Technical Report MPI-SWS-2014-004
February 2014

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 unexpected behavior that one can expect to observe in multi-threaded 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, rely-guarantee "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 its soundness directly in terms of the axiomatic C11 weak memory model.

Contents

1 Introduction **2**

2 The C11 memory model **2**

3 GPS: a logic for release-acquire consistency **4**

4 Case studies **8**

5 The semantics and soundness of GPS **9**

6 Related work **10**

A Language **13**
- A.1 Syntax . . . 13
- A.2 Semantics . . . 13
- A.3 Memory model . . . 13

B Logic **16**
- B.1 Semantic structures . . . 16
- B.2 Local safety . . . 16
- B.3 Global safety . . . 18
- B.4 Syntax and semantics . . . 19
- B.5 Proof theory . . . 19

C Metatheory **21**
- C.1 Basic properties of semantics domains and ghost moves . . . 21
- C.2 Proof rules: local soundness . . . 22
- C.3 Global soundness . . . 22

D Examples **29**
- D.1 One-shot message passing . . . 29
- D.2 Spinlocks . . . 31
- D.3 Ernie Cohen's lock example . . . 33
- D.4 Michael-Scott queue . . . 34
- D.5 Circular buffer . . . 39
- D.6 Bounded ticket locks . . . 44

1. Introduction

There are many good reasons to run code out of order. CPUs maximize productivity by scheduling instructions according to the availability of data. Write buffers mask memory latency by updating caches asynchronously. Even mundane compiler optimizations like common subexpression elimination change the ordering of reads.

These and other critical optimizations are expected, of course, to respect the semantics—of *sequential* code. For concurrent code, reorderings have a visible effect: they destroy the illusion of a single memory shared between all threads, allowing different threads to observe writes in different orders. Since the optimizations are considered too important to give up, architectures and languages instead codify their effect abstractly, in terms of *weak memory models* that specify which observations are possible and which aren't [2].

Weak memory complicates the already-difficult task of reasoning about correctness of concurrent code. Worse, it renders impotent our most effective weaponry: the sophisticated formal methods that have been developed to help tame concurrency, which almost universally assume a strong (*i.e.,* sequentially consistent) memory model. Even basic techniques like history invariants and auxiliary state seem to rely on threads witnessing events in the same order.

So we are left with a pressing question: is there any way to retain the advances in modern concurrency verification when our fundamental assumptions about memory have to change?

Recovering strong memory One answer that has emerged in recent years is to try somehow to "recover" the assumption of strong memory. Most memory models satisfy the so-called *fundamental property* [27]: they guarantee sequential consistency for "sufficiently synchronized" code. (Synchronization operations like memory fences effectively thwart compiler and CPU optimizations.) Thus, if a concurrency logic enforces a strong synchronization discipline, it can support strong memory reasoning [7, 10, 14, 24]. The downside is that the logic can only be used to verify programs that follow the discipline, and, of course, can only verify program modules whose behavior is sequentially consistent. That rules out some of the more subtle (and thus important to verify) algorithms used in practice, including several of the case studies in §4.

Another way of recovering strong memory is to explicitly model low-level hardware details (*e.g.,* per-processor write buffers) within one's logic [26, 30], or to transform the program being verified so that interactions with write buffers, for instance, are made manifest in its code [4]. 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.

Navigating weak memory Here we take a different approach: rather than trying to recover strong memory, we embrace weak memory as it is.[1] Our goal is to develop a program logic with high-level, structured reasoning principles for weak memory.

An important first step toward this goal is the recent work of Vafeiadis and Narayan on Relaxed Separation Logic (RSL) [29], the first logic for the C11 memory model [18, 19]. RSL supports simple, high-level reasoning about resource invariants and ownership transfer à la the concurrent separation logic (CSL) [23]. But it does not support some other useful features of modern concurrency logics, such as "ghost" (auxiliary) state and rely-guarantee reasoning.

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

[1] A similar perspective has recently been advocated in the context of model checking ("*Weakness is a virtue*" [3]), albeit with a rather different motivation; here, we investigate its consequences for program logics.

A major obstacle to developing such a logic is the global nature of weak memory models. For example, most language-level models are given in terms of *event graphs* whose nodes include every read and write performed in a program execution. To determine the values that might be returned by a given read operation, one may have to in principle consider every write event in the graph, using the model's axioms to deduce whether the write is visible to the read or not. Global axioms are well-suited to giving a precise language semantics, but they do not easily yield thread-local reasoning.

Our key technique for coping with global graphs is to force reasoning to be *even more local* than in logics for strong memory. Because compilers and CPUs must respect the semantics of sequential code, they generally do not reorder writes to the *same* location. Weak memory models therefore guarantee *coherence*: writes to any single location will appear in the same order to all threads (the so-called *modification order*). Thus, we can recover the full toolkit of concurrency reasoning *if* we restrict it to a single location at a time.

GPS encodes such location-local reasoning through *per-location protocols* (PL-protocols), which govern the writes to a single location according to an abstract state transition system (§3). Although seemingly focused on single locations, PL-protocols are in fact the key to *cross-location reasoning*: by discovering that one location ℓ is in a state s in its protocol, one can learn that another location ℓ' is *at least* in some state s' in its protocol. This kind of "transitive visibility" is at the heart of weak memory models, and PL-protocols provide a structured, thread-local way to reason about it.

PL-protocols are modeled after similar mechanisms in recent SC concurrency logics [28], and as such, support a flexible combination of rely-guarantee (by abstractly characterizing interference between threads), ghost state (by tracking the history of writes to a location), and CSL-style resource invariants. GPS also offers two other facilities for ghost instrumentation—*ghosts* and *escrows*—which we explain in §3. The need for multiple mechanisms stems from the fact that ghost state is not sound in general under a weak memory semantics because it induces additional synchronization to the program, which at the extreme can be used to regain SC. Adapting ghost state to weak memory thus required us to isolate several different usage patterns that do not induce additional synchronization and do remain sound under weak memory assumptions.

GPS targets the recent C11 [18, 19] memory model, which offers portable but fine-grained control over memory consistency guarantees. To keep the presentation focused, we consider the two most important consistency modes—nonatomic and release-acquire (see §2). The logic is, however, sound under the full axioms of C11. We have defined its semantics and proven its soundness directly in terms of the axioms of C11, as summarized in §5. The supplementary materials [1] provide many further details, as well as a Coq mechanization of the soundness proof of GPS.

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. We conclude in §6 with related 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 leads naturally to an interleaving semantics of concurrency, where threads interact 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 reflecting the possible reorderings performed by the CPU or compiler.

The C11 memory model [18, 19] 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. The lack of data races effectively prevents the program from observing reorderings, so nonatomic accesses can enjoy the full suite of compiler/CPU optimizations and still appear SC.

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.

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]:

$[x]_{\texttt{at}} := 1$ $\texttt{if } [y]_{\texttt{at}} == 0 \texttt{ then}$ /* *crit. section* */	$[y]_{\texttt{at}} := 1$ $\texttt{if } [x]_{\texttt{at}} == 0 \texttt{ then}$ /* *crit. section* */

We presume at the outset that x and y are pointers to distinct locations, both with initial value 0.[2] 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 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 data structure from one thread to another (here represented as a single value, 37):

$[x]_{\texttt{na}} := 37;$ $[y]_{\texttt{at}} := 1;$	$\texttt{repeat } [y]_{\texttt{at}} \texttt{ end};$ $[x]_{\texttt{na}}$

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 does not guarantee that threads see the globally-latest write to a location, but it 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 can only read from the latest.

A final point about the message-passing example: 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. Thus the code is data-race free, despite its nonatomic accesses to x.

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

Event graphs We now present the C11 model formally, following Batty et al. [6] and subsequent simplifications [7, 29]. Our presentation makes several further simplifications due to our focus on release-acquire atomics, *but GPS is also sound for the full-blown C11 axioms*. The appendix includes more details [1].

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 one of the following *actions*:

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

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 [29]).

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 is a bound on 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:[3]

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

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

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

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

Syntax

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

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

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

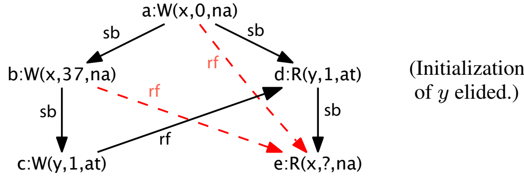
$$
\begin{array}{c}
\langle T \uplus [i \mapsto (a, K[\texttt{fork } e])]; G\rangle \longrightarrow \\
\langle T \uplus [i \mapsto (a, K[0])] \uplus [j \mapsto (a, e)]; G\rangle
\end{array}
$$

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

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

$$
\mathsf{hb}(a, b) \implies \begin{array}{ll} \neg\mathsf{mo}(b, a), & \neg\mathsf{mo}(\mathsf{rf}(b), a), \\ \neg\mathsf{mo}(\mathsf{rf}(b), \mathsf{rf}(a)), & \neg\mathsf{mo}(b, \mathsf{rf}(a)) \end{array}
$$

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).[4] We want to 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 hence $\mathsf{hb}(a, b)$, we have $\neg\mathsf{mo}(b, a)$. But mo is a total order on writes to each location, so $\mathsf{mo}(a, b)$.
- Because $\mathsf{rf}(d) = c$, we have $\mathsf{sw}(c, d)$ and thus $\mathsf{hb}(c, d)$. By transitivity of hb, we know that $\mathsf{hb}(b, d)$ 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.

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.

Altogether, we write $\mathsf{consistentC11}(G)$ to say that a graph G satisfies the axioms above (plus one more for uninitialized reads). The full set of axioms is in the technical appendix [1].

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 keep the semantics streamlined, we adopt A-normal form [15], which requires intermediate computations to be named through `let`-binding, which is 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 then 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 [1]), then $[\![e]\!] = \textbf{err}$; 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: a logic for release-acquire consistency

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 also 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 (release-acquire) C11 memory model with a program logic that (1) captures intuitions about transitive visibility more directly and (2) supports thread-local reasoning. This section presents GPS, which achieves both goals through

- *per-location protocols* that abstract away event graphs, and
- *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

[4] Formally, $\mathsf{rf}(d) = c$; graphically, we draw an rf edge from c to d, so that the arrow points in the direction of hb.

operators of *multi-sorted* first-order logic:

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

where θ ranges over *sorts* (for now, just Val) and t ranges over *terms*. We write $t : \theta$ if t has sort θ, and assume that variables X are broken into classes by sort (ℓ, x, y, z for variables of sort Val).

Per-location protocols We start with a twist on message passing:

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

Intuitively, this variant, with multiple threads sending the same same message (37), 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 $(37, 1)$. Alas, this kind of reasoning 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.

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* $\sqsubseteq_\tau$, which is a partial order on states.
- A *state interpretation* $\tau(s, z)$, which is a resource assertion in which s and z appear free (*i.e.,* it is 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.

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

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

$$\frac{P' \Rrightarrow P \quad \{P\}\, e\, \{x.\ Q\} \quad \forall x.\ Q \Rrightarrow Q'}{\{P'\}\, e\, \{x.\ Q'\}} \qquad \frac{P \Rightarrow Q}{P \Rrightarrow Q} \qquad \frac{P \Rrightarrow Q}{P * R \Rrightarrow Q * R}$$

Figure 2. A selection of basic logical rules for GPS

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 $\mathsf{Flg}(x)$, a thread must (1) write 1 to y and (2) have already observed that $\boxed{x : \mathbf{37} \mid \mathsf{Dat}}$, 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:[5]

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

The Hoare triple takes as a precondition some pre-existing knowledge about ℓ's protocol. (For the message receiver, this knowledge will be $\boxed{y : \mathbf{0} \mid \mathsf{Flg}(x)}$.) 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:

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

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.

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 $s' \sqsupseteq_\tau s$. Finally, PL-protocols have allowed us not just to abstract away from the event graph, but also 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.

Physical resources GPS makes the simplifying assumption that each location is either always used nonatomically (*i.e.,* for data),

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

or always used atomically (*i.e.,* for synchronization).[6] 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

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

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:

$$\{\mathsf{true}\}\ \mathtt{alloc}(n)\ \{x.\ \mathsf{uninit}(x) * \cdots * \mathsf{uninit}(x+n-1)\}$$
$$\{\mathsf{uninit}(\ell) \vee \ell \hookrightarrow -\}\ [\ell]_{\mathtt{na}} := v\ \{\ell \hookrightarrow v\}$$
$$\{\ell \hookrightarrow v\}\quad [\ell]_{\mathtt{na}}\quad \{x.\ x = v * \ell \hookrightarrow v\}$$

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

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

but since atomic locations are shared, separation enforces only that different observations about the state of ℓ's protocol are coherent:

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

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

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 [13, 20, 21], we model ghosts as *partial commutative monoids* (PCMs). In particular, GPS is parameterized by a collection of PCMs μ, such that

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

The unit represents the empty permission, while $t \cdot_\mu t'$ combines the permissions t and t'. In general, we do *not* want all compositions to be defined: we want certain permissions to be exclusive. 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

$$\boxed{\gamma : \diamond \mid \mathsf{Tok}} * \boxed{\gamma : \diamond \mid \mathsf{Tok}} \quad \Rightarrow \quad \mathsf{false}$$

Hence, GPS ensures the token for ghost γ cannot be owned twice. (We use this PCM in an example at the end of the section.)

Taking stock: resource ownership versus 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 $t = t'$). GPS has a modality $\Box$ for knowledge, where $\Box P$ holds if P is true and does not depend on resource ownership. Knowledge includes assertions that are "pure" in the parlance of separation logic, like equalities on terms, but it also includes protocol observations:

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

Knowledge does not include ownership: $\Box(\ell \hookrightarrow v) \Rightarrow \mathsf{false}$. But knowledge can be shared freely, while resource ownership must be carefully managed to avoid duplication. Thus, we have

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

where the second axiom can be used, together with the frame rule, to show that knowledge is retained no matter what an expression does. Finally, the $\Box$ modality distributes over $\wedge$, $\vee$, $\forall$, and $\exists$.

Ownership transfer through protocols To explain physically-based ownership transfers, we consider a simple spinlock:

```
newLock() ≜ let x = alloc(1) in [x]at := unlocked; x
lock(x)   ≜ repeat CAS(x, unlocked, locked) end
unlock(x) ≜ [x]at := unlocked
```

where `unlocked` = 0 and `locked` = 1. We want to reason about this lock in the style of concurrent separation logic [23], *i.e.,* we want to be able to prove the following triples:

$$\{P\}\ \ \mathtt{newLock}\ \ \{x.\ \Box\mathsf{isLock}(x)\}$$
$$\{\mathsf{isLock}(x)\}\ \ \mathtt{lock}(x)\ \ \{P\}$$
$$\{\mathsf{isLock}(x) * P\}\ \mathtt{unlock}(x)\ \{\mathsf{true}\}$$

Here, the assertion P is an arbitrary *resource invariant* (*e.g.,* 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 **LP** having a single state Inv, where

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

Intuitively, whenever a thread releases the lock, it must have reestablished the resource invariant P, which it then relinquishes,

[6] This assumption is in line with the C and C++ standards, which require variable declarations to specify whether the variables will be accessed atomically or nonatomically.

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 $\tau(s, v)$:

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

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

Subsequently, we can reason about `CAS` as follows:

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

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, in general, reading works as follows:

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

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 `CAS`es), 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 `CAS`es, 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.
- `CAS`es both relinquish and gain ownership when successful, and behave like reads when unsuccessful.

Ownership transfer through escrows We have just seen how GPS axiomatizes the intrinsic, physical synchronization offered by `CAS`, but of course programs can and do build up their own means of synchronization without using `CAS` (which is relatively expensive). Take the following algorithm, related to us by Ernie Cohen:

```
[x]at := choose(1,2);        ||  [y]at := choose(1,2);
repeat [y]at end;            ||  repeat [x]at end;
if [x]at == [y]at then       ||  if [x]at != [y]at then
  /* crit. section */        ||    /* crit. section */
```

If we extend the language with `choose` (for nondeterministic choice), this algorithm guarantees mutual exclusion between critical sections under C11's memory model, without using `CAS`. The key is that the two threads have agreed on a *logical* condition for synchronization: the first thread wins if the values pointed to by x and y are equal, and the second wins if they are not.

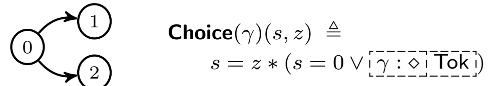
In GPS terms, we again imagine some resource invariant P to which the critical sections provide access—but we need some way to reflect the logical synchronization condition of the algorithm. More generally, we need a way for threads to gain ownership of resources not because they won a physical, `CAS`-mediated race, but because they have met some logical condition. But if we are to avoid duplication of ownership, we must ensure that the logical condition is "exclusive", so that it can be met at most once.

Thus we are led to the final concept in GPS: *escrows*.[7] The idea is that resources are placed "under escrow" (*i.e.,* temporarily given up) until some exclusive, logical condition is met, at which point the thread meeting the condition gains ownership of the resources. 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, and must be exclusive: $P * P \Rightarrow \mathsf{false}$. 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 (as we will see in §4). 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 above example, we can now apply the full apparatus of GPS. First, we have a protocol $\mathbf{Choice}(\gamma)$ with states 0, 1 and 2 (here we pun abstract states and concrete numbers):



This protocol captures not just the irreversible choices made for x or y, but also control over *who* can make these choices; we do so through a ghost token $\diamond$, where the identity γ is taken as a parameter to the protocol. Only the owner of $\boxed{\gamma : \diamond}$ will be able to transition from the 0 state to the 1 or 2 states.

Second, we have an escrow type $\mathbf{PE}(\gamma^x, \gamma^y)$ for P:

$$\mathbf{PE}(\gamma^x, \gamma^y) : \exists i, j > 0.\ \boxed{x : i} * \boxed{y : j} * \left(\begin{array}{l} \boxed{\gamma^x : \diamond} * i = j \\ \vee \boxed{\gamma^y : \diamond} * i \neq j \end{array}\right) \rightsquigarrow P$$

Here we have elided the protocol and ghost types, which are **Choice** and Tok respectively. The escrow condition says that a thread must know that x and y are both in nonzero states, and that either the states are equal and the γ^x token is owned, or they are

[7] As we discuss in Section 6, escrows are closely related to Bugliesi *et al.*'s notion of "exponential serialization" [9].

distinct and the γ^y token is owned. The fact that the escrow condition is exclusive follows from the combined use of tokens and protocol assertions; the latter dictate that the existentials can only be instantiated in one way.

We assume at the outset that we are given ownership of P, which we then want to allow the two threads to race for. We can create ghost tokens and put P under escrow using ghost moves (we elide the $\exists$ quantifers):

$$P \Rrightarrow \boxed{\gamma_1^x : \diamond} * \boxed{\gamma_2^x : \diamond} * \boxed{\gamma_1^y : \diamond} * \boxed{\gamma_2^y : \diamond} * [\mathbf{PE}(\gamma_1^x, \gamma_1^y)]$$

where the 1 subscript is for escrow tokens, while the 2 subscript is for protocol tokens. We then have

$$\{\mathsf{uninit}(x)\}\ [x]_{\mathtt{at}} := 0 \left\{ \boxed{x : 0 \mid \mathbf{Choice}(\gamma_2^x)} \right\}$$

when initializing x, and likewise for y. Finally, we give the first thread ownership of $\boxed{\gamma_1^x : \diamond} * \boxed{\gamma_2^x : \diamond}$ and give the other tokens to the second thread. With this setup, the rest of the proof is straightforward; it, along with the other examples in this section, can be found in detail in the appendix [1].

4. Case studies

We have applied GPS to three challenging case studies for weak memory reasoning: *Michael and Scott's lock-free queue* [22], as well as *circular buffers* [17] and *bounded ticket locks* [11] (both drawn from the Linux kernel). Note that the first two of these exhibit non-SC behavior (cf. §1). 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 our appendix [1].

Circular buffers (Linux kernel) Figure 3 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 $q + \mathtt{buf}$), 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 non-atomically. All synchronization is performed through the reader/writer indices. Note, however, that (as in Cohen's example from the previous section) this synchronization is entirely *logical*: 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 be operating simultaneously.

A spec for circular buffers We will prove the following spec:

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

The spec is parameterized over a predicate P that should hold of all the elements in the buffer; it guarantees that $P(x)$ 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.[8] 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 3) 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 (excerpted in Figure 3) depends on all the features of GPS working in concert.

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** provide a way to communicate awareness of these escrows back and forth.

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 aforementioned **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 3 as elements of the ghost PCM $\wp(\mathbb{N})^4$ (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 state of the writer index is at least 0 and less than N cells ahead of the absolute state of the 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

[8] 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.

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 3 displays the proof outline for $\texttt{tryProd}(q, x)$. (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 $\square$ are only written once and then used freely in the rest of the proof since they hold true forever.

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 $i < j_0 + N \leq j + N$, 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. The semantics and soundness of GPS

Axiomatic models like C11's pose a challenge for the semantics and soundness of program logics: they do not provide the global notion of "current state" that such logics usually depend on, making it difficult to even define the meaning of a Hoare triple.

Here we briefly summarize the semantics and soundness of GPS. The supplementary material [1] includes a technical appendix containing the complete semantics, a decomposition of our soundness theorem into key lemmas, and further details about its proof. It also includes a Coq development mechanizing the entire logic and its soundness proof.

Overview Reasoning in GPS is compositional: we prove triples about each expression, and link them together using the `let` and `fork` rules. But the expression semantics is global: it assumes a whole, closed program. The semantics of GPS must bridge this gap.

```
newBuffer()
let q = alloc(N+2)
[q+ri]at := 0;
[q+wi]at := 0;
q

where wi ≜ 0,
ri ≜ 1, buf ≜ 2
```

```
tryProd(q, x)
let w = [q+wi]at
let r = [q+ri]at
let w' = w+1 mod N
if w' == r then 0
else
  [q+buf+w]na := x;
  [q+wi]at := w';  1
```

```
tryCons(q)
let w = [q+wi]at
let r = [q+ri]at
let r' = r+1 mod N
if w == r then 0
else
  let x = [q+buf+r]na
  [q+ri]at := r';  x
```

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

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

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

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

$$\mathbf{PE}(\gamma, q, i) : \boxed{\gamma : \mathsf{escP}(i)} \rightsquigarrow \mathsf{uninit}(q + \mathtt{buf} + (i \bmod N)) \vee (q + \mathtt{buf} + (i \bmod N)) \hookrightarrow -$$

$$\mathbf{CE}(\gamma, q, j) : \boxed{\gamma : \mathsf{escC}(j)} \rightsquigarrow \exists x.\ P(x) * (q + \mathtt{buf} + (j \bmod N)) \hookrightarrow x$$

$$\mathsf{all} \triangleq (\mathbb{N}, \mathbb{N}, \mathbb{N}, \mathbb{N})$$
$$\mathsf{restP}(i) \triangleq (\{j \mid j > i\}, \{j \mid j \geq i\}, \emptyset, \emptyset)$$
$$\mathsf{restC}(i) \triangleq (\emptyset, \emptyset, \{j \mid j > i\}, \{j \mid j \geq i\})$$
$$\mathsf{protP}(i) \triangleq (\{i\}, \emptyset, \emptyset, \emptyset)$$
$$\mathsf{escP}(i) \triangleq (\emptyset, \{i\}, \emptyset, \emptyset)$$
$$\mathsf{protC}(i) \triangleq (\emptyset, \emptyset, \{i\}, \emptyset)$$
$$\mathsf{escC}(i) \triangleq (\emptyset, \emptyset, \emptyset, \{i\})$$

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

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

Figure 3. Proof excerpt for the circular buffer case study

This kind of gap is always present in concurrent program logics, but one normally bridges it by appeal to the heap (thereby assuming SC semantics). In particular, threads view each other's activities through constraints on heap evolution—either simple invariants, or rely/guarantee relations—and bounding the possible heaps is all that is needed to determine *e.g.,* what reads will return.

Fortunately, the rules of GPS point the way forward: rather than mediate thread activity through a global heap, we can understand it through the various kinds of ownership and knowledge introduced in §3, which give us a *purely logical* way of predicting *e.g.,* what reads will return—a form of rely/guarantee reasoning that does not depend on a global heap. We therefore formulate a notion of *local safety* for a thread, which says that the actions it controls conform to its guarantee, assuming that the actions its environment controls (*e.g.,* the contents of a read event) follow its rely. We can then easily show that the proof rules of GPS preserve local safety.

We are then left with another gap: local safety makes no reference to the C11 axioms; it instead just assumes that the events it observes obey the rely constraints. We therefore formulate a notion of *global safety* for an event graph, which includes a *labeling* of the edges of the event graph with resource/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 expression 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.

Resources In the semantics of GPS, a *resource* r is a tuple (Π, g, Σ) of a *physical location map* Π, a *ghost identity map* g, and *known escrow set* Σ. 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

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

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

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

and similarly for atomic locations, where the protocol state is allowed to advance. 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 $\models \{P\}\, e\, \{x.Q\} \triangleq \forall n, r \in [\![P]\!].\ r \Rrightarrow \mathsf{LSafe}_n(e, [\![x.\ Q]\!])$.

Theorem 1 (Local soundness)**.** All of the proof rules given in §3 are sound for this semantics of Hoare triples.

Theorem 1 has been mechanized entirely in Coq; see `GpsLogic.v`.

Global safety We then define *global safety* $\mathsf{GSafe}_n(\mathcal{T}, G, \mathcal{L})$ over an *instrumented thread pool* $\mathcal{T}$, an event graph G, and a *labeling* $\mathcal{L}$. The instrumented thread pool maps each thread to a tuple (a, e, r, Φ) giving the thread's last event a in the graph, its continuation e, its current resources r, 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. Finally, the labeling must globally ensure:

- *Compatibility*: any set of concurrent resource transfers must be composable, *i.e.,* resources are never duplicated.
- *Conformance*: if $\mathsf{mo}(a, b)$ for two atomic writes/updates to ℓ with protocol τ, the labeled protocol states are related by $\sqsubseteq_\tau$.

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

Theorem 2 (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 $\mathsf{GSafe}_n(\mathcal{T}', G', \mathcal{L}')$.

Theorem 2 has been mechanized in Coq; see `GpsAdequate.v` for details.

Our main result, *adequacy*, is an easy corollary; it connects the proof theory all the way to the C11 execution (for closed e):

$$\{\mathsf{true}\}\, e\, \{x.\ P\} \quad \Longrightarrow \quad [\![e]\!] \subseteq \{V \mid [\![P[V/x]]\!] \neq \emptyset\}$$

6. Related work

Direct influences

The closest related work to GPS is the recent *Relaxed Separation Logic* (RSL) introduced by Vafeiadis and Narayan [29], which is the only prior program logic for the C11 memory model. The goal of RSL is to support simple CSL-style reasoning about release-acquire accesses: it is possible for a release write to *directly* transfer resource ownership to an acquire read. 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 a given value can only be used to perform a transfer *once* per location.

While GPS draws inspiration from RSL, there are many significant differences. Most importantly, GPS offers a much more flexible way of coordinating ownership and knowledge transfers between threads—including rely-guarantee reasoning—through its protocols and ghosts. This fact, together with escrows, allows us to lift several restrictions from RSL, including the one on repeated writes of the same value—crucial for handling the indices in the circular buffer, and the ticket numbers in the bounded ticket lock. To our knowledge, none of our case studies can be verified in RSL.

The semantics of GPS is also structured differently from that of RSL, which does not employ an intermediate step like local safety, and must therefore deal with compositionality directly at the level of event graphs using "contextual executions". We expect the two-step factoring of GPS to be easier to extend in the future.

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 concurrency.

Per-location (PL) protocols are inspired by CaReSL [28], another recent concurrency logic (for strong memory), which includes abstract STSs for governing shared state. The primary difference

is that our PL-protocols govern a single location, while CaReSL's STSs govern arbitrary heap regions. CaReSL also couples a notion of "tokens" directly with STSs, while GPS supports ghost state separately using *ghost PCMs* [13, 20, 21]. GPS's separation of orthogonal mechanisms has the side benefit of lifting 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.

Escrows are very similar to "exponential serialization", a mechanism recently proposed by Bugliesi *et al.* [9] 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 a non-duplicable 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—is established semantically (by proving $P * P \Rightarrow \mathsf{false}$ in the logic of GPS). In contrast, since the primitive affine predicates of Bugliesi *et al.*'s type system have no underlying semantic interpretation, exponential serialization requires a more complex and syntactic "guardedness" check on contexts.

Our use of *names* for protocol and escrow types is inspired by Gotsman et al. [16], who use named lock invariants. In both cases, the motivation for names is to break what would otherwise be a semantic circularity: statements about protocols (respectively, lock invariants) can appear within assertions, but their definitions *involve* arbitrary assertions. See [16] for more details.

Finally, while not directly related to our work on program logic, recent work suggests that model checking can also benefit by embracing weak memory as-is, rather than reducing it to SC [3, 5].

Alternative approaches

As we discussed in §1, 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

- Owens [24] proves that data-race free and "triangular-race" free programs on x86-TSO have SC behavior.
- Batty et al. [7] prove that for C11 restricted to nonatomics and SC-atomics, data-race freedom ensures SC behavior.
- Cohen and Schirmer [10] prove that programs following a certain ownership discipline and flushing write buffers at certain times on TSO models have SC behavior.
- Ferreira et al. [14] 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 our work with 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 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.

Recovering SC through low-level reasoning We are also aware of two program logics for reasoning about weak memory by directly incorporating a hardware memory model into the logic.

Ridge [26] provides a program logic for x86-TSO that supports rely-guarantee reasoning. The logic works directly with the operational x86-TSO model [25], 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 [30] 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.

Both of the above logics permit SC-like reasoning, but this reasoning applies only indirectly, since writes are actually routed through explicit write buffers. GPS, by contrast, provides proof rules whose restrictions directly and abstractly encompass the effect of reordering on local reasoning.

Acknowledgments

We gratefully acknowledge support by the EC FP7 FET project ADVENT.

References

[1] Supplemental material for this paper: `http://plv.mpi-sws.org/gps/`.

[2] S. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, 1996.

[3] J. Alglave. Weakness is a virtue. In *EC*^{2}, 2013.

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

[5] J. Alglave, D. Kroening, and M. Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *CAV*, 2013.

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

[7] 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.

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

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

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

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

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

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

[14] R. Ferreira, X. Feng, and Z. Shao. Parameterized memory models and concurrent separation logic. In *ESOP*, volume 6012 of *LNCS*, 2010.

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

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

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

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

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

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

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

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

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

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

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

[26] T. Ridge. A rely-guarantee proof system for x86-TSO. In *VSTTE*, volume 6217 of *LNCS*, 2010.

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

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

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

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

A. Language

A.1 Syntax

$$
\begin{array}{llll}
\textit{Val} & V & ::= & n \\
\textit{OVal} & v & ::= & x \mid V \\
\textit{Exp} & e & ::= & v \mid v \texttt{ + } v \mid v \texttt{ == } v \mid v \texttt{ mod } v \mid \texttt{let } x = e \texttt{ in } e \mid \texttt{repeat } e \texttt{ end} \mid \texttt{fork } e \\
 & & \mid & \texttt{if } v \texttt{ then } e \texttt{ else } e \mid \texttt{alloc}(n) \mid [v]_O \mid [v]_O \texttt{ := } v \mid \texttt{CAS}(v, v, v) \mid \texttt{FAI}(v) \\
\textit{OrderAnn} & O & ::= & \texttt{at} \mid \texttt{na} \\
\textit{EvalCtx} & K & ::= & [\,] \mid \texttt{let } x = K \texttt{ in } e \\
\textit{Action} & \alpha & ::= & \mathbb{S} \mid \mathbb{A}(\ell..\ell') \mid \mathbb{W}(\ell, V, O) \mid \mathbb{R}(\ell, V, O) \mid \mathbb{U}(\ell, V, V) \\
\textit{ActName} & a & & \text{(an infinite set)} \\
\textit{ActMap} & A & \in & \textit{ActName} \overset{\text{fin}}{\rightharpoonup} \textit{Action} \\
\textit{Graph} & G & ::= & (A, \mathsf{sb}, \mathsf{mo}, \mathsf{rf}) \qquad \mathsf{sb}, \mathsf{mo} \subseteq \mathrm{dom}(A) \times \mathrm{dom}(A), \mathsf{rf} \in \mathrm{dom}(A) \rightharpoonup \mathrm{dom}(A) \\
\textit{ThreadMap} & T & \in & \mathbb{N} \overset{\text{fin}}{\rightharpoonup} (\textit{ActName} \times \textit{Exp})
\end{array}
$$

A.2 Semantics

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

$$
\begin{array}{rcll}
n \texttt{ + } m & \xrightarrow{\mathbb{S}} & k & k = n + m \\
n \texttt{ mod } m & \xrightarrow{\mathbb{S}} & k & k = n \bmod m \\
n \texttt{ == } m & \xrightarrow{\mathbb{S}} & 1 & n = m \\
n \texttt{ == } m & \xrightarrow{\mathbb{S}} & 0 & n \neq m \\
\texttt{let } x = V \texttt{ in } e & \xrightarrow{\mathbb{S}} & e[V/x] & \\
\texttt{repeat } e \texttt{ end} & \xrightarrow{\mathbb{S}} & \texttt{let } x = e \texttt{ in if } x \texttt{ then } x \texttt{ else repeat } e \texttt{ end} & \\
\texttt{if } V \texttt{ then } e_1 \texttt{ else } e_2 & \xrightarrow{\mathbb{S}} & e_1 & V \neq 0 \\
\texttt{if } V \texttt{ then } e_1 \texttt{ else } e_2 & \xrightarrow{\mathbb{S}} & e_2 & V = 0 \\
\\
\texttt{alloc}(n) & \xrightarrow{\mathbb{A}(\ell..\ell+n-1)} & \ell & \\
[\ell]_O & \xrightarrow{\mathbb{R}(\ell,V,O)} & V & \\
[\ell]_O \texttt{ := } V & \xrightarrow{\mathbb{W}(\ell,V,O)} & 0 & \\
\texttt{CAS}(\ell, V_o, V_n) & \xrightarrow{\mathbb{U}(\ell,V_o,V_n)} & 1 & \\
\texttt{CAS}(\ell, V_o, V_n) & \xrightarrow{\mathbb{R}(\ell,V',\texttt{at})} & 0 & V' \neq V_o \\
\texttt{FAI}(\ell) & \xrightarrow{\mathbb{U}(\ell,V,V')} & V & V' = (V+1) \bmod \mathbf{C} \\
K[e] & \xrightarrow{\alpha} & K[e'] & e \xrightarrow{\alpha} e'
\end{array}
$$

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

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

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

We discuss the validity of these operational rules in section A.3 below.

$$
\mathsf{execs}(e) \triangleq \{\ (e', G) \ \mid\ \langle [i \mapsto (\mathsf{start}, e)]; ([\mathsf{start} \mapsto \mathbb{S}], \emptyset, \emptyset, \emptyset)\rangle \longrightarrow^* \langle [i \mapsto (_, e')] \uplus T; G\rangle\ \}
$$

$$
[\![e]\!] \triangleq \begin{cases} \mathbf{err} & \exists(_, G) \in \mathsf{execs}(e).\ \mathsf{dataRace}(G) \vee \mathsf{memoryError}(G) \\ \{\ V \ \mid\ (V, _) \in \mathsf{execs}(e)\ \} & \text{otherwise} \end{cases}
$$

A.3 Memory model

A.3.1 The C11 atomic access modes

The C11 standard [18, 19] includes several kinds of atomic accesses: sequentially-consistent, release-acquire, release-consume, and fully relaxed. We have focused on release-acquire, because:

- Sequentially-consistent accesses are already well-understood.
- Release-consume atomics are useful only for specific architectures (PowerPC and Arm), but substantially complicate the memory model.

- Fully relaxed accesses, as formalized by Batty et al. [6], suffer from several known problems. First, they allow *out-of-thin-air* reads, which the text of the standard explicitly forbids [18, 19]—but it is not known how to rule out these reads without also obstructing key compiler optimizations. On the other hand, even as formalized, fully relaxed access do not permit certain basic optimizations [29]. They also pose severe problems for compositional reasoning [8, 29].

As we explain in section A.3.4, however, GPS is sound for the full C11 model as formalized by Batty et al. [6].

A.3.2 The formal C11 model

The C11 memory model we use is based on the formalization of Batty et al. [6], as simplified by Batty et al. [7] in the absence of release-consume atomics. We also incorporate the following simplifications introduced by Vafeiadis and Narayan [29]:

- The sb and sw orders are not transitive; *e.g.,* sb relates each event only to its immediate successors in program order. This simplifies both the operational semantics of the language and the semantics of GPS. Since hb *is* transitively closed, this has no effect on the memory model axioms.
- The "additional synchronized with" edges are incorporated into sb rather than sw, which again makes no difference for the axioms but simplifies the semantics.
- For uniformity, the sw edges include sb-related events, whereas in [7] these are ruled out. Since hb includes both sw and sb, this makes no difference to the axioms.

In addition to these simplifications, our formalization of the memory model drops release sequences, instead requiring sw edges only between immediate atomic read/write pairs. Consequently, our axioms are strictly *weaker* than those in *e.g.,* Batty et al. [6], since we require strictly fewer sw edges. GPS does not have proof rules that take advantage of release sequences, so it is sound with or without them. See section A.3.4.

A.3.3 Justifying the operational semantics

The C11 axioms are generally understood to apply to an entire program execution, but the operational semantics we have given assumes that we can construct the event graph in a step-by-step fashion while guaranteeing consistency with C11 *at every step*. It also assumes that we can introduce read events in such an order that they always read from write events already appearing in the event graph. It does *not* assume, however, that new write events appear at the end of mo sequences.

These assumptions are justified by the fact that, in the absence of release-consume and relaxed operations, we know that for any complete execution satisfying the C11 axioms:

- The hb order is acyclic, and
- If $\mathsf{rf}(b) = a$ then $\mathsf{hb}(a, b)$.

Since the sb order is dictated by the program text and is a sub-order of hb, there is some sequence of events in program order that is also in hb order. Based on the second bullet above, we know we can generate the event graph of the complete execution step-by-step using an operational semantics, while assuming that new nodes read only from previous nodes. Finally, because the event graphs generated in each step are prefixes of the complete event graph that are closed under rf and hb-predecessors, we know that if the axioms hold of the complete graph, they hold of the prefixes.

It is therefore also possible to instead work with the usual semantics, in which consistency is only assumed for the entire execution, which is how soundness of RSL is proved [29]. We chose to check prefix consistency mainly to simplify the soundness proof for GPS, and in particular the statement of the instrumented execution theorem (*i.e.,* Theorem 5).

A.3.4 Soundness for the full C11 model

Despite all of the above-mentioned simplifications, GPS is trivially sound for the full C11 model: any program verified by GPS is guaranteed to only use release-acquire atomics, which formally justifies most of the simplifications made above [see 7]. The additional simplification we have made here of dropping release sequences is easy to justify: doing so strictly weakens the axioms, so any execution consistent under the semantics of [7] is consistent under the axioms given below.

A.3.5 Axioms

$$
\begin{array}{ll}
\mathsf{consistentC11}(A, \mathsf{sb}, \mathsf{mo}, \mathsf{rf}) \triangleq & \\
\quad \forall a, b.\ \mathsf{mo}(a, b) \implies \exists \ell.\ \mathsf{writes}(a, \ell, -),\ \mathsf{writes}(b, \ell, -) & \text{(ConsistentMO1)} \\
\quad \forall \ell.\ \mathsf{strictTotalOrder}(\{a \mid \mathsf{writes}(a, \ell, -)\}, \mathsf{mo}) & \text{(ConsistentMO2)} \\
\quad \forall b.\ \ \mathsf{rf}(b) \neq \bot \iff \exists \ell, a.\ \mathsf{writes}(a, \ell, -),\ \mathsf{reads}(b, \ell, -),\ \mathsf{hb}(a, b) & \text{(ConsistentRF1)} \\
\quad \forall a, b.\ \mathsf{rf}(b) = a \implies \exists \ell, V.\ \mathsf{writes}(a, \ell, V),\ \mathsf{reads}(b, \ell, V),\ \neg\mathsf{hb}(b, a) & \text{(ConsistentRF2)} \\
\quad \forall a, b.\ \mathsf{rf}(b) = a, (\mathsf{isNonatomic}(a) \vee \mathsf{isNonatomic}(b)) \implies \mathsf{hb}(a, b) & \text{(ConsistentRFNA)} \\
\quad \forall a, b.\ \mathsf{hb}(a, b) \implies & \\
\qquad a \neq b,\ \ \neg\mathsf{mo}(\mathsf{rf}(b), \mathsf{rf}(a)),\ \ \neg\mathsf{mo}(\mathsf{rf}(b), a),\ \ \neg\mathsf{mo}(b, \mathsf{rf}(a)),\ \ \neg\mathsf{mo}(b, a) & \text{(Coherence)} \\
\quad \forall a, c.\ \mathsf{isUpd}(c),\ \mathsf{rf}(c) = a \implies \mathsf{mo}(a, c),\ \nexists b.\ \mathsf{mo}(a, b),\ \mathsf{mo}(b, c) & \text{(AtomicCAS)} \\
\quad \forall a \neq b, \vec{\ell}, \vec{\ell'}.\ A(a) = \mathbb{A}(\vec{\ell}),\ A(b) = \mathbb{A}(\vec{\ell'}) \implies \vec{\ell} \pitchfork \vec{\ell'} & \text{(ConsistentAlloc)}
\end{array}
$$

where $\mathsf{hb} \triangleq (\mathsf{sb} \cup \mathsf{sw})^+$

$$
\begin{array}{l}
\mathsf{sw} \triangleq \{(a, b) \mid \mathsf{rf}(a) = b,\ \mathsf{isAtomic}(a),\ \mathsf{isAtomic}(b)\} \\
\mathsf{reads}(a, \ell, V) \triangleq A(a) \in \{\mathbb{R}(\ell, V, -), \mathbb{U}(\ell, V, -)\} \\
\mathsf{writes}(a, \ell, V) \triangleq A(a) \in \{\mathbb{W}(\ell, V, -), \mathbb{U}(\ell, -, V)\} \\
\mathsf{strictTotalOrder}(S, R) \triangleq (\nexists a.\ R(a, a)), \\
\qquad\qquad\qquad\qquad\qquad (\forall a, b, c.\ R(a, b), R(b, c) \implies R(a, c)),
\end{array}
$$

$$(\forall a, b \in S.\ a \neq b \implies R(a,b) \vee R(b,a))$$

$\mathsf{dataRace}(A, \mathsf{sb}, \mathsf{mo}, \mathsf{rf}) \triangleq \exists \ell.\ \exists a \neq b \in \mathrm{dom}(A).$
$\quad \mathsf{accessesLoc}(a, \ell), \mathsf{accessesLoc}(b, \ell), \mathsf{writes}(a, -, -) \vee \mathsf{writes}(b, -, -),$
$\quad \mathsf{isNonatomic}(a) \vee \mathsf{isNonatomic}(b), \neg\mathsf{hb}(a, b), \neg\mathsf{hb}(b, a)$
where $\mathsf{hb} \triangleq (\mathsf{sb} \cup \mathsf{sw})^+$

$\mathsf{memoryError}(A, \mathsf{sb}, \mathsf{mo}, \mathsf{rf}) \triangleq \exists \ell.\ \exists b \in \mathrm{dom}(A).$
$\quad \mathsf{accessesLoc}(b, \ell),$
$\quad \nexists a \in \mathrm{dom}(A).\ A(a) = \mathbb{A}(\vec{\ell}),\ \ \ell \in \vec{\ell},\ \ \mathsf{hb}(a, b)$
where $\mathsf{hb} \triangleq (\mathsf{sb} \cup \mathsf{sw})^+$

B. Logic

B.1 Semantic structures

B.1.1 Parameters

We assume:

- The following domains, with associated metavariables:

$$\begin{array}{lllllll} \mathsf{s} & \in & \mathit{State} & \text{(a set)} \qquad & \tau & \in & \mathit{ProtTy} \quad \text{(a set)} \\ \sigma & \in & \mathit{EscrowTy} & \text{(a set)} \qquad & \mu & \in & \mathit{PCMTy} \quad \text{(a set)} \end{array}$$

- For each μ, a partial commutative monoid $[\![\mu]\!]$ with unit ε_μ, multiplication $\cdot_\mu$, and a homomorphism $|-| : [\![\mu]\!] \to [\![\mu]\!]$ such that
  (1) $m \cdot_\mu m' = \varepsilon_\mu \implies m = m' = \varepsilon_\mu$, (positivity)
  (2) $m = m \cdot_\mu |m|$, (duplicability)
  (3) $m \cdot_\mu m' \leq_\mu m \implies |m'| = m'$, and (maximality)
  (4) $m \cdot_\mu m_1 = m \cdot_\mu m_2 \implies |m_1| = |m_2|$ (partial cancellativity)
  where $m \leq_\mu m'$ iff $\exists m''.\ m \cdot_\mu m'' = m'$.
- For each τ a partial order $\sqsubseteq_\tau \subseteq \mathit{State} \times \mathit{State}$.

B.1.2 Domains

$$\begin{array}{rcl} \pi \in \mathit{Prot} & ::= & \bot \mid \mathsf{uninit} \mid \mathtt{na}(V) \mid \mathtt{at}(\tau, S) \text{ where } S \in \mathit{Trace}(\tau) \\ r \in \mathit{Resource} & \triangleq & \left\{ (\Pi, g, \Sigma) \;\middle|\; \Pi \in \mathbb{N} \to \mathit{Prot},\ \Sigma \subseteq \mathit{EscrowTy} \right\} \\ \mathit{Trace}(\tau) & \triangleq & \left\{ S \overset{\text{fin}}{\subseteq} \mathit{State} \;\middle|\; S \text{ totally ordered by } \sqsubseteq_\tau \right\} \\ \mathit{Ghost} & \triangleq & \left\{ g \in \prod_{\mu \in \mathit{PCMTy}} \mathbb{N} \to [\![\mu]\!] \;\middle|\; \forall \mu \in \mathit{PCMTy}.\ g(\mu)(n) = \varepsilon_\mu \text{ for infinitely many } n \right\} \end{array}$$

B.1.3 Resource composition

Protocol composition is given by the following partial commutative operator:

$$\begin{array}{rcl} \bot \oplus \pi = \pi \oplus \bot & \triangleq & \pi \\ \mathtt{na}(V) \oplus \mathtt{na}(V) & \triangleq & \mathtt{na}(V) \\ \mathtt{at}(\tau, S_1) \oplus \mathtt{at}(\tau, S_2) & \triangleq & \mathtt{at}(\tau, S_1 \cup S_2) \text{ when well-typed} \end{array}$$

and is lifted pointwise to protocol maps. Composition on ghosts is likewise defined pointwise.

$$\begin{array}{rcl} (\Pi, g, \Sigma) \oplus (\Pi', g', \Sigma') & \triangleq & (\Pi \oplus \Pi', g \oplus g', \Sigma \cup \Sigma') \\ r[\ell] & \triangleq & r.\Pi(\ell) \\ (\Pi, g, \Sigma)[\ell := \pi] & \triangleq & (\Pi[\ell := \pi], g, \Sigma) \end{array} \qquad \begin{array}{rcl} \mathsf{emp} & \triangleq & ((\lambda n.\ \bot), (\lambda \mu.\ \lambda n.\ \varepsilon_\mu), \emptyset) \\ r \leq r'' & \triangleq & \exists r'.\ r \oplus r' = r'' \\ r \# r' & \triangleq & r \oplus r' \text{ defined} \end{array}$$

B.1.4 Resource stripping

$$|(\Pi, g, \Sigma)| \triangleq (|\Pi|, |g|, \Sigma) \qquad |g| \triangleq \lambda \mu.\ \lambda n.\ |g(\mu)(n)| \qquad |\Pi| \triangleq \lambda \ell. \begin{cases} \Pi(\ell) & \Pi(\ell) = \mathtt{at}(-,-) \\ \bot & \text{otherwise} \end{cases}$$

B.1.5 Propositions

$$\begin{array}{rcl} \mathit{Prop} & \triangleq & \left\{ \mathcal{P} \subseteq \mathit{Resource} \;\middle|\; \forall r \in \mathcal{P}.\ \forall r' \# r.\ r \oplus r' \in \mathcal{P} \right\} \\ \lfloor r \rfloor & \triangleq & \left\{ r \oplus r' \;\middle|\; r' \in \mathit{Resource} \right\} \\ \mathcal{P}_1 * \mathcal{P}_2 & \triangleq & \left\{ r_1 \oplus r_2 \mid r_1 \in \mathcal{P}_1, r_2 \in \mathcal{P}_2 \right\} \end{array}$$

B.1.6 Escrow and protocol type interpretations

We assume we are given the following interpretation functions:

$$\mathsf{interp}(\tau) \in \mathit{State} \times \mathit{Val} \to \mathit{Prop} \qquad \mathsf{interp}(\sigma) \in \mathit{Prop} \times \mathit{Prop}$$

where if $\mathsf{interp}(\sigma) = (\mathcal{P}, \mathcal{P}')$ then $\mathcal{P} * \mathcal{P} = \emptyset$.

B.2 Local safety

B.2.1 Protocols

$$\begin{array}{rcl} \mathtt{at}(\tau, S) \sqsubseteq_{\mathtt{at}} \mathtt{at}(\tau, S') & \triangleq & \forall \mathsf{s} \in S.\ \exists \mathsf{s}' \in S'.\ \mathsf{s} \sqsubseteq_\tau \mathsf{s}' \\ \pi \equiv_{\mathtt{at}} \pi' & \triangleq & \pi \sqsubseteq_{\mathtt{at}} \pi' \wedge \pi' \sqsubseteq_{\mathtt{at}} \pi \\ r_{\mathsf{rf}} \in \mathsf{envMove}(r, \ell, V) & \triangleq & \exists \tau, \mathsf{s}.\ \ r_{\mathsf{rf}} \in \mathsf{interp}(\tau)(\mathsf{s}, V),\ \ r[\ell] \sqsubseteq_{\mathtt{at}} r_{\mathsf{rf}}[\ell] \equiv_{\mathtt{at}} \mathtt{at}(\tau, \{\mathsf{s}\}),\ \ r_{\mathsf{rf}} \# r \\ (r_{\mathsf{sb}}, r_{\mathsf{rf}}) \in \mathsf{atGuar}(r, \ell, V) & \triangleq & \exists \tau, \mathsf{s}, S.\ \ r_{\mathsf{rf}} \in \mathsf{interp}(\tau)(\mathsf{s}, V),\ \ (r_{\mathsf{sb}} \oplus r_{\mathsf{rf}}) = r[\ell := \mathtt{at}(\tau, S \cup \{\mathsf{s}\})],\ \ r_{\mathsf{sb}}[\ell] = r_{\mathsf{rf}}[\ell] \\ & & \text{either } r[\ell] = \mathsf{uninit},\ \ S = \emptyset \text{ or } r[\ell] = \mathtt{at}(\tau, S),\ \ \forall \mathsf{s}_0 \in S.\ \mathsf{s}_0 \sqsubseteq_\tau \mathsf{s} \end{array}$$

B.2.2 Rely/guarantee

α	r'	$r' \in \mathsf{rely}(r, \alpha)$ if
$\mathbb{R}(\ell, V, \mathtt{na})$	r	$r[\ell] = \mathtt{na}(V') \implies V = V'$
$\mathbb{R}(\ell, V, \mathtt{at})$	$r \oplus \lfloor r_{\mathsf{rf}} \rfloor$	$r[\ell] = \mathtt{at}(-) \implies r_{\mathsf{rf}} \in \mathsf{envMove}(r, \ell, V)$
$\mathbb{U}(\ell, V, V')$	$r \oplus r_{\mathsf{rf}}$	$r[\ell] = \mathtt{at}(-) \implies r_{\mathsf{rf}} \in \mathsf{envMove}(r, \ell, V)$
$\mathbb{W}(\ell, V, \mathtt{at})$	r	$r[\ell] = \mathtt{at}(-) \implies \exists V'.\ \mathsf{envMove}(r, \ell, V') \neq \emptyset$
otherwise	r	always

α	$(r_{\mathsf{sb}}, r_{\mathsf{rf}}) \in \mathsf{guar}(r_{\mathrm{pre}}, r, \alpha)$ if
$\mathbb{S}$	$r_{\mathsf{rf}} = \mathsf{emp},\ r_{\mathsf{sb}} = r$
$\mathbb{A}(\ell..\ell')$	$r_{\mathsf{rf}} = \mathsf{emp},\ r_{\mathsf{sb}} = r[\ell..\ell' := \mathsf{uninit}]$
$\mathbb{R}(\ell, V, \mathtt{na})$	$r_{\mathsf{rf}} = \mathsf{emp},\ r_{\mathsf{sb}} = r,\ r[\ell] = \mathtt{na}(-)$
$\mathbb{R}(\ell, V, \mathtt{at})$	$r_{\mathsf{rf}} = \mathsf{emp},\ r_{\mathsf{sb}} = r,\ r[\ell] = \mathtt{at}(-)$
$\mathbb{W}(\ell, V, \mathtt{na})$	$r_{\mathsf{rf}} = \mathsf{emp},\ r_{\mathsf{sb}} = r[\ell := \mathtt{na}(V)],\ r[\ell] \in \{\mathsf{uninit}, \mathtt{na}(-)\}$
$\mathbb{W}(\ell, V, \mathtt{at})$	$(r_{\mathsf{sb}}, r_{\mathsf{rf}}) \in \mathsf{atGuar}(r, \ell, V'),\ \forall r_E \in \mathsf{envMove}(r_{\mathrm{pre}}, \ell, -).\ r_E[\ell] \sqsubseteq_{\mathtt{at}} r_{\mathsf{rf}}[\ell]$
$\mathbb{U}(\ell, V, V')$	$(r_{\mathsf{sb}}, r_{\mathsf{rf}}) \in \mathsf{atGuar}(r, \ell, V'),\ r[\ell] = \mathtt{at}(-)$

B.2.3 Ghost moves

$$\frac{r \in \mathcal{P}}{r \Rrightarrow \mathcal{P}} \qquad \frac{r_0 \Rrightarrow \mathcal{P} \qquad \forall r \in \mathcal{P}.\ r \Rrightarrow \mathcal{P}'}{r_0 \Rrightarrow \mathcal{P}'} \qquad \frac{m \in \llbracket \mu \rrbracket}{r \Rrightarrow \lfloor r \rfloor * \{(\bot, [\mu \mapsto [i \mapsto m]], \emptyset) \mid i \in \mathbb{N}\}} \qquad \frac{\forall g_F \# g.\ g_F \# g'}{(\Pi, g, \Sigma) \Rrightarrow \lfloor (\Pi, g', \Sigma) \rfloor}$$

$$\frac{\mathsf{interp}(\sigma) = (\mathcal{P}, \mathcal{P}') \qquad r' \in \mathcal{P}'}{(\Pi, g, \Sigma) \oplus r' \Rrightarrow \lfloor (\Pi, g, \Sigma \cup \{\sigma\}) \rfloor} \qquad \frac{\begin{array}{c}\mathsf{interp}(\sigma) = (\mathcal{P}, \mathcal{P}') \\ \sigma \in r.\Sigma \qquad r \in \mathcal{P}\end{array}}{r_0 \oplus r \Rrightarrow \lfloor r_0 \rfloor * \mathcal{P}'}$$

B.2.4 Protocol equivalence for writes

α	$(r_{\mathrm{pre}}, r') \in \mathsf{wpe}(\alpha)$ if
$\mathbb{A}(\ell_1..\ell_n)$	$\forall i. 1 \leq i \leq n \Rightarrow r'(\ell_i) = \bot$
$\mathbb{W}(\ell, -, \mathtt{at})$	$r_{\mathrm{pre}}[\ell] = \mathtt{at}(-) \wedge r'[\ell] = \mathtt{at}(-) \implies \exists r_E \in \mathsf{envMove}(r_{\mathrm{pre}}, \ell, -).\ r_E[\ell] = r'[\ell]$
$\mathbb{U}(\ell, -, -)$	$r_{\mathrm{pre}}[\ell] = \mathtt{at}(-) \implies r'[\ell] \equiv r_{\mathrm{pre}}[\ell]$
otherwise	always

B.2.5 Local safety

$r \in \mathsf{LSafe}_0(e, \Phi) \triangleq$ always

$r \in \mathsf{LSafe}_{n+1}(e, \Phi) \triangleq$

If $e \in \mathit{Val}$ then $r \Rrightarrow \Phi(e)$

If $e = K[\mathtt{fork}\ e']$ then $r \in \mathsf{LSafe}_n(K[0], \Phi) * \mathsf{LSafe}_n(e', \mathsf{true})$

If $e \xrightarrow{\alpha} e'$ then $\forall r_F \# r.\ \forall r_{\mathrm{pre}} \in \mathsf{rely}(r \oplus r_F, \alpha).\ \exists \mathcal{P}.\ r_{\mathrm{pre}} \Rrightarrow \mathcal{P}$ and

$\forall r' \in \mathcal{P}.\ (r_{\mathrm{pre}}, r') \in \mathsf{wpe}(\alpha) \implies \exists r_{\mathrm{post}}.\ (r_{\mathrm{post}} \oplus r_F, -) \in \mathsf{guar}(r_{\mathrm{pre}}, r', \alpha),\ r_{\mathrm{post}} \in \mathsf{LSafe}_n(e', \Phi)$

B.3 Global safety

B.3.1 Domains

$$
\begin{array}{rcl}
\mathit{Tag}(G) & ::= & \{(\mathtt{sb}, a, b) \mid (a,b) \in G.\mathsf{sb} \vee b = \bot\} \cup \{(\mathtt{esc}, a, b) \mid (a,b) \in G.\mathsf{hb} \vee b = \bot\} \\
 & \cup & \{(\mathtt{rf}, a, b) \mid G.\mathsf{rf}(a) = b \vee b = \bot\} \cup \{(\mathtt{cond}, a, \bot)\} \\
\mathcal{L} \in \mathit{Labeling}(G) & \triangleq & \mathit{Tag}(G) \rightarrow \mathit{Resource} \\
\mathcal{I} \in \mathit{EscrowIntros} & \triangleq & \wp_{\text{fin}}(\mathit{EscrowTy} \times \mathit{Resource}) \\
\mathcal{T} \in \mathit{IThreadMap} & \triangleq & \mathbb{N} \xrightarrow{\text{fin}} (\mathit{ActName} \times \mathit{Exp} \times \mathit{Resource} \times (\mathit{Val} \rightarrow \mathit{Prop}))
\end{array}
$$

B.3.2 Valid ghost moves

$r \Rrightarrow_{\mathcal{I}} r' \;\triangleq\; \exists g.\ r' = (r.\Pi,\ g,\ r.\Sigma \cup \{\sigma \mid (\sigma, -) \in \mathcal{I}\})$

B.3.3 Valid edge labels for a node

$$
\begin{array}{l}
b \in \mathsf{valid}(G, \mathcal{L}, N) \triangleq \exists r, \mathcal{I}. \\
\quad \mathcal{L} \in \mathit{Labeling}(G) \\
\quad \mathsf{in}(\mathtt{sb}) \oplus \mathsf{in}(\mathtt{rf}) \oplus \mathsf{in}(\mathtt{esc}) \Rrightarrow_{\mathcal{I}} r \oplus \mathsf{out}(\mathtt{esc}) \oplus \mathsf{out}(\mathtt{cond}) \\
\quad (\mathsf{out}(\mathtt{sb}), \mathsf{out}(\mathtt{rf})) \in \mathsf{guar}(\mathsf{in}(\mathtt{sb}) \oplus \mathsf{in}(\mathtt{rf}), r, \alpha) \\
\quad (\forall c \in N.\ \mathsf{isUpd}(c) \wedge \mathsf{rf}(c) = b \implies \mathcal{L}(\mathtt{rf}, b, c) = \mathsf{out}(\mathtt{rf})) \\
\quad (\nexists c \in N.\ \mathsf{isUpd}(c) \wedge \mathsf{rf}(c) = b) \implies \mathcal{L}(\mathtt{rf}, b, \bot) = \mathsf{out}(\mathtt{rf}) \\
\quad |\mathcal{L}(\mathtt{rf}, b, \bot)| = |\mathsf{out}(\mathtt{rf})| \\
\quad \forall (\sigma, r_E) \in \mathcal{I}.\ \mathsf{interp}(\sigma) = (\mathcal{Q}, \mathcal{Q}') \implies r_E \in \mathcal{Q}' \\
\quad \mathcal{L}(\mathtt{esc}, b, \bot) = \bigoplus \left\{ r_E \;\middle|\; \begin{array}{l} (\sigma, r_E) \in \mathcal{I},\ \mathsf{interp}(\sigma) = (\mathcal{P}, \mathcal{P}'), \\ r_E \in \mathcal{P}',\ (\nexists c.\ \mathsf{hb}^{=}(b, c) \wedge \mathcal{L}(\mathtt{cond}, c, \bot) \in \mathcal{P}) \end{array} \right\}
\end{array}
$$

where

$$
\begin{array}{l}
\quad G \;=\; (A, \mathsf{sb}, \mathsf{mo}, \mathsf{rf}) \\
\quad \alpha \;\triangleq\; A(b) \\
\quad \mathsf{in}(x) \quad \triangleq\; \bigoplus \{\mathcal{L}(x, a, b) \mid (x, a, b) \in \mathrm{dom}(\mathcal{L})\} \\
\quad \mathsf{out}(x) \;\triangleq\; \bigoplus \{\mathcal{L}(x, b, c) \mid (x, b, c) \in \mathrm{dom}(\mathcal{L})\}
\end{array}
$$

B.3.4 Concurrent compatibility

$\mathsf{compat}(G, \mathcal{L}) \triangleq \forall \mathcal{E} \subseteq \mathrm{dom}(\mathcal{L}).\ \ \mathsf{tgt}(\mathcal{E}); G.\mathsf{hb}^* \pitchfork \mathsf{src}(\mathcal{E}) \implies \bigoplus_{\eta \in \mathcal{E}} \mathcal{L}(\eta)$ defined

B.3.5 Protocol conformance

$\mathsf{conform}(G, \mathcal{L}, N) \triangleq \forall \ell.\ \forall a, b \in N.\ \ G.\mathsf{mo}_{\ell,\mathtt{at}}(a, b) \implies \mathsf{out}(\mathcal{L}, a, \mathtt{rf})[\ell] \sqsubseteq_{\mathtt{at}} \mathsf{out}(\mathcal{L}, b, \mathtt{rf})[\ell]$

B.3.6 Global safety

$$
\begin{array}{l}
\mathsf{GSafe}_n(\mathcal{T}, G, \mathcal{L}) \triangleq \\
\quad \mathsf{valid}(G, \mathcal{L}, N) = N \\
\quad \mathsf{compat}(G, \mathcal{L}) \\
\quad \mathsf{conform}(G, \mathcal{L}, N) \\
\quad \forall a \in N.\ \mathcal{L}(\mathtt{sb}, a, \bot) = \bigoplus \{r \mid \exists i.\ \mathcal{T}(i) = (a, -, r, -)\} \\
\quad \forall i.\ \mathcal{T}(i) = (a, e, r, \Phi) \implies r \in \mathsf{LSafe}_n(e, \Phi) \\
\text{where } N \triangleq \mathrm{dom}(G.A)
\end{array}
$$

B.4 Syntax and semantics

B.4.1 Parameters

We assume:

- A syntax of states and PCM terms, with appropriate sorting rules, as part of the term syntax given below.
- A term interpretation function $[\![t]\!]^\rho$ for state and PCM terms.

B.4.2 Syntax

$$
\begin{array}{llll}
\text{Sort} & \theta & ::= & \mathsf{Val} \mid \mathsf{State} \mid \mathsf{PCM}_\mu \\
\text{Var} & X & ::= & \ell \mid x \mid s \\
\text{Term} & t & ::= & X \mid n \mid \varepsilon_\mu \mid t \cdot_\mu t \mid \cdots \\
\text{Proposition} & P & ::= & t = t \mid P \wedge P \mid P \vee P \mid P \Rightarrow P \mid \forall X : \theta.\, P \mid \exists X : \theta.\, P \\
 & & \mid & \Box P \mid P * P \mid \mathsf{uninit}(t) \mid t \hookrightarrow t \mid \boxed{t : t \mid \tau} \mid t \sqsubseteq_\tau t \mid \boxed{t : t \mid \mu} \mid [\sigma]
\end{array}
$$

B.4.3 Proposition semantics

R	$r \in [\![R]\!]^\rho$ iff
$t = t'$	$[\![t]\!]^\rho = [\![t']\!]^\rho$
$t \sqsubseteq_\tau t'$	$[\![t]\!]^\rho \sqsubseteq_\tau [\![t']\!]^\rho$
$P \wedge Q$	$r \in [\![P]\!]^\rho \cap [\![Q]\!]^\rho$
$P \vee Q$	$r \in [\![P]\!]^\rho \cup [\![Q]\!]^\rho$
$P \Rightarrow Q$	$\lfloor r \rfloor \cap [\![P]\!]^\rho \subseteq [\![Q]\!]^\rho$
$\forall X.\, P$	$r \in \bigcap_{d \in \mathsf{sort}(X)} [\![P]\!]^{\rho[X \mapsto d]}$
$\exists X.\, P$	$r \in \bigcup_{d \in \mathsf{sort}(X)} [\![P]\!]^{\rho[X \mapsto d]}$

R	$r \in [\![R]\!]^\rho$ iff
$\Box P$	$\lvert r \rvert \in [\![P]\!]^\rho$
$P_1 * P_2$	$r \in [\![P_1]\!]^\rho * [\![P_2]\!]^\rho$
$\mathsf{uninit}(t)$	$r.\Pi([\![t]\!]^\rho) = \mathsf{uninit}$
$t \hookrightarrow t'$	$r.\Pi([\![t]\!]^\rho) = \mathtt{na}([\![t']\!]^\rho)$
$\boxed{t : t' \mid \tau}$	$r.\Pi([\![t]\!]^\rho) \geq \mathtt{at}(\tau, \{[\![t']\!]^\rho\})$
$\boxed{t : t' \mid \mu}$	$r.g(\mu)([\![t]\!]^\rho) \geq [\![t']\!]^\rho$
$[\sigma]$	$\sigma \in r.\Sigma$

B.4.4 Ghost move semantics

$$\rho \models P \Rrightarrow Q \quad \triangleq \quad \forall r \in [\![P]\!]^\rho \,.\, r \Rrightarrow [\![Q]\!]^\rho$$

B.4.5 Hoare triple semantics

$$\mathsf{LSafe}(e, \Phi) \quad \triangleq \quad \bigcap_n \mathsf{LSafe}_n(e, \varphi)$$

$$\rho \models \{P\}\, e\, \{x.\, Q\} \quad \triangleq \quad r \in [\![P]\!]^\rho \,.\, r \Rrightarrow \mathsf{LSafe}(e, \lambda V.\ [\![Q]\!]^\rho\, [x \mapsto v])$$

B.5 Proof theory

B.5.1 Necessitation

$$\Box P \Rightarrow P \qquad \Box P \Rightarrow \Box\Box P \qquad \Box P * Q \Leftrightarrow \Box P \wedge Q \qquad t = t' \Rightarrow \Box t = t' \qquad \boxed{t : t' \mid \tau} \Rightarrow \Box\, \boxed{t : t' \mid \tau} \qquad \frac{t \cdot_\mu t = t}{\boxed{\gamma : t \mid \mu} \Rightarrow \Box\, \boxed{\gamma : t \mid \mu}}$$

$$[\sigma] \Rightarrow \Box[\sigma]$$

B.5.2 Separation

$$\boxed{\gamma : t \mid \mu} * \boxed{\gamma : t' \mid \mu} \Leftrightarrow \boxed{\gamma : t \cdot_\mu t' \mid \mu} \qquad\qquad \boxed{\ell : s \mid \tau} * \boxed{\ell : s' \mid \tau'} \Rightarrow \tau = \tau' \wedge (s \sqsubseteq_\tau s' \vee s' \sqsubseteq_\tau s)$$

B.5.3 Ghost moves

$$\frac{P \Rightarrow Q}{P \Rrightarrow Q} \qquad \frac{P \Rrightarrow Q}{P * R \Rrightarrow Q * R} \qquad \frac{P \Rrightarrow Q \qquad Q \Rrightarrow R}{P \Rrightarrow R} \qquad \frac{\sigma : P \rightsquigarrow Q}{Q \Rrightarrow [\sigma]} \qquad \frac{\sigma : P \rightsquigarrow Q}{P * [\sigma] \Rrightarrow Q} \qquad \frac{P_1 \Rrightarrow Q \qquad P_2 \Rrightarrow Q}{P_1 \vee P_2 \Rrightarrow Q}$$

$$\frac{P \Rrightarrow Q}{\exists X. P \Rrightarrow Q} \qquad \mathsf{true} \Rrightarrow \exists \gamma.\, \boxed{\gamma : t \mid \mu} \qquad \frac{\forall t_F : \mathsf{PCM}_\mu.\ t_1 \#_\mu t_F \Rightarrow t_2 \#_\mu t_F}{\boxed{\gamma : t_1 \mid \mu} \Rrightarrow \boxed{\gamma : t_2 \mid \mu}}$$

B.5.4 Hoare logic

Allocation

$$\{\mathsf{true}\}\ \mathtt{alloc}(n)\ \{x.\ x \neq 0 * \mathsf{uninit}(x) * \cdots * \mathsf{uninit}(x + n - 1)\}$$

Acquire/release protocol rules

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

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

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

$$\frac{\forall s' \sqsupseteq_\tau s.\ \forall z.\ \tau(s', z) * P \Rrightarrow \exists s'' \sqsupseteq_\tau s'.\ \tau(s'', (z+1) \bmod \mathbf{C}) * Q}{\left\{\boxed{\ell : s \mid \tau} * P\right\}\ \mathtt{FAI}(\ell)\ \left\{z.\ \exists s''.\ \boxed{\ell : s'' \mid \tau} * Q\right\}}$$

Nonatomics

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

Structural rules

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

Axioms for pure reductions

$$\begin{array}{lcl}\{\mathsf{true}\} & v & \{x.\ x = v\} \\ \{\mathsf{true}\} & v \;\mathtt{+}\; v' & \{x.\ x = v + v'\} \\ \{\mathsf{true}\} & v \;\mathtt{==}\; v' & \{x.\ x = 1 \Leftrightarrow v = v'\}\end{array} \qquad \frac{\begin{array}{c}\{P * v \neq 0\}\, e_1\, \{x.\ Q\} \\ \{P * v = 0\}\, e_2\, \{x.\ Q\}\end{array}}{\{P\}\ \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\ \{x.\ Q\}} \qquad \frac{\{P\}\, e\, \{x.\ Q\} \qquad \forall x.\ \{Q\}\, e'\, \{y.\ R\}}{\{P\}\ \mathtt{let}\ x = e\ \mathtt{in}\ e'\ \{y.\ R\}}$$

$$\frac{\{P\}\, e\, \{\mathsf{true}\}}{\{P\}\ \mathtt{fork}\ e\ \{\mathsf{true}\}} \qquad \frac{\{P\}\, e\, \{x.\ (x = 0 * P) \vee (x \neq 0 * Q)\}}{\{P\}\ \mathtt{repeat}\ e\ \mathtt{end}\ \{x.\ Q\}}$$

C. Metatheory

The metatheory of GPS has been formalized in Coq and mechanically checked. The `README.txt` file explains the contents of the various Coq files. Below we report on the main conceptual effort of the soundness proof: finding a decomposition of global soundness into a sequence of lemmas.

C.1 Basic properties of semantics domains and ghost moves

This subsection gives a few of the basic lemmas about our semantic domains and ghost moves. All of these claims, and many others besides, have been formalized and mechanically checked in `GpsModel.v`, `GpsModelLemmas.v`, and `GpsLogic.v`.

C.1.1 Resources

Lemma 1. Protocols and resources form partial commutative monoids.

Corollary 1. If $(r \oplus r')\#r''$ then $r\#r''$.

Lemma 2. If $r\#r'$ then $|r \oplus r'| = |r| \oplus |r'|$.

Lemma 3. $||r|| = |r|$.

Lemma 4. $r = r \oplus |r|$.

Lemma 5. $|r| = |r| \oplus |r|$.

C.1.2 Propositions

Lemma 6. $[\![P]\!]^\rho \in \mathit{Prop}$.

Lemma 7. *Prop* forms a BI algebra.

Proof. Follows immediately from Lemma 1. □

C.1.3 Protocols

Lemma 8. If $\pi \sqsubseteq_{\mathtt{at}} \pi' \sqsubseteq_{\mathtt{at}} \pi''$ then $\pi \sqsubseteq_{\mathtt{at}} \pi''$.

Lemma 9. If $\pi\#\pi_F$ then $\pi \sqsubseteq_{\mathtt{at}} (\pi \oplus \pi_F)$.

Lemma 10. If $\pi \oplus \pi' = \mathtt{at}(\tau, S)$ then either $\pi \equiv_{\mathtt{at}} \mathtt{at}(\tau, S)$ or $\pi' \equiv_{\mathtt{at}} \mathtt{at}(\tau, S)$.

C.1.4 Ghost moves

Lemma 11. $r \Rrightarrow \lfloor r \rfloor$.

Lemma 12. If $r \Rrightarrow \mathcal{P}$ and $r_F\#r$ then $r \oplus r_F \Rrightarrow \mathcal{P} * \lfloor r_F \rfloor$.

C.2 Proof rules: local soundness

Theorem 3 (Local safety for ghost moves)**.** If $P \Rrightarrow Q$ then for all closing ρ we have $\rho \models P \Rrightarrow Q$.

Proof. Checked entirely in Coq; see `GpsLogic.v`. □

Theorem 4 (Local safety for Hoare triples)**.** If $\{P\}\, e\, \{x.\ Q\}$ then for all closing ρ we have $\rho \models \{P\}\, e\, \{x.\ Q\}$

Proof. Checked entirely in Coq; see `GpsLogic.v`. □

C.3 Global soundness

The proof of global soundness breaks into two pieces:

- First, we prove a sequence of easy "visibility" lemmas: given that we know global safety for a graph constructed so far, resource knowledge at any point in the graph connects to some associated fact about happens-before visibility. For example, if a given node b claims to know that an atomic location ℓ is in a particular state s, then there must be some node a such that $\mathsf{hb}(a, b)$ and a wrote to ℓ while moving to s. These lemmas are given in section C.3.1.
- Second, we prove a sequence of lemmas that lead to the main instrumented step theorem (Theorem 5). Each of these lemmas accounts for a piece of the definition of local safety: rely (§C.3.2), ghost moves (§C.3.3), protocol equivalence for writes (§C.3.4) and guarantee (§C.3.5). We set up ("prepare") for taking a step with a final lemma in section C.3.6. Each of these lemmas takes an existing labeling of the graph and transforms it by labelling the associated edges:
  - Step preparation: labels the incoming sb edge
  - Rely: labels any incoming rf edges
  - Ghost: labels hb edges for all escrow-related activity, including transferring the escrowed resource and consuming the escrow condition
  - Guarantee: labels the outgoing sb and rf edges.

  Outgoing resources from a new node initially go to a sink node $\bot$. These resources are subsequently moved to label *e.g.,* rf edges as further nodes are added to the graph.

C.3.1 Visibility

Definition 1. We say that a set of nodes $N \subseteq \mathrm{dom}(G.A)$ is *G-prefix closed* (written $N \in \mathsf{prefix}(G)$) if $\forall b \in N.\ \forall a.\ G.\mathsf{hb}(a, b) \implies a \in N$.

Lemma 13 (Allocation visibility)**.** If

$\mathsf{consistentC11}(G)$
$N \in \mathsf{prefix}(G)$
$N \cup \{b\} \in \mathsf{prefix}(G)$
$N \subseteq \mathsf{valid}(G, \mathcal{L}, N')$
$\mathsf{in}(\mathcal{L}, b, \mathsf{all})[\ell] \neq \bot$

then $\exists a.$

$G.\mathsf{hb}(a, b)$
$G.A(a) = \mathbb{A}(\vec{\ell})$
$\ell \in \vec{\ell}$

Proof. See `visible_allocation` in `GpsVisible.v`. □

Lemma 14 (Nonatomic protocol visibility)**.** If

$$\begin{array}{l} \mathsf{consistentC11}(G) \\ N \in \mathsf{prefix}(G) \\ N \cup \{b\} \in \mathsf{prefix}(G) \\ N \subseteq \mathsf{valid}(G, \mathcal{L}, N') \\ \mathsf{compat}(G, \mathcal{L}) \\ \mathsf{in}(\mathcal{L}, b, \mathsf{all})[\ell] = \mathtt{na}(V) \end{array}$$

then $\exists a$.

$$\begin{array}{l} G.\mathsf{hb}(a, b) \\ G.A(a) = \mathbb{W}(\ell, V, \mathtt{na}) \\ \forall a' \in N.\ G.A(a') = \mathbb{W}(\ell, -, -) \implies (G.\mathsf{hb}^{=}(b, a') \vee G.\mathsf{hb}^{=}(a', a)) \end{array}$$

Proof. See `visible_na` in `GpsVisible.v`. □

Lemma 15 (Uninitialized visibility)**.** If

$$\begin{array}{l} \mathsf{consistentC11}(G) \\ N \in \mathsf{prefix}(G) \\ N \cup \{b\} \in \mathsf{prefix}(G) \\ N \subseteq \mathsf{valid}(G, \mathcal{L}, N') \\ \mathsf{compat}(G, \mathcal{L}) \\ \mathsf{in}(\mathcal{L}, b, \mathsf{all})[\ell] = \mathsf{uninit} \\ a \in N \\ G.A(a) = \mathbb{W}(\ell, -, -) \end{array}$$

then

$$G.\mathsf{hb}^{=}(b, a)$$

Proof. See `visible_uninit` in `GpsVisible.v`. □

Lemma 16 (Atomic protocol visibility)**.** If

$$\begin{array}{l} \mathsf{consistentC11}(G) \\ N \in \mathsf{prefix}(G) \\ N \cup \{b\} \in \mathsf{prefix}(G) \\ N \subseteq \mathsf{valid}(G, \mathcal{L}, N') \\ \mathsf{compat}(G, \mathcal{L}) \\ \mathsf{in}(\mathcal{L}, b, \mathsf{all})[\ell] = \mathtt{at}(-) \end{array}$$

then $\exists a, S'$.

$$\begin{array}{l} G.\mathsf{hb}(a, b) \\ \mathsf{writes}(G.A(a), \ell, -) \\ \mathsf{isAtomic}(a) \\ \mathsf{out}(\mathcal{L}, a, \mathtt{sb})[\ell] \equiv_{\mathtt{at}} \mathsf{in}(\mathcal{L}, b, \mathsf{all})[\ell] \end{array}$$

Proof. See `visible_atomic` in `GpsVisible.v`. □

Lemma 17 (Escrow visibility)**.** If

$\mathsf{consistentC11}(G)$
$N \in \mathsf{prefix}(G)$
$N \cup \{b\} \in \mathsf{prefix}(G)$
$N \subseteq \mathsf{valid}(G, \mathcal{L}, N')$
$\mathsf{compat}(G, \mathcal{L})$
$\sigma \in \mathsf{in}(\mathcal{L}, b, \mathsf{all}).\Sigma$

then $\exists a.$

$G.\mathsf{hb}(a, b)$
$\sigma \notin \mathsf{in}(\mathcal{L}, a, \mathsf{all}).\Sigma$
$\sigma \in \mathsf{out}(\mathcal{L}, b, \mathsf{all}).\Sigma$

Proof. See `visible_escrow` in `GpsVisible.v`. □

C.3.2 Rely

Lemma 18 (Rely step)**.** If

$G.A(a) = \alpha$
$\mathrm{dom}(G.A) = N \uplus \{a\}$
$N \in \mathsf{prefix}(G)$
$N \subseteq \mathsf{valid}(G, \mathcal{L}, N)$
$\mathsf{in}(\mathcal{L}, a, \mathsf{all}) = \mathsf{out}(\mathcal{L}, a, \mathsf{all}) = \mathsf{emp}$
$\mathsf{compat}(G, \mathcal{L})$
$\mathsf{conform}(G, \mathcal{L}, N)$
$\mathsf{consistentC11}(G)$
$\mathsf{in}(\mathcal{L}, a, \mathtt{rf}) = \mathsf{emp}$
$\mathsf{in}(\mathcal{L}, a, \mathtt{esc}) = \mathsf{emp}$
$\mathsf{out}(\mathcal{L}, a, \mathsf{all}) = \mathsf{emp}$

then $\exists \mathcal{L}'$.

$N \subseteq \mathsf{valid}(G, \mathcal{L}', \mathrm{dom}(G.A))$
$\mathsf{compat}(G, \mathcal{L}')$
$\mathsf{conform}(G, \mathcal{L}', N)$
$\mathsf{in}(\mathcal{L}', a, \mathtt{sb}) \oplus \mathsf{in}(\mathcal{L}', a, \mathtt{rf}) \in \mathsf{rely}(\mathsf{in}(\mathcal{L}', a, \mathtt{sb}), \alpha)$
$\mathsf{in}(\mathcal{L}', a, \mathtt{esc}) = \mathsf{out}(\mathcal{L}', a, \mathsf{all}) = \mathsf{emp}$
$\forall b, c.\ \mathcal{L}'(\mathtt{sb}, b, c) = \mathcal{L}(\mathtt{sb}, b, c)$
$\forall b.\ \mathcal{L}'(\mathtt{sb}, b, \bot) = \mathcal{L}(\mathtt{sb}, b, \bot)$

Proof. See `rely_step` in `GpsRelyGhost.v`. ☐

C.3.3 Ghost moves

Lemma 19 (Ghost step)**.** If

$$\begin{array}{l}
\mathrm{dom}(G.A) = N \uplus \{a\} \\
\mathsf{consistentC11}(G) \\
N \in \mathsf{prefix}(G) \\
N \subseteq \mathsf{valid}(G, \mathcal{L}, \mathrm{dom}(G.A)) \\
\mathsf{compat}(G, \mathcal{L}[(\mathtt{esc}, a, \bot) := \mathcal{L}(\mathtt{esc}, a, \bot) \oplus r]) \\
\mathsf{conform}(G, \mathcal{L}, N) \\
r_{\text{before}} \triangleq \mathsf{in}(\mathcal{L}, a, \mathtt{sb}) \oplus \mathsf{in}(\mathcal{L}, a, \mathtt{rf}) \oplus \mathsf{in}(\mathcal{L}, a, \mathtt{esc}) \\
r_{\text{after}} \triangleq r \oplus \mathsf{out}(\mathcal{L}, a, \mathtt{esc}) \oplus \mathsf{out}(\mathcal{L}, a, \mathtt{cond}) \\
r_{\text{before}} \Rrightarrow_{\mathcal{I}} r_{\text{after}} \\
|r_0| \leq r \\
\forall c.\ \mathcal{L}(\mathtt{esc}, a, c) = \mathsf{emp} \\
\forall (\sigma, r_E) \in \mathcal{I}.\ \mathsf{interp}(\sigma) = (\mathcal{Q}, \mathcal{Q}') \implies r_E \in \mathcal{Q}' \\
\mathcal{L}(\mathtt{esc}, a, \bot) = \bigoplus \left\{ r_E \ \middle|\ \begin{array}{l} \mathsf{interp}(\sigma) = (\mathcal{Q}, \mathcal{Q}'),\ r_E \in \mathcal{Q}', \\ (\nexists b.\ \mathsf{hb}^{=}(a, b) \wedge \mathcal{L}(\mathtt{cond}, b, \bot) \in \mathcal{Q}) \end{array} \right\}
\end{array}$$

then $\exists \mathcal{L}', \mathcal{I}', r', r'_{\text{before}}, r'_{\text{after}} \in \mathcal{P}$.

$$\begin{array}{l}
N \subseteq \mathsf{valid}(G, \mathcal{L}', \mathrm{dom}(G.A)) \\
\mathsf{compat}(G, \mathcal{L}'[(\mathtt{esc}, a, \bot) := \mathcal{L}'(\mathtt{esc}, a, \bot) \oplus r']) \\
\mathsf{conform}(G, \mathcal{L}', N) \\
r'_{\text{before}} \triangleq \mathsf{in}(\mathcal{L}', a, \mathtt{sb}) \oplus \mathsf{in}(\mathcal{L}', a, \mathtt{rf}) \oplus \mathsf{in}(\mathcal{L}', a, \mathtt{esc}) \\
r'_{\text{after}} \triangleq r' \oplus \mathsf{out}(\mathcal{L}', a, \mathtt{esc}) \oplus \mathsf{out}(\mathcal{L}', a, \mathtt{cond}) \\
r'_{\text{before}} \Rrightarrow_{\mathcal{I}'} r'_{\text{after}} \\
\forall b.\ \mathcal{L}'(\mathtt{sb}, b, \bot) = \mathcal{L}(\mathtt{sb}, b, \bot) \\
\forall b.\ \mathcal{L}'(\mathtt{rf}, b, \bot) = \mathcal{L}(\mathtt{rf}, b, \bot) \\
\forall b, c.\ \mathcal{L}'(\mathtt{sb}, b, c) = \mathcal{L}(\mathtt{sb}, b, c) \\
\forall b, c.\ \mathcal{L}'(\mathtt{rf}, b, c) = \mathcal{L}(\mathtt{rf}, b, c) \\
\forall c.\ \mathcal{L}(\mathtt{esc}, a, c) = \mathsf{emp} \\
\forall (\sigma, r_E) \in \mathcal{I}'.\ \mathsf{interp}(\sigma) = (\mathcal{Q}, \mathcal{Q}') \implies r_E \in \mathcal{Q}' \\
\mathcal{L}(\mathtt{esc}, a, \bot) = \bigoplus \left\{ r_E \ \middle|\ \begin{array}{l} (\sigma, r_E) \in \mathcal{I}',\ \mathsf{interp}(\sigma) = (\mathcal{Q}, \mathcal{Q}'),\ r_E \in \mathcal{Q}', \\ (\nexists b.\ \mathsf{hb}^{=}(a, b) \wedge \mathcal{L}(\mathtt{cond}, b, \bot) \in \mathcal{Q}) \end{array} \right\}
\end{array}$$

Proof. See `ghost_step` in `GpsRelyGhost.v`. □

C.3.4 Protocol equivalence for writes

Lemma 20 (Protocol equivalence for writes)**.** If

$$\begin{array}{l}
\mathrm{dom}(G.A) = N \uplus \{a\} \\
\mathsf{consistentC11}(G) \\
N \in \mathsf{prefix}(G) \\
N \subseteq \mathsf{valid}(G, \mathcal{L}, \mathrm{dom}(G.A)) \\
\mathsf{in}(\mathcal{L}', a, \mathtt{sb}) \oplus \mathsf{in}(\mathcal{L}', a, \mathtt{rf}) \in \mathsf{rely}(\mathsf{in}(\mathcal{L}', a, \mathtt{sb}), \alpha) \\
\mathsf{compat}(G, \mathcal{L}[(\mathtt{esc}, a, \bot) := \mathcal{L}(\mathtt{esc}, a, \bot) \oplus r]) \\
\mathsf{conform}(G, \mathcal{L}, N) \\
\mathsf{in}(\mathcal{L}, a, \mathsf{all}) \Rrightarrow_{\mathcal{I}} r \oplus \mathsf{out}(\mathcal{L}, a, \mathtt{esc}) \oplus \mathsf{out}(\mathcal{L}, a, \mathtt{cond}) \\
|\mathsf{in}(\mathcal{L}, a, \mathtt{sb}) \oplus \mathsf{in}(\mathcal{L}, a, \mathtt{rf})| \leq r
\end{array}$$

then

$$(\mathsf{in}(\mathcal{L}, a, \mathtt{sb}) \oplus \mathsf{in}(\mathcal{L}, a, \mathtt{rf}), \mathsf{in}(\mathcal{L}, a, \mathsf{all})) \in \mathsf{wpe}(G.A(a))$$

Proof. See pwpe in GpsGuarPrep.v and wpe in GpsGuarPrep.v. ☐

C.3.5 Guarantee

Lemma 21 (Guar step)**.** If

$$\begin{array}{l}
\alpha = G.A(a) \\
\mathrm{dom}(G.A) = N \uplus \{a\} \\
N \in \mathsf{prefix}(G) \\
N \subseteq \mathsf{valid}(G, \mathcal{L}, \mathrm{dom}(G.A)) \\
\mathsf{consistentC11}(G) \\
\mathsf{compat}(G, \mathcal{L}[(\mathtt{esc}, a, \bot) := \mathcal{L}(\mathtt{esc}, a, \bot) \oplus r) \\
\mathsf{conform}(G, \mathcal{L}, N) \\
r_{\mathrm{pre}} = \mathsf{in}(\mathcal{L}, a, \mathtt{sb}) \oplus \mathsf{in}(\mathcal{L}, a, \mathtt{rf}) \\
r_{\mathrm{pre}} \in \mathsf{rely}(-, \alpha) \\
\mathsf{in}(\mathcal{L}, a, \mathsf{all}) \Rrightarrow_{\mathcal{I}} r \oplus \mathsf{out}(\mathcal{L}, a, \mathtt{esc}) \oplus \mathsf{out}(\mathcal{L}, a, \mathtt{cond}) \\
\forall (\sigma, r_E) \in \mathcal{I}'.\ \mathsf{interp}(\sigma) = (\mathcal{Q}, \mathcal{Q}') \implies r_E \in \mathcal{Q}' \\
\mathcal{L}(\mathtt{esc}, a, \bot) = \bigoplus \left\{ r_E \;\middle|\; \begin{array}{l} (\sigma, r_E) \in \mathcal{I}',\ \mathsf{interp}(\sigma) = (\mathcal{Q}, \mathcal{Q}'),\ r_E \in \mathcal{Q}', \\ (\nexists b.\ \mathsf{hb}^{=}(a, b) \wedge \mathcal{L}(\mathtt{cond}, b, \bot) \in \mathcal{Q}) \end{array} \right\} \\
|\mathsf{in}(\mathcal{L}, a, \mathsf{all})| \leq r \\
\mathsf{out}(\mathcal{L}, b, \mathtt{rf}) = \mathsf{emp} \\
\mathsf{out}(\mathcal{L}, b, \mathtt{sb}) = \mathsf{emp} \\
(r_{\mathsf{sb}}, r_{\mathsf{rf}}) \in \mathsf{guar}(r_{\mathrm{pre}}, r, \alpha) \\
\mathsf{wpe}(\alpha, r_{\mathrm{pre}}, \mathsf{in}(\mathcal{L}, a, \mathsf{all}))
\end{array}$$

then $\exists \mathcal{L}'$.

$$\begin{array}{l}
\mathrm{dom}(G.A) = \mathsf{valid}(G, \mathcal{L}', \mathrm{dom}(G.A)) \\
\mathsf{compat}(G, \mathcal{L}') \\
\mathsf{conform}(G, \mathcal{L}', \mathrm{dom}(G.A)) \\
\forall b \neq a.\ \mathcal{L}'(\mathtt{sb}, b, \bot) = \mathcal{L}(\mathtt{sb}, b, \bot) \\
\mathcal{L}'(\mathtt{sb}, a, \bot) = r_{\mathsf{sb}}
\end{array}$$

Proof. See guar_step in GpsGuarPrep.v. ☐

C.3.6 Step preparation

Lemma 22 (Step preparation)**.** If

$$\begin{array}{l}
\mathsf{consistentC11}(G) \\
\mathsf{consistentC11}(G') \\
\mathrm{dom}(G'.A) = \mathrm{dom}(G.A) \uplus \{b\} \\
\mathcal{L}(\mathtt{sb}, a, \bot) = r \oplus r_{\mathrm{rem}} \\
\mathrm{dom}(G) \subseteq \mathsf{valid}(G', \mathcal{L}, \mathrm{dom}(G.A)) \\
\mathsf{compat}(G, \mathcal{L}) \\
\mathsf{conform}(G, \mathcal{L}, \mathrm{dom}(G)) \\
\forall c \in \mathrm{dom}(G.A).\ G.A(c) = G'.A(c) \\
G'.\mathsf{sb} = G.\mathsf{sb} \uplus \{[a, b)\} \\
\forall c \in \mathrm{dom}(G.A).\ G.\mathsf{rf}(c) = G'.\mathsf{rf}(c) \\
G'.\mathsf{mo} \supseteq G.\mathsf{mo}
\end{array}$$

then $\exists \mathcal{L}'.$

$$\begin{array}{l}
\mathsf{valid}(G', \mathcal{L}', \mathrm{dom}(G.A)) = \mathrm{dom}(G) \\
\mathsf{compat}(G', \mathcal{L}') \\
\mathsf{conform}(G', \mathcal{L}', \mathrm{dom}(G')) \\
\mathcal{L}'(\mathtt{sb}, a, \bot) = r_{\mathrm{rem}} \\
\mathsf{in}(\mathcal{L}', b, \mathtt{sb}) = r \\
\mathsf{in}(\mathcal{L}', b, \mathtt{rf}) = \mathsf{emp} \\
\mathsf{in}(\mathcal{L}', b, \mathtt{esc}) = \mathsf{emp} \\
\forall a' \neq a.\ \mathcal{L}'(\mathtt{sb}, a', b) = \mathsf{emp} \\
\mathsf{out}(\mathcal{L}', b, \mathsf{all}) = \mathsf{emp} \\
\forall a' \neq a.\ \mathcal{L}'(\mathtt{sb}, a', \bot) = \mathcal{L}(\mathtt{sb}, a', \bot)
\end{array}$$

Proof. See `prepare_step` in `GpsGuarPrep.v`. □

C.3.7 Instrumented execution and adequacy

We define functions erase : *IThreadMap* → *ThreadMap* and post : *IThreadMap* → $\mathbb{N} \xrightarrow{\text{fin}}$ *Val* → *Prop* as follows:

$$\begin{array}{rcl}
\mathsf{erase}(\mathcal{T}) & \triangleq & \lambda i.\ (a, e) \text{ if } \mathcal{T}(i) = (a, e, -, -) \\
\mathsf{post}(\mathcal{T}) & \triangleq & \lambda i.\ \mathcal{P} \text{ if } \mathcal{T}(i) = (-, -, -, \mathcal{P})
\end{array}$$

Theorem 5 (Instrumented execution)**.** If $\mathsf{GSafe}_{n+1}(\mathcal{T}, G, \mathcal{L}, \mathrm{dom}(G.A))$ and $\langle \mathsf{erase}(\mathcal{T}); G\rangle \longrightarrow \langle T'; G'\rangle$ then there exist $\mathcal{T}', \mathcal{L}'$ such that $\mathsf{erase}(\mathcal{T}') = T'$ and $\mathsf{GSafe}_n(\mathcal{T}', G', \mathcal{L}', \mathrm{dom}(G'.A))$ and $\mathsf{post}(\mathcal{T}) \subseteq \mathsf{post}(\mathcal{T}')$.

Proof. See `gsafe_pres` in `GpsAdequate.v`. □

Theorem 6 (Adequacy)**.** If e is closed and $\{\mathsf{true}\}\, e\, \{x.\ P\}$ then $[\![e]\!] \subseteq \{V \mid [\![P]\!]^{[x \mapsto V]} \neq \emptyset\}$.

Proof. See `adequacy` in `GpsAdequate.v`. □

D. Examples

D.1 One-shot message passing

D.1.1 Code

```
let x = alloc(1) in
let y = alloc(1) in
[x]_na := 0;
[y]_at := 0;
fork [x]_na := 1; [y]_at := 1;
repeat [y]_at end;
[x]_na
```

D.1.2 Proof setup

A ghost PCM for tokens We set up a ghost PCM named Token with carrier $\wp(\{1\})$ and $\uplus$ as composition. Let $\diamond$ denote $\{1\}$.

Escrows We define a single escrow, $\mathbf{XE}(\gamma, x)$, as follows:

$$\mathbf{XE}(\gamma, x): \quad \boxed{\gamma : \diamond} \quad \rightsquigarrow \quad x \hookrightarrow 1$$

Protocols We define a single protocol, $\mathbf{YP}(\gamma, x)$, with states 0 and 1 and transition relation $\leq$. State interpretations are as follows:

$$\mathbf{YP}(\gamma, x)(0, z) \quad \triangleq \quad z = 0$$
$$\mathbf{YP}(\gamma, x)(1, z) \quad \triangleq \quad z = 1 * [\mathbf{XE}(\gamma, x)]$$

D.1.3 Verification

$\left\{\mathsf{true}\right\}$

$\left\{\exists\gamma.\ \boxed{\gamma : \diamond}\right\}$

let $x = \mathtt{alloc}(1)$ in

$\left\{\boxed{\gamma : \diamond} * \mathsf{uninit}(x)\right\}$

let $y = \mathtt{alloc}(1)$ in

$\left\{\boxed{\gamma : \diamond} * \mathsf{uninit}(x) * \mathsf{uninit}(y)\right\}$

$[x]_{\mathtt{na}} := 0;$

$\left\{\boxed{\gamma : \diamond} * x \hookrightarrow 0 * \mathsf{uninit}(y)\right\}$

$[y]_{\mathtt{at}} := 0;$

$\left\{\boxed{\gamma : \diamond} * x \hookrightarrow 0 * \boxed{y : 0 \mid \mathbf{YP}(\gamma, x)}\right\}$

fork

$\quad\left\{x \hookrightarrow 0 * \boxed{y : 0 \mid \mathbf{YP}(\gamma, x)}\right\}$

$\quad[x]_{\mathtt{na}} := 1;\ \left\{x \hookrightarrow 1 * \boxed{y : 0 \mid \mathbf{YP}(\gamma, x)}\right\}$

$\quad\left\{[\mathbf{XE}(\gamma, x)] * \boxed{y : 0 \mid \mathbf{YP}(\gamma, x)}\right\}$

$\quad[y]_{\mathtt{at}} := 1;$

$\quad\left\{\boxed{y : 1 \mid \mathbf{YP}(\gamma, x)}\right\}$

$\left\{\boxed{\gamma : \diamond} * \boxed{y : 0 \mid \mathbf{YP}(\gamma, x)}\right\}$

repeat $[y]_{\mathtt{at}}$ end;

$\left\{\boxed{\gamma : \diamond} * \boxed{y : 1 \mid \mathbf{YP}(\gamma, x)} * [\mathbf{XE}(\gamma, x)]\right\}$

$\left\{\boxed{y : 1 \mid \mathbf{YP}(\gamma, x)} * x \hookrightarrow 1\right\}$

$[x]_{\mathtt{na}}$

$\left\{z.\ z = 1\right\}$

D.2 Spinlocks

D.2.1 Parameters

Fix some assertion P for the resources protected by the lock.

D.2.2 Code

$\texttt{newLock} \triangleq$
  $\texttt{let } x = \texttt{alloc}(1) \texttt{ in}$
  $[x]_{\texttt{at}} := 1;$
  x
$\texttt{spin}(x) \triangleq$
  $\texttt{repeat } [x]_{\texttt{at}} \texttt{ end}$
$\texttt{lock}(x) \triangleq$
  $\texttt{repeat spin}(x); \texttt{CAS}(x, 1, 0) \texttt{ end}$
$\texttt{unlock}(x) \triangleq$
  $[x]_{\texttt{at}} := 1$

D.2.3 Proof setup

Top-level spec

$$\{P\}\ \texttt{newLock}\ \{x. \Box\mathsf{isLock}(x)\}$$
$$\{\mathsf{isLock}(x)\}\ \texttt{lock}(x)\ \{P\}$$
$$\{\mathsf{isLock}(x) * P\}\ \texttt{unlock}(x)\ \{\mathsf{true}\}$$

Protocols We assume a protocol **LP** with a single state Inv, interpreted as follows:

$$\mathbf{LP}(\mathsf{Inv}, x) \quad \triangleq \quad (x = 1 * P) \vee x = 0$$

High-level predicates

$$\mathsf{isLock}(x) \quad \triangleq \quad \boxed{x : \mathsf{Inv} \mid \mathbf{LP}}$$

D.2.4 Verification of `newLock`

$\left\{P\right\}$
$\texttt{let } x = \texttt{alloc}(1) \texttt{ in}$
$\left\{P * \mathsf{uninit}(x)\right\}$
$[x]_{\texttt{at}} := 1;$
$\left\{\boxed{x : \mathsf{Inv} \mid \mathbf{LP}}\right\}$
x
$\left\{\Box(\mathsf{isLock}(x))\right\}$

D.2.5 Verification of `spin`

$\left\{\boxed{x : \mathsf{Inv} \mid \mathbf{LP}}\right\}\ \texttt{repeat } [x]_{\texttt{at}} \texttt{ end}\ \left\{\boxed{x : \mathsf{Inv} \mid \mathbf{LP}}\right\}$

D.2.6 Verification of `lock`

$\left\{\mathsf{isLock}(x)\right\}$
$\left\{\boxed{x : \mathsf{Inv} \mid \mathbf{LP}}\right\}$
$\texttt{repeat}\ \left\{\boxed{x : \mathsf{Inv} \mid \mathbf{LP}}\right\}$
  $\texttt{spin}(x);$
  $\left\{\boxed{x : \mathsf{Inv} \mid \mathbf{LP}}\right\}$
  $\texttt{CAS}(x, 1, 0)$
  $\left\{z. \boxed{x : \mathsf{Inv} \mid \mathbf{LP}} * (z = \mathsf{true} \Rightarrow P)\right\}$
$\texttt{end}$
$\left\{P\right\}$

D.2.7 Verification of `unlock`

$\left\{\mathsf{isLock}(x) * P\right\}$

$\left\{ \boxed{x : \mathsf{Inv}} \boxed{\mathbf{LP}} * P \right\}$
$[x]_{\mathtt{at}} := 1$
$\left\{ \mathsf{true} \right\}$

D.3 Ernie Cohen's lock example

D.3.1 Parameters

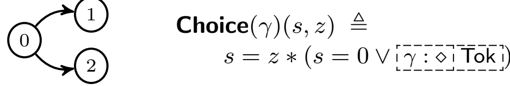
Fix some assertion P for the resources to be raced for.

D.3.2 Code

```
[x]at := choose(1,2);          ||  [y]at := choose(1,2);
repeat [y]at end;              ||  repeat [x]at end;
if [x]at == [y]at then         ||  if [x]at != [y]at then
  /* critical section */       ||    /* critical section */
```

D.3.3 Proof setup

Protocols



$$\textsf{Choice}(\gamma)(s,z) \triangleq s = z * (s = 0 \vee \boxed{\gamma : \diamond}\ \textsf{Tok})$$

Escrows We have an escrow type $\textsf{PE}(\gamma_1, \gamma_2)$ for the resource P:

$$\textsf{PE}(\gamma^x, \gamma^y) : \exists i, j > 0.\ \boxed{x : i} * \boxed{y : j} * \begin{pmatrix} \boxed{\gamma^x : \diamond} * i = j \\ \vee\ \boxed{\gamma^y : \diamond} * i \neq j \end{pmatrix} \rightsquigarrow P$$

D.3.4 Proof

$\{P\}$
let $x = \texttt{alloc}(1)$
$\{P * \textsf{uninit}(x)\}$
let $y = \texttt{alloc}(1)$
$\{P * \textsf{uninit}(x) * \textsf{uninit}(y)\}$
$\{P * \textsf{uninit}(x) * \textsf{uninit}(y) * \exists \gamma_1^x, \gamma_2^x, \gamma_1^y, \gamma_2^y.\ \boxed{\gamma_1^x : \diamond} * \boxed{\gamma_2^x : \diamond} * \boxed{\gamma_1^y : \diamond} * \boxed{\gamma_2^y : \diamond}\}$
$\{\textsf{uninit}(x) * \textsf{uninit}(y) * \boxed{\gamma_1^x : \diamond} * \boxed{\gamma_2^x : \diamond} * \boxed{\gamma_1^y : \diamond} * \boxed{\gamma_2^y : \diamond} * \Box([\textsf{PE}(\gamma_1^x, \gamma_1^y)])\}$
$[x]_{\texttt{at}} := 0$
$\{\textsf{uninit}(y) * \boxed{\gamma_1^x : \diamond} * \boxed{\gamma_2^x : \diamond} * \boxed{\gamma_1^y : \diamond} * \boxed{\gamma_2^y : \diamond} * \Box\left(\boxed{x : 0 \mid \textsf{Choice}(\gamma_2^x)}\right)\}$
$[y]_{\texttt{at}} := 0$
$\{\boxed{\gamma_1^x : \diamond} * \boxed{\gamma_2^x : \diamond} * \boxed{\gamma_1^y : \diamond} * \boxed{\gamma_2^y : \diamond} * \Box\left(\boxed{y : 0 \mid \textsf{Choice}(\gamma_2^y)}\right)\}$

Left thread:

$\{\boxed{\gamma_1^x : \diamond} * \boxed{\gamma_2^x : \diamond}\}$
$[x]_{\texttt{at}} := \texttt{choose}(1, 2);$
$\{\boxed{\gamma_1^x : \diamond} * \exists i > 0.\ \Box\left(\boxed{x : i \mid \textsf{Choice}(\gamma_2^x)}\right)\}$
repeat $[y]_{\texttt{at}}$ end;
$\{\boxed{\gamma_1^x : \diamond} * \exists j > 0.\ \Box\left(\boxed{y : j \mid \textsf{Choice}(\gamma_2^y)}\right)\}$
if $[x]_{\texttt{at}}$ == $[y]_{\texttt{at}}$ then
  $\{\boxed{\gamma_1^x : \diamond} * i = j\}$
  $\{P\}$
  /* critical section */

Right thread:

$\{\boxed{\gamma_1^y : \diamond} * \boxed{\gamma_2^y : \diamond}\}$
$[y]_{\texttt{at}} := \texttt{choose}(1, 2);$
$\{\boxed{\gamma_1^y : \diamond} * \exists j > 0.\ \Box\left(\boxed{y : j \mid \textsf{Choice}(\gamma_2^y)}\right)\}$
repeat $[x]_{\texttt{at}}$ end;
$\{\boxed{\gamma_1^y : \diamond} * \exists i > 0.\ \Box\left(\boxed{x : i \mid \textsf{Choice}(\gamma_2^x)}\right)\}$
if $[x]_{\texttt{at}}$!= $[y]_{\texttt{at}}$ then
  $\{\boxed{\gamma_1^y : \diamond} * i \neq j\}$
  $\{P\}$
  /* critical section */

D.4 Michael-Scott queue

D.4.1 Parameters

Fix some per-element predicate $P(x)$.

Let $\texttt{head} = 0$, $\texttt{tail} = 1$, $\texttt{data} = 0$, $\texttt{link} = 1$. We will use these values as field offsets.

D.4.2 Code

```
newBuffer ≜
  let s = alloc(2)    /* initial sentinel node */
  [s + link]_at := 0;
  let q = alloc(2);   /* queue = head and tail pointers */
  [q + head]_at := s;
  [q + tail]_at := s;
  q

findTail(q) ≜
  let n = [q + tail]_at
  let n′ = [n + link]_at
  if n′ == 0 then n
  else [q + tail]_at := n′; 0

tryEnq(q, x) ≜
  let n = alloc(2);
  [n + data]_na := x;
  [n + link]_at := 0;
  let t = repeat findTail(q) end
  if CAS(t + link, 0, n)
    then [q + tail]_at := n; 1
  else 0

tryDeq(q) ≜
  let s = [q + head]_at
  let n = [s + link]_at
  if n == 0 then 0
  else if CAS(q + head, s, n) then [n + data]_na
  else 0
```

Note that the use of release-acquire operations causes the Michael-Scott queue to exhibit behavior that is not sequentially consistent. This places it out of reach for the type of verification methods (mentioned in the Introduction to the paper) that recover SC reasoning by demanding the use of strong synchronization disciplines. In particular, suppose we have two queues, q and r, both initially empty, and then we run the following:

```
repeat tryEnq(q, 1) end;  ||  repeat tryEnq(r, 2) end;
let x = tryDeq(r)         ||  let y = tryDeq(q)
```

Using our release-acquire implementation of the queue, there is an execution in which both threads return $x = y = 0$ (*i.e.,* observing each other's queue to be empty). But that is not a possible SC execution. This is really just a simple encoding of the canonical example where release-acquire differs from SC, but lifted to a higher-level data structure rather than just using primitive reads/writes.

D.4.3 Proof setup

Top-level spec

$$\begin{array}{rcl} \{\mathsf{true}\} & \mathtt{newBuffer}() & \{q.\ \Box\mathsf{Queue}(q)\} \\ \{\mathsf{Queue}(q) * P(x)\} & \mathtt{tryEnq}(q, x) & \{z.\ z \neq 0 \vee P(x)\} \\ \{\mathsf{Queue}(q)\} & \mathtt{tryDeq}(q) & \{x.\ x = 0 \vee P(x)\} \end{array}$$

A ghost PCM for tokens We set up a ghost PCM named Token with carrier $\wp(\{1\})$ and $\uplus$ as composition. Let $\diamond$ denote $\{1\}$.

Escrows We define a single escrow, $\mathbf{DEQ}(\ell, \gamma, \gamma')$, defined as follows:

$$\mathbf{DEQ}(\ell, \gamma, \gamma') : \quad \boxed{\gamma : \diamond} \quad \rightsquigarrow \quad \exists x.\ \ell \hookrightarrow x * P(x) * \boxed{\gamma' : \diamond}$$

Protocols We have a protocol $\mathbf{Link}(\gamma)$ for $\mathtt{link}$ pointers with states Null, $\mathsf{Linked}(\ell)$ and transitions from Null to $\mathsf{Linked}(\ell)$ for any ℓ (plus the usual reflexive, transitive closure), with state interpretations:

$$\begin{array}{rcl} \mathbf{Link}(\gamma)(\mathsf{Null}, x) & \triangleq & x = 0 \\ \mathbf{Link}(\gamma)(\mathsf{Linked}(\ell), x) & \triangleq & x = \ell \neq 0 * \exists \gamma'.\ [\mathbf{DEQ}(\ell + \mathtt{data}, \gamma, \gamma')] * \boxed{\ell + \mathtt{link} : \mathsf{Null} \mid \mathbf{Link}(\gamma')} \end{array}$$

We also have two protocols, **Head** and **Tail**, each with a single state called Inv, interpreted as:

$$\begin{array}{rcl} \mathbf{Head}(\mathsf{Inv}, x) & \triangleq & \exists \gamma.\ \boxed{x + \mathtt{link} : \mathsf{Null} \mid \mathbf{Link}(\gamma)} * \boxed{\gamma : \diamond} \\ \mathbf{Tail}(\mathsf{Inv}, x) & \triangleq & \exists \gamma.\ \boxed{x + \mathtt{link} : \mathsf{Null} \mid \mathbf{Link}(\gamma)} \end{array}$$

High-level predicates

$$\mathsf{Queue}(q) \quad \triangleq \quad \boxed{q + \mathtt{head} : \mathsf{Inv} \mid \mathbf{Head}} * \boxed{q + \mathtt{tail} : \mathsf{Inv} \mid \mathbf{Tail}}$$

D.4.4 Verification of `newBuffer`

$\left\{\text{true}\right\}$
$\left\{\exists\gamma.\ \boxed{\gamma : \diamond}\right\}$
let $s = \texttt{alloc}(2)$
$\left\{\boxed{\gamma : \diamond} * \text{uninit}(s + \texttt{data}) * \text{uninit}(s + \texttt{link})\right\}$
$\left\{\boxed{\gamma : \diamond} * \text{uninit}(s + \texttt{link})\right\}$ /* *leak memory* */
$[s + \texttt{link}]_{\text{at}} := 0;$
$\left\{\boxed{\gamma : \diamond} * \boxed{s + \texttt{link} : \text{Null} \mid \textbf{Link}(\gamma)}\right\}$
let $q = \texttt{alloc}(2);$
$\left\{\boxed{\gamma : \diamond} * \boxed{s + \texttt{link} : \text{Null} \mid \textbf{Link}(\gamma)} * \text{uninit}(q + \texttt{head}) * \text{uninit}(q + \texttt{tail})\right\}$
$[q + \texttt{head}]_{\text{at}} := s;$
$\left\{\boxed{q + \texttt{head} : \text{Inv} \mid \textbf{Head}} * \boxed{s + \texttt{link} : \text{Null} \mid \textbf{Link}(\gamma)} * \text{uninit}(q + \texttt{tail})\right\}$
$[q + \texttt{tail}]_{\text{at}} := s;$
$\left\{\boxed{q + \texttt{head} : \text{Inv} \mid \textbf{Head}} * \boxed{q + \texttt{tail} : \text{Inv} \mid \textbf{Tail}}\right\}$
q
$\left\{q.\ \Box\text{Queue}(q)\right\}$

D.4.5 Verification of `findTail`

$\left\{\Box(\text{Queue}(q))\right\}$
$\left\{\boxed{q + \texttt{tail} : \text{Inv} \mid \textbf{Tail}}\right\}$
let $n = [q + \texttt{tail}]_{\text{at}}$
$\left\{\Box\left(\exists\gamma.\ \boxed{n + \texttt{link} : \text{Null} \mid \textbf{Link}(\gamma)}\right)\right\}$
let $n' = [n + \texttt{link}]_{\text{at}}$
$\left\{n' \neq 0 \Rightarrow \exists\gamma'.\ \boxed{n' + \texttt{link} : \text{Null} \mid \textbf{Link}(\gamma')}\right\}$
if $n' == 0$ then
  $\left\{\text{true}\right\}$
  n
  $\left\{n.\ \Box\text{Queue}(q) * \exists\gamma.\ \boxed{n + \texttt{link} : \text{Null} \mid \textbf{Link}(\gamma)}\right\}$
else
  $\left\{\exists\gamma'.\ \boxed{n' + \texttt{link} : \text{Null} \mid \textbf{Link}(\gamma')}\right\}$
  $[q + \texttt{tail}]_{\text{at}} := n';$
  $\left\{\text{true}\right\}$
  0
  $\left\{z.\ z = 0 * \Box\text{Queue}(q)\right\}$

D.4.6 Verification of `tryEnq`

$\left\{ P(x) * \Box(\mathsf{Queue}(q)) \right\}$

$\texttt{let}\ n = \texttt{alloc}(2);$

$\left\{ P(x) * \mathsf{uninit}(n + \texttt{data}) * \mathsf{uninit}(n + \texttt{link}) * \Box(n \neq 0) \right\}$

$[n + \texttt{data}]_{\texttt{na}} := x;$

$\left\{ P(x) * (n + \texttt{data}) \hookrightarrow x * \mathsf{uninit}(n + \texttt{link}) \right\}$

$\left\{ P(x) * (n + \texttt{data}) \hookrightarrow x * \mathsf{uninit}(n + \texttt{link}) * \exists \gamma'.\ \boxed{\gamma' : \diamond} \right\}$

$[n + \texttt{link}]_{\texttt{at}} := 0;$

$\left\{ P(x) * (n + \texttt{data}) \hookrightarrow x * \boxed{n + \texttt{link} : \mathsf{Null}\ \Big|\ \mathbf{Link}(\gamma')} * \boxed{\gamma' : \diamond} \right\}$

$\texttt{let}\ t = \texttt{repeat findTail}(q)\ \texttt{end}$

$\left\{ P(x) * (n + \texttt{data}) \hookrightarrow x * \boxed{n + \texttt{link} : \mathsf{Null}\ \Big|\ \mathbf{Link}(\gamma')} * \boxed{\gamma' : \diamond} * \exists \gamma.\ \boxed{t + \texttt{link} : \mathsf{Null}\ \Big|\ \mathbf{Link}(\gamma)} \right\}$

$\texttt{if CAS}(t + \texttt{link}, 0, n)\ \texttt{then}$

$\quad \left\{ \boxed{t + \texttt{link} : \mathsf{Linked}(n)\ \Big|\ \mathbf{Link}(\gamma)} * \boxed{n + \texttt{link} : \mathsf{Null}\ \Big|\ \mathbf{Link}(\gamma')} \right\}$

$\quad \left\{ \boxed{n + \texttt{link} : \mathsf{Null}\ \Big|\ \mathbf{Link}(\gamma')} * \boxed{q + \texttt{tail} : \mathsf{Inv}\ \Big|\ \mathbf{Tail}} \right\}$

$\quad [q + \texttt{tail}]_{\texttt{at}} := n;$

$\quad \left\{ \mathsf{true} \right\}$

$\quad 1$

$\quad \left\{ z.\ z = 1 \right\}$

$\texttt{else}$

$\quad \left\{ P(x) * \boxed{t + \texttt{link} : \mathsf{Null}\ \Big|\ \mathbf{Link}(\gamma)} \right\}$

$\quad 0$

$\quad \left\{ z.\ z = 0 * P(x) \right\}$

D.4.7 Verification of `tryDeq`

$\left\{ \mathsf{Queue}(q) \right\}$

$\left\{ \Box\left(\boxed{q + \texttt{head} : \mathsf{Inv}} \boxed{\textbf{Head}} \right) \right\}$

$\texttt{let } s = [q + \texttt{head}]_{\texttt{at}}$

$\left\{ \exists \gamma.\ \boxed{s + \texttt{link} : \mathsf{Null}} \boxed{\textbf{Link}(\gamma)} \right\}$

$\texttt{let } n = [s + \texttt{link}]_{\texttt{at}}$

$\left\{ n \neq 0 \Rightarrow \exists \gamma'.\ [\textbf{DEQ}(n + \texttt{data}, \gamma, \gamma')] * \boxed{n + \texttt{link} : \mathsf{Null}} \boxed{\textbf{Link}(\gamma')} \right\}$

$\texttt{if } n \texttt{ == } 0 \texttt{ then}$

$\quad \left\{ \mathsf{true} \right\}$

$\quad 0$

$\quad \left\{ x.\ x = 0 \right\}$

$\texttt{else}$

$\quad \left\{ \boxed{q + \texttt{head} : \mathsf{Inv}} \boxed{\textbf{Head}} * [\textbf{DEQ}(n + \texttt{data}, \gamma, \gamma')] * \boxed{n + \texttt{link} : \mathsf{Null}} \boxed{\textbf{Link}(\gamma')} \right\}$

$\quad \texttt{if CAS}(q + \texttt{head}, s, n) \texttt{ then}$

$\quad\quad \left\{ \boxed{q + \texttt{head} : \mathsf{Inv}} \boxed{\textbf{Head}} * \exists x.\ n + \texttt{data} \hookrightarrow x * P(x) \right\}$

$\quad\quad [n + \texttt{data}]_{\texttt{na}}$

$\quad\quad \left\{ x.\ P(x) \right\}$

$\quad \texttt{else}$

$\quad\quad \left\{ \boxed{q + \texttt{head} : \mathsf{Inv}} \boxed{\textbf{Head}} \right\}$

$\quad\quad 0$

$\quad\quad \left\{ x.\ x = 0 \right\}$

D.5 Circular buffer

D.5.1 Parameters

- Fix some choice of buffer size $N > 1$; the actual capacity is $N - 1$.
- Fix some per-element predicate $P(x)$.

Let $\texttt{wi} = 0$, $\texttt{ri} = 1$, $\texttt{buf} = 2$. We will use these values as field offsets.

D.5.2 Code

Based on circular buffers from the Linux kernel [17].

```
newBuffer() ≜
  let q = alloc(N + 2)    /* queue = writer index, reader index, buffer */
  [q + ri]_at := 0;
  [q + wi]_at := 0;
  q

tryProd(q, x) ≜
  let w = [q + wi]_at
  let r = [q + ri]_at
  let w' = w + 1 mod N
  if w' == r then 0
  else [q + buf + w]_na := x;
       [q + wi]_at := w';
       1

tryCons(q) ≜
  let w = [q + wi]_at
  let r = [q + ri]_at
  if w == r then 0
  else let x = [q + buf + r]_na
       [q + ri]_at := r + 1 mod N;
       x
```

In real implementations, this data structure provides an operation returning a bound on the size of the buffer, which can then be used to efficiently batch a series of reads/writes without checking the indices each time. It would be straightforward to generalize our proof to handle such an operation.

Note also that this data structure exhibits non-SC behavior for essentially the same reasons as the Michael-Scott queue does. (One can construct a similar example to the one given in §D.4.2.)

D.5.3 Proof setup

Top-level spec

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

A ghost PCM for natural numbers We set up a ghost PCM with carrier $\wp(\mathbb{N})^4$ with composition $\uplus$ component-wise. We define the following terms over this PCM:

$$\begin{array}{rcl}
\mathsf{all} & \triangleq & (\mathbb{N}, \mathbb{N}, \mathbb{N}, \mathbb{N}) \\
\mathsf{restP}(i) & \triangleq & (\{j \mid j > i\}, \{j \mid j \geq i\}, \emptyset, \emptyset) \\
\mathsf{restC}(i) & \triangleq & (\emptyset, \emptyset, \{j \mid j > i\}, \{j \mid j \geq i\}) \\
\mathsf{protP}(i) & \triangleq & (\{i\}, \emptyset, \emptyset, \emptyset) \\
\mathsf{escP}(i) & \triangleq & (\emptyset, \{i\}, \emptyset, \emptyset) \\
\mathsf{protC}(i) & \triangleq & (\emptyset, \emptyset, \{i\}, \emptyset) \\
\mathsf{escC}(i) & \triangleq & (\emptyset, \emptyset, \emptyset, \{i\})
\end{array}$$

Escrows We define two escrows, $\mathbf{PE}(\gamma, q, i)$ and $\mathbf{CE}(\gamma, q, i)$, as follows:

$$\begin{array}{rcl}
\mathbf{PE}(\gamma, q, i) : \ \boxed{\gamma : \mathsf{escP}(i)} & \rightsquigarrow & \mathsf{uninit}(q + \texttt{buf} + (i \bmod N)) \\
 & \vee & (q + \texttt{buf} + (i \bmod N)) \hookrightarrow - \\
\mathbf{CE}(\gamma, q, j) : \ \boxed{\gamma : \mathsf{escC}(j)} & \rightsquigarrow & \exists x.\ P(x) * (q + \texttt{buf} + (j \bmod N)) \hookrightarrow x
\end{array}$$

Protocols We assume STS states for every natural number. We assume two protocols, $\mathbf{PP}(\gamma, q)$ and $\mathbf{CP}(\gamma, q)$ over natural number states, with transition relations

$$\sqsubseteq_{\mathbf{PP}} \ \triangleq \ \sqsubseteq_{\mathbf{CP}} \ \triangleq \ \leq$$

and state interpretations

$$\begin{array}{rcl}
\mathbf{PP}(\gamma, q)(i, x) & \triangleq & \square(x = i \bmod N * \forall j < i.\ [\mathbf{CE}(\gamma, q, j)]) \ * \ \boxed{\gamma : \mathsf{protP}(i)} \\
\mathbf{CP}(\gamma, q)(j, x) & \triangleq & \square(x = j \bmod N * \forall i < j + N.\ [\mathbf{PE}(\gamma, q, i)]) \ * \ \boxed{\gamma : \mathsf{protC}(j)}
\end{array}$$

High-level predicates

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

D.5.4 Verification of `newBuffer`

$\left\{\mathsf{true}\right\}$

$\left\{\exists\gamma.\ \boxed{\gamma : \mathsf{all}}\right\}$

$\texttt{let}\ q = \texttt{alloc}(N + 2)$

$\left\{\boxed{\gamma : \mathsf{all}} * \mathsf{uninit}(q) * \cdots * \mathsf{uninit}(q + N + 1)\right\}$

$\left\{\boxed{\gamma : \mathsf{all}} * \mathsf{uninit}(q) * \cdots * \mathsf{uninit}(q + N + 1)\right\}$

$\left\{\boxed{\gamma : \mathsf{all}} * \mathsf{uninit}(q + \texttt{wi}) * \mathsf{uninit}(q + \texttt{ri}) * [\mathbf{PE}(\gamma, q, 0)] * \cdots * [\mathbf{PE}(\gamma, q, N - 1)]\right\}$

$[q + \texttt{ri}]_{\texttt{at}} := 0;$

$\left\{\boxed{q + \texttt{ri} : 0 \mid \mathbf{CP}(\gamma, q)} * \boxed{\gamma : \mathsf{restC}(0)} * \boxed{\gamma : \mathsf{protP}(0)} * \boxed{\gamma : \mathsf{restP}(0)} * \mathsf{uninit}(q + \texttt{wi})\right\}$

$[q + \texttt{wi}]_{\texttt{at}} := 0;$

$\left\{\boxed{q + \texttt{ri} : 0 \mid \mathbf{CP}(\gamma, q)} * \boxed{q + \texttt{wi} : 0 \mid \mathbf{PP}(\gamma, q)} * \boxed{\gamma : \mathsf{restC}(0)} * \boxed{\gamma : \mathsf{restP}(0)}\right\}$

q

$\left\{\mathsf{Prod}(q) * \mathsf{Cons}(q)\right\}$

D.5.5 Verification of `tryProd`

$\left\{ \mathsf{Prod}(q) * P(x) \right\}$

$\left\{ \boxed{\gamma : \mathsf{restP}(i)} * P(x) * \Box\Big(i < j_0 + N \wedge \boxed{q + \mathtt{wi} : i \mid \mathbf{PP}(\gamma, q)} \wedge \boxed{q + \mathtt{ri} : j_0 \mid \mathbf{CP}(\gamma, q)}\Big) \right\}$

$\mathtt{let}\ w = [q + \mathtt{wi}]_{\mathtt{at}}$

$\left\{ \boxed{\gamma : \mathsf{restP}(i)} * P(x) * \Box(w = i \bmod N \wedge \forall k < i.\ [\mathbf{CE}(\gamma, q, k)]) \right\}$

$\mathtt{let}\ r = [q + \mathtt{ri}]_{\mathtt{at}}$

$\left\{ \boxed{\gamma : \mathsf{restP}(i)} * P(x) * \Box\Big(r = j \bmod N \wedge \forall k < j + N.\ [\mathbf{PE}(\gamma, q, k)] \wedge j_0 \leq j \wedge \boxed{q + \mathtt{ri} : j \mid \mathbf{CP}(\gamma, q)}\Big) \right\}$

$\left\{ \boxed{\gamma : \mathsf{restP}(i)} * P(x) * \Box(i < j + N \wedge [\mathbf{PE}(\gamma, q, i)]) \right\}$

$\mathtt{let}\ w' = w + 1\ \mathtt{mod}\ N$

$\left\{ \boxed{\gamma : \mathsf{restP}(i)} * P(x) * \Box(w' = w + 1 \bmod N) \right\}$

$\mathtt{if}\ w' == r\ \mathtt{then}$

$\quad \left\{ \boxed{\gamma : \mathsf{restP}(i)} * P(x) \right\}$

$\quad 0$

$\quad \left\{ z.\ \mathsf{Prod}(q) * z = 0 * P(x) \right\}$

$\mathtt{else}$

$\quad \left\{ \boxed{\gamma : \mathsf{restP}(i)} * P(x) * \Box(w' \neq r) \right\}$

$\quad \left\{ \boxed{\gamma : \mathsf{restP}(i)} * P(x) * \Box(i + 1 < j + N) \right\}$

$\quad \left\{ \boxed{\gamma : \mathsf{restP}(i+1)} * \boxed{\gamma : \mathsf{protP}(i+1)} * P(x) * \boxed{\gamma : \mathsf{escP}(i)} \right\}$

$\quad \left\{ \boxed{\gamma : \mathsf{restP}(i+1)} * \boxed{\gamma : \mathsf{protP}(i+1)} * P(x) * (\mathsf{uninit}(q + \mathsf{buf} + w) \vee (q + \mathsf{buf} + w) \hookrightarrow -) \right\}$

$\quad [q + \mathsf{buf} + w]_{\mathtt{na}} := x;$

$\quad \left\{ \boxed{\gamma : \mathsf{restP}(i+1)} * \boxed{\gamma : \mathsf{protP}(i+1)} * P(x) * (q + \mathsf{buf} + w) \hookrightarrow x \right\}$

$\quad \left\{ \boxed{\gamma : \mathsf{restP}(i+1)} * \boxed{\gamma : \mathsf{protP}(i+1)} * [\mathbf{CE}(\gamma, q, i)] \right\}$

$\quad [q + \mathtt{wi}]_{\mathtt{at}} := w';$

$\quad \left\{ \boxed{\gamma : \mathsf{restP}(i+1)} * \boxed{q + \mathtt{wi} : i + 1 \mid \mathbf{PP}(\gamma, q)} \right\}$

$\quad 1$

$\quad \left\{ z.\ \mathsf{Prod}(q) * z = 1 \right\}$

D.5.6 Verification of `tryCons`

$\left\{\mathsf{Cons}(q)\right\}$

$\left\{\boxed{\gamma : \mathsf{restC}(j)} * \square\left(j \leq i_0 \wedge \boxed{q + \mathtt{wi} : i_0 \mid \mathbf{PP}(\gamma, q)} \wedge \boxed{q + \mathtt{ri} : j \mid \mathbf{CP}(\gamma, q)}\right)\right\}$

$\mathtt{let}\ w = [q + \mathtt{wi}]_{\mathtt{at}}$

$\left\{\boxed{\gamma : \mathsf{restC}(j)} * \square\left(w = i \bmod N \wedge \forall k < i.\ [\mathbf{CE}(\gamma, q, k)] \wedge i_0 \leq i \wedge \boxed{q + \mathtt{wi} : i \mid \mathbf{PP}(\gamma, q)}\right)\right\}$

$\left\{\boxed{\gamma : \mathsf{restC}(j)} * \square(j \leq i)\right\}$

$\mathtt{let}\ r = [q + \mathtt{ri}]_{\mathtt{at}}$

$\left\{\boxed{\gamma : \mathsf{restC}(j)} * \square(r = j \bmod N \wedge \forall k < j + N.\ [\mathbf{PE}(\gamma, q, k)])\right\}$

$\mathtt{if}\ w == r\ \mathtt{then}$

$\quad\left\{\boxed{\gamma : \mathsf{restC}(j)}\right\}\ 0$

$\quad\left\{x.\ \mathsf{Cons}(q) * x = 0\right\}$

$\mathtt{else}$

$\quad\left\{\boxed{\gamma : \mathsf{restC}(j)} * w \neq r\right\}$

$\quad\left\{\boxed{\gamma : \mathsf{restC}(j)} * \square(j < i)\right\}$

$\quad\left\{\boxed{\gamma : \mathsf{restC}(j+1)} * \boxed{\gamma : \mathsf{protC}(j+1)} * \boxed{\gamma : \mathsf{escC}(j)}\right\}$

$\quad\left\{\boxed{\gamma : \mathsf{restC}(j+1)} * \boxed{\gamma : \mathsf{protC}(j+1)} * \exists x.\ P(x) * (q + \mathsf{buf} + r) \hookrightarrow x\right\}$

$\quad\mathtt{let}\ x = [q + \mathtt{buf} + r]_{\mathtt{na}}$

$\quad\left\{\boxed{\gamma : \mathsf{restC}(j+1)} * \boxed{\gamma : \mathsf{protC}(j+1)} * P(x) * (q + \mathsf{buf} + r) \hookrightarrow x\right\}$

$\quad\left\{\boxed{\gamma : \mathsf{restC}(j+1)} * P(x) * \boxed{\gamma : \mathsf{protC}(j+1)} * [\mathbf{PE}(\gamma, q, j + N)]\right\}$

$\quad[q + \mathtt{ri}]_{\mathtt{at}} := r + 1\ \mathtt{mod}\ N;$

$\quad\left\{\boxed{\gamma : \mathsf{restC}(j+1)} * P(x) * \boxed{q + \mathtt{ri} : j + 1 \mid \mathbf{CP}(\gamma, q)}\right\}$

$\quad x$

$\quad\left\{x.\ \mathsf{Cons}(q) * P(x)\right\}$

D.6 Bounded ticket locks

D.6.1 Parameters

- Fix some resource invariant P to be protected by the lock.
- C is a constant representing the maximum unsigned integer value of the machine.
  FAI increments modulo C (see the semantics of the language in Appendix §A.2).

Let $\mathtt{ns} = 0, \mathtt{tc} = 1$. We will use these values as field offsets.

D.6.2 Code

```
newLock() ≜
  let x = alloc(2)
  [x+ns]_at := 0;
  [x+tc]_at := 0;
  x

lock(x) ≜
  let y = FAI(x+tc)
  repeat
    let z = [x+ns]_at
    y == z
  end

unlock(x) ≜
  let z = [x+ns]_at
  [x+ns]_at := (z+1) mod C
```

Ticket locks [11] are a fair locking mechanism employed by the Linux kernel. The data structure involves two atomic locations, one ($x + \mathtt{tc}$) storing a "ticket" counter, and the other ($x + \mathtt{ns}$) a "now-serving" counter. Both are initially 0. To acquire the lock, a thread first atomically obtains the current value t of the ticket counter and increments it (using a fetch-and-add), and then spins until the now-serving counter is equal to the "ticket" t that it received. To release the lock, the thread just increments the now-serving counter to $t + 1$.

Ticket locks rely crucially on the invariant that no two threads trying to acquire the lock at the same time have the same ticket. If the ticket counter is modeled as an unbounded natural number, this invariant is easy to ensure. But in reality, ticket counters are bounded by the maximum unsigned integer value C of a machine word, and they wrap around to 0 once hitting $\mathsf{C} - 1$. Ticket locks thus only behave correctly if the number of threads concurrently trying to acquire the lock is at most C.

We have proven correctness for an implementation of bounded ticket locks, where the updates to the ticket counter are performed using a physically atomic fetch-and-increment operation FAI, but the reads and writes to the now-serving counter are release-acquire. By arranging for the acquire operation to consume a MayAcquire permission, and by only giving the client a fixed budget of C such permissions, our spec restricts the client from spawning more than C threads to acquire the lock at one time. While similar in certain ways to the proof of the circular buffer, the proof of bounded ticket locks is surprisingly subtle and employs a somewhat more elaborate ghost PCM. It also relies on the following "frame-preserving update" rule for ghost moves, which we have proven sound:

$$\frac{\forall t_F : \mathsf{PCM}_\mu.\ t_1 \#_\mu t_F \Rightarrow t_2 \#_\mu t_F}{\boxed{\gamma : t_1}^{\mu} \Rrightarrow \boxed{\gamma : t_2}^{\mu}}$$

This rule, familiar from recent work on separation logic [13, 20, 21], enables one to make an arbitrary update to one's ghost resource, so long as the update is guaranteed to preserve compatibility with arbitrary frame resources. It is used in the proof of the (UseUnPerm) and (GetTicket) axioms below.

To our knowledge, this is the first formal proof of correctness for bounded ticket locks in a weak memory setting.

D.6.3 Proof setup

Top-level spec

$$\{P\}\ \mathtt{newLock}()\ \left\{x.\ \mathop{\Large\ast}_{i<\mathsf{C}} \mathsf{MayAcquire}(x)\right\}$$
$$\{\mathsf{MayAcquire}(x)\}\ \ \mathtt{lock}(x)\ \ \{P * \mathsf{MayRelease}(x)\}$$
$$\{P * \mathsf{MayRelease}(x)\}\ \mathtt{unlock}(x)\ \{\mathsf{MayAcquire}(x)\}$$

Default sorts of variables

- t, n range over $\mathbb{N}$.
- i, j range over $\mathrm{Ids} = \{0, \ldots, \mathsf{C} - 1\}$, where C is the word size (the modulus of FAI).
- T ranges over $\wp(\mathbb{N})$.
- M ranges over $\mathbb{N} \rightharpoonup \mathrm{Ids}$ such that $\exists t.\ \mathrm{dom}(M) = \{t' \mid t' < t\}$.
- $\mathcal{I}$ ranges over $\mathrm{Ids} \rightharpoonup \mathbb{N}$.

Abstract predicates For purposes of modularity, we give the following axioms about a set of abstract ghost predicates, which are all you need in order to do the proof. Later on, we will show that these predicates are implementable (and axioms satisfiable) in terms of a suitable ghost PCM.

$$\mathsf{Perms}^{\gamma}_{\geq}(t) \Rightarrow \mathsf{LkPerm}^{\gamma}(t) * \mathsf{UnPerm}^{\gamma}(t) * \mathsf{Perms}^{\gamma}_{\geq}(t+1) \quad \text{(GetPerms)}$$
$$\mathsf{LkPerm}^{\gamma}(t) * \mathsf{LkPerm}^{\gamma}(t) \Rightarrow \bot \quad \text{(LkPermExclusive)}$$
$$\mathsf{UsedUP}^{\gamma}_{<}(t) * \mathsf{UnPerm}^{\gamma}(t') \Rightarrow t \leq t' \quad \text{(UnusedUnPerms)}$$
$$\mathsf{UsedUP}^{\gamma}_{<}(t) * \mathsf{UnPerm}^{\gamma}(t) \Rrightarrow \mathsf{UsedUP}^{\gamma}_{<}(t+1) \quad \text{(UseUnPerm)}$$
$$\mathsf{UsedUP}^{\gamma}_{<}(t) \Rightarrow \Box(\mathsf{UsedUP}^{\gamma}_{<}(t)) \quad \text{(UsedPermsPure)}$$
$$\mathsf{MyTkts}^{\gamma}(i,T) * \mathsf{AllTkts}^{\gamma}(M) \Rightarrow (\forall t.\ M(t) = i \Leftrightarrow t \in T) \quad \text{(MyAllCoherence)}$$
$$\mathsf{MyTkts}^{\gamma}(i,T) * \mathsf{AllTkts}^{\gamma}(M) * t = |\mathrm{dom}(M)| \Rrightarrow \mathsf{MyTkts}^{\gamma}(i, T \uplus \{t\}) * \mathsf{AllTkts}^{\gamma}(M \uplus [t \mapsto i]) \quad \text{(GetTicket)}$$
$$\mathsf{true} \Rrightarrow \exists\gamma.\ \mathsf{AllTkts}^{\gamma}(\emptyset) * \Big(\mathop{\Large *}_{i<\mathbf{C}} \mathsf{MyTkts}^{\gamma}(i,\emptyset)\Big) * \mathsf{Perms}^{\gamma}_{\geq}(0) * \mathsf{UsedUP}^{\gamma}_{<}(0) \quad \text{(NewGhost)}$$

Escrows We define one resource escrow $\mathbf{Esc}(\gamma, n)$, which is used to pass control over the lock-protected resource from the lock-releaser to the next lock-acquirer (with ticket n).

$$\mathbf{Esc}(\gamma,n): \quad \mathsf{LkPerm}^{\gamma}(n) \quad \rightsquigarrow \quad P$$

Protocols The protocol $\mathbf{NSP}(\gamma)$ describes a protocol on the now-serving counter $x + \mathtt{ns}$, with states for every natural number n and the usual ordering ($\leq$) on states. Here, n represents the "absolute" value of the counter, as opposed to the actual value, which is n mod $\mathbf{C}$.

$$\mathbf{NSP}(\gamma)(n,z) \quad \triangleq \quad \Box(z = n \bmod \mathbf{C} * \mathsf{UsedUP}^{\gamma}_{<}(n) * [\mathbf{Esc}(\gamma,n)])$$

The protocol $\mathbf{TCP}(\gamma, x)$ describes an invariant protocol on the ticket counter $x + \mathtt{tc}$, with single state Inv.

$$\begin{aligned}\mathbf{TCP}(\gamma,x)(\mathsf{Inv},y) \quad \triangleq \quad & \exists t,n,M.\ (t = |\mathrm{dom}(M)|) * (y = t \bmod \mathbf{C}) * (t \leq n + \mathbf{C}) \\ & * \boxed{x + \mathtt{ns} : n \mid \mathbf{NSP}(\gamma)} * (\forall t_1 < t_2 < t.\ M(t_1) = M(t_2) \Rightarrow t_1 < n) \\ & * \mathsf{Perms}^{\gamma}_{\geq}(t) * \mathsf{AllTkts}^{\gamma}(M)\end{aligned}$$

Derived predicates Here are some useful predicates defined in terms of the above predicates.

$$\begin{aligned}\mathsf{UsedTkts}^{\gamma}(x,T) \quad &\triangleq \quad \boxed{x + \mathtt{tc} : \mathsf{Inv} \mid \mathbf{TCP}(\gamma,x)} * \forall t \in T.\ \boxed{x + \mathtt{ns} : t+1 \mid \mathbf{NSP}(\gamma)} \\ \mathsf{HoldingTkt}^{\gamma}(i,T,t) \quad &\triangleq \quad \mathsf{LkPerm}^{\gamma}(t) * \mathsf{UnPerm}^{\gamma}(t) * \mathsf{MyTkts}^{\gamma}(i, T \uplus \{t\}) \\ \mathsf{MayAcquire}^{\gamma}(x,i,T) \quad &\triangleq \quad \Box(\mathsf{UsedTkts}^{\gamma}(x,T)) * \mathsf{MyTkts}^{\gamma}(i,T) \\ \mathsf{MayRelease}^{\gamma}(x,i,T,t) \quad &\triangleq \quad \Box(\mathsf{UsedTkts}^{\gamma}(x,T)) * \mathsf{MyTkts}^{\gamma}(i, T \uplus \{t\}) * \mathsf{UnPerm}^{\gamma}(t) * \boxed{x + \mathtt{ns} : t \mid \mathbf{NSP}(\gamma)} \\ \mathsf{MayAcquire}(x) \quad &\triangleq \quad \exists\gamma,i,T.\ \mathsf{MayAcquire}^{\gamma}(x,i,T) \\ \mathsf{MayRelease}(x) \quad &\triangleq \quad \exists\gamma,i,T,t.\ \mathsf{MayRelease}^{\gamma}(x,i,T,t)\end{aligned}$$

D.6.4 Verification of `newLock`

$\{P\}$

$\Big\{P * \exists\gamma.\ \mathsf{UsedUP}^{\gamma}_{<}(0) * \mathsf{Perms}^{\gamma}_{\geq}(0) * \mathsf{AllTkts}^{\gamma}(\emptyset) * \Big(\mathop{\Large *}_{i<\mathbf{C}} \mathsf{MyTkts}^{\gamma}(i,\emptyset)\Big)\Big\}$

$\mathtt{let}\ x = \mathtt{alloc}(2)$

$\Big\{P * \mathsf{uninit}(x + \mathtt{ns}) * \mathsf{uninit}(x + \mathtt{tc}) * \mathsf{UsedUP}^{\gamma}_{<}(0) * \mathsf{Perms}^{\gamma}_{\geq}(0) * \mathsf{AllTkts}^{\gamma}(\emptyset) * \Big(\mathop{\Large *}_{i<\mathbf{C}} \mathsf{MyTkts}^{\gamma}(i,\emptyset)\Big)\Big\}$

$\Big\{[\mathbf{Esc}(\gamma,0)] * \mathsf{UsedUP}^{\gamma}_{<}(0) * \mathsf{uninit}(x + \mathtt{ns}) * \mathsf{uninit}(x + \mathtt{tc}) * \mathsf{Perms}^{\gamma}_{\geq}(0) * \mathsf{AllTkts}^{\gamma}(\emptyset) * \Big(\mathop{\Large *}_{i<\mathbf{C}} \mathsf{MyTkts}^{\gamma}(i,\emptyset)\Big)\Big\}$

$[x + \mathtt{ns}]_{\mathtt{at}} := 0;$

$\Big\{\boxed{x + \mathtt{ns} : 0 \mid \mathbf{NSP}(\gamma)} * \mathsf{uninit}(x + \mathtt{tc}) * \mathsf{Perms}^{\gamma}_{\geq}(0) * \mathsf{AllTkts}^{\gamma}(\emptyset) * \Big(\mathop{\Large *}_{i<\mathbf{C}} \mathsf{MyTkts}^{\gamma}(i,\emptyset)\Big)\Big\}$

$[x + \mathtt{tc}]_{\mathtt{at}} := 0;$

$\Big\{\boxed{x + \mathtt{tc} : \mathsf{Inv} \mid \mathbf{TCP}(\gamma,x)} * \Big(\mathop{\Large *}_{i<\mathbf{C}} \mathsf{MyTkts}^{\gamma}(i,\emptyset)\Big)\Big\}$

$\Big\{\Box(\mathsf{UsedTkts}^{\gamma}(x,\emptyset)) * \Big(\mathop{\Large *}_{i<\mathbf{C}} \mathsf{MyTkts}^{\gamma}(i,\emptyset)\Big)\Big\}$

x

$\Big\{\mathop{\Large *}_{i<\mathbf{C}} \mathsf{MayAcquire}^{\gamma}(x,i,\emptyset)\Big\}$

$\Big\{\mathop{\Large *}_{i<\mathbf{C}} \mathsf{MayAcquire}(x)\Big\}$

D.6.5 Verification of `lock`

$\left\{\mathsf{MayAcquire}(x)\right\}$
$\left\{\exists \gamma, i, T.\ \mathsf{MayAcquire}^{\gamma}(x, i, T)\right\}$
$\left\{\mathsf{MyTkts}^{\gamma}(i, T) * \square(\mathsf{UsedTkts}^{\gamma}(x, T))\right\}$
$\left\{\mathsf{MyTkts}^{\gamma}(i, T) * \boxed{x + \mathtt{tc} : \mathsf{Inv}\ \middle|\ \mathbf{TCP}(\gamma, x)} * \forall t \in T.\ \boxed{x + \mathtt{ns} : t + 1\ \middle|\ \mathbf{NSP}(\gamma)}\right\}$
let $y = \mathtt{FAI}(x + \mathtt{tc})$
$\left\{\exists t, n_0.\ \square\left(\boxed{x + \mathtt{ns} : n_0\ \middle|\ \mathbf{NSP}(\gamma)} * (y = t \bmod \mathbf{C}) * (t < n_0 + \mathbf{C})\right) * \mathsf{HoldingTkt}^{\gamma}(i, T, t)\right\}$
repeat
  let $z = [x + \mathtt{ns}]_{\mathtt{at}}$
  $\left\{\exists n.\ \square\left(\boxed{x + \mathtt{ns} : n\ \middle|\ \mathbf{NSP}(\gamma)} * (z = n \bmod \mathbf{C}) * (n \geq n_0) * \mathsf{UsedUP}^{\gamma}_{<}(n) * [\mathbf{Esc}(\gamma, n)]\right) * \mathsf{HoldingTkt}^{\gamma}(i, T, t)\right\}$
  $\left\{\square(n \leq t < n_0 + \mathbf{C} \leq n + \mathbf{C}) * \mathsf{HoldingTkt}^{\gamma}(i, T, t)\right\}$
  $y == z$
  $\left\{b.\ (b = 0 * \mathsf{HoldingTkt}^{\gamma}(i, T, t)) \vee (b = 1 * \mathsf{HoldingTkt}^{\gamma}(i, T, t) * (t \bmod \mathbf{C} = n \bmod \mathbf{C}))\right\}$
end
$\left\{\mathsf{HoldingTkt}^{\gamma}(i, T, t) * (t \bmod \mathbf{C} = n \bmod \mathbf{C})\right\}$
$\left\{\mathsf{HoldingTkt}^{\gamma}(i, T, t) * \square(t = n)\right\}$
$\left\{\mathsf{MyTkts}^{\gamma}(i, T \uplus \{t\}) * \mathsf{UnPerm}^{\gamma}(t) * \mathsf{LkPerm}^{\gamma}(t) * [\mathbf{Esc}(\gamma, t)]\right\}$
$\left\{P * \mathsf{MyTkts}^{\gamma}(i, T \uplus \{t\}) * \mathsf{UnPerm}^{\gamma}(t)\right\}$
$\left\{P * \mathsf{MayRelease}^{\gamma}(x, i, T, t)\right\}$
$\left\{P * \mathsf{MayRelease}(x)\right\}$

D.6.6 Verification of `unlock`

$\left\{P * \mathsf{MayRelease}(x)\right\}$
$\left\{P * \exists \gamma, i, T, t.\ \mathsf{MayRelease}^{\gamma}(x, i, T, t)\right\}$
$\left\{P * \mathsf{UnPerm}^{\gamma}(t) * \mathsf{MyTkts}^{\gamma}(i, T \uplus \{t\}) * \square\left(\mathsf{UsedTkts}^{\gamma}(x, T) * \boxed{x + \mathtt{ns} : t\ \middle|\ \mathbf{NSP}(\gamma)}\right)\right\}$
let $z = [x + \mathtt{ns}]_{\mathtt{at}}$
$\left\{P * \mathsf{UnPerm}^{\gamma}(t) * \mathsf{MyTkts}^{\gamma}(i, T \uplus \{t\}) * \square(z = t \bmod \mathbf{C} * \mathsf{UsedUP}^{\gamma}_{<}(t))\right\}$
$\left\{[\mathbf{Esc}(\gamma, t + 1)] * \mathsf{UnPerm}^{\gamma}(t) * \mathsf{MyTkts}^{\gamma}(i, T \uplus \{t\})\right\}$
$[x + \mathtt{ns}]_{\mathtt{at}} := (z + 1) \bmod \mathbf{C}$
$\left\{\boxed{x + \mathtt{ns} : t + 1\ \middle|\ \mathbf{NSP}(\gamma)} * \mathsf{MyTkts}^{\gamma}(i, T \uplus \{t\})\right\}$
$\left\{\square(\mathsf{UsedTkts}^{\gamma}(x, T \uplus \{t\})) * \mathsf{MyTkts}^{\gamma}(i, T \uplus \{t\})\right\}$
$\left\{\mathsf{MayAcquire}^{\gamma}(x, i, T \uplus \{t\})\right\}$
$\left\{\mathsf{MayAcquire}(x)\right\}$

D.6.7 Substantiating the axioms about the abstract predicates

Here we define the abstract predicates that we used in the above proof in terms of a suitable ghost PCM. It is easy to check that the axioms about these predicates that we used are sound w.r.t. the PCM. In the case of the ghost "transition" axioms (UseUnPerm) and (GetTicket), *i.e.,* where there is a "state change" (using up an unlock permission, or assigning the next ticket to a particular index i), the soundness of the axiom relies on the frame-preserving ghost update rule. For the ghost allocation axiom (NewGhost), soundness follows from the ghost allocation rule.

The ghost PCM is a Cartesian product of three sub-PCMs:

$$\begin{array}{rrcll} \textit{Ticket Allocations} & \mathcal{T} & ::= & \mathsf{My}(\mathcal{I}) & \\ & & \mid & \mathsf{All}(\mathcal{I}, M) & \forall t.\ \forall i \in \mathrm{dom}(\mathcal{I}).\ t \in \mathcal{I}(i) \Leftrightarrow i = M(t) \\ \textit{Lock Permissions} & \mathcal{L} & ::= & L & L \subseteq \mathbb{N} \\ \textit{Unlock Permissions} & \mathcal{U} & ::= & (U, n) & U \subseteq \mathbb{N} \wedge \forall t \in U.\ t \geq n \end{array}$$

A ticket allocation is either $\mathsf{My}(\mathcal{I})$—which asserts the right to lock for the indices in $\mathrm{dom}(\mathcal{I})$, as well as the knowledge of all tickets allocated to those indices ($\mathcal{I}$ itself)—or else $\mathsf{All}(\mathcal{I}, M)$, which asserts the right to lock for the indices in $\mathrm{dom}(\mathcal{I})$, as well as the knowledge of *all* tickets allocated so far. There is a side condition on well-definedness of $\mathsf{All}(\mathcal{I}, M)$ insisting that $\mathcal{I}$ and M are coherent.

The unit of the ticket allocation monoid is $\mathsf{My}(\emptyset)$. Composition is defined as follows:

$$\begin{array}{rcll} \mathsf{My}(\mathcal{I}_1) \cdot \mathsf{My}(\mathcal{I}_2) & \triangleq & \mathsf{My}(\mathcal{I}_1 \uplus \mathcal{I}_2) & \\ \mathsf{My}(\mathcal{I}_1) \cdot \mathsf{All}(\mathcal{I}_2, M) & \triangleq & \mathsf{All}(\mathcal{I}_1 \uplus \mathcal{I}_2, M) & \text{if that is well-defined} \\ \mathsf{All}(\mathcal{I}_1, M) \cdot \mathsf{My}(\mathcal{I}_2) & \triangleq & \mathsf{All}(\mathcal{I}_1 \uplus \mathcal{I}_2, M) & \text{if that is well-defined} \end{array}$$

Lock permissions are the usual powerset monoid with disjoint union as composition and $\emptyset$ as unit. The set L represents the set of lock permissions one holds.

Unlock permissions (U, n) are similar, except that here we have the possibility of using a ticket up, after which point the "knowledge" that it is used up becomes a boxable (permanently true) assertion. The U represents the set of unlock permissions one holds, while n represents a lower bound on the unlock permissions that *anyone* can hold. (All unlock permissions less than n are to be viewed as "used up".) The side condition on well-definedness of monoid elements enforces that one cannot own an unlock permission that one knows has already been used up.

The unit of the monoid is $(\emptyset, 0)$. Composition is defined as follows:

$$(U_1, n_1) \cdot (U_2, n_2) \quad \triangleq \quad (U_1 \uplus U_2, \max(n_1, n_2)) \qquad \text{if that is well-defined}$$

We are now ready to define the abstract predicates used in the proof:

$$\begin{array}{rcl} \mathsf{Perms}^{\gamma}_{\geq}(i) & \triangleq & \boxed{\gamma : (\mathsf{My}(\emptyset), \{j \mid j \geq i\}, (\{j \mid j \geq i\}, 0))} \\ \mathsf{LkPerm}^{\gamma}(i) & \triangleq & \boxed{\gamma : (\mathsf{My}(\emptyset), \{i\}, (\emptyset, 0))} \\ \mathsf{UnPerm}^{\gamma}(i) & \triangleq & \boxed{\gamma : (\mathsf{My}(\emptyset), \emptyset, (\{i\}, 0))} \\ \mathsf{UsedUP}^{\gamma}_{<}(i) & \triangleq & \boxed{\gamma : (\mathsf{My}(\emptyset), \emptyset, (\emptyset, i))} \\ \mathsf{MyTkts}^{\gamma}(i, T) & \triangleq & \boxed{\gamma : (\mathsf{My}([i \mapsto T]), \emptyset, (\emptyset, 0))} \\ \mathsf{AllTkts}^{\gamma}(M) & \triangleq & \boxed{\gamma : (\mathsf{All}(\emptyset, M), \emptyset, (\emptyset, 0))} \end{array}$$