text{tid}(a_k)}$. Further, it is easy to see that $\langle \epsilon_1, \pi_1 \rangle, \ldots, \langle \epsilon_n, \pi_n \rangle, \langle \text{lab}(a_k), \text{tid}(a_k) \rangle$ is also a trace of $e$, and so $G_0; [0 \mapsto e] \not\rightarrow^* \bot_{\text{race}}$.
  - Both $a_j \not\in W^n_a$ and $a_k \not\in W^n_a$. Then, either $a_k \in W^n_a$ and $a_j \in W^n_a^*$, or $a_j \in W^n_a$ and $a_k \in W^n_a^*$. Suppose, w.l.o.g., that the former holds. Let $B = \{a \in E | \langle a, a_k \rangle \in (G.\text{sb} \cup G.\text{rf})^+ \lor \langle a, a_j \rangle \in (G.\text{sb} \cup G.\text{rf})^*\}$, and let $G' = G \cap B$. Then, $G'$ is strongly consistent, not strongly racy, and not strongly uninitialized, and by Lemma 27, $G_0 \xrightarrow{\xi_1 \pi_1} \ldots \xrightarrow{\xi_n \pi_n} G'$ for some trace $\langle \epsilon_1, \pi_1 \rangle, \ldots, \langle \epsilon_n, \pi_n \rangle$ of $e$. Since $\langle a_j, a_k \rangle \not\in (G.\text{sb} \cup G.\text{rf})^*$, we have that $\langle a_j, b \rangle \not\in (G'.\text{sb} \cup G'.\text{rf})^*$ for every $b \in G'.G'\epsilon^{\text{tid}(a_k)}$. Since $a_j \in G'.G'\text{loc}(a_k)$, we have $G' \xrightarrow{\text{lab}(a_k) \times \text{tid}(a_k)} \bot_{\text{race}}$. Furthermore, it is easy to see that $\langle \epsilon_1, \pi_1 \rangle, \ldots, \langle \epsilon_n, \pi_n \rangle, \langle \text{lab}(a_k), \text{tid}(a_k) \rangle$ is also a trace of $e$, and so $G_0; [0 \mapsto e] \not\rightarrow^* \bot_{\text{race}}$.

- $G \cap \{a_1, \ldots, a_k\}$ is strongly uninitialized: Then, $a_k \in RU$ and $\langle a, a_k \rangle \not\in (\text{sb} \cup \text{rf})^*$-sb for every $a \in \text{Wloc}(a_k)$. Let $B = \{a \in E | \langle a, a_k \rangle \in (G.\text{sb} \cup G.\text{rf})^+\}$, and let $G' = G \cap B$. Then, $G'$ is strongly consistent, not strongly racy, and not strongly uninitialized, and by Lemma 27, $G_0 \xrightarrow{\xi_1 \pi_1} \ldots \xrightarrow{\xi_n \pi_n} G'$ for some trace $\langle \epsilon_1, \pi_1 \rangle, \ldots, \langle \epsilon_n, \pi_n \rangle$ of $e$. Since $\langle a_j, a_k \rangle \not\in (G.\text{sb} \cup G.\text{rf})^*$, $G.\text{sb}$ for every $a \in G.\text{Wloc}(a_k)$, we have $G' \xrightarrow{\text{lab}(a_k) \times \text{tid}(a_k)} \bot_{\text{uninit}}$. Furthermore, it is easy to see that $\langle \epsilon_1, \pi_1 \rangle, \ldots, \langle \epsilon_n, \pi_n \rangle, \langle \text{lab}(a_k), \text{tid}(a_k) \rangle$ is also a trace of $e$, and so $G_0; [0 \mapsto e] \not\rightarrow^* \bot_{\text{uninit}}$.

($\Leftarrow$) Suppose that $G_0; [0 \mapsto e] \not\rightarrow^* \bot_{\text{race}}$ or $G_0; [0 \mapsto e] \not\rightarrow^* \bot_{\text{uninit}}$. Then, there exists a trace $\langle \epsilon_1, \pi_1 \rangle, \ldots, \langle \epsilon_n, \pi_n \rangle$ of $e$, such that $G_0 \xrightarrow{\xi_1 \pi_1} \ldots \xrightarrow{\xi_n \pi_n} \bot_{\text{race}}$ or $G_0 \xrightarrow{\xi_1 \pi_1} \ldots \xrightarrow{\xi_n \pi_n} \bot_{\text{uninit}}$. Let $G_{n-1}$ such that $G_0 \xrightarrow{\xi_1 \pi_1} \ldots \xrightarrow{\xi_{n-1} \pi_{n-1}} G_{n-1} \xrightarrow{\xi_n \pi_n} \bot_{\text{race}}$, or $G_0 \xrightarrow{\xi_1 \pi_1} \ldots \xrightarrow{\xi_{n-1} \pi_{n-1}} G_{n-1} \xrightarrow{\xi_n \pi_n} \bot_{\text{uninit}}$. By Lemma 26, $G_{n-1}$ is a strongly consistent execution of $e$. Since $G_{n-1} \xrightarrow{\xi_n \pi_n} \bot_{\text{race}}$ or $G_{n-1} \xrightarrow{\xi_n \pi_n} \bot_{\text{uninit}}$, at least one of the following hold:
The physical state $T$, that satisfies $\varepsilon \vdash G \vdash \eta$ and the message induced by $G \vdash \eta$. Next, we show that the timestamp machine presented in §2 is equivalent to the operational $\varepsilon \vdash G \vdash \eta$. Clearly, $G'$ is an execution of $\varepsilon$. Using Lemma 21, as well as the receptiveness property of traces, we can obtain an execution $G''$ of $\varepsilon$ that is consistent, and satisfies $\langle a, b \rangle \not\in G''$. Since $\text{mod}(b) = n$, it follows that $G''$ is racy.

The following lemmas (tediously but straightforwardly) follow from the definitions.

A.3 Timestamp machine

Next, we show that the timestamp machine presented in §2 is equivalent to the operational machine above.

**Definition 30.** A timestamp assignment for an execution graph $G$ is a function $ts : \mathbb{W} \rightarrow \text{Time}$. Given a timestamp assignment $ts$ for $G$ and an event $a \in \mathbb{W} \cup \mathbb{U}$, the view of an event $a$ in $G$ according to $ts$, denoted $\text{view}(a, G, ts)$, and the message induced by $a$ in $G$ according to $ts$, denoted $\text{msg}(a, G, ts)$, are given by:

$$\text{view}(a, G, ts) = \lambda \ell. \max \{ ts(b) \mid b \in \mathbb{W}_{\ell}, \langle b, a \rangle \in (s\mathbb{b} \cup r\mathbb{f})^* \}$$

$$\text{msg}(a, G, ts) = (\text{loc}(a), \text{val}_{\ell}(a), ts(a), \text{view}(a, G, ts))$$

If $\{ ts(b) \mid b \in \mathbb{W}_{\ell}, \langle b, a \rangle \in (s\mathbb{b} \cup r\mathbb{f})^* \}$ is empty, we take $\text{view}(a, G, ts)(f) = \bot$ (undefined).

**Definition 31.** Let $G$ be an execution graph and $ts$ be a timestamp assignment $ts$ for $G$. The physical state $(M^ts_G, T^ts_G, N^ts_G)$ is given by:

$$M^ts_G = \{ \text{msg}(a, G, ts) \mid a \in \mathbb{W} \}$$

$$T^ts_G = \lambda \ell. \lambda \ell. \max \{ ts(a) \mid a \in \mathbb{W}_{\ell}, \exists b^\ell. \langle a, b \rangle \in (s\mathbb{b} \cup r\mathbb{f})^* \}$$

(If $\{ ts(a) \mid a \in \mathbb{W}_{\ell} \}$ is empty, we take $T^ts_G(\pi)(f) = \bot$)

$$N^ts_G = \lambda \ell. \max \{ ts(a) \mid a \in \mathbb{W}^\text{na} \}$$

(If $\{ ts(a) \mid a \in \mathbb{W}^\text{na} \}$ is empty, we take $N^ts_G(\ell) = \bot$)

The following lemmas (tediously but straightforwardly) follow from the definitions.

**Lemma 32.** Suppose that $(M, T, N) \xrightarrow{\varepsilon} (M', T', N')$, and let $G$ be an execution graph such that $G \sim (M, T, N)$. Then, $G \xrightarrow{\varepsilon} G'$ for some execution graph $G'$ such that $G' \sim (M', T', N')$.

**Lemma 33.** Suppose that $(M, T, N) \xrightarrow{\varepsilon} \bot_{\text{race}}$, and let $G$ be an execution graph such that $G \sim (M, T, N)$. Then, $G \xrightarrow{\varepsilon} \bot_{\text{race}}$.

**Lemma 34.** Suppose that $(M, T, N) \xrightarrow{\varepsilon} \bot_{\text{uninit}}$, and let $G$ be an execution graph such that $G \sim (M, T, N)$. Then, $G \xrightarrow{\varepsilon} \bot_{\text{uninit}}$.

**Lemma 35.** Suppose that $G \xrightarrow{\varepsilon} G'$, and let $ts'$ be a timestamp assignment for $G'$. Let $ts = ts'|_{G, \mathbb{W}}$. Then, $ts$ is a timestamp assignment for $G$, and $(M^ts_G, T^ts_G, N^ts_G) \xrightarrow{\varepsilon} (M'^ts_G, T'^ts_G, N'^ts_G)$. 
Lemma 36. Suppose that $G_0 \xrightarrow{\varepsilon_1 \pi_1} \ldots \xrightarrow{\varepsilon_n \pi_n} G_n$. Then, $(M_0, T_0, N_0) \xrightarrow{\varepsilon_1 \pi_1} \ldots \xrightarrow{\varepsilon_n \pi_n} (M_n, T_n, N_n)$ for some physical state $(M_n, T_n, N_n)$.

**Proof.** Let $ts_n$ be some timestamp assignment for $G_n$, and for every $0 \leq k < n$, let $ts_k = ts_n|_{G_k}$. For every $0 \leq k \leq n$, let $(M_k, T_k, N_k) = (M_{G_k}, T_{G_k}, N_{G_k})$. (Clearly, this definition of $(M_k, T_k, N_k)$ agrees with the definition of the initial physical state in Definition 2). By Lemma 35, we have $(M_{k-1}, T_{k-1}, N_{k-1}) \xrightarrow{\varepsilon_k \pi_k} (M_k, T_k, N_k)$ for every $1 \leq i \leq n$.

Lemma 37. Suppose that $G \xrightarrow{\varepsilon_k \pi_k} \bot_{\text{race}}$ and let $(M, T, N)$ be a physical state such that $G \sim (M, T, N)$. Then, $(M, T, N) \xrightarrow{\varepsilon_k \pi} \bot_{\text{race}}$.

Lemma 38. Suppose that $G \xrightarrow{\varepsilon_k \pi} \bot_{\text{uninit}}$ and let $(M, T, N)$ be a physical state such that $G \sim (M, T, N)$. Then, $(M, T, N) \xrightarrow{\varepsilon_k \pi} \bot_{\text{uninit}}$.

Finally, by combining the results above, we obtain the following:

Corollary 39. Let $e$ be an expression.
- If $e$ is not buggy, then a trace $tr = \langle \varepsilon_1, \pi_1 \rangle, \ldots, \langle \varepsilon_n, \pi_n \rangle$ is allowed for $e$ iff
  $$(M_0, T_0, N_0); [0 \mapsto e] \xrightarrow{\varepsilon_1 \pi_1} \ldots \xrightarrow{\varepsilon_n \pi_n} (M, T, N); TS$$
  for some thread $\pi$, physical state $(M, T, N)$, and threadpool $TS$.
- $e$ is buggy if $(M_0, T_0, N_0); [0 \mapsto e] \xrightarrow{\ast} \bot_{\text{race}}$ or $(M_0, T_0, N_0); [0 \mapsto e] \xrightarrow{\ast} \bot_{\text{uninit}}$ for some thread $\pi$. 

B  Single-writer protocols’ proof rules

\[ \ell : s \tau_R \Rightarrow \square \ell : s \tau_R \]
\[ \ell : s_1 \tau_R \ast \ell : s_2 \tau_R \Rightarrow s_1 \subseteq s_2 \vee s_2 \subseteq s_1 \]
\[ \ell : s_1 \tau_W \ast \ell : s_2 \tau_W \Rightarrow s_2 \subseteq s_1 \]
\[ \ell : s \tau_W \Rightarrow \ell : s \tau_W \ast \ell : s \tau_R \]
\[ \ell : s \tau_W \Rightarrow \triangleright \tau_{\text{full}}(s, v) \ast \triangleright \tau_{\text{read}}(s, v) \ast Q \]
\[ \{ \ell \leftrightarrow _= \} \ell_{[\text{na}]} := v \{ Q \} \]
\[ \{ \ell : s \tau_W \} \ell_{[\text{at}]} \{ v. \ell : s \tau_W \ast \tau_{\text{read}}(s, v) \} \]
\[ \forall s, s', v. \ P \ast \tau_{\text{read}}(s', v) \Rightarrow Q \]
\[ \{ \ell : s \tau_R \ast P \} \ell_{[\text{at}]} \{ v. \exists s' \supseteq s. \ell : s' \tau_R \ast Q \} \]
\[ P \ast \ell : s'' \tau_W \ast \tau_{\text{full}}(s, \_ \_ \,) \Rightarrow \tau_{\text{full}}(s'', v) \ast Q \quad s'' \supseteq s \]
\[ \{ \ell : s \tau_W \ast P \} \ell_{[\text{at}]} := v \{ \ell : s'' \tau_R \ast Q \} \]

C  Escrows and exchanges’ proof rules

In addition to escrows, we have mechanized a weaker variant of exchanges [27]. Our variant is weaker in the sense that the resources put in the exchanges have to be justified at the same view. Unlike the general exchanges, our variant can be seen as trivially true. In order to enforce that the resources are put back at the same view as when they are taken out, we need to expose the identity of views in the logic. However, since we only need to know equalities of views, we just need to bind views to names and expose the names. We call those names abstract views. The assertion \( P \beta \) asserts that \( P \) holds at some abstract view \( \beta \). To use \( P \) a thread needs an extra knowledge \( \text{Seen}(\beta) \) which states that it has observed the view with name \( \beta \).

The assertion \( [P \leadsto Q]_{\beta}^{\mathcal{N}} \) asserts the knowledge of an exchange at some abstract view \( \beta \) under \( \mathcal{N} \).
RSL proof rules

RSL focuses on the message-passing style transferring of resources through release-write/acquire-read pairs. The two main assertions of RSL are \( \text{Rel}(\ell, Q) \), representing the permission to write to \( \ell \), and \( \text{Acq}(\ell, Q) \), representing the permission to read from \( \ell \). The resource \( Q(v) \) is transferred through the write/read pair of value \( v \). The assertion \( \text{RMWAcq}(\ell, P) \) is another permission that is used to CAS on the location.

We present in Figure 15 the proof rules of iRSL, which was already checked by Coq. The encoding extends the original assertions with a predicate \( D \) which is duplicable, to represent the restricted interpretation of the underlying invariant governing \( \ell \) (in a similar fashion to what we did in iGPS). For more details please refer to the Coq development, which also contains a fractional version of the variant.

\[
\begin{align*}
\text{RSL-WRel} & \quad \{ Q(v) \star \ell \} \ell_{[\text{at}]} := v \{ \text{Init}(\ell) \star \text{Rel}(\ell, Q, D) \} \\
\text{RSL-RAcq} & \quad \{ \text{Init}(\ell) \star \text{Acq}(\ell, R, D) \} \ell_{[\text{at}]} = v \{ R(v) \star \ell \} \star \text{Acq}(\ell, R, v := \top) \}
\end{align*}
\]

\[
\begin{align*}
\text{RSL-CAS} & \quad P \star \ell \equiv \ell \equiv \ell \equiv R(0, v) \\
\{ \text{Init}(\ell) \star \text{RMWAcq}(\ell, Q, D) \} \star \text{cas}(\ell, v, v) \{ \exists v. R(b, v) \star \text{Init}(\ell) \star \text{RMWAcq}(\ell, Q, D) \}
\end{align*}
\]

\[
\begin{align*}
\text{RSL-Rel-SplitJoin} & \quad \text{Rel}(\ell, Q_1, D) \star \text{Rel}(\ell, Q_2, D) \equiv \text{Rel}(\ell, \lambda v. Q_1(v) \lor Q_2(v), D) \\
\text{RSL-Acq-SplitJoin} & \quad \text{Acq}(\ell, R_1, D) \star \text{Acq}(\ell, R_2, D) \equiv \text{Acq}(\ell, \lambda v. R_1(v) \land R_2(v), D) \\
\text{RSL-RMW-Acq-Split} & \quad \text{RMWAcq}(\ell, P, D) \equiv \text{RMWAcq}(\ell, P, D) \star \text{Acq}(\ell, \text{True}, D)
\end{align*}
\]

\[
\begin{align*}
\text{RSL-Init-Persistent} & \quad \text{Init}(\ell) \vdash \Box \text{Init}(\ell) \\
\text{RSL-RMW-Persistent} & \quad \text{RMWAcq}(\ell, P, D) \vdash \Box \text{RMWAcq}(\ell, P, D) \\
\text{RSL-Rel-Persistent} & \quad \text{Rel}(\ell, Q, D) \vdash \Box \text{Rel}(\ell, Q, D) \\
\text{RSL-Aloc-R} & \quad \{ \top \} \text{alloc}(1) \{ \ell. \text{Rel}(\ell, P) \star \text{Acq}(\ell, P) \}
\end{align*}
\]

\[
\begin{align*}
\text{RSL-Init-M} & \quad \{ \top \} \text{alloc}(1) \{ \ell. \text{Rel}(\ell, P) \star \text{RMWAcq}(\ell, P) \}
\end{align*}
\]

\[
\begin{align*}
\text{RSL-Init-R} & \quad \{ \text{Own}(\ell) \star \ell \} \ell_{[\text{na}]} := v \{ \text{Init}(\ell) \star \text{Rel}(\ell, Q, D) \star \text{Acq}(\ell, Q, D) \}
\end{align*}
\]

\[
\begin{align*}
\text{RSL-Init-M} & \quad \{ \text{Own}(\ell) \star \ell \} \ell_{[\text{na}]} := v \{ \text{Init}(\ell) \star \text{Rel}(\ell, Q, D) \star \text{RMWAcq}(\ell, Q, D) \}
\end{align*}
\]

\[\text{Figure 15} \, \text{Proof rules of iRSL.}\]
Let \( ns \equiv 0, tc \equiv 1 \).

\[
\begin{align*}
\text{newLock()} & \triangleq \\
& \ egin{align*}
\text{let } x & = \text{alloc}(2) \text{ in} \\
x + ns[n] & := 0; \\
x + tc[n] & := 0; \\
x & . 
\end{align*} \\
\text{lock}(x) & \triangleq \\
& \begin{align*}
\text{let } y & = \text{FAI}(x + tc) \text{ in} \\
\text{repeat} \\
& \ egin{align*}
\text{let } z & = x + ns[\text{at}] \text{ in} \\
y & = z
\end{align*} \\
\text{unlock}(x) & \triangleq \\
& \begin{align*}
\text{let } z & = x + ns[\text{at}] \text{ in} \\
x + ns[\text{at}] & := (z + 1) \mod C
\end{align*}
\end{align*}
\]

\[\text{Figure 16} \text{ Bounded ticket lock in } \lambda_{\text{RN}}.\]

\section*{E Bounded ticket locks}

Ticket locks are a fair locking mechanism found in the Linux kernel. The first formal proof of ticket locks in RA+NA was given in the appendix of the GPS paper. We follow that variant and its proof’s intuition closely, although our proof, thanks to single-writer protocols, is significantly simpler.

The code of the bounded ticket lock variant in \( \lambda_{\text{RN}} \) is given in Figure 16. The lock has two atomic fields, a ticket counter \( x + tc \) and a now-serving counter \( x + ns \), which are initialized with 0. A thread can acquire the lock by first asking for a ticket \( t \) from the ticket counter, which is a fetch-and-increment (FAI) operation that hands out the current value of the ticket counter as the new ticket, and increases the counter by 1. After obtaining the ticket, the thread waits for the now-serving counter to reach its ticket (it keeps checking \( x + ns \)), by which time it is guaranteed that previous tickets have been served, so it is safe to acquire the lock. To release the lock, the thread only needs to write \( t + 1 \) to the now-serving counter, signifying that the next ticket can now be served.

In the bounded variant of ticket locks, the counters are stored in a fixed-size machine word with the maximum unsigned value \( C \). Once the counters reach \( C - 1 \), the next increment will reset them back to 0. This additional flavor of reality induces the main complication of the verification. In unbounded ticket locks, it is easy to maintain the invariant that no two threads have the same ticket. Here, however, tickets are being reused, and the ticket counter must ensure that a ticket \( t \) has already been served before re-issuing it. In order to circumvent this, the number of concurrent threads trying to acquire the lock is restricted at most \( C \). Additionally, before getting a new ticket, each thread must prove to the ticket counter that its most recent ticket has been served. Consequently, the ticket counter must track which tickets have been issued to which threads and by its best the value of the now-serving counter, in order to keep the number of tickets waiting under \( C \).

The now-serving counter is where the actually transfer of resources takes place: the lock resource \( P \) is transferred from the release write to \( x + ns \) (by the unlock call of the previous ticket holder) to the acquire read by the lock call of the current ticket holder. Since a read cannot get ownership, escrows are used instead to transfer \( P \). The releasing thread creates an escrow to store \( P \) and deliver it to the holder of ticket \( t + 1 \). The escrow is acquired by the \( t + 1 \) ticket holder when the loop of its call to lock stops. By that time the acquiring thread should also have the correct exclusive token to use the escrow and obtain \( P \). The releasing thread saves the proof that it has released the lock, in order to present that to the ticket counter when it requests another ticket later.

Our proof significantly simplifies the original proof by the following points:

- **Using single-writer protocols for the now-serving counter.** The original proof uses a normal protocol for the counter, in combination with two other ghost components
We explain the technical construction of the proof. The complete definitions of monoids, protocols and escrows needed are given in Figure 17.

**E.1 Proof setup for bounded ticket locks**

We use protocols \( \text{TCP}(\gamma, x) \) and \( \text{NSP}(\gamma) \) to govern the ticket counter and the now-serving counter respectively. The \( \text{TCP}(\gamma, x) \) protocol only has a single state \( \text{Inv} \), while the states of \( \text{NSP}(\gamma) \) are natural numbers, which represent abstract tickets. Though working differently, both counters record the abstract ticket \( t \) they are currently working with, who requires the concrete counter value \( t_c \) to follow the equation \( t_c = t \mod C \). The abstract ticket of each counter can only increase over time (the state relation is \( \leq \) for \( \text{NSP}(\gamma) \)). By exploiting abstract tickets, we do not need to deal with reusing concrete tickets.

The interpretations of \( \text{NSP}(\gamma) \) and \( \text{TCP}(\gamma, x) \) depend respectively on the first and second components of a product monoid for our ghost states. The first component \( \text{Disj}(\mathbb{N}) \)
XX:40 Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris

is a disjoint-union monoid of natural numbers, whose composition is only defined for disjoint sets of natural numbers: $X \cdot Y = X \cup Y$ if $X \cap Y = \emptyset$. It is used for abstract ticket ownerships. The second component $\text{AUTH}(\mathbb{N} \rightarrow \mathbb{N})$ is an authoritative monoid of a finite partial map from $\mathbb{N}$ to exclusive elements from $\mathbb{N} \cup \{-1\}$. It is used for tracking issued abstract tickets for all participating threads. The composition works in the same way as that of the heap monoid in $\lambda_{\text{SC}}$ and the history and thread-view monoids in $\lambda_{\text{RN}}$: it has the same rules for exclusiveness, agreement, and updates as those monoids. The composition of the product monoid is then lifted piecewise.

The now-serving counter protocol

The assertion $\text{Perm}^\gamma(t)$ embeds the ownership of the abstract ticket $t$ in its first component (while the second component is an empty map which contains zero knowledge or ownership). $\text{Perm}^\gamma(t)$ satisfies exclusiveness, realizing the requirement that a ticket can only be uniquely owned:

$$\text{Perm}^\gamma(t) \ast \text{Perm}^\gamma(t) \Rightarrow \text{False}$$

Exclusiveness of $\text{Perm}^\gamma(t)$ allows us to define the escrow used to transfer the lock’s resource, which contains $P$ and the full single-writer permission of the now-serving counter. This is reflected in the definition of $\text{Escs}(x, \gamma, n)$, which asserts the existence of the escrows for all abstract tickets up to $n$. Additionally, $\text{Escs}(x, \gamma, n)$ subsumes the proofs that all abstract tickets up to (but excluding) $n$ have been served, due to the fact that the lock’s resource can only be stored in exactly one of the escrows at a time, and that the verification maintains that the resource only moves from the escrow of some ticket $t$ to that of $t + 1$. These proofs are needed when a thread updates the now-serving counter $x + ns$, to release the lock. This is enforced in $\text{NSP}(\gamma)$. When increasing the counter with value $z = n \mod C$ to $(n + 1) \mod C$, the lock-releasing thread moves the protocol state from $n$ to $n + 1$, and must re-establish $\text{NSP}(\gamma)$ by allocating the escrow for the ticket $n + 1$ for the next lock-acquiring thread.

The ticket counter protocol

Tracking tickets is the main role of the ticket counter. The tracking $M$ is a finite map from participant ids to their corresponding most recent abstract tickets. The domain of $M$, i.e. the set of ids, is restricted to the set $[0, ..., C - 1]$, since there are at most $C$ threads participating in the lock. If a thread has never obtained any ticket, then it is mapped to $-1$. The authoritative part of the map $\bullet M$ is part of $\text{AllTkt}^\gamma(M)$, which in turn is stored in the protocol $\text{TCP}(\gamma, x)$ that governs the ticket counter $x + tc$. The non-authoritative $\circ M$ is split into $\text{MyTkt}^\gamma[i \mapsto M(i)]$’s for $i \in [0, ..., C - 1]$. Each $\text{MyTkt}^\gamma[i \mapsto M(i)]$ represents the knowledge of the most recent ticket $M(i)$, as well as the exclusive permission required to acquire a new ticket for the thread $i$. These permissions are allocated when the lock is created, and then distributed to the threads that wish to use the lock.

$\text{TCP}(\gamma, x)$ always keep its authoritative tracker $\text{AllTkt}^\gamma(M)$. The maximum abstract ticket of $M$ is $t - 1$, which is also the most recently issued ticket by the ticket counter. Meanwhile, the next ticket to be issued is $t$, and the current value $y$ of the ticket counter must be the corresponding concrete ticket of $t$. $M$ of course is injective, since no two threads should get the same ticket. $\text{TCP}(\gamma, x)$ also holds permissions $\text{Perm}^\gamma(j)$ of to-be-issued abstract tickets. When requesting a ticket, a thread obtains the abstract ticket $t$ (and the concrete ticket $y$), as well as $\text{Perm}^\gamma(t)$ to use the escrow later. The thread also increases the
ticket counter value (concretely and abstractly), and, by doing so, it must show that it has observed the now-serving counter to be at least some state that is not too far away from the ticket counter, which is realized by the conditions between \( t \) and \( n \) in the definition of \( \text{TCP} \).

These conditions say that the ticket counter is always ahead of the now-serving counter, but their distance cannot surpass the number of waiting tickets known to the ticket counter, i.e. those that have been issued but are still not reported to the ticket counter as served.

**Specification for bounded ticket lock**

The specification that we need to verify is very simple, as given in Figure 18.

\[
\begin{align*}
\{ \text{newLock} \} & \ x. \ \bigwedge_{i < C} \text{MayAcquire}(x) \\
\{ \text{MayAcquire}(x) \} & \ \text{lock}(x) \ \{ \text{MayRelease}(x) \} \\
\{ \text{MayRelease}(x) \} & \ \text{unlock}(x) \ \{ \text{MayAcquire}(x) \}
\end{align*}
\]

**Figure 18** Specification of bounded ticket locks.

E.2 Proof steps for lock

The proof outline for the \( \text{lock} \) procedure is given in Figure 21. We explain the main reasoning below.

- With the permission \( \text{MyTkt}^\gamma [i \mapsto t] \), a thread \( i \) acquires a new abstract ticket \( t \) (concretely \( y \)) along with the permission \( \text{Perm}^\gamma(t) \), and updates the tracker for \( i \) to \( t \).

  It also increases the ticket counter concretely to \((y + 1) \mod C\) along with an abstract ticket \( t + 1 \), by which the ticket counter demands \( x + ns : n_0 \\text{NSP}(\gamma) \) for some good enough \( n_0 \). This demand is fulfilled either by what the thread has locally, or what is already in the interpretation previously. In either way the thread updates its observation of \( \text{NSP}(\gamma) \) to \( n_0 \).

- By entering the loop, the thread keeps updating its observation of \( \text{NSP}(\gamma) \) to some state \( n \) later that \( n_0 \). When the loop exits, the thread knows that the two concrete tickets \( y \) and \( t + 1 \) are observed by some party.
\[
\begin{align*}
\{&>P\} \\
\text{let } x = \text{alloc}(2) \text{ in} \\
\{&>P \cdot x + \text{ns} \rightarrow A \cdot x + \text{tc} \rightarrow A\} \\
\{&>P \cdot x + \text{ns} \rightarrow A \cdot x + \text{tc} \rightarrow \exists \gamma, M. \text{ Perm}^\gamma(t) \cdot \text{AllTkte}(M-1) \cdot \text{ MyTkte}^\gamma [i \mapsto -1]\}
\end{align*}
\]

\[
\begin{align*}
x + \text{ns}[na] & := 0; \\
\{&>P \cdot x + \text{ns} \rightarrow 0\} \\
\{&>P \cdot x + \text{tc} \rightarrow 0\}
\end{align*}
\]

\[
\begin{align*}
\{&>P \cdot \text{MayAcquire}(x)\}
\end{align*}
\]

\textbf{Figure 19} Proof outline for \texttt{newLock}

\[
\begin{align*}
\{&>P \cdot \text{MayRelease}(x)\} \\
\exists \gamma, i, t, >P \cdot \text{MyTkte}^\gamma [i \mapsto t] & \cdot \bigg( x + \text{tc} \cdot \text{Inv (TCP(\gamma, x))} \bigg) \\
\text{Ctx: } x + \text{tc} & \cdot \text{Inv (TCP(\gamma, x))}
\end{align*}
\]

\[
\begin{align*}
\text{let } z = x + \text{ns}[na] \text{ in} \\
\{&z = t \mod \text{C} \cdot >P \cdot \text{MyTkte}^\gamma [i \mapsto t] \cdot \bigg( x + \text{ns} : t \cdot \text{NSP(} \gamma \bigg) \bigg)\} \\
x + \text{ns}[na] & := (z + 1) \mod \text{C} \\
\{&\text{MyTkte}^\gamma [i \mapsto t] \cdot \bigg( x + \text{ns} : t + 1 \cdot \text{NSP(} \gamma \bigg) \bigg)\} \\
\{&\text{MayAcquire}(x)\}
\end{align*}
\]

\textbf{Figure 20} Proof outline for \texttt{unlock}
\{\text{MayAcquire}(x)\}
\{\exists \gamma, t, i. \text{MyTkt}^\gamma[i \mapsto t_i] * x + tc : \text{Inv} [\text{TCP}(\gamma, x)] * t, i \neq -1 \implies x + ns : t_i + 1 [\text{NSP}(\gamma)]_R\}
\text{let } y = \text{FAI}(x + tc) \text{ in}
\{\exists t, n_0. y = t \mod C \land t < n_0 + C * \text{MyTkt}^\gamma[i \mapsto t] * \text{Perm}^\gamma(t) \implies x + ns : n_0 [\text{NSP}(\gamma)]_R\}
\text{Ctx: } y = t \mod C \land t < n_0 + C
\text{Ctx: } x + tc : \text{TCP}(\gamma, x)
\text{repeat}
\{\exists n \geq n_0. \text{MyTkt}^\gamma[i \mapsto t] * \text{Perm}^\gamma(t) * x + ns : n [\text{NSP}(\gamma)]_R\}
\text{let } z = x + ns_{[at]} \text{ in}
\{\exists n' \geq n, z = n' \mod C * \Box \text{Escs}(x, \gamma, n')\}
\text{MyTkt}^\gamma[i \mapsto t] * \text{Perm}^\gamma(t) * x + ns : n' [\text{NSP}(\gamma)]_R
\text{y} \equiv 0
\{\exists n \geq n_0. y = z + z = n \mod C * \Box \text{Escs}(x, \gamma, n)\}
\text{MyTkt}^\gamma[i \mapsto t] * \text{Perm}^\gamma(t) * x + ns : n [\text{NSP}(\gamma)]_R
\text{have } n \leq t < n_0 + C \leq n + C \text{ and } t \mod C = n \mod C
\text{Ctx: } t = n
\{\text{MyTkt}^\gamma[i \mapsto t] * \text{Perm}^\gamma(t) * \text{Perm}^\gamma(t) \implies x + ns : t [\text{NSP}(\gamma)]_R\}
\{\text{MyTkt}^\gamma[i \mapsto t] * \text{Perm}^\gamma(t) * P \implies x + ns : t [\text{NSP}(\gamma)]_R\}
\{P * \text{MyTkt}^\gamma[i \mapsto t] * x + ns : t [\text{NSP}(\gamma)]_R\}
\{P * \text{MayRelease}(x)\}

\begin{itemize}
  \item Figure 21 Proof outline for lock
\end{itemize}

and z are the same, and this fact should imply that their corresponding abstract tickets t and n are also the same. To show this we need to use a subtle reasoning with the single writer protocol.

Since we already have t mod C = n mod C and t < n + C by acquiring the ticket and loop-checking the counter, we only need to show n < t to show n = t. This matches the intuition that the now-serving counter cannot surpass the unserved ticket, since we have locally the permission Perm^\gamma(t). We assume the contrary t < n and derive a contradiction.

Note that by reading the now-serving counter we also obtain Escs(x, \gamma, n). Then we can use Perm^\gamma(t) with the escrow for t to obtain x + ns : t [\text{NSP}(\gamma)]_R. Combining with our locally-known x + ns : n [\text{NSP}(\gamma)]_R, it must be that n < t: \textit{contradiction}!

Knowing that n = t, we use the escrow with Perm^\gamma(t) to acquire the lock, and gain \textit{P}. We also gain P * x + ns : t [\text{NSP}(\gamma)]_R, which we combine with x + ns : t [\text{NSP}(\gamma)]_R to remove the later, and complete the MayRelease(x) permission.
The model of iGPS protocols consists of two parts: a global protocol invariant \( \text{PrInv}_\tau(\ell) \) shared by all protocol variants, and local protocol assertions given out to clients, which depend on the particular protocol variant. We will now explain both parts in detail. The definitions are given in Figure 22b.

The protocol invariant owns, among other resources, the location’s history \( \text{Hist}(\ell, \{ (v, t, V) | (s, v, t, V) \in \Delta \}) \) though \( h \) will be expressed differently, as we explain below. In addition to the physical history, the invariant also owns the authoritative fragment of the logical history \( \Delta \) – an augmented history that write events paired with protocol states.

The protocol invariant further owns the protocol interpretation for every state in \( \Delta \). As all states are potentially targets of reads, we store \( \tau_{\text{read}} \) for every state in the history. To enable ownership transfer via CAS, we additionally need to store \( \tau_{\text{full}} \) for those states whose associated write event can still be target of an update operation.

This condition is sufficient for non-exclusive protocols. However, single-writer protocols offer an additional mechanism of transferring out resources from the protocol interpretation: exclusive writes. To support both protocol variants, the invariant tracks the most recent exclusive write \( (s_x, v_x, V_x) \) and stores \( \tau_{\text{full}} \) for \( \delta_x \) as well as any later write that can be target of CAS. In the case of non-exclusive protocols, \( \delta_x \) is set to the initial protocol write and all CAS-able writes maintain their full interpretation \( \tau_{\text{full}} \).

We track \( \delta_x \) with an additional authoritative piece of ghost state. The protocol invariant holds the authoritative fragment \( \bullet \Delta_x \) while the non-authoritative fragment \( \circ \Delta_x \) is given to the writer – or put into the shared invariant \( \text{PrShared}_\tau \) in the case of non-exclusive protocols.

Apart from those resources, the protocol invariant imposes three conditions on the physical history \( h \) and the logical history \( \Delta \). The most important one is consistency: the modification order on \( h \) must correspond to the protocol order on \( \Delta \). The \textbf{Consistent} assertion captures exactly this property.

\footnote{The definitions differ slightly from the Coq mechanization to simplify the presentation.}
Reader(\(\delta\)) \(\triangleq\) \(\exists \delta \in \Delta, \forall \delta' \in \Delta. \delta' \cdot t \leq \delta \cdot t\)

Writer(\(\delta\)) \(\triangleq\) \(\exists \Delta. \delta \cdot \delta' \cdot \Delta\)

\(\Pr_{\text{Shared}}(t_x) \triangleq\) \(\exists \delta, \delta x. \Pr_{\text{Inv}}(\ell) \cdot \delta \cdot \delta x \cdot XW\)

\(\ell : s \tau \triangleq \lambda V. \Pr_{\text{Inv}}(\ell) \cdot \exists V \subseteq V, t_x \leq V(\ell). \Pr_{\text{Shared}}(t_x)\)

Additionally, we need to make sure that the logical history does not assign multiple protocol states to a write event. To this end, \textit{Unique} asserts that \(\Delta\) contains any write event at most once. With this restriction in place, we can now express \(h\) in terms of \(\Delta\) by forgetting the protocol states in \(\Delta\), i.e.

\(h = \{ (v, t, V) | (s, v, t, V) \in \Delta \}\).

Finally, we need an additional piece of information on the logical history: \(\Delta\) contains only final states, i.e. states reachable from any other state, in the disconnected segment. We call this property \textit{FinalInv}. The disconnected segment – illustrated in Figure 22a – spans all writes which are not connected to the most recent exclusive write. The write events contained in that segment share an important property: they could be \(mo\)-after newly inserted writes (which are always \(mo\)-after the most recent exclusive write). To re-establish \textit{Consistent} for new writes, we need to be able to relate their protocol state to any existing \(mo\)-later write. \textit{FinalInv} guarantees that the state associated with the newly inserted write comes before any state in the disconnected segment.

Before we turn to the local protocol assertions, we define the PCMs used in \(\Pr_{\text{Inv}}\). The PCM for the most recent exclusive write \(\delta x\) is a simple \textit{Auth} construction similar to the ones shown in §3, providing both an authoritative fragment for \(\Pr_{\text{Inv}}\) as well as an exclusive non-authoritative fragment to be given to the writer or to \(\Pr_{\text{Shared}}\). The PCM for logical histories is more involved, as it needs to provide three kinds of fragments: (1) an authoritative fragment held by \(\Pr_{\text{Inv}}\), (2) an exclusive fragment given to the writer or to \(\Pr_{\text{Shared}}\), representing the right to change the logical history – and thus also the physical one – and (3) a persistent fragment used in the (reader) protocol assertion, representing a lower bound on protocol state. To fulfill these requirements, we construct the rather intimidating double-\textit{Auth} PCM LH below.

\(\text{LE} \triangleq \text{State} \times \text{Val} \times \text{Time} \times \text{View} \) (logical event)
\(\text{LES} \triangleq \mathcal{P}(\text{LE}) \) (logical event set)
\(\text{XW} \triangleq \text{Auth}(\text{Ex}(\text{LE})) \) (most recent excl. event)
\(\text{LH} \triangleq \text{Auth}(\text{Auth}(\text{LES})) \) (auth. logical event set)

With these definitions we can now define protocol assertions, which are presented in Figure 23. As before, we elide ghost and invariant names, which add some bookkeeping overhead to the definitions.