GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation

Aaron Turon Viktor Vafeiadis Derek Dreyer
Max Planck Institute for Software Systems (MPI-SWS)
{turon,viktor,dreyer}@mpi-sws.org

Abstract

Weak memory models formalize the inconsistent behaviors that one can expect to observe in multithreaded programs running on modern hardware. In so doing, however, they complicate the already-difficult task of reasoning about correctness of concurrent code. Worse, they render impotent the sophisticated formal methods that have been developed to tame concurrency, which almost universally assume a strong (*i.e.,* sequentially consistent) memory model.

This paper introduces **GPS**, the first program logic to provide a full-fledged suite of modern verification techniques—including ghost state, protocols, *and* separation logic—for high-level, structured reasoning about weak memory. We demonstrate the effectiveness of GPS by applying it to challenging examples drawn from the Linux kernel as well as lock-free data structures. We also define the semantics of GPS and prove in Coq that it is sound with respect to the axiomatic C11 weak memory model.

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

Keywords Concurrency; Weak memory models; C/C++; Program logic; Separation logic

1. Introduction

When reasoning about the behavior of a multithreaded program, what can we assume about the interactions between concurrent threads and the shared memory they operate on? In the vast majority of the research on concurrent program verification, it is assumed that shared memory accesses are *sequentially consistent (SC)—i.e.,* there is a single global RAM, threads take turns interacting with it, and any memory update performed by one thread is immediately visible to all other threads. Even assuming sequential consistency, concurrent program verification is a highly challenging problem, since one must account for the myriad interleavings of threads. But fortunately there has been tremendous progress in recent years on advanced program logics and verification tools to help tame the complexity of interleaved execution [9, 11, 13, 15, 24, 29, 30, 36, 37, 39].

Unfortunately, the assumption of sequential consistency is unrealistically "strong": the synchronization required to implement it on modern architectures precludes useful compiler optimizations that reorder memory operations, and is thus considered by many to be too expensive in general [6]. Instead, languages like C/C++ [20, 21] and Java [26] support *weak* (or *relaxed*) models of memory, in which different threads may observe operations on shared memory occurring in different orders. To characterize precisely what types of inconsistent observations are permitted, these language-level memory models eschew the fiction of a single global RAM and an interleaving semantics; rather, they model valid program executions using *event graphs*, which track dependencies between memory accesses subject to a variety of consistency axioms, *e.g.,* "if *this* event is visible to a thread t, then so are *these other* events."

In short, weak memory models are useful in enabling compilers and hardware to aggressively optimize memory accesses, but they also invalidate the basic assumptions underlying existing verification tools and complicate the semantics of concurrent code. As such, they have led to a serious gap between the theory and practice of concurrency.

This paper takes a substantial step toward closing that gap by presenting the first concurrent program logic that is sound under weak memory assumptions but also supports a full suite of modern verification techniques: ghost state, protocols, and separation (GPS). Below, we briefly explain why these techniques have proven important in reasoning under strong memory assumptions (§1.1) and the obstacles we face in adapting them to weak memory (§1.2), before describing our contributions in more detail (§1.3).

OOPSLA '14, October 20–24, 2014, Portland, OR, USA.

ACM 978-1-4503-2585-1/14/10. . . $15.00.
http://dx.doi.org/10.1145/2660193.2660243

1.1 Concurrent program logics: the state of the art

The goal of most program logics is to prove "deep" correctness properties of code, and to do so in a *modular* fashion, whereby different components of a program can be verified in isolation, given only logical specifications (specs) of the other components. Modern logics for SC concurrency meet this goal through a variety of mechanisms—among the most widespread and effective are the following:

Ownership and separation. Concurrent programs are often inherently modular in the sense that different threads within a program control (or "own") disjoint pieces of the program state. This modularity is important for simplifying verification: if a thread owns a piece of state, one should be able to verify the thread's manipulations of that state without worrying about interference from other threads. Modern logics encapsulate this kind of reasoning through the mechanisms of *ownership* and *separation*.

Consider, for instance, the parallel composition rule of *concurrent separation logic (CSL)* [30]:

$$\frac{\{P_1\}\, e_1\, \{Q_1\} \qquad \{P_2\}\, e_2\, \{Q_2\}}{\{P_1 * P_2\}\ e_1 \mid\mid e_2\ \{Q_1 * Q_2\}}$$

Here, P_i and Q_i not only describe the facts that hold before and after the execution of thread e_i, they also characterize the piece of the program state that e_i "owns". The rule thus says that e_1 and e_2 may be safely run in parallel—without *any* interference checking—so long as P_1 and P_2 describe disjoint pieces of state, as enforced implicitly by the use of the "separating conjunction" $P_1 * P_2$ in the precondition.

Protocols. Separation lets one dispense with interference implicitly when threads do not in fact interfere. But sometimes explicit reasoning about interference is unavoidable, *e.g.,* when reasoning about racy (lock-free) data structures. In such cases, the most basic mechanism for restoring modular reasoning is the *invariant*, which describes a property holding of a piece of shared state at all times. With an invariant installed, different threads can be verified modularly so long as they all respect the invariant.

More generally, since invariants can be overly restrictive, modern logics support various forms of *protocols* for legislating interference. The best-known protocol mechanism is *rely-guarantee* [23], which describes the state transitions a thread may perform (the guarantee) vs. those its environment may perform (the rely). Recent protocol mechanisms improve upon rely-guarantee by supporting more abstract/concise forms of shared state transition systems [37].

Ghost state. Last but not least, *ghost* (or auxiliary) state refers generally to any behavior-preserving instrumentation of a program (or its proof) with additional "logical" state for the purposes of verification. Ghost state is often used to expose control flow, or to summarize execution history, in a way that could not be done just in terms of the "physical" state manipulated by the program. Furthermore, it is essential for the completeness of basic concurrency logics.

In newer logics, ghost state, protocols, and separation are used in tandem to great effect. For example, ghost state can be used to encode logical "permissions" (or "tokens"), which are ownable resources that control the ability to make certain transitions in shared state protocols. Ownership of permissions can then be transferred back and forth between threads *via* the same shared protocols, in turn providing a way to model the dynamic "role-playing" that occurs in realistic concurrent code. Logics such as RGSep [39], LRG [16], Deny-Guarantee [15], VCC [9], Chalice [24], CAP [13], CaReSL [37], FCSL [29], iCAP [36], and TaDA [11] depend on such a synthesis of ghost state, protocols, and separation.

1.2 Obstacles to modular weak memory reasoning

While the aforementioned mechanisms provide powerful, modular reasoning about concurrency, there are serious obstacles to adapting them to weak memory models like those of C/C++ or Java:

Separation obstacles. Models of concurrent separation logics have generally assumed the existence of a single global RAM (pieces of which may be owned by different threads) and a single global notion of "time" (based on an interleaving semantics). However, in weak memory models based on event graphs, there is no clear global notion of a heap or of time, making it unclear how to model basic notions like Hoare triples and separation.

Protocol obstacles. Most logics support protocols that govern multiple memory locations simultaneously, connecting the value of one location to another. But even this simple mechanism is unsound for weak memory: updates to different locations may appear in contradictory orders to different threads, so a thread can appear to be following the protocol from its own point of view while violating it from the point of view of other threads.

Ghost state obstacles. Traditional ghost state is incorporated by introducing explicit reads and writes to a program text, with the constraint that these operations must not change the code's observable behavior. But in weak memory models it is not clear how to usefully incorporate such reads and writes without also introducing events and ordering into the event graph that ultimately affect the program's behavior.

An important first step toward overcoming these obstacles is the recent work of Vafeiadis and Narayan on Relaxed Separation Logic (RSL) [38], the first logic for the C11 memory model. RSL supports simple, high-level reasoning about resource invariants and ownership transfer à la concurrent separation logic (CSL) [30]—a particularly simple combination of protocols and separation. But RSL provides no support for ghost state or for more complex forms of protocol (*e.g.,* rely-guarantee) or ownership transfer.

1.3 This paper

In this paper, we present **GPS**, the first logic to support ghost state, protocols and separation in a weak memory setting.

GPS builds on the groundwork laid by RSL, extending and generalizing it in several useful ways:

- **Protocols.** GPS supports *per-location (PL) protocols*, which are modeled after the protocols in recent concurrency logics but restricted in order to be sound under weak memory. The key to regaining soundness is to insist that a protocol may only precisely dictate the evolution of a *single* shared memory location, although it may make bounded assertions about the state of other memory locations, *e.g.,* "x's value may only grow over time, and when x contains n, y must contain *at least* n as well."
- **Ghost state.** The states of PL-protocols already constitute a useful form of ghost state for summarizing, *e.g.,* the history of an execution. To support ownable logical resources (*e.g.,* permissions), GPS offers an additional facility called *ghosts*. Ghosts enable one to create and manipulate whatever kind of logical resource one needs for a particular verification, so long as it can be formulated as a partial commutative monoid [14, 22, 25].
- **Ownership transfer.** In prior SC logics, threads can transfer ownership of resources to other threads through the medium of a shared protocol. GPS's PL-protocols also support ownership transfer between threads, but for soundness purposes it is somewhat restricted: the acquiring thread must perform an explicit synchronization operation like CAS in order to ensure that it is the exclusive recipient of the transfer. To facilitate ownership transfer even when the threads use only plain reads and writes, we introduce an additional mechanism called *escrows*.

We demonstrate the use of the above logical features through a series of concrete motivating examples in §3.

GPS targets the recent C11 [20, 21] memory model, which offers portable but fine-grained control over memory consistency guarantees. GPS supports verification of programs that use the three most important consistency modes for C11: nonatomic, release-acquire and sequentially-consistent (see §2). Since sequentially-consistent reasoning is relatively well understood, the paper presents the details only for the first two modes (but see §6 for further discussion of the SC mode). It is nonetheless worth stressing that verifications in GPS hold good under the full axioms of the C11 model (and thus for any compliant compiler). Moreover, **the entire logic, model and soundness proof of GPS have been formalized in Coq** [1]. For space reasons, we focus in this paper almost entirely on the proof theory of GPS; §3.7 sketches some details of the semantic model and soundness proof, but for full details we refer the reader to the appendix and Coq development.

To evaluate GPS, we have applied it to several challenging case studies drawn from the Linux kernel and lock-free data structures, as we describe in §4. These examples extend the reach of existing program logics: we know of no other logic that can verify them under C11's weak memory assumptions. We conclude in §5 and §6 with related and future work.

2. The C11 memory model

Memory models answer a seemingly simple question: when a thread reads from a location, what values can it encounter?

- Sequential consistency (SC) provides an equally simple answer: threads read the last value written. SC is based on interleaving, where threads interact atomically through a global heap holding each location's current value.
- In weaker consistency models, the "last value written" to a location plays no special role—it may not even be well-defined. Instead, threads can read out-of-date values due to CPU or compiler optimizations.

The C11 memory model [20, 21] strikes a careful balance between these extremes by offering a menu of consistency levels. Broadly, memory operations are classified as either *nonatomic* (the default) or *atomic*. Nonatomic accesses are intended for "normal data", while atomic accesses are used for synchronization.

Nonatomics are governed by a peculiar contract: the programmer can assume them to be SC, but must (under this assumption!) never create a *data race*—roughly, a thread must never write nonatomically if another thread might access the same location concurrently. This rule prevents the program from observing compiler/CPU optimizations on nonatomics.

Atomics offer the opposite tradeoff: concurrent threads may race to *e.g.,* update a location atomically, but the memory model provides weaker guarantees (and admits fewer optimizations) for atomic accesses in general. The precise guarantees are determined by an "ordering annotation", ranging from SC to fully relaxed. In this paper, we focus on the *release-acquire* ordering, which is the primary building block for non-SC synchronization. As such, we will use two ordering annotations, $O \in \{\texttt{at}, \texttt{na}\}$, for atomic (release-acquire) and nonatomic accesses, respectively. The full version of the memory model and GPS logic, as formalized in Coq [1], also includes sequentially-consistent accesses.

Examples Before introducing C11 formally, we build some intuition through two classic examples. The first is a simplified version of Dekker's algorithm, which provided the first solution to the mutual-exclusion problem [12]:

```
[x]_at := 1                  ||  [y]_at := 1
if [y]_at == 0 then          ||  if [x]_at == 0 then
  /* crit. section */        ||    /* crit. section */
```

We presume at the outset that x and y are pointers to distinct locations, both with initial value 0.[1] The two threads race to announce their *intent* to enter a critical section; each thread then checks whether it announced first. In this simplified version, even under SC, it is possible for both threads to lose. Unfortunately, in the C11 model, it is also possible for both threads to win! The intuition is that C11 allows the reads to

[1] We are using here the program logic notation for pointer dereferencing, $[-]$, which avoids ambiguity with the $*$ of separation logic.

Syntax

$$
\begin{array}{rcl}
v & ::= & x \mid V \text{ where } V \in \mathbb{N} \\
e & ::= & v \mid v + v \mid v == v \mid v \bmod v \\
& \mid & \texttt{let } x = e \texttt{ in } e \mid \texttt{repeat } e \texttt{ end} \\
& \mid & \texttt{if } v \texttt{ then } e \texttt{ else } e \mid \texttt{fork } e \\
& \mid & \texttt{alloc}(n) \mid [v]_O \mid [v]_O \texttt{ := } v \\
& \mid & \texttt{CAS}(v, v, v) \\
K & ::= & [\,] \mid \texttt{let } x = K \texttt{ in } e \\
T & \in & \mathbb{N} \xrightarrow{\text{fin}} (\mathit{ActName} \times \mathit{Exp})
\end{array}
$$

Event steps $\boxed{e \xrightarrow{\alpha} e}$

$$
\begin{array}{rcl}
K[e] & \xrightarrow{\alpha} & K[e'] \quad \text{if} \quad e \xrightarrow{\alpha} e' \\
\texttt{alloc}(n) & \xrightarrow{\mathbb{A}(\ell..\ell+n-1)} & \ell \\
[\ell]_O & \xrightarrow{\mathbb{R}(\ell,V,O)} & V \\
[\ell]_O \texttt{ := } V & \xrightarrow{\mathbb{W}(\ell,V,O)} & 0 \\
\texttt{CAS}(\ell, V_o, V_n) & \xrightarrow{\mathbb{U}(\ell,V_o,V_n)} & 1 \\
\texttt{CAS}(\ell, V_o, V_n) & \xrightarrow{\mathbb{R}(\ell,V',\texttt{at})} & 0 \quad \text{if} \quad V' \neq V_o
\end{array}
$$

Machine steps $\boxed{\langle T; G\rangle \longrightarrow \langle T'; G'\rangle}$

$$
\frac{\begin{array}{l} e \xrightarrow{\alpha} e' \qquad \mathsf{consistentC11}(G') \\ G'.A = G.A \uplus [a' \mapsto \alpha] \\ G'.\mathsf{sb} = G.\mathsf{sb} \uplus (a, a') \\ G'.\mathsf{mo} \supseteq G.\mathsf{mo} \\ G'.\mathsf{rf} \in \{G.\mathsf{rf},\ G.\mathsf{rf} \uplus [a' \mapsto b]\} \end{array}}{\begin{array}{c} \langle T \uplus [i \mapsto (a, e)]; G\rangle \longrightarrow \\ \langle T \uplus [i \mapsto (a', e')]; G'\rangle \end{array}}
$$

$$
\langle T \uplus [i \mapsto (a, K[\texttt{fork } e])]; G\rangle \longrightarrow \langle T \uplus [i \mapsto (a, K[0])] \uplus [j \mapsto (a, e)]; G\rangle
$$

Figure 1. Syntax and semantics of a language for C11 concurrency

be performed before the writes have become visible to all threads: the two threads can read stale values.

The second example illustrates a case where C11 atomics *do* enforce some ordering. The goal is to pass a "message" (in this case, just a single value, 37, but more generally a data structure) from one thread to another:

$$
\begin{array}{l|l} [x]_{\texttt{na}} \texttt{ := } 37; & \texttt{repeat } [y]_{\texttt{at}} \texttt{ end}; \\ [y]_{\texttt{at}} \texttt{ := } 1; & [x]_{\texttt{na}} \end{array}
$$

Again, we presume x and y are pointers to distinct locations, initially 0. The `repeat` construct executes an expression repeatedly until its value is nonzero, so the second thread will "spin" until it sees the write to y by the first thread. Unlike in Dekker's algorithm, here C11 will guarantee that the subsequent read from x will return 37. The key difference is that reading 1 from y yields *positive* information about what the first thread has done: if an atomic (release) write by a given thread is seen by another thread, so is everything that "happened before" the write, including all the writes that appear prior to it in the thread's code. Dekker's algorithm, by contrast, draws conclusions from *not* seeing a write by another thread.

In general, then, release-acquire in C11 doesn't guarantee that threads see the "globally-latest" write to a location, but does guarantee that (1) if a thread sees a *particular* write, it also sees everything that happened before it, and (2) of the writes a thread sees for a location, it reads from the latest.

A final point about the code: its use of y guarantees that the write to x by the first thread happens *before*—not concurrently with—the read of x by the second thread. So the code is data-race free, despite nonatomic accesses to x.

Event graphs We now present the C11 model formally, following Batty et al. [3] and subsequent simplifications [4, 38]. Our presentation makes some further simplifications due to our focus on release-acquire atomics, but GPS is sound for reasoning about non-atomic, release-acquire and SC accesses under the official C11 axioms. Reasoning about more advanced features of the C11 model—namely, *relaxed* and *consume* atomics—is left as future work and discussed in §6.

Since weak memory models allow threads to see stale values, they must track the history of an execution and use it to specify the values a read can return. The C11 model takes the *axiomatic* approach: it treats each step of a program execution as a node in a graph, and then constrains the graph through a collection of global axioms on several kinds of edges. Each node is labeled with an *action*:

$$
\alpha \ ::= \ \mathbb{S} \mid \mathbb{A}(\ell..\ell') \mid \mathbb{R}(\ell, V, O) \mid \mathbb{W}(\ell, V, O) \mid \mathbb{U}(\ell, V, V')
$$

where $O \in \{\texttt{na}, \texttt{at}\}$. The actions are $\mathbb{S}$kip (no memory interaction), $\mathbb{A}$llocate, $\mathbb{R}$ead, $\mathbb{W}$rite, and atomic $\mathbb{U}$pdate. Reads and writes record the location, value read/written, and ordering annotation. An atomic update $\mathbb{U}(\ell, V_o, V_n)$ simultaneously reads the value V_o from location ℓ and updates it with the new value V_n (used for *e.g.,* compare-and-set).

We assume an infinite set of event IDs; an *action map* A is then a finite partial map from event IDs to actions, which defines the nodes (and node labels) of a graph. An *event graph* $G = (A, \mathsf{sb}, \mathsf{mo}, \mathsf{rf})$ connects the nodes with three kinds of directed edges:

Sequenced-before ($\mathsf{sb} \subseteq \mathrm{dom}(A) \times \mathrm{dom}(A)$), which records the order of events as they appear in the code (*i.e.,* "program order"). For convenience, sb is *not* transitive: it relates each node only to its immediate successors in program order (see [38]).

Modification order ($\mathsf{mo} \subseteq \mathrm{dom}(A) \times \mathrm{dom}(A)$), which is a strict, total order on all the writes to each location, but does not relate writes to different locations. It determines which of any pair of (possibly concurrent) writes to a location is considered to "take effect" first—a determination that is agreed upon globally.

Reads-from ($\mathsf{rf} \in \mathrm{dom}(A) \rightharpoonup \mathrm{dom}(A)$), which maps each read to the unique write, if any, that it is reading from. It is undefined for reads from uninitialized locations.

The goal of the C11 axioms is to constrain the rf relation so that it provides the guarantees mentioned informally above. The axioms rely on a pair of derived relations:

Synchronized-with ($\mathsf{sw} \subseteq \mathrm{dom}(A) \times \mathrm{dom}(A)$) defines those read-write pairs that induce "transitive visibility", as in

the message-passing example above. In the release-acquire fragment of C11, these include any read/write pair marked as atomic:

$$\mathsf{sw} \triangleq \{(a,b) \mid \mathsf{rf}(b) = a,\ \mathsf{isAtomic}(a),\ \mathsf{isAtomic}(b)\}$$

Happens-before ($\mathsf{hb} \triangleq (\mathsf{sb} \cup \mathsf{sw})^+$) is the heart of the model: $\mathsf{hb}(a,b)$ means that if a thread has observed event b, then it has observed event a as well; it bounds staleness.

Axioms Only the sb order is determined by the program as written. The other orders are chosen arbitrarily—except that they must satisfy C11's axioms. These axioms include some sanity checks:

- hb is acyclic (an event cannot happen before itself),
- a location cannot be allocated more than once,
- rf maps reads to writes of the same location and value, and it is not possible to read a value from a write that happens later:[2]

$$\mathsf{rf}(b) = a \implies \exists \ell, V.\ \mathsf{writes}(a,\ell,V),\ \mathsf{reads}(b,\ell,V),\ \neg\mathsf{hb}(b,a)$$

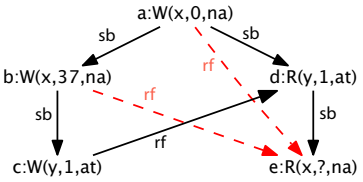
- atomic updates must, in fact, be atomic: the update must *immediately* follow the event it reads from in mo:

$$\mathsf{isUpd}(c),\ \mathsf{rf}(c) = a \implies \mathsf{mo}(a,c),\ \nexists b.\ \mathsf{mo}(a,b),\ \mathsf{mo}(b,c)$$

But the heavy lifting of the C11 model is done by a final axiom, called *coherence*, which connects mo, rf, and hb:

$$\mathsf{hb}(a,b) \implies \neg\mathsf{mo}(b,a),\ \neg\mathsf{mo}(\mathsf{rf}(b),a),\ \neg\mathsf{mo}(b,\mathsf{rf}(a)),\ \neg\mathsf{mo}(\mathsf{rf}(b),\mathsf{rf}(a))$$

To see how coherence formally ensures the intuitive guarantees we gave above, we apply it to the simple message-passing example, this time in graph form:



(Initialization of y elided.)

In the depicted execution, the event d in the second thread reads from the event c in the first thread (which writes 1 to y). We use coherence to deduce that the subsequent read of x in event e *must* read from event b (which writes 37 to x):

- Since $\mathsf{sb}(a,b)$, and thus $\mathsf{hb}(a,b)$, we have $\neg\mathsf{mo}(b,a)$. But mo is a total order on writes to a location, so $\mathsf{mo}(a,b)$.
- Since $\mathsf{rf}(d) = c$, we have $\mathsf{sw}(c,d)$ and thus $\mathsf{hb}(c,d)$. By transitivity of hb, we have $\mathsf{hb}(b,d)$ and hence $\mathsf{hb}(b,e)$.
- Coherence then says that $\neg\mathsf{mo}(\mathsf{rf}(e),b)$, *i.e.,* that e cannot read from any write earlier (in mo) than b; in particular, e cannot read from a. It must read from b.

[2] The reads and writes functions extract the locations and values from normal read/writes as well as atomic updates.

The key is the second step, where we deduce the existence of an sw edge (and thus the transitive visibility, by hb, of previous writes). In Dekker's algorithm, by contrast, when one thread reads the other's flag, there are no hb edges that ensure it sees the "latest" write.

We write $\mathsf{consistentC11}(G)$ if a graph G satisfies the axioms above (plus one more for uninitialized reads).

A language for C11 concurrency Figure 1 gives a simple language of expressions e with allocation, pointer arithmetic, thread forking and order-annotated memory operations. To streamline the semantics, we adopt A-normal form [18], which requires intermediate computations to be named through `let`-binding (the only evaluation context K). The `if` expression takes the `then` branch when its guard is non-zero. Similarly, `repeat` executes the given subexpression until it produces a non-zero value, which is returned.

The semantics is given in two layers. First, expressions e freely generate actions α through the relation $e \xrightarrow{\alpha} e'$. Pure expressions generate the $\mathbb{S}$ action (*e.g.,* $\texttt{let}\ x = V\ \texttt{in}\ e \xrightarrow{\mathbb{S}} e[V/x]$), while expressions that interact with memory generate corresponding memory model actions. Note that reading generates an $\mathbb{R}$ action for an *arbitrary* value. The actual value read is constrained by the second layer, which governs *machine configurations* $\langle T; G\rangle$.

Machine configurations track the current pool of threads, T, and the event graph built up so far, G. For each thread, the pool maintains (1) the identity of the last event produced by the thread and (2) an expression giving the thread's continuation. To take a (non-`fork`) step, a thread's continuation must generate some action α, which is then incorporated into an updated event graph G', where it is placed in sb order after the thread's previous event. The mo order for G' can arbitrarily extend the one for G, but because it is a strict total order on writes, the extension will only add relationships to the new node. The rf order can likewise only add a read for the new node, which must read from some previously-existing write. Finally, the new graph G' is assumed to satisfy the C11 axioms, constraining both the possible events and edges. The validity of this semantics for C11 is discussed in the appendix [1].

We write $[\![e]\!]$ for the set of final values e can produce, starting with a single-node event graph (where the start node is action $\mathbb{S}$). If at any point e creates a data race or memory error (defined formally in the appendix), then $[\![e]\!] = \mathbf{err}$; the C11 semantics leaves such programs undefined. Any expression verified by GPS is guaranteed to be free of data races on non-atomic locations and memory errors.

3. GPS

The C11 memory model successfully serves as a contract between compiler and programmer, making it possible—in principle—to resolve disputes (can a read of x here return 0?) by reference to global axioms. These axioms—again, in principle—also support certain intuitions about, *e.g.,*

transitive visibility. But, even with an example as simple as one-shot message passing (§2), the intuitions are not *directly* captured by the axioms. Rather, they emerge through chains of subtle reasoning showing that certain edges must, or must not, exist. Axiomatic reasoning is relentlessly global: a read event can potentially read from any write in the graph, so the axioms must be applied to each write to rule it in or out.

Our goal is to supplement the C11 memory model with a program logic that (1) directly captures intuitions about transitive visibility and (2) supports thread-local reasoning. GPS achieves both goals through

- *per-location protocols* that abstract away event graphs;
- *ghosts* and *escrows*, which govern logical permissions in the style of recent separation logics.

Setup GPS is a separation logic for an expression language. Its central judgment is the Hoare triple, $\{P\}\, e\, \{x.\ Q\}$, which says that when given resources described by P, the expression e is memory safe and data-race free. If, moreover, e terminates with a value V, it will do so with resources satisfying $Q[V/x]$. We will introduce assertions P gradually. For now, we assume they include the basic operators of *multi-sorted* first-order logic:

$$P \ ::= \ t = t \ \mid \ P \wedge P \ \mid \ P \vee P \ \mid \ P \Rightarrow P \ \mid \ \forall X.P \ \mid \ \exists X.P$$

where t ranges over *terms*. We write $t : \theta$ if t has sort θ, and assume that variables X are broken into classes by sort. For now, the only sort is Val, ranged over by variables ℓ, x, y, z.

3.1 Per-location protocols

We start with a slight variation on the message-passing example from §2:

$$\begin{array}{l}[x]_{\mathtt{at}} := 37;\\ [y]_{\mathtt{at}} := 1;\end{array} \ \Big\|\ \cdots\ \Big\| \ \begin{array}{l}[x]_{\mathtt{at}} := 37;\\ [y]_{\mathtt{at}} := 1;\end{array} \ \Big\| \ \begin{array}{l}\mathtt{repeat}\ [y]_{\mathtt{at}}\ \mathtt{end};\\ [x]_{\mathtt{at}}\end{array}$$

In this variant, there are multiple threads sending the same message (37),[3] and intuitively this works for the same reason the original does: transitive visibility. We want to articulate this common intuition in a way that doesn't depend on how many threads are sending the message 37 or involve global reasoning about the event graph.

A tempting starting point is to simply say that the values that x and y point to progress from $(0, 0)$ to $(37, 0)$ to $(37, 1)$. In other words, we would like to impose a *protocol* on the evolution of x and y. Such protocols are the lifeblood of prior SC concurrency logics (see §5.1 for details), but alas, the kind of reasoning they support is unsound for weak memory in general: it assumes that all threads will see writes to different locations in the same order. In actuality, independent (*i.e.,* hb-unrelated) writes to different locations can appear to threads in different orders, which is why Dekker's algorithm fails. If we want thread-local reasoning, we need an approach that accounts for what *our thread* may see, while capturing the happens-before relationship between writes.

[3] Note that the writes to x here must be atomic to avoid data races.

The key insight of GPS is that we *can* constrain the evolution of values if we focus on one location at a time: mo provides a linear order, seen by all threads, on the writes to a given location. Toward this end, GPS provides *per-location protocols*, which are state transition systems governing a single shared location. Using protocols, we can express the changes to x and y independently:

These transition systems offer an abstraction of the event graph: each state represents *a set* of write events, while edges represent mo relationships between them. Thus, for x, we see that all of the writes of 37 are mo-later than the initial write of 0. But these independent constraints alone are not enough: we must ensure that y can only be in state 1 if x is "known" to be in state 37.

In general, protocol states are abstract; the labels on the transition systems above are merely suggestive. Each state is given an *interpretation*, which constrains the values that may be written to the location in that state, but may also impose other constraints—including, as we will see, constraints on *other protocols*. (Treating states abstractly allows us to, in effect, associate a ghost variable with each memory location, as §4 will show.)

Formally, we assume a sort State of protocol states, ranged over by variables s. GPS is parameterized by (1) the grammar of terms of sort State and (2) a set of *protocol types* (metavariable τ). For each protocol type τ, the user of the logic specifies:

- A *transition relation* $\sqsubseteq_\tau$, a partial order on states.
- A *state interpretation* $\tau(s, z)$, an assertion in which s and z appear free (*i.e.,* a predicate on s and z). The assertion represents what must be true of a value z for a thread to be permitted to write it to the location in state s.

For the message passing example, we introduce a protocol type Dat governing location x. Writing abstract states in bold, we say $\mathbf{0} \sqsubseteq_{\mathsf{Dat}} \mathbf{0}$, $\mathbf{0} \sqsubseteq_{\mathsf{Dat}} \mathbf{37}$, $\mathbf{37} \sqsubseteq_{\mathsf{Dat}} \mathbf{37}$, and define

$$\mathsf{Dat}(s, z) \ \triangleq \ (s = \mathbf{0} \wedge z = 0) \vee (s = \mathbf{37} \wedge z = 37)$$

To give the protocol for y, however, we need a way of talking about the protocol for x in its state interpretations. For this purpose, GPS offers *protocol assertions*, $\boxed{\ell : s \mid \tau}$, which say that location ℓ is governed by the protocol type τ, and has been observed in state s, thus giving a *lower bound* on the current protocol state.

We can now give the protocol for y. We introduce a protocol type $\mathsf{Flg}(\ell)$ that is parameterized over a location ℓ (which we will instantiate with x). Again writing abstract states in bold, we say $\mathbf{0} \sqsubseteq_{\mathsf{Flg}} \mathbf{0}$, $\mathbf{0} \sqsubseteq_{\mathsf{Flg}} \mathbf{1}$, $\mathbf{1} \sqsubseteq_{\mathsf{Flg}} \mathbf{1}$, and

$$\begin{array}{rcl}\mathsf{Flg}(\ell)(s, z) & \triangleq & (s = \mathbf{0} \wedge z = 0)\\ & \vee & (s = \mathbf{1} \wedge z = 1 \wedge \boxed{\ell : \mathbf{37} \mid \mathsf{Dat}})\end{array}$$

Thus, to move to state $\mathbf{1}$ in $\mathbf{Flg}(x)$, a thread must (1) write 1 to y and (2) have already observed that $\boxed{x : \mathbf{37} \mid \mathbf{Dat}}$, which it can ensure by first writing 37 to x itself.

What happens when a thread reads y? GPS supports the following Hoare triple for atomic reads of a location ℓ:[4]

$$\frac{\forall s' \sqsupseteq_\tau s.\ \forall z.\ \tau(s', z) \Rightarrow Q}{\left\{\boxed{\ell : s \mid \tau}\right\}\ [\ell]_{\mathtt{at}}\ \left\{z.\ \exists s'.\ \boxed{\ell : s' \mid \tau} \wedge Q\right\}}$$

The precondition requires some pre-existing knowledge about ℓ's protocol. (For the message receiver, this knowledge will be $\boxed{y : \mathbf{0} \mid \mathbf{Flg}(x)}$.) The pre-existing knowledge gives a lower bound on the possible writes the read could read from: they must be at least as far as state s in the protocol. The premise of the rule then quantifies, abstractly, over the write we might be reading from: it must have moved to some future state s' in the protocol, and have written some value z such that $\tau(s', z)$ holds. From all such possible writes, we derive a common assertion Q—but note that s' and z can appear in Q, so it can tie together the value read and the state observed.

Altogether, we have:

$$\left\{\boxed{y : \mathbf{0} \mid \mathbf{Flg}(x)}\right\}\ [y]_{\mathtt{at}}\ \left\{\begin{array}{l} z.\ \boxed{y : \mathbf{0} \mid \mathbf{Flg}(x)} \wedge z = 0 \\ \vee\ \boxed{y : \mathbf{1} \mid \mathbf{Flg}(x)} \wedge z = 1 \wedge \boxed{x : \mathbf{37} \mid \mathbf{Dat}} \end{array}\right\}$$

So if a thread reads 1 from y, it learns a lower bound on the protocol state for x. If it subsequently reads x, it is guaranteed to see 37.

$$\left\{\boxed{y : \mathbf{1} \mid \mathbf{Flg}(x)} \wedge \boxed{x : \mathbf{37} \mid \mathbf{Dat}}\right\}\ [x]_{\mathtt{at}}\ \left\{z.\ \boxed{y : \mathbf{1} \mid \mathbf{Flg}(x)} \wedge \boxed{x : \mathbf{37} \mid \mathbf{Dat}} \wedge z = 37\right\}$$

Before describing the rest of GPS, we briefly consider the connection to the C11 model. GPS assertions say what is known at each point in a thread's code, with each such point corresponding to a node in the event graph. A thread will only be able to claim $\boxed{\ell : s \mid \tau}$ if a write moving ℓ to (abstract) state s happens before the corresponding node in the event graph. But because writes to ℓ in mo order correspond to moves within the protocol, the thread can subsequently read only from a write in some state $s' \sqsupseteq_\tau s$. PL-protocols have allowed us to abstract away from the event graph and to reason thread-locally: the thread receiving the message does not need to know anything about the code/events of the sending threads except that they follow the protocols.

3.2 Physical resources

GPS makes the simplifying assumption that each location is either always used nonatomically (*i.e.,* for data), or always used atomically (*i.e.,* for synchronization). Atomic locations can be freely shared between threads, which can only make protocol assertions about them; since protocol assertions are just lower bounds, they are invariant under interference from other threads. Nonatomic locations, on the other hand, must be treated as resources to ensure that only one thread can write to them at a time, in order to avoid data races. GPS thus includes the assertions

$$P \ ::= \ \cdots \ \mid \mathsf{uninit}(\ell) \mid \ell \hookrightarrow v \mid P * P$$

which resemble traditional separation logic, except that locations begin uninitialized. The heap assertion $\ell \hookrightarrow v$ means that ℓ is classified as nonatomic, and currently points to value v. We thus get the usual separation logic axioms for nonatomic locations:

$$\{\mathsf{true}\}\ \mathtt{alloc}(n)\ \left\{\begin{array}{r} x.\ \mathsf{uninit}(x) * \cdots \\ * \mathsf{uninit}(x + n - 1) \end{array}\right\}$$

$$\{\mathsf{uninit}(\ell) \vee \ell \hookrightarrow -\}\ [\ell]_{\mathtt{na}} := v\ \{\ell \hookrightarrow v\}$$

$$\{\ell \hookrightarrow v\}\ \ [\ell]_{\mathtt{na}}\ \ \{x.\ x = v * \ell \hookrightarrow v\}$$

The separating conjunction $P * Q$ requires that resources claimed by P are disjoint from those of Q, *e.g.,*

$$\mathsf{uninit}(\ell) * \mathsf{uninit}(\ell') \Rightarrow \ell \neq \ell'$$
$$\ell \hookrightarrow v * \boxed{\ell' : s \mid \tau} \Rightarrow \ell \neq \ell'$$

but since atomic locations are shared, separation enforces only that different observations about their state cohere:

$$\boxed{\ell : s \mid \tau} * \boxed{\ell : s' \mid \tau'} \quad \Rightarrow \quad \tau = \tau' \wedge (s \sqsubseteq_\tau s' \ \vee \ s' \sqsubseteq_\tau s)$$

In addition to these axioms, GPS supports the usual rules for a concurrent separation logic; see Figure 2.

$$\frac{\{P\}\, e\, \{x.\ Q\}}{\{P * R\}\, e\, \{x.\ Q * R\}} \qquad \frac{\{Q\}\, e\, \{\mathsf{true}\}}{\{P * Q\}\ \mathtt{fork}\ e\, \{P\}}$$

$$\frac{\{P\}\, e\, \{x.\ Q\} \qquad \forall x.\ \{Q\}\, e'\, \{y.\ R\}}{\{P\}\ \mathtt{let}\ x = e\ \mathtt{in}\ e'\, \{y.\ R\}} \qquad \frac{P \Rrightarrow Q}{P \Rrightarrow Q}$$

$$\frac{\{P\}\, e\, \{x.\ (x = 0 \wedge P) \vee (x \neq 0 \wedge Q)\}}{\{P\}\ \mathtt{repeat}\ e\ \mathtt{end}\, \{x.\ Q\}} \qquad \frac{P \Rrightarrow Q}{P * R \Rrightarrow Q * R}$$

$$\frac{P' \Rrightarrow P \qquad \{P\}\, e\, \{x.\ Q\} \qquad \forall x.\ Q \Rrightarrow Q'}{\{P'\}\, e\, \{x.\ Q'\}}$$

Figure 2. A selection of basic logical rules for GPS

3.3 Ghost resources

Our earlier presentation of protocols implicitly assumed that all threads can make the same moves within a protocol. But we often want to say that only certain threads have the right to make a particular move. To do so, we add non-physical resources—*ghosts*—to GPS. These purely logical resources are used to express arbitrary notions of permission that can be divided amongst threads. Here we explain what ghosts are; the subsequent subsections explain how they are used together with protocols.

Following recent work in separation logic [14, 22, 25], we model ghosts as *partial commutative monoids* (PCMs): GPS is parameterized by a collection of PCMs μ, such that

[4] This rule is sound only for the assertions we have introduced so far; the general rule is given in "Ownership transfer through protocols", below.

- There is a sort PCM_μ for each μ,
- Terms of sort PCM_μ include *unit* ε_μ and *composition* $\cdot_\mu$.

The unit represents the empty permission, while $t \cdot_\mu t'$ combines the permissions t and t'. We do *not* want all compositions to be defined: we want certain permissions to be *exclusive*, meaning that they do not compose with themselves. So composition is a partial function, but is commutative and associative where defined (and $\varepsilon_\mu \cdot_\mu t = t$ for any t).

Within the logic, we add *ghost assertions*, $\boxed{\gamma : t \mid \mu}$, which claim ownership of the ghost permission t drawn from some PCM μ. Since we may want to use many instances of a particular PCM, ghosts have an *identity* γ. Being nonphysical, ghosts are manipulated entirely through the rule of consequence, which is generalized to allow *ghost moves* $\Rrightarrow$, rather than just implications; see Figure 2. These moves allow new ghosts t to appear out of thin air, with a fresh identity: $\mathsf{true} \Rrightarrow \exists\gamma.\, \boxed{\gamma : t \mid \mu}$. Once a ghost is created, it can be split apart using $*$, as follows:

$$\boxed{\gamma : t \cdot_\mu t' \mid \mu} \;\Leftrightarrow\; \boxed{\gamma : t \mid \mu} * \boxed{\gamma : t' \mid \mu}$$

We take $\boxed{\gamma : t \cdot_\mu t' \mid \mu}$ to be false if $t \cdot_\mu t'$ is undefined.

A very simple but useful kind of permission is a *token*, which is meant to be owned by exactly one thread at a time. We can model this as a PCM, Tok, with two elements, ε and $\diamond$ (the token), with $\varepsilon \cdot \diamond = \diamond = \diamond \cdot \varepsilon$. We leave the composition $\diamond \cdot \diamond$ undefined, so that $\boxed{\gamma : \diamond \mid \mathsf{Tok}} * \boxed{\gamma : \diamond \mid \mathsf{Tok}} \;\Rightarrow\; \mathsf{false}$. Hence, GPS ensures the token for ghost γ cannot be owned twice. (We use this PCM in several examples in §3.6.)

3.4 Taking stock: resource ownership *vs*. knowledge

We have now seen the full complement of resource ownership assertions (physical and ghost) provided by GPS, with $*$ combining or separating them. Ownership can be divided by the `fork` rule (Figure 2), which allows the parent thread to donate some of its resources to the child thread. But we will also need to transfer ownership between already-running threads—while ensuring, of course, that claims of ownership are not duplicated in the process. GPS provides two mechanisms for doing so, one physical and the other nonphysical, described in the next two subsections.

Both mechanisms rely on a fundamental distinction between assertions possibly involving *resource ownership* (like $\ell \hookrightarrow v$) and assertions only involving *knowledge* (like $t = t'$). The key is that, while ownership can come and go, knowledge remains true forever.

GPS has a modality $\Box$ for knowledge, where $\Box P$ holds if P is true and does not depend on resource ownership—and therefore will remain true forever. These properties of knowledge are captured in two axioms:

$$\Box P \Rightarrow P \qquad\qquad \Box P \Leftrightarrow \Box P * \Box P$$

Using the second axiom and the frame rule, we can derive:

$$\frac{\{P * \Box R\}\, e\, \{x.\ Q\}}{\{P * \Box R\}\, e\, \{x.\ Q * \Box R\}}$$

Knowledge is retained no matter what an expression does.

Knowledge includes assertions that are "pure" in the parlance of separation logic, like equalities on terms, but it also includes protocol observations:

$$t = t' \Rightarrow \Box(t = t') \qquad\qquad \boxed{\ell : s \mid \tau} \;\Rightarrow\; \Box\boxed{\ell : s \mid \tau}$$

On the other hand, assertions about ownership never constitute knowledge: the axiom $\Box(\ell \hookrightarrow v) \Rightarrow \mathsf{false}$ says that it is impossible to treat nonatomic ownership as knowledge.

Finally, the $\Box$ modality distributes over $\wedge$, $\vee$, $\forall$, and $\exists$.

3.5 Ownership transfer through protocols

To explain physically-based ownership transfers, we consider a simple spinlock:

$$\begin{array}{rcl}
\texttt{newLock}() & \triangleq & \texttt{let } x = \texttt{alloc}(1) \texttt{ in } [x]_{\mathtt{at}} \texttt{ := unlocked};\ x \\
\texttt{lock}(x) & \triangleq & \texttt{repeat CAS}(x, \texttt{unlocked}, \texttt{locked}) \texttt{ end} \\
\texttt{unlock}(x) & \triangleq & [x]_{\mathtt{at}} \texttt{ := unlocked}
\end{array}$$

where `unlocked` = 0 and `locked` = 1. We want to reason about this lock in the style of concurrent separation logic [30], *i.e.,* we want to be able to prove the following triples:

$$\begin{array}{rl}
\{P\}\ \texttt{newLock}()\ & \{x.\ \Box\mathsf{isLock}(x)\} \\
\{\mathsf{isLock}(x)\}\ \ \texttt{lock}(x)\ & \{P\} \\
\{\mathsf{isLock}(x) * P\}\ \texttt{unlock}(x)\ & \{\mathsf{true}\}
\end{array}$$

Here, the assertion P is an arbitrary *resource invariant* (*e.g.,* claiming ownership of nonatomic locations) protected by the lock, while isLock represents the permission to use the lock. These triples reflect a transfer of ownership of the resources satisfying P, first upon creation of the lock, and then between each successive thread that acquires the lock. But the whole point of the lock is to ensure that when multiple threads race to acquire it, only one will win—and it is the use of `CAS` that guarantees this, by physical atomicity. We want to leverage the fact that `CAS` physically arbitrates races to logically arbitrate ownership transfers.

To do so, we revise our understanding of protocol state interpretations: rather than just a way to communicate knowledge between threads, they are more generally a way to transfer resource ownership between threads. For the spinlock, we can get away with a simple protocol type $\mathbf{LP}$ having a single state Inv, where

$$\mathbf{LP}(\mathsf{Inv}, z) \;\triangleq\; (z = \texttt{unlocked} * P) \vee z = \texttt{locked}$$

Intuitively, whenever a thread releases the lock, it must have reestablished the resource invariant P, which it then relinquishes, allowing P to be transferred to the next thread acquiring the lock. We can then define $\mathsf{isLock}(x) \triangleq \boxed{x : \mathsf{Inv} \mid \mathbf{LP}}$.

To initialize an atomic location ℓ with state s and value v, a thread must relinquish resources $\tau(s, v)$:

$$\{\mathsf{uninit}(\ell) * \tau(s, v)\}\ [\ell]_{\mathtt{at}} \texttt{ := } v\ \left\{\boxed{\ell : s \mid \tau}\right\}$$

which is reflected in the triple for `newLock()` above.

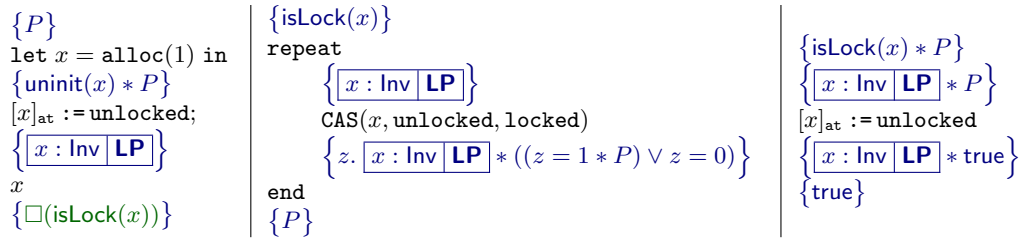


Figure 3. Proof outlines for the simple spinlock implementation

Subsequently, we can reason about CAS as follows:

$$\frac{\begin{array}{c}\forall s' \sqsupseteq_\tau s.\ \tau(s', V_o) * P \Rrightarrow \exists s'' \sqsupseteq_\tau s'.\ \tau(s'', V_n) * Q \\ \forall s'' \sqsupseteq_\tau s.\ \forall y \neq V_o.\ \tau(s'', y) * P \Rightarrow \Box R\end{array}}{\left\{\begin{array}{l}\boxed{\ell : s \mid \tau} \\ * P\end{array}\right\}\ \mathtt{CAS}(\ell, V_o, V_n)\ \left\{\begin{array}{l}z.\ \exists s''.\ \boxed{\ell : s'' \mid \tau} * \\ ((z = 1 * Q) \vee (z = 0 * P * \Box R))\end{array}\right\}}$$

The two premises of the rule correspond to the CAS succeeding or failing, respectively. In the successful case, we observe the protocol in some state s', and *choose* a new state s'' that is reachable from it. To make the move from s' to s'', we (1) gain the resources $\tau(s', V_o)$, because we won the race to CAS, but (2) must relinquish resources $\tau(s'', V_n)$, which can be transferred to the next successful CAS on ℓ. We can use any resources P we owned beforehand, and we get to keep any leftover resources Q.

The failure case works like an atomic read, except that we do not learn the exact value observed; we know only that it differs from the expected value V_o. Since multiple threads can read from the same write, it should not be possible to gain resources by reading alone—but it should still be possible to gain knowledge. Thus the full read rule is:

$$\frac{\forall s' \sqsupseteq_\tau s.\ \forall z.\ \tau(s', z) * P \Rightarrow \Box Q}{\left\{\boxed{\ell : s \mid \tau} * P\right\}\ [\ell]_{\mathtt{at}}\ \left\{z.\ \exists s'.\ \boxed{\ell : s' \mid \tau} * P * \Box Q\right\}}$$

This rule differs from the version we gave earlier in two respects. First, the assertion Q is placed under the $\Box$ modality, ensuring that readers only gain knowledge, not resources, through the protocol. Second, the precondition includes an arbitrary assertion P, which we combine via $*$ with the interpretation of the state we are reading.

The inclusion of the assertion P enables *rely-guarantee* reasoning through protocols. For the protocol to be in state s', some thread must have written z to ℓ while also giving up resources $\tau(s', z)$. If we read from this write, we know that the resources involved must be disjoint from any resources P we currently own. We can therefore *rule out* certain protocol states on this basis. The typical way to do so is through ghosts: we can require that, to move to a certain protocol state s', a thread must give up a ghost t (*e.g.,* a token). Thus, if a thread owns some ghost t' such that $t \cdot t'$ is undefined, then the thread knows that the protocol cannot be in state s'. We illustrate this kind of reasoning in the next subsection.

Finally, we have a rule for atomic writes:

$$\frac{P \Rrightarrow \tau(s'', V) * Q \qquad \forall s' \sqsupseteq_\tau s.\ \tau(s', -) * P \Rightarrow s'' \sqsupseteq_\tau s'}{\left\{\boxed{\ell : s \mid \tau} * P\right\}\ [\ell]_{\mathtt{at}} := V\ \left\{\boxed{\ell : s'' \mid \tau} * Q\right\}}$$

Writes are surprisingly subtle. Prior to writing, our thread knows some lower bound s on the protocol state. But because the write may be racing with unknown other writes (or CASes), we do not know (or learn!) the "current" state of the protocol. Instead, we must move to a state s'' that is reachable from *any* state $s' \sqsupseteq_\tau s$ that concurrent threads may be moving to. As with reads and CASes, though, we know that any such state s' must be satisfiable with resources disjoint from our resources, P. In particular, if $\tau(s', -) * P \Rightarrow \mathsf{false}$, then we do *not* have to show that $s'' \sqsupseteq_\tau s'$.

In summary:

- Reads relinquish nothing and gain knowledge.
- Writes relinquish ownership and gain nothing.
- CASes relinquish and gain ownership when successful, and behave like reads when unsuccessful.

Returning to the simple spinlock example introduced at the beginning of this subsection, Figure 3 contains the proof outlines for newLock(), lock(x), and unlock(x). The proofs are straightforward and follow immediately from the rules for atomic accesses given in this subsection and the rules of Figure 2.

3.6 Ownership transfer through escrows

We have just shown how GPS axiomatizes ownership transfer for programs that use the *explicit*, built-in form of synchronization offered by CAS. But in fact programs can and do build up their own *implicit* mechanisms for ownership transfer without using CAS (which is relatively expensive). We already saw such implicit synchronization at work in the original version of our "message-passing" example from §2:

$$\begin{array}{l}[x]_{\mathtt{na}} := 37; \\ [y]_{\mathtt{at}} := 1;\end{array} \quad \Big\| \quad \begin{array}{l}\mathtt{repeat}\ [y]_{\mathtt{at}}\ \mathtt{end}; \\ [x]_{\mathtt{na}}\end{array}$$

Unlike the version of this example that we showed how to verify in §3.1, this version transfers *ownership* of a *nonatomic* location ($x \hookrightarrow 37$) from *one* thread to another, and it does so without using CAS. Intuitively, the reason this works is that the threads have agreed ahead of time—implicitly—that once y is set to 1, the second thread will have the exclusive permission to take ownership of x. (Indeed, the transfer would

be unsound if there were two copies of the second thread operating concurrently.) However, since the second thread does not use CAS, it cannot transfer ownership of x directly out of y's protocol—some additional mechanism is needed.

Thus we are led to the final concept in GPS: *escrows*.[5] The idea is that a thread may *indirectly* transfer a resource to another thread by placing it "under escrow": it is then inaccessible to any thread until some exclusive, logical condition is met, at which point the thread meeting the condition gains ownership of it. GPS is parameterized over a set of escrow types (metavariable σ) and definitions, written $\sigma : P \rightsquigarrow Q$. Here Q represents the resource to be placed under escrow, while P represents the *transfer condition*, which must be *exclusive* ($P * P \Rrightarrow \mathsf{false}$) to ensure that ownership of Q is only transferred out of the escrow to *one* receiving thread.

Escrows are created and used via ghost moves, where the assertion $[\sigma]$ says that an escrow of type σ is known to exist:

$$\frac{\sigma : P \rightsquigarrow Q}{Q \Rrightarrow [\sigma]} \qquad \frac{\sigma : P \rightsquigarrow Q}{P \wedge [\sigma] \Rrightarrow Q} \qquad [\sigma] \Rightarrow \Box[\sigma]$$

The first rule allows Q to be put under escrow; ownership is lost, in exchange for the *knowledge* $[\sigma]$—and because $[\sigma]$ is knowledge, it can be learned about through reading. When later extracting the resource Q from the escrow $[\sigma]$, the condition P is *consumed*; this fact, together with the exclusivity of P, ensures that an escrow can only be used to transfer ownership once.

Returning to the message-passing example, the idea is to define an escrow type, $\mathbf{XE}(\gamma)$, which governs the transfer of the resource $x \hookrightarrow 37$. The escrow type is parameterized by γ, which is the name of an exclusive ghost token, $\boxed{\gamma : \diamond \mid \mathsf{Tok}}$, that will be used to guard the escrow (*i.e.,* as its transfer condition). The second thread will start out as the (unique) owner of this token, but then exchange it for ownership of x. Formally, we define $\mathbf{XE}(\gamma)$ as follows:

$$\mathbf{XE}(\gamma) : \quad \boxed{\gamma : \diamond \mid \mathsf{Tok}} \quad \rightsquigarrow \quad x \hookrightarrow 37$$

We then define a single protocol governing y, namely $\mathbf{YP}(\gamma)$, with states 0 and 1 and transition relation $\leq$, and the following state interpretations:

$$\begin{aligned} \mathbf{YP}(\gamma)(0, z) &\triangleq z = 0 \\ \mathbf{YP}(\gamma)(1, z) &\triangleq z = 1 * [\mathbf{XE}(\gamma)] \end{aligned}$$

This protocol enforces that y progresses from 0 to 1, and when it is set to 1, the escrow $\mathbf{XE}(\gamma)$ must exist. Thus, before the first thread sets y to 1, it must first transfer the resource $x \hookrightarrow 37$ into the escrow $\mathbf{XE}(\gamma)$ so that it can then pass the *knowledge* of this escrow's existence into the protocol. Once the second thread receives this knowledge from the protocol (by reading y as 1), it can trade in its ghost token for ownership of the resource $x \hookrightarrow 37$, as desired. This reasoning is summarized in the proof outline in Figure 4 (omitting the Tok type in the ghost assertions for brevity).

[5] As we discuss in Section 5, escrows are closely related to Bugliesi *et al.*'s notion of "exponential serialization" [7].

$\{x \hookrightarrow 0 * \boxed{y : 0 \mid \mathbf{YP}(\gamma)}\}$
$[x]_{\mathtt{na}} := 37;$
$\{x \hookrightarrow 37 * \boxed{y : 0 \mid \mathbf{YP}(\gamma)}\}$
$\{[\mathbf{XE}(\gamma)] * \boxed{y : 0 \mid \mathbf{YP}(\gamma)}\}$
$[y]_{\mathtt{at}} := 1;$
$\{\boxed{y : 1 \mid \mathbf{YP}(\gamma)}\}$

$\{\boxed{\gamma : \diamond} * \boxed{y : 0 \mid \mathbf{YP}(\gamma)}\}$
repeat $[y]_{\mathtt{at}}$ end;
$\{\boxed{\gamma : \diamond} * \boxed{y : 1 \mid \mathbf{YP}(\gamma)} * [\mathbf{XE}(\gamma)]\}$
$\{x \hookrightarrow 37\}$
$[x]_{\mathtt{na}}$
$\{z.\ z = 37 * x \hookrightarrow 37\}$

Figure 4. Proof outline for nonatomic message-passing

A more challenging application of escrows Although the above example succinctly illustrates the basic idea of escrows, it is perhaps not the most compelling one, given that it can be handled by other means in prior logics (such as RSL [38]).

We therefore turn now to an interesting synchronization algorithm (suggested to us by Ernie Cohen), whose GPS verification demonstrates an elegant use of escrows and which, to our knowledge, is beyond the reach of prior logics:

$[x]_{\mathtt{at}} :=$ choose$(1, 2)$;	$[y]_{\mathtt{at}} :=$ choose$(1, 2)$;
repeat $[y]_{\mathtt{at}}$ end;	repeat $[x]_{\mathtt{at}}$ end;
if $[x]_{\mathtt{at}} ==$ $[y]_{\mathtt{at}}$ then	if $[x]_{\mathtt{at}}$!= $[y]_{\mathtt{at}}$ then
/* *crit. section* */	/* *crit. section* */

The goal of this algorithm is to guarantee mutual exclusion using release-acquire atomics, but without using CAS. The idea is that each thread sets its respective variable (x or y) to either 1 or 2 (using a nondeterministic choice operator, choose) and then checks the value chosen by the other thread. This enables the threads to synchronize implicitly based on a logical condition: the first thread wins if the values pointed to by x and y are equal, and the second wins if they are not.

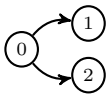
Implicit in the algorithm is the invariant that once each thread sets its variable to 1 or 2, it will not change it further. As a consequence, unlike in Dekker's algorithm (§2), each thread in Cohen's algorithm relies only on *positive* information about the progress of the other thread—*e.g.,* has y been set to *some* nonzero value yet and, if so, what?—in order to determine if it has won the race. Intuitively, it is this restriction to positive reasoning that makes Cohen's algorithm work under release-acquire semantics while Dekker's doesn't.

We now sketch the verification of Cohen's algorithm (full details are given in the appendix [1]). Suppose that the winning thread should gain exclusive access to some shared resource P. To verify Cohen's algorithm, our basic idea is to place P under an escrow $\mathbf{PE}$ at the beginning (prior to the execution of either thread). The transfer condition for this escrow will be defined so as to be satisfiable only by whichever thread wins the race. Thus, once that thread knows it has won, it can unlock the escrow and gain access to P.

Formally, at the beginning of the proof, four tokens will be created and passed to the two threads: the first thread (which sets x) will be given the tokens $\boxed{\gamma_1^x : \diamond}$ and $\boxed{\gamma_2^x : \diamond}$, and the second thread (which sets y) will be given the tokens $\boxed{\gamma_1^y : \diamond}$ and $\boxed{\gamma_2^y : \diamond}$. The γ_1's will be used to guard access to the

aforementioned escrow, and the γ_2's will be used to guard access to transitions in the protocols governing x and y.

Speaking of which: when x and y are initialized to 0, they will be associated with $\mathsf{Choice}(\gamma_2^x)$ and $\mathsf{Choice}(\gamma_2^y)$, respectively, where $\mathsf{Choice}(\gamma)$ is the following protocol with states 0, 1, 2:



$$\mathsf{Choice}(\gamma)(s,z) \triangleq s = z \wedge (s = 0 \vee \boxed{\gamma : \diamond \mid \mathsf{Tok}})$$

This protocol captures not just the irreversible choices made for x and y, but also control over *who* can make these choices: only the owner of the ghost token $\boxed{\gamma_2^x : \diamond}$—*i.e.,* the first thread—will be able to change the state of x and transition from the 0 state of $\mathsf{Choice}(\gamma_2^x)$ to the 1 or 2 states, and similarly only the second thread will be able to change the state of y's protocol.

Finally, we come to the definition of the **PE** escrow, under which P is placed at the beginning of the proof:

$$\mathsf{PE}(\gamma_1^x, \gamma_1^y) : \exists i, j > 0.\ \begin{array}{l} \boxed{x : i \mid \mathsf{Choice}(\gamma_2^x)} \wedge \left(\begin{array}{c} \boxed{\gamma_1^x : \diamond \mid \mathsf{Tok}} \wedge i = j \\ \vee\ \boxed{\gamma_1^y : \diamond \mid \mathsf{Tok}} \wedge i \neq j \end{array} \right) \rightsquigarrow P \\ \boxed{y : j \mid \mathsf{Choice}(\gamma_2^y)} \end{array}$$

The transfer condition on this escrow says that, in order to access P, a thread must know that x and y are both in nonzero states—*i.e.,* that both threads have made their irreversible choices—and that either the states are equal and the thread owns γ_1^x, or they are distinct and the thread owns γ_1^y. In either case, the thread that owns the relevant token is the winning thread. Note that the fact that the transfer condition is exclusive, which is necessary in order for it to be a valid transfer condition, follows easily from the combined use of tokens and protocol assertions. In particular, the proof rule shown at the end of §3.2 dictates that if $\boxed{x : i}$ and $\boxed{x : i'}$ for $i, i' > 0$, then $i = i'$ (and similarly for y, j, j').

3.7 Soundness

The main soundness result for GPS is simple to state:

Theorem 1 (Soundness)**.** If $\{\mathsf{true}\}\, e\, \{x.\ P\}$ is provable then $[\![e]\!] \subseteq \{V \mid [\![P[V/x]]\!] \neq \emptyset\}$.

The theorem says that Hoare triples proved in GPS accurately predict the final result of a closed program, according to the C11 memory model.

But to prove this theorem, we must be able to relate the Hoare triples of the logic to C11's event graphs, which do not provide the global notion of "current state" that the semantics of triples usually depend on.

Overview of the model and proof We structure the semantics of triples into two layers: *local safety* and *global safety*.

Local safety steps through the execution of a single thread, but instead of using an event graph and the C11 axioms to restrict the actions produced by the thread, it essentially replays the rules of GPS. For example, when a thread performs an acquire read, local safety enforces that the location being read is governed by a PL-protocol, and the value read is then constrained by that protocol—exactly mirroring the logic's rule for acquire reads. Thus local safety provides a kind of "rely/guarantee semantics": it checks that an abstract execution of a given thread follows the rules of the logic (*i.e.,* making valid protocol moves) while relying on protocols to predict the outcome of reads (*i.e.,* valid protocols moves made by other threads). When a new thread is forked, local safety is checked independently for the new thread and its parent. Since local safety is just a restatement of GPS's rules in small-step form, it is easy to show that the proof rules of GPS preserve local safety.

Global safety applies to labeled event graphs, where the labels annotate graph edges with resource and knowledge transfers from the point of view of GPS. By imposing appropriate constraints on the labeling, global safety connects the logical assumptions made in local safety with the physical reality of the event graph.

The heart of the soundness argument is then to show that if a whole program is locally safe, it is globally safe. We do this by building up the C11 event graph step-by-step (much like the operational semantics), showing for each new event that (1) the existing labeling implies the rely for the event, and (2) the event's guarantee, which we know by local safety, implies that we can extend the labeling to include it.

Unfortunately, space constraints prevent us from describing the semantic model and proof—both of which are substantial—in full detail here. Below, we sketch a few of the key details. The full semantic model is described in the appendix, and the entire model and soundness proof (as well as an extension of the logic to handle SC accesses) have been formalized and checked in our Coq development [1].

Resources In the semantics of GPS, a *resource* r is a tuple (Π, g, Σ) containing:

- A *physical location map* Π from locations to either a value (for nonatomics) or a protocol and state (for atomics).
- A *ghost identity map* g from ghost names to an element of the corresponding ghost PCM.
- A *known escrow set* Σ containing all escrow types currently in play.

Resources form a PCM with composition $\oplus$, and assertions are interpreted as sets of resources, *e.g.,*

$$r \in [\![P_1 * P_2]\!] \triangleq \exists r_1, r_2.\ r = r_1 \oplus r_2, r_1 \in [\![P_1]\!], r_2 \in [\![P_2]\!]$$

The structure of resources and definition of $\oplus$ are designed to support the axioms on assertions we gave in §3.

Local safety With resources in hand, we can define a semantic version of ghost moves $r \Rrightarrow \mathcal{P}$, which says that from resource r it is possible to take a ghost move to resources described by the (semantic) assertion $\mathcal{P}$. We can also define two functions

$$\mathsf{rely}, \mathsf{guar} : \mathit{Resource} \times \mathit{Action} \to \mathbb{P}(\mathit{Resource})$$

that describe the rely and guarantee constraints on updating resources, given that we are performing some action α. For example, if $\alpha = \mathbb{R}(\ell, V, \mathtt{na})$ and r claims that $\ell \hookrightarrow V'$, then

$$\mathsf{rely}(r, \alpha) = \text{ if } V = V' \text{ then } \{r\} \text{ else } \emptyset$$

which says that the action is only possible if the value it claims to read is the one the logic says the location has. This precisely mirrors the rule for nonatomic reading, and in particular yields no new resources. For atomic locations, the protocol state is allowed to advance, again mirroring the logic's rule for atomic reads.

We can then define local safety:

$r_{\mathrm{pre}} \in \mathsf{LSafe}_0(e, \Phi) \triangleq$ always
$r_{\mathrm{pre}} \in \mathsf{LSafe}_{n+1}(e, \Phi) \approx$ ***(simplified; see appendix)***
  If $e \in \mathit{Val}$ then $r_{\mathrm{pre}} \Rrightarrow \Phi(e)$
  If $e = K[\mathtt{fork}\ e']$ then
    $r_{\mathrm{pre}} \in \mathsf{LSafe}_n(K[0], \Phi) * \mathsf{LSafe}_n(e', \mathsf{true})$
  If $e \xrightarrow{\alpha} e'$ then $\forall r \in \mathsf{rely}(r_{\mathrm{pre}}, \alpha).\ \exists \mathcal{P}.\ r \Rrightarrow \mathcal{P}$ and
    $\forall r' \in \mathcal{P}.\ \exists r_{\mathrm{post}} \in \mathsf{guar}(r', \alpha).\ r_{\mathrm{post}} \in \mathsf{LSafe}_n(e', \Phi)$

which is indexed by the number of steps for which we demand safety. (An expression is "locally safe" if LSafe_n holds for all n.) Local safety can be understood as giving *weakest preconditions*: $\mathsf{LSafe}_n(e, \Phi)$ is the set of starting resources for which e can safely execute for n steps with postcondition Φ (a semantic predicate). We then define

$$\models \{P\}\, e\, \{x.Q\} \triangleq \forall n.\ \forall r \in [\![P]\!].\ r \Rrightarrow \mathsf{LSafe}_n(e, [\![x.\ Q]\!])$$

Theorem 2 (Local soundness)**.** All of the proof rules given in §3 are sound for this semantics of Hoare triples.

Global safety We then define a notion of *global safety*, written $\mathsf{GSafe}_n(\mathcal{T}, G, \mathcal{L})$, over an *instrumented thread pool* $\mathcal{T}$, an event graph G, and a *labeling* $\mathcal{L}$.

The instrumented thread pool $\mathcal{T}$ maps each thread to a tuple (a, e, r, Φ), specifying the last event a that the thread performed in the event graph, the remainder e of its computation, the resources r that it currently holds, and its postcondition Φ. Global safety at n assumes that each thread is locally safe for n more steps, given its resources and postcondition.

The labeling $\mathcal{L}$ annotates hb edges of the graph with *resource transfers* between the nodes, and is constrained to ensure that each node obeys the corresponding guar condition. Since each atomic write to a location ℓ is associated logically with a move in ℓ's protocol, the labeling $\mathcal{L}$ also annotates each write event for ℓ with information about the corresponding state to which ℓ's protocol was updated.

The labeling must then globally ensure the following:

- *Compatibility*: any set of "concurrent" resource transfers (*i.e.,* roughly, those that are not hb-related) must be composable with one another, ensuring that exclusive resources are never duplicated.
- *Conformance*: if $\mathsf{mo}(a, b)$ for two atomic writes/updates to ℓ with protocol τ, the protocol states with which a and b are labeled must be related by $\sqsubseteq_\tau$.

Soundness The key theorem is a kind of simulation between the expression semantics and global safety:

Theorem 3 (Instrumented execution)**.** If $\mathsf{GSafe}_{n+1}(\mathcal{T}, G, \mathcal{L})$ and $\langle \mathsf{erase}(\mathcal{T}); G\rangle \longrightarrow \langle T'; G'\rangle$ then there is some $\mathcal{T}', \mathcal{L}'$ such that $\mathsf{erase}(\mathcal{T}') = T'$ and $\mathsf{GSafe}_n(\mathcal{T}', G', \mathcal{L}')$.

Our main soundness result, given at the beginning of the section, is then a corollary connecting the proof theory all the way to the C11 execution (for closed expressions).

4. Case studies

We have applied GPS to three challenging case studies for weak memory reasoning: *Michael and Scott's lock-free queue* [28], as well as *circular buffers* [19] and *bounded ticket locks* [10] (both adapted from the Linux kernel). Note that the first two of these exhibit non-SC behavior to their clients (cf. §5). For space reasons, we focus here on the proof for circular buffers, which we describe in some detail. For full details of all three examples, see the appendix [1].

Circular buffers Figure 6 shows the code for a simplified variant of the circular buffer data structure drawn from the Linux kernel. It is a fixed-size queue, implemented using an array that "wraps around". Specifically, the queue pointed to by q consists of an N-cell array (at $q + \mathtt{b}$), together with a reader index (at $q + \mathtt{ri}$) specifying the array offset of the next item to be consumed, and a writer index (at $q + \mathtt{wi}$) specifying the array offset of the next item to be produced. The "active" part of the queue consists of the array elements starting at the reader index and ending at the one prior to the writer index, wrapping around modulo N. Hence, if the two indices are equal, then the buffer is empty, and if the writer index is one before the reader index (modulo N), then the buffer is full (with $N - 1$ elements).

The `tryProd` and `tryCons` operations first check the two indices to see whether the buffer is full or empty, respectively. If so, they return 0. Otherwise, they proceed by writing/reading the element at the writer/reader index and then incrementing that index (modulo N). Since accesses to the actual data in the buffer are completely synchronized, the cells comprising the array itself can be read and written nonatomically. All synchronization is performed through the reader/writer indices. Note, however, that (as in Cohen's example from §3.6) this synchronization is entirely *implicit*: the algorithm uses plain writes, not `CAS`, to increment the indices. While this is an efficiency win (*e.g.,* on x86, the algorithm requires no fences), it means that only one producer and one consumer can operate simultaneously.

The specification We will prove the following spec:

$$\begin{array}{c}
\{\mathsf{true}\}\ \mathtt{newBuffer}()\ \{q.\ \mathsf{Prod}(q) * \mathsf{Cons}(q)\} \\
\{\mathsf{Prod}(q) * P(x)\}\,\mathtt{tryProd}(q, x)\,\{z.\ \mathsf{Prod}(q) * (z \neq 0 \vee P(x))\} \\
\{\mathsf{Cons}(q)\}\ \ \mathtt{tryCons}(q)\ \ \{x.\ \mathsf{Cons}(q) * (x = 0 \vee P(x))\}
\end{array}$$

The spec is parameterized over a predicate P that should hold of all the elements in the buffer; it guarantees that $P(x)$

$$\mathsf{all} \triangleq (\mathbb{N},\mathbb{N},\mathbb{N},\mathbb{N})$$

$$\mathsf{restP}(i) \triangleq ((>i),(\geq i),\emptyset,\emptyset)$$
$$\mathsf{restC}(i) \triangleq (\emptyset,\emptyset,(>i),(\geq i))$$

$$\mathsf{protP}(i) \triangleq (\{i\},\emptyset,\emptyset,\emptyset)$$
$$\mathsf{escP}(i) \triangleq (\emptyset,\{i\},\emptyset,\emptyset)$$
$$\mathsf{protC}(i) \triangleq (\emptyset,\emptyset,\{i\},\emptyset)$$
$$\mathsf{escC}(i) \triangleq (\emptyset,\emptyset,\emptyset,\{i\})$$

$$\mathsf{Prod}(q) \triangleq \exists \gamma, i, j.\ i < j + N * \boxed{q + \texttt{wi} : i \mid \mathbf{PP}(\gamma, q)} * \boxed{q + \texttt{ri} : j \mid \mathbf{CP}(\gamma, q)} * \boxed{\gamma : \mathsf{restP}(i)}$$
$$\mathsf{Cons}(q) \triangleq \exists \gamma, i, j.\ j \leq i \quad * \boxed{q + \texttt{wi} : i \mid \mathbf{PP}(\gamma, q)} * \boxed{q + \texttt{ri} : j \mid \mathbf{CP}(\gamma, q)} * \boxed{\gamma : \mathsf{restC}(j)}$$

$$\mathbf{PP}(\gamma, q)(i, x) \triangleq \boxed{\gamma : \mathsf{protP}(i)} \wedge \Box x = i \bmod N \wedge \Box \forall j < i. \qquad [\mathbf{CE}(\gamma, q, j)]$$
$$\mathbf{CP}(\gamma, q)(j, x) \triangleq \boxed{\gamma : \mathsf{protC}(j)} \wedge \Box x = j \bmod N \wedge \Box \forall i < j + N. \ [\mathbf{PE}(\gamma, q, i)]$$

$$\mathbf{PE}(\gamma, q, i) : \boxed{\gamma : \mathsf{escP}(i)} \rightsquigarrow \mathsf{uninit}(q + \mathtt{b} + (i \bmod N)) \vee (q + \mathtt{b} + (i \bmod N)) \hookrightarrow -$$
$$\mathbf{CE}(\gamma, q, j) : \boxed{\gamma : \mathsf{escC}(j)} \rightsquigarrow \exists x.\ P(x) * (q + \mathtt{b} + (j \bmod N)) \hookrightarrow x$$

Figure 5. Technical setup for the circular buffer case study

holds of all elements x that the consumer consumes so long as it holds of all elements x that the producer produces. This predicate can thus be used in typical separation-logic style to transfer ownership of data structures from producer to consumer.[6] The spec also employs two predicates $\mathsf{Prod}(q)$ and $\mathsf{Cons}(q)$, which describe the privilege of acting as producer or consumer, respectively. These predicates are exclusive resources, ensuring that there can only be one call to `tryProd` and one call to `tryCons` running concurrently. Their definitions (in Figure 5) are described below.

Note that this spec is rather weak because it does not enforce that the buffer actually implements a queue. This is merely for simplicity—it is easy to generalize our proof to handle a stronger spec, *e.g.,* in which P, Prod, and Cons are allowed to keep track of the entire sequence of elements produced thus far.

High-level picture Our proof of the above spec (Figure 6) depends on all the features of GPS working in concert. Figure 5 shows the technical setup for the proof.

First, we use *protocols* **PP** and **CP** to govern the states of the writer and reader indices, respectively. The state of each of these protocols tracks the "absolute state" of the corresponding index, meaning the total number of writes/reads that have ever occurred, which can only increase over time (the state ordering is $\leq$). The state interpretation of **PP**/**CP** then dictates that the "physical state" of the writer/reader index equal the absolute state modulo N.

Second, since the buffer does not use `CAS`, it is not possible to use the **PP** and **CP** protocols to *directly* transfer ownership of the cells in the buffer between the producer and consumer. Fortunately, we can *indirectly* exchange ownership of the buffer cells instead, by (a) placing the cells under *escrows*, and (b) using **PP** and **CP** as a conduit for the *knowledge* that these escrows, once created, exist. Specifically, after filling a buffer cell with a new element, the producer will pass control of the cell to the consumer via the **CE** escrow (see Step 10 in the proof of `tryProd`); upon consumption, the consumer will pass control of the cell back to the producer via the **PE** escrow. The state interpretations of **PP** and **CP** offer a way to communicate awareness of these escrows back and forth.

[6] In the case that the buffer is full, *i.e.,* return value $z = 0$, the `tryProd` operation simply returns ownership of $P(x)$ to the caller.

Third, we use *ghost tokens* in a manner similar to the proof of Cohen's example from the previous section. The $\mathsf{protP}(i)$ and $\mathsf{protC}(i)$ tokens are needed in order to transition to (absolute) state i of the **PP** and **CP** protocols, respectively, while the escP and escC tokens are used as transfer conditions for the **PE** and **CE** escrows. In both cases, the producer and consumer each start out with all the tokens they will ever need (*i.e.,* $\mathsf{restP}(0)$ and $\mathsf{restC}(0)$) as part of their exclusive resource predicates $\mathsf{Prod}(q)$ and $\mathsf{Cons}(q)$, and they proceed to "spend" one protocol token and one escrow token upon each call to `tryProd`/`tryCons`. All these tokens are defined in Figure 5 as elements of the ghost PCM $\mathbb{P}(\mathbb{N}) \times \mathbb{P}(\mathbb{N}) \times \mathbb{P}(\mathbb{N}) \times \mathbb{P}(\mathbb{N})$ (with composition defined as componentwise $\uplus$).

Finally, tying everything together, $\mathsf{Prod}(q)$ and $\mathsf{Cons}(q)$ assert *bounded knowledge* about the states of the **PP** and **CP** protocols, thus enforcing the *fundamental invariant of circular buffers*:

> The absolute writer index is **at least** 0 and **less than** N cells ahead of the absolute reader index.

Now, the reader (of this paper, not the buffer) may rightly wonder: how can this fundamental invariant possibly be enforced in the weak memory setting, given that it concerns the states of two separate cells being updated by different threads? The answer is that, although neither the producer nor the consumer can fully assume or maintain this invariant themselves, they are each able to enforce a piece of it sufficient to verify their own correctness. In particular, the consumer controls the progress of the reader index, and can therefore assume and maintain the invariant that the reader index never overtakes the writer index (the "at least 0" part), while the producer controls the progress of the writer index, and can therefore assume and maintain the invariant that the writer index never leaves the reader index more than $N - 1$ cells behind (the "less than N" part). Together, these piecemeal enforcements of the fundamental invariant are enough to perform the full verification.

Proof outline for `tryProd` Figure 6 displays the proof outline for `tryProd`(q, x). (The proof for `tryCons` is almost dual, and the proof for `newBuffer` is comparatively simple; see the appendix.) We explain here some of the most important steps in the proof. Throughout, note that assertions under

$\texttt{wi} \triangleq 0, \texttt{ri} \triangleq 1, \texttt{b} \triangleq 2$

$\underline{\texttt{newBuffer}()}$

$\texttt{let } q = \texttt{alloc}(N{+}2)$
$[q + \texttt{ri}]_{\texttt{at}} := 0;$
$[q + \texttt{wi}]_{\texttt{at}} := 0;$
q

$\underline{\texttt{tryProd}(q, x)}$

$\texttt{let } w = [q + \texttt{wi}]_{\texttt{at}}$
$\texttt{let } r = [q + \texttt{ri}]_{\texttt{at}}$
$\texttt{let } w' = w + 1 \bmod N$
$\texttt{if } w' == r \texttt{ then}$
  0
$\texttt{else}$
  $[q + \texttt{b} + w]_{\texttt{na}} := x;$
  $[q + \texttt{wi}]_{\texttt{at}} := w';$
  1

$\underline{\texttt{tryCons}(q)}$

$\texttt{let } w = [q + \texttt{wi}]_{\texttt{at}}$
$\texttt{let } r = [q + \texttt{ri}]_{\texttt{at}}$
$\texttt{let } r' = r + 1 \bmod N$
$\texttt{if } w == r \texttt{ then}$
  0
$\texttt{else}$
  $\texttt{let } x = [q + \texttt{b} + r]_{\texttt{na}}$
  $[q + \texttt{ri}]_{\texttt{at}} := r';$
  x

Proof outline for $\texttt{tryProd}(q, x)$:

$\{\mathsf{Prod}(q) * P(x)\}$
(1) $\left\{\boxed{\gamma : \mathsf{restP}(i)} * P(x) * \Box\left(i < j_0 + N \wedge \boxed{q + \texttt{wi} : i \mid \mathbf{PP}(\gamma, q)} \wedge \boxed{q + \texttt{ri} : j_0 \mid \mathbf{CP}(\gamma, q)}\right)\right\}$
$\texttt{let } w = [q + \texttt{wi}]_{\texttt{at}}$
(2) $\left\{\boxed{\gamma : \mathsf{restP}(i)} * P(x) * \Box(w = i \bmod N \wedge \forall k < i.\ [\mathbf{CE}(\gamma, q, k)])\right\}$
$\texttt{let } r = [q + \texttt{ri}]_{\texttt{at}}$
(3) $\left\{\boxed{\gamma : \mathsf{restP}(i)} * P(x) * \Box\left(\begin{array}{l} r = j \bmod N \wedge \boxed{q + \texttt{ri} : j \mid \mathbf{CP}(\gamma, q)} \\ \quad \wedge\ j_0 \le j \wedge \forall k < j + N.\ [\mathbf{PE}(\gamma, q, k)] \end{array}\right)\right\}$
(4) $\left\{\boxed{\gamma : \mathsf{restP}(i)} * P(x) * \Box(i < j + N \wedge [\mathbf{PE}(\gamma, q, i)])\right\}$
$\texttt{let } w' = w + 1 \bmod N$
$\left\{\boxed{\gamma : \mathsf{restP}(i)} * P(x) * \Box(w' = w + 1 \bmod N)\right\}$
(5) $\texttt{if } w' == r \texttt{ then } \left\{\boxed{\gamma : \mathsf{restP}(i)} * P(x)\right\}\ 0\ \{z.\ \mathsf{Prod}(q) * z = 0 * P(x)\}$
$\texttt{else } \left\{\boxed{\gamma : \mathsf{restP}(i)} * P(x) * \Box(w' \neq r)\right\}$
(6) $\left\{\boxed{\gamma : \mathsf{restP}(i)} * P(x) * \Box(i + 1 < j + N)\right\}$
(7) $\left\{\boxed{\gamma : \mathsf{restP}(i+1)} * \boxed{\gamma : \mathsf{protP}(i+1)} * P(x) * \boxed{\gamma : \mathsf{escP}(i)}\right\}$
(8) $\left\{\boxed{\gamma : \mathsf{restP}(i+1)} * \boxed{\gamma : \mathsf{protP}(i+1)} * P(x) * (\mathsf{uninit}(q + \texttt{b} + w) \vee (q + \texttt{b} + w) \hookrightarrow -)\right\}$
$[q + \texttt{b} + w]_{\texttt{na}} := x;$
(9) $\left\{\boxed{\gamma : \mathsf{restP}(i+1)} * \boxed{\gamma : \mathsf{protP}(i+1)} * P(x) * (q + \texttt{b} + w) \hookrightarrow x\right\}$
(10) $\left\{\boxed{\gamma : \mathsf{restP}(i+1)} * \boxed{\gamma : \mathsf{protP}(i+1)} * [\mathbf{CE}(\gamma, q, i)]\right\}$
$[q + \texttt{wi}]_{\texttt{at}} := w';$
(11) $\left\{\boxed{\gamma : \mathsf{restP}(i+1)} * \boxed{q + \texttt{wi} : i + 1 \mid \mathbf{PP}(\gamma, q)}\right\}$
1
(12) $\{z.\ \mathsf{Prod}(q) * z = 1\}$

Figure 6. Proof excerpt for the circular buffer case study

$\Box$ are only written once and then used freely in the rest of the proof since they hold true forever after.

Step 1: By unfolding $\mathsf{Prod}(q)$, we gain access to our piece of the fundamental invariant, namely that the absolute writer index i is less than N past the absolute reader index, which is at least j_0.

Step 2: The reason we know *exactly* what i is—but merely have a lower bound on j_0—is that we own the protocol tokens $\mathsf{protP}(k)$ for all $k > i$, constraining the possible "rely" moves that other threads can make in the **PP** protocol. In this step, we exploit that knowledge to assert that the value w we read is exactly $i \bmod N$.

Step 3: Here we read the current reader index r, whose absolute state j must be at least j_0 (as mentioned already). From the read of protocol **CP** at state j, we also gain knowledge of the escrows $\mathbf{PE}(\gamma, q, k)$ for all $k < j + N$.

Step 4: Since $i < j_0 + N \le j + N$, the escrows we just learned about in the previous step include $\mathbf{PE}(\gamma, q, i)$, which we need later.

Step 5: If the buffer is full, *i.e.,* $r = (w + 1) \bmod N$, then the operation is a no-op and we simply return $P(x)$ back to the caller.

Step 6: Otherwise, $r \neq (w + 1) \bmod N$. We know from Step 4 that $i < j + N$, and we want to show $i + 1 < j + N$ because this is the piece of the fundamental invariant that we are responsible for maintaining when we bump up the writer index at the end of the operation (Step 12). To prove this, we must establish $i + 1 \neq j + N$. So suppose the opposite is true: $i + 1 = j + N$. Then, since $w = i \bmod N$, we obtain $(w + 1) \bmod N = (i + 1) \bmod N = (j + N) \bmod N = j \bmod N = r$. Contradiction.

Step 7: From our stash of tokens ($\mathsf{restP}(i)$), we peel off a protocol token ($\mathsf{protP}(i + 1)$) for advancing to the $(i + 1)$-th state of the **PP** protocol, and an escrow token ($\mathsf{escP}(i)$) for accessing the escrow $\mathbf{PE}(\gamma, q, i)$ that we learned about in Step 4.

Step 8: We access the escrow, thereby gaining ownership of the buffer cell at index w.

Step 9: We non-atomically write x to the buffer cell.

Step 10: We pass control of the buffer cell back to the consumer by placing it under the consumer escrow $\mathbf{CE}(\gamma, q, i)$.

Step 11: We advance the absolute writer index (*i.e.,* the state of the **PP** protocol) to $i+1$, which we can do because (a) we own the token $\mathsf{protP}(i + 1)$, and (b) we have knowledge of $\mathbf{CE}(\gamma, q, i)$.

Step 12: Thanks to Step 6, we have preserved the "less than N" part of the fundamental invariant, as demanded by $\mathsf{Prod}(q)$.

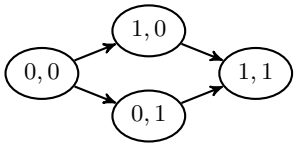
5. Related work

5.1 From SC reasoning to weak memory reasoning

As explained in the introduction, the various logical mechanisms employed by GPS are not fundamentally new: they are all either descendants or restrictions of mechanisms proposed in prior logics for strong (SC) concurrency.

First and foremost, many recent SC logics support some form of *protocol* for describing fine-grained invariants on shared state; GPS's *per-location (PL) protocols* are inspired most directly by the protocols of CaReSL [37]. CaReSL's protocols take the form of state transition systems (STSs) wherein each STS state is associated with an invariant about some underlying shared state. The primary difference between GPS's protocols and CaReSL's protocols is that CaReSL's protocols are not restricted to governing the contents of a single location: they may govern arbitrary heap regions, and this additional flexibility renders them suitable for verifying programs that assume sequential consistency.

For instance, the CaReSL protocol for verifying Dekker's algorithm (§2) would look something like this:



Here, each protocol state governs the contents of x and y simultaneously. In the $(1, 0)$ state the first thread has won the race; in the $(0, 1)$ state the second thread has won the race; and in the $(1, 1)$ state the race is over (and it is impossible to tell who won). The verification of Dekker's algorithm just has to ensure that (a) each thread only makes state changes according to the protocol, which is easy since updating x or y from 0 to 1 is always legal according to the protocol, and (b) each thread only accesses the shared resource once it has observed the protocol being in its respective winning state.

In the weak memory setting, the kind of simultaneous invariant represented by the above protocol, relating the "current" states of x and y, is unsound because the updates to x and y may appear in different orders to the first and second threads. It is a key original insight in the design of GPS that the soundness of CaReSL-style protocols for weak memory can in fact be regained if we simply restrict them to governing a single location at a time.

GPS's support for ghost state is also inspired by CaReSL, but the mechanisms are somewhat different. CaReSL supports ghost state through "tokens", which are coupled with its protocol mechanism, whereas in GPS ghost state is handled separately via *ghost PCMs* [14, 22, 25]. (In this paper, we have only made use of simple token-like ghost PCMs, but the "bounded ticket lock" example, shown in the appendix [1], employs a much more interesting PCM.) GPS's separation of orthogonal mechanisms has the side benefit of removing CaReSL's "token purity" restriction—*e.g.,* in the circular buffer example from Section 4, we did not require any side condition on the per-item predicate $P(x)$, whereas an analogous proof in CaReSL would have required that $P(x)$ be a "token-pure" (*i.e.,* duplicable) assertion.

GPS's *escrows*, $P \rightsquigarrow Q$, can be viewed as yet another kind of CaReSL-style protocol, restricted in a different way than PL-protocols are. Escrows are essentially protocols with two states: *before* and *after* the resource being held in escrow, Q, has been exchanged for the escrow condition, P. Escrows are sound in the weak memory setting because the only thread that can observe anything at all about the protocol is the thread that exchanges P for Q. Since that thread owns P, and P is exclusive, it can deduce that the escrow is in the *before* state, and therefore safely transition to the *after* state, without any concern about the observations of other threads.

Although in the context of concurrency logics the escrow mechanism is unusual, there is some precedent for it: escrows are very similar to "exponential serialization", a mechanism proposed by Bugliesi *et al.* [7] as part of an affine type system for verifying cryptographic protocols. Bugliesi *et al.* employ this mechanism for much the same reasons we do—namely, as a way of indirectly transferring control of an exclusive resource from one thread to another across a duplicable, "knowledge-only" channel. However, in their case the channel takes the form of a cryptographic signing key, whereas for us it is a shared memory location. Logically, the main difference between escrows and exponential serialization is that the precondition of escrow creation—*i.e.,* that the escrow transfer condition P is exclusive ($P * P \Rightarrow \mathsf{false}$)—is something we can prove easily within the logic of GPS. In contrast, since the primitive affine predicates of Bugliesi *et al.*'s type system have no underlying semantic interpretation, they can only ensure the analogous exclusiveness condition via a complex and syntactic "guardedness" check on typing contexts.

5.2 Relaxed Separation Logic (RSL)

The closest related work to GPS is the recent *Relaxed Separation Logic* (RSL) introduced by Vafeiadis and Narayan [38], which is the only prior program logic for the C11 memory model. The goal of RSL is to support simple reasoning about release-acquire accesses in the style of Concurrent Separation Logic (CSL) [30]. Unlike in GPS, it is possible in RSL for a release write to *directly* transfer resource ownership to an acquire read (*e.g.,* in verifying the nonatomic message-passing example, for which GPS required escrows). To manage such transfers, RSL employs release/acquire *permissions* describing the resources to be transferred upon a write to a given location. The choice of resources depends solely on the *value* being written, and so any given value can only be used to perform a transfer *once* per location.

GPS draws much inspiration from RSL, particularly in its proof of soundness, whose structure is based closely on RSL's. There are many significant differences, however. Most importantly, GPS offers a much more flexible way of coordinating ownership and knowledge transfers between threads—including rely-guarantee reasoning—through its

protocols, ghosts, and escrows. These mechanisms refactor and generalize the permission-based reasoning of RSL, thus allowing us to lift several of RSL's restrictions, including the one on repeated writes of the same value. Lifting this restriction is crucial for handling the indices in the circular buffer, and the ticket numbers in the bounded ticket lock, as these are cases where the same value is "recycled" (*i.e.,* written to a location multiple times, and each time used to perform a different resource transfer). To our knowledge, none of our case studies can be verified in RSL.

5.3 Alternative approaches

Most existing approaches to reasoning about weak memory rely in some way on recovering strong memory assumptions, either by imposing a synchronization discipline or by reasoning directly about low-level hardware details.

Recovering SC by synchronization discipline Most memory models satisfy the so-called *fundamental property* [34]: they guarantee sequential consistency for "sufficiently synchronized" code. (Synchronization operations like memory fences effectively thwart compiler and CPU optimizations.) Thus, if one can use a concurrency logic or some other means to enforce a strong synchronization discipline, one can recover strong memory reasoning for programs that follow that discipline. Instances of this approach include:

- Owens [31] proves that data-race free and "triangular-race" free programs on x86-TSO have SC behavior.
- Batty et al. [4] prove that for C11 restricted to nonatomics and SC-atomics, data-race freedom ensures SC behavior.
- Cohen and Schirmer [8] prove that programs following a certain ownership discipline and flushing write buffers at certain times on TSO models have SC behavior.
- Ferreira et al. [17] prove that concurrent separation logic is sound for a class of weak memory models satisfying a data-race freedom guarantee.

All of these disciplines force programs to use enough synchronization to keep weak memory behavior unobservable. We view them as complementary to GPS: they delimit an important subset of programs for which SC reasoning is sound within a weak memory model. Ultimately, our goal is to derive such disciplines *within* a more general weak memory program logic like GPS. Our treatment of locks in §3 already does this for the simple case of recovering CSL-style reasoning within weak memory: our lock spec provides the key concurrency rules for CSL as a *derived* set of rules in GPS.

We believe the extra generality of GPS is important because it enables us to verify a wider class of weak memory programs, including those whose observable behavior is *not* SC. The circular buffer and Michael-Scott queue are good examples of this (see the appendix [1]). Singh *et al.* [35] argue that one should not expose the high-level programmer to such non-SC data structures, but GPS shows that in fact it is possible to reason sensibly and modularly about them.

Recovering SC through low-level reasoning Another way of recovering strong memory is to explicitly model low-level hardware details (*e.g.,* per-processor write buffers) within one's logic [33, 40], or to transform the program being verified so that interactions with write buffers, for instance, are made manifest in its code [2]. While this type of approach can accommodate arbitrary programs and enable the reuse of existing SC techniques, it provides little abstraction or modularity: users of such an approach must reason directly with the low-level hardware details, with relatively little help given in structuring this reasoning.

Ridge [33] provides a program logic for x86-TSO that supports rely-guarantee reasoning. The logic works directly with the operational x86-TSO model [32], and includes assertions about both program counters and write buffers. Rely constraints must be stable under the (nondeterministic) flushing of write buffers.

Wehrman and Berdine [40] propose a separation logic for x86-TSO which directly models store buffers and provides both temporal and spatial separating conjunctions, as well as resource invariants in the style of CSL. Unfortunately, the logic as proposed has some (known) soundness gaps, and to our knowledge a sound version has not yet been developed.

6. Future work

While GPS makes a significant step forward in reasoning about release/acquire semantics, there is much work left to do to develop a full understanding of the C11 memory model.

Interaction with SC In our Coq development, we show that the reasoning principles for release-acquire atomics apply to SC atomics as well. In addition, we believe that if each memory location were uniquely used in conjunction with one access mode (*e.g.,* always release-acquire or always SC), then it would be straightforward to supplement GPS with completely separate (and stronger) reasoning principles for SC atomics, along the lines of prior SC logics. However, the C11 model allows programmers to freely mix memory orderings, and ideally program logics should support such mixed reasoning as well. Early investigation suggests that the C11 model has some corner cases when mixing memory orderings that may obstruct compositional reasoning principles.

Consume reads The C11 memory model supports a weaker mode for reads, called *consume* reads, under which happens-before relationships are only introduced for subsequent actions that depend on the value that was read. Such consume reads are used crucially, for example, in the implementation of read-copy-update (RCU) synchronization in the Linux kernel [27]. We believe it should be possible to extend GPS with support for consume reads, and that reasoning about compositionally about them will likely require the introduction of a modality encapsulating possible data dependencies.

Relaxed operations Finally, C11 offers fully relaxed memory orderings, which induce no happens-before relationships.

If both relaxed reads and writes are allowed, the formal C11 model permits causal cycles: an execution can produce a value "out of thin air" through a cycle of relaxed read and write operations [5]. As noted in the RSL paper [38], these cycles inhibit even very basic forms of logical reasoning, including single-location invariants, and they also inhibit program analyses that are routinely used for optimization. (RSL includes rules for reasoning about relaxed accesses, but only under a severely restricted version of the C11 memory model.) We therefore believe that C11 should be revised to rule out these and other causal cycles, which will enable us to find sound reasoning principles for relaxed operations.

Acknowledgments

This work is partially supported by the EC FP7 FET project ADVENT. We would also like to thank Xiao Jia, Ralf Jung, and Joe Tassarotti for helpful comments and corrections.

References

[1] Appendix and Coq development for this paper available at the following URL: http://plv.mpi-sws.org/gps/.

[2] J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig. Software verification for weak memory via program transformation. In *ESOP*, 2013.

[3] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL*, 2011.

[4] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: From C++11 to POWER. In *POPL*, 2012.

[5] M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency. In *POPL*, 2013.

[6] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *PLDI*, 2008.

[7] M. Bugliesi, S. Calzavara, F. Eigner, and M. Maffei. Logical foundations of secure resource management in protocol implementations. In *POST*, 2013.

[8] E. Cohen and B. Schirmer. From total store order to sequential consistency: A practical reduction theorem. In *ITP*, 2010.

[9] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs*, 2009.

[10] J. Corbet. Ticket spinlocks, 2008. http://lwn.net/Articles/267968/.

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

[12] E. W. Dijkstra. EWD123: Cooperating Sequential Processes. Technical report, 1965.

[13] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP 2010*, volume 6183 of *LNCS*, pages 504–528. Springer, 2010.

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

[15] M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. In *ESOP*, 2009.

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

[17] R. Ferreira, X. Feng, and Z. Shao. Parameterized memory models and concurrent separation logic. In *ESOP*, 2010.

[18] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI*, 1993.

[19] D. Howells and P. E. McKenney. Circular buffers. https://www.kernel.org/doc/Documentation/circular-buffers.txt.

[20] ISO/IEC 14882:2011. Programming language C++, 2011.

[21] ISO/IEC 9899:2011. Programming language C, 2011.

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

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

[24] K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In *Foundations of Security Analysis and Design V*, volume 5705 of *LNCS*. 2009.

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

[26] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL*, 2005.

[27] P. McKenney. *Exploiting deferred destruction: an analysis of read-copy-update techniques in operating system kernels*. PhD thesis, Oregon Graduate Institute, 2004.

[28] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *JPDC*, 51(1):1–26, 1998.

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

[30] P. O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1):271–307, 2007.

[31] S. Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *ECOOP*, 2010.

[32] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *TPHOLs*, 2009.

[33] T. Ridge. A rely-guarantee proof system for x86-TSO. In *VSTTE*, 2010.

[34] V. A. Saraswat, R. Jagadeesan, M. Michael, and C. von Praun. A theory of memory models. In *PPoPP*, 2007.

[35] A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi. End-to-end sequential consistency. In *ISCA*, 2012.

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

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

[38] V. Vafeiadis and C. Narayan. Relaxed separation logic: A program logic for C11 concurrency. In *OOPSLA*, 2013.

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

[40] I. Wehrman and J. Berdine. A proposal for weak-memory local reasoning. In *LOLA*, 2011.