EUROPEAN COMMISSION
SEVENTH FRAMEWORK PROGRAMME
FP7-ICT-2011-C
Grant agreement no.: 308830

Deliverable D3.1

Intermediate report for WP3: Reasoning about software/hardware interfaces

Project acronym	ADVENT
Project title	Architecture-driven verification of systems software
Funding scheme	FP7 FET Young Explorers
Scientific coordinator	Dr. Alexey Gotsman, IMDEA Software Institute, Alexey.Gotsman@imdea.org, +34 911 01 22 02

1 Summary

Work Package 3 in the ADVENT project is concerned with reasoning about the software/hardware interfaces, and specifically with how to deal the implications of low-level hardware effects when reasoning about systems software. Through the first year of the project, we have focused on *relaxed* or *weak consistency* as the most important hardware effect that affects systems software and yet evades formal reasoning.

To enable formal reasoning about relaxed consistency, we have devised both decomposition principles (abstraction theorems) as well as program logics for verifying concurrent programs running under various weak memory consistency models, such as the TSO memory model adopted by the Sparc and x86 architectures and the C11 memory model adopted by C and C++ language standards. In more detail, we have developed:

- An *abstraction theorem for the C11 memory model* [1], which allows us to decompose the reasoning about concurrent programs into modular reasoning about each of its components. In essence, our abstraction theorem extends the definition of linearisability to the C11 setting, where executions are not longer linear event sequences.
- *Relaxed separation logic* (RSL) [15], an extension of concurrent separation logic to C11 model, allowing threads to assert ownership of part of the shared state and to transfer ownership by synchronisation operations.
- *GPS* [14], a more advanced program logic for C11 that incorporates adaptations of the three features of the more modern concurrent program logics, namely *ghost state*, *protocols*, and *separation*. GPS thus enables reasoning not only about shared state invariants, but also about its evolution over time.
- A *program logic for TSO* [7] that is designed to reason effectively about certain common synchronisation patterns that appear in TSO programs.

2 Motivation

In a sequential setting, the behavior of memory is simple: when a program reads a memory location, it receives the value that it wrote most recently to that location. In a concurrent setting, where the program does not define a total order on all memory accesses, however, this simple rule no longer applies. The question of what values may be yielded by reads in concurrent programs is known as the platform's *memory consistency model* (or *memory model* for short). In WP1, we identified three memory models as being of particular interest:

(1) the x86-TSO model followed by x86 and x64 architectures,

(2) the Power/ARM model, and

(3) the C11 memory model adopted by the 2011 revisions of the C and C++ standards.

In this work package, we have focused mainly on the C11 memory model because it is what C/C++ programmers face and it is easily compilable to the x86-TSO and Power/ARM hardware memory models. We have developed three logics, two for reasoning about concurrent C11 the C11 memory model and one about x86-TSO. The general goal is to help programmers by providing them with reasoning principles for concurrent programs that are sound despite the rather weak consistency properties guaranteed by the memory models.

3 Reasoning Principles for C11 Concurrency

Library Decomposition [1] Our first contribution was to propose techniques for decomposing the reasoning about programs using concurrent libraries on the C11 memory model into separate reasoning about the libraries and their clients. (This work was done in between the submission of the project proposal and the project start, and appeared at POPL'13. We briefly described it here as it is relevant to the project.)

In more detail, we have proposed a criterion that allows to abstract a concurrent library by a simpler library serving as its specification in reasoning about the client. This criterion satisfies the *Abstraction Theorem*: if one library (a specification) abstracts another (an implementation), then the behaviours of any client using the implementation are contained in the behaviours of the client using the specification. This result allows complex library code to be replaced by simpler specifications, for verification or informal reasoning. To justify the practicality of the proposed criterion for library abstraction we have applied it two typical concurrent algorithms: a non-blocking stack and an array-based queue.

The challenge in getting such a result on the C/C++ memory model lied in soundly capturing the client-library interactions through the subtle synchronisation effects arising from the memory model. This was made only more challenging by the fact that the C11 memory model is defined axiomatically, whereas existing techniques for library abstraction, such as linearizability, have focused on operational trace-based models. To deal with this, we proposed a novel notion of a *history*, which records all interactions between a client and a library. Histories in our work consist of several partial orders on call and return actions. This is in contrast to variants of linearizability, where histories are linear sequences. We then defined an abstraction relation on histories as inclusion over partial orders, and lift this relation to give our abstraction criterion for libraries: one library abstracts another if any history of the former can be reproduced in abstracted form by the latter.

During this work, we uncovered a problem with the C11 memory model: some aspects of the model conflict with abstraction. The C11 model permits *satisfaction cycles*, where the effect of actions executed down a conditional branch is what causes the branch to be taken in the first place. This breaks the straightforward assumption that faults are confined to either client or library code: a misbehaving client can cause misbehaviour in a library, which can in turn cause the original client misbehaviour! For these reasons, we actually defined *two* distinct library abstraction criteria: one for general C11, and one for a language without the feature leading to satisfaction cycles. The former requires an a priori check that the client and the library do not access each others' internal memory locations, which hinders compositionality. The latter lifts this restriction (albeit for a C11 model modified to admit incomplete program runs) and thus provides evidence that satisfaction cycles are to blame for non-compositional behaviour. Our results have motivated an ongoing effort to revise the memory model:

`http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3710.html`

Relaxed separation logic (RSL) [15] While having developed decomposition principles is clearly a very useful step for reasoning about complex software, we also need to have techniques for verifying the individual components with respect to their specifications. To achieve this, we have developed two program logics for C11 concurrency, *RSL* and *GPS*.

The goal of relaxed separation logic (RSL) was to show that C11 concurrency supports resource reasoning in the style of concurrent separation logic (CSL) [11], and that proofs written in CSL can easily be adapted to work under weaker consistency guarantees.

Specifically, we focused on the various different kinds of memory accesses supported by C11: nonatomic, relaxed atomic, acquire atomic, release atomic, and sequentially consistent (SC) atomic, each of which provides different consistency guarantees, and has different implementation costs. On the one end of the spectrum, races on non-atomic accesses result in completely

undefined behaviour (they are treated as programming errors); on the other end, SC-atomic accesses are globally synchronized. The implementation cost and the guarantees provided by relaxed, acquire, and release accesses lie somewhere in between: different threads can observe them happening in different orders, but fewer or no memory fences are needed (depending on the architecture).

What we managed to show with RSL, is that ownership-based reasoning is sound, and moreover that ownership can be transferred along acquire/release atomic memory accesses and does not require the more expensive SC-accesses.

In more detail, RSL follows the resourceful reading of separation logic triples. As in standard separation logic, when we assert the Hoare triple $\{P\}\ C\ \{Q\}$, we say that the command C will not access any memory other than that given by its precondition, P, or subsequently acquired during its execution. We thus support the parallel composition rule of separation logic,

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \qquad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * P_2\}\ C_1 \| C_2\ \{Q_1 * Q_2\}} \tag{Par}$$

which ensures that the two threads do not have any races on nonatomic memory accesses, a condition required by C11. To handle acquire/release atomics, we introduce two new assertion kinds, $\mathsf{Rel}(\ell, \mathcal{Q}_1)$ and $\mathsf{Acq}(\ell, \mathcal{Q}_2)$. These denote respectively the permissions to perform a release-write of some value v at location ℓ and give away ownership of the resource described by $\mathcal{Q}_1(v)$, or an acquire-read and gain ownership of $\mathcal{Q}_2(v)$. With these assertion forms we provide simple proof rules for release writes and acquire reads, similar to those for releasing and acquiring mutual exclusion locks in CSL.

The main challenge in this work was to prove the soundness of the program logic, as there were two really significant obstacles we had to overcome.

1. *No global notions of state and time*: Traditionally, $\{P\}\ C\ \{Q\}$ asserts that if we execute C in an initial state satisfying P and it terminates, then the final state will satisfy Q. In C11 concurrency, however, the terms "initial state" and "final state" are ill-defined, because there exist no global notions of time or state. To interpret triples, we thus resorted to *logical* local notions of time and state. We defined a logical notion of local state at each event of a program execution, and thread the logical state through C11 "happens-before" edges.

2. *No operational semantics*: Concurrent program logics are typically proved sound over an operational or a trace semantics. In either case, the meaning of Hoare triples can be defined in terms of an auxiliary predicate by induction over the length of an execution trace. These definitions cannot directly be extended to the C11 axiomatic model as there is no obvious total order for the induction. Our solution was to order the C11 events according to the total number of events that happen before them.

The paper introducing RSL was published at OOPSLA 2013 [15] and is attached. We are currently working on extensions of RSL to handle memory barriers.

GPS [14] Our next contribution was *GPS*, a more advanced program logic for C11 concurrency, published as a MPI-SWS technical report. A slightly updated version of the technical report is currently under submission to OOPSLA 2014.

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

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

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

  More generally, since invariants can be overly restrictive, modern logics support various forms of *protocols* for legislating interference. The best-known protocol mechanism is *rely-guarantee* [8], which describes the state transitions a thread may perform (the guarantee) vs. those its environment may perform (the rely). Recent protocol mechanisms improve upon rely-guarantee by supporting more abstract/concise forms of shared state transition systems [13].

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

In newer logics, ghost state, protocols, and separation are used in tandem to great effect. For example, ghost state can be used to encode logical "permissions" (or "tokens"), which are ownable resources that control the ability to make certain transitions in shared state protocols. Ownership of permissions can then be transferred back and forth between threads *via* the same shared protocols, in turn providing a way to model the dynamic "role-playing" that occurs in realistic concurrent code. Logics such as RGSep [16], LRG [6], Deny-Guarantee [5], VCC [2], Chalice [9], CAP [4], CaReSL [13], FCSL [10], iCAP [12], and TaDA [3] depend on such a synthesis of ghost state, protocols, and separation.

While the aforementioned mechanisms provide powerful, modular reasoning about concurrency, there are serious obstacles to adapting them to weak memory models. Besides those mentioned in the section about RSL, we also have:

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

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

GPS is the first logic that properly supports ghost state, protocols and separation in a weak memory setting. GPS builds on the groundwork laid by RSL, extending it as follows:

- *Protocols.* GPS supports *per-location (PL) protocols*, which are modeled after the protocols in recent concurrency logics but restricted in order to be sound under weak memory. The key to regaining soundness is to insist that a protocol may only precisely dictate the evolution of a *single* shared memory location, although it may make bounded assertions about the state of other memory locations, e.g., "x's value may only grow over time, and when x contains n, y must contain *at least* n as well."

- *Ghost state.* The states of PL-protocols already constitute a useful form of ghost state for summarizing, e.g., the history of an execution. To support ownable logical resources (e.g., permissions), GPS offers an additional facility called *ghosts*. Ghosts enable one to create and manipulate whatever kind of logical resource one needs for a particular verification, so long as it can be formulated as a partial commutative monoid.

As with RSL, the entire logic, model and soundness proof of GPS have been formalized in Coq and are available online.

4 Reasoning Principles for x86-TSO

Besides reasoning about the C11 memory model, we have also developed a program logic for reasoning about concurrent programs running under the TSO memory model. We have extended separation logic with a notion of *TSO spaces*. TSO spaces are similar to the shared resources of Concurrent Separation Logic (CSL) [11], except that memory owned by a TSO space may be accessed directly via TSO operations (memory accesses with TSO semantics) rather than using classical critical sections. Also, where CSL associates a resource invariant with each resource, we associate an *abstract state space* with each TSO space, as well as an *abstraction predicate* that associates each abstract state with a corresponding separation logic formula. The abstract state space is equipped with an *abstract reachability* pre-order $\preceq$.

Knowledge about the state of a TSO space in the proof of a thread is represented as an abstract state, representing the thread's view of the state of the TSO space. A thread's view is a lower bound (under the abstract reachability pre-order) on the actual state of the TSO space.

Each TSO write operation is associated with an *abstract state transition function*, mapping an abstract pre-state to an abstract post-state. These abstract state transition functions must respect the abstract reachability relation and properly abstract the concrete behavior of the TSO operation. Also, crucially, to account for TSO's relaxed behavior, the abstract state transition functions must be *monotonic*: they must respect reachability and be sound with respect to concrete behavior not only in the current abstract state, but also in all reachable future abstract states. In a thread's proof, when the thread performs a TSO write operation, its local view of the TSO space is updated per the abstract state transition function.

Each TSO read operation is associated with a function, f, mapping result values to new abstract states. This function must satisfy the property that for any result value, v, and for any future abstract state, α', if the target location may have value v in this abstract state, then $f(v) \preceq \alpha'$. In a thread's proof, when the thread performs a TSO read operation, its local view of the TSO space is replaced with the (hopefully more precise) lower bound given by function f.

For example, consider an implementation of a Java virtual machine, where field accesses are implemented as TSO operations. Threads may create objects and leak references to those objects to other threads without synchronization. The VM implementation must ensure nonetheless that each thread only sees objects with properly initialized run-time type information (such as the pointer to the virtual method dispatch table).

To verify such a JVM implementation using the proposed logic, we put the heap in a TSO space. As the abstract state space, we adopt the powerset of addresses in this heap, each state representing the set of the addresses of the currently allocated and initialized objects.

The subset relation serves as the reachability order. The abstraction predicate states (1) that allocated objects occupy disjoint heap space, (2) that they properly point to an existing virtual method dispatch table, and (3) that their fields point to allocated objects. Each thread is aware of the addresses of objects it allocated itself, as well as addresses read from fields of known objects. That is, reading a field updates the thread's view by inserting the newly discovered object address into the abstract state. After a thread allocates an object and initializes its run-time type information, it performs a no-op operation on the TSO space to update its local view of the set of allocated objects, inserting the newly allocated object into the abstract state. Writing the value of a local variable to a field corresponds to the identity function at the abstract level, since all object references a thread holds in local variables are already in the thread's local view.

The approach is described in more detail in [7]. Besides a definition and soundness argument of the approach, this document also reports on a preliminary encoding of this approach into the logic of the VeriFast program verifier, and on three example proofs: an example that captures the virtual machine scenario, a lock implementation, and a producer-consumer example.

References

[1] M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency. In *POPL*, pages 235–248. ACM, 2013.

[2] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.

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

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

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

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

[7] B. Jacobs. Verifying TSO programs. Technical Report CW-660, Department of Computer Science, KU Leuven, 2014.

[8] C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, 1983.

[9] K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In *FOSAD*, volume 5705 of *LNCS*, pages 195–222. Springer, 2009.

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

[11] P. O'Hearn. Resources, concurrency and local reasoning. *TCS*, 2007.

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

[13] A. Turon, D. Dreyer, and L. Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*, pages 377–390. ACM, 2013.

[14] A. Turon, V. Vafeiadis, and D. Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. Technical Report MPI-SWS-2014-004, MPI-SWS, 2014.

[15] V. Vafeiadis and C. Narayan. Relaxed separation logic: a program logic for c11 concurrency. In *OOPSLA*, pages 867–884, 2013.

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

List of Attached Papers

[15] Viktor Vafeiadis and Chinmay Narayan. Relaxed separation logic: a program logic for c11 concurrency. In *OOPSLA'13*, pages 867–884. ACM 2013.

[14] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. Technical Report MPI-SWS-2014-004, MPI-SWS, 2014.

[7] Bart Jacobs. Verifying TSO programs. Technical Report CW-660, Department of Computer Science, KU Leuven, 2014.

Relaxed Separation Logic: A Program Logic for C11 Concurrency

Viktor Vafeiadis
Max Planck Institute for Software Systems (MPI-SWS)
viktor@mpi-sws.org

Chinmay Narayan
Indian Institute of Technology, Delhi
chinmay@cse.iitd.ac.in

Abstract

We introduce *relaxed separation logic* (RSL), the first program logic for reasoning about concurrent programs running under the C11 relaxed memory model. From a user's perspective, RSL is an extension of concurrent separation logic (CSL) with proof rules for the various kinds of C11 atomic accesses. As in CSL, individual threads are allowed to access non-atomically only the memory that they own, thus preventing data races. Ownership can, however, be transferred via certain atomic accesses. For SC-atomic accesses, we permit arbitrary ownership transfer; for acquire/release atomic accesses, we allow ownership transfer only in one direction; whereas for relaxed atomic accesses, we rule out ownership transfer completely. We illustrate RSL with a few simple examples and prove its soundness directly over the axiomatic C11 weak memory model.

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

Keywords Concurrency; Weak memory models; C/C++; Proof system; Separation logic

1. Introduction

Wanting to enable many hardware and software optimizations, modern programming language definitions provide rather weak guarantees on the semantics of concurrent memory accesses allowing, for example, different threads to observe shared operations happening in different orders. One such case is the concurrency model adopted by the 2011 revisions of the C and C++ standards (ISO/IEC 9899:2011; ISO/IEC 14882:2011), which we will study in this paper and refer to as the C11 model.

C11 provides several kinds of memory accesses—non-atomic, relaxed atomic, acquire atomic, release atomic, and sequentially consistent (SC) atomic—each providing different consistency guarantees. On the one end of the spectrum, races on non-atomic accesses result in completely undefined behaviour (they are treated as programming errors); on the other end, SC-atomic accesses are globally synchronized. The guarantees provided by relaxed, acquire, and release accesses lie somewhere in between: different threads can observe them happening in different orders.

The reason for having all these kinds of accesses is that they map differently to the various common architectures, and have very different implementation costs. Non-atomic and relaxed atomic accesses are generally rather cheap as they correspond to vanilla machine loads and stores, and may be reordered by the compiler and/or by an out-of-order execution unit. At the other end of the spectrum, SC accesses are very expensive because their implementation involves a full memory barrier. The cost of acquire and release accesses depends a lot on the architecture. On x86, they are compiled down to plain reads and writes (Batty et al. 2011) and are therefore cheap. On PowerPC and ARM, the cost is somewhat higher as they induce a memory barrier, but of a weaker kind than full memory barriers (Sarkar et al. 2012).

Our goal is to help C11 programmers by providing them with sound reasoning principles for concurrent programs. We show that C11 concurrency supports resource reasoning in the style of separation logic (O'Hearn 2007); in particular, ownership can be transferred along acquire/release atomic memory accesses and does not require SC-accesses.

We develop *relaxed separation logic* (RSL), a program logic that follows the resourceful reading of separation logic triples. When we assert the Hoare triple $\{P\}\ Cmd\ \{Q\}$, we say that the command Cmd will not access any memory other than that given by its precondition, P, or subsequently acquired during its execution. We thus support the parallel composition rule of separation logic,

$$\frac{\{P_1\}\ Cmd_1\ \{Q_1\} \qquad \{P_2\}\ Cmd_2\ \{Q_2\}}{\{P_1 * P_2\}\ Cmd_1 \| Cmd_2\ \{Q_1 * Q_2\}} \quad \text{(PAR)}$$

OOPSLA '13, October 29–31, 2013, Indianapolis, Indiana, USA.

ACM 978-1-4503-2374-1/13/10. . . $15.00.
http://dx.doi.org/10.1145/2509136.2509532

which ensures that the two threads do not have any races on non-atomic memory accesses, a condition required by C11.

To handle acquire/release atomics, we introduce two new assertion kinds, $\mathsf{Rel}(\ell, \mathcal{Q}_1)$ and $\mathsf{Acq}(\ell, \mathcal{Q}_2)$. These denote respectively the permissions to perform a release-write of some value v at location ℓ and give away ownership of the resource described by $\mathcal{Q}_1(v)$, or an acquire-read and gain ownership of $\mathcal{Q}_2(v)$. With these assertion forms we provide simple proof rules for release writes and acquire reads, similar to those for releasing and acquiring mutual exclusion locks in concurrent separation logic.

Besides RSL itself, the main contribution of this work was to define the meaning of Hoare triples in a relaxed memory model setting, so as to prove the soundness of RSL. This was rather challenging for three main reasons.

No global state/time: Traditionally, $\{P\}\ Cmd\ \{Q\}$ asserts that if we execute Cmd in an initial state satisfying P and it terminates, then the final state will satisfy Q. In C11 concurrency, however, the terms "initial state" and "final state" are ill-defined, because there exist no global notions of time or state.

To interpret triples, we thus resort to *logical* local notions of time and state. We define a logical notion of local state at each event of a program execution, and thread the logical state through C11 "happens-before" edges.

Assertions in heaps: Our assertions for dealing with acquire and release atomics require that the logical heaps used to interpret them contain assertions. This results in a circularity in the model of assertions, which for simplicity we resolve by storing syntactic assertions.

No operational semantics: Concurrent program logics are typically proved sound over an operational or a trace semantics. In either case, the meaning of Hoare triples can be defined in terms of an auxiliary predicate by induction over the length of an execution trace. These definitions cannot directly be extended to the C11 model as there is no obvious total order for the induction. Our solution is to order the C11 events according to the total number of events that happen before them.

As a secondary contribution, we observed that the semantics of relaxed atomic memory accesses in C11 is too weak to permit even the most basic reasoning principles about them, which in turn renders basic compiler optimizations unsound. In order to allow such reasoning principles, we proposed a crude fix to C11, which we discuss in Section 6.

In the remainder of this paper, we define a minimal concurrent programming language (§2), review the C11 concurrency model (§3), describe the assertions and proofs rules of RSL (§4), verify a few examples using RSL (§5), explain the problems caused by relaxed accesses and their resolution (§6), present the semantics of assertions and Hoare triples and sketch the main parts of the soundness proof (§7). We conclude with a discussion of related and future work (§8).

A Coq formalization of the soundness proof of RSL can be found at the following URL.

`http://www.mpi-sws.org/~viktor/rsl/`

2. Programming Language

In order to focus on the concurrency aspects of C11 and to avoid the inherent complexity of a large language like C, we introduce a minimal concurrent programming language featuring the various kinds of memory accesses supported by C11. Following Batty et al. (2012), we omit consume reads from the model, because they are relevant only for a few architectures (PowerPC and ARM) and substantially complicate the model. For simplicity, we also omit memory fences.

To make the local sequential execution order explicit, we present the grammar of expressions in A-normal form (cf. Flanagan et al. 1993). Atomic expressions, $e \in \mathsf{AExp}$, consist of variables and values (locations and numbers). Program expressions, $E \in \mathsf{Exp}$, consist of atomic expressions, let-bound computations, conditionals, loops, parallel composition, memory allocation, loads, stores, and atomic compare-and-swap ($\mathbf{CAS}$) instructions.

$$
\begin{aligned}
v \in \mathsf{Val} &::= \ell \mid n && \text{where } \ell \in \mathsf{Loc}, n \in \mathbb{N} \\
e \in \mathsf{AExp} &::= x \mid v && \text{where } x \in \mathsf{Var} \\
E \in \mathsf{Exp} &::= e \mid \mathbf{let}\ x = E\ \mathbf{in}\ E' \mid \mathbf{if}\ e\ \mathbf{then}\ E\ \mathbf{else}\ E' \\
&\quad \mid \mathbf{repeat}\ E\ \mathbf{end} \mid E_1 \| E_2 \mid \mathbf{alloc}() \\
&\quad \mid [e]_X \mid [e]_Y := e' \mid \mathbf{CAS}_{Z,W}(e, e', e'')
\end{aligned}
$$

where $X \in \{\mathrm{sc}, \mathrm{acq}, \mathrm{rlx}, \mathrm{na}\}, Y \in \{\mathrm{sc}, \mathrm{rel}, \mathrm{rlx}, \mathrm{na}\}$,
$Z \in \{\mathrm{sc}, \mathrm{rel\_acq}, \mathrm{acq}, \mathrm{rel}, \mathrm{rlx}\}, W \in \{\mathrm{sc}, \mathrm{acq}, \mathrm{rlx}\}$

As in C, in conditional expressions we treat zero as false and non-zero values as true. The construct $\mathbf{repeat}\ E\ \mathbf{end}$ executes E repeatedly until it returns a non-zero value.

Memory accesses are annotated by their mode: sequentially consistent (sc), acquire (acq), release (rel), combined release-acquire ($\mathrm{rel\_acq}$), relaxed (rlx), or non-atomic (na). According to the C standard, not all modes are available for all accesses: reads cannot be releases, writes cannot be acquires, $\mathbf{CAS}$s cannot be non-atomic. These restrictions are to avoid redundancy in the language. For example, an acquire write, if such a thing were allowed, would behave exactly the same as a relaxed write.

$\mathbf{CAS}$ is an atomic operation used to heavily in lock-free concurrent algorithms. It takes a location, ℓ, and two values, v' and v'', as arguments. It atomically checks if the value at location is v' or not. If the value is same as v', then $\mathbf{CAS}$ succeeds: it atomically writes v'' to ℓ and returns the old value. If the value is different, $\mathbf{CAS}$ fails: it returns that value and does not modify the location. $\mathbf{CAS}$ is annotated with two access modes: one to be used for the successful case, and one for the unsuccessful case.

For conciseness in examples, we will often write expressions such as $[E]_{\mathrm{na}}$ instead of $\mathbf{let}\ x = E\ \mathbf{in}\ [x]_{\mathrm{na}}$. We also write $E_1; E_2$ instead of $\mathbf{let}\ x = E_1\ \mathbf{in}\ E_2$ when $x \notin \mathsf{fv}(E_2)$.

$$
\begin{aligned}
\mathit{new\_lock}() &= \mathbf{let}\ x = \mathbf{alloc}()\ \mathbf{in}\ [x]_{\mathrm{rel}} := 1; x \\
\mathit{spin}(x) &= \mathbf{repeat}\ [x]_{\mathrm{rlx}}\ \mathbf{end} \\
\mathit{lock}(x) &= \mathbf{repeat}\ \mathit{spin}(x); \mathbf{CAS}_{\mathrm{acq,rlx}}(x, 1, 0)\ \mathbf{end} \\
\mathit{unlock}(x) &= [x]_{\mathrm{rel}} := 1
\end{aligned}
$$

Figure 1. Simple spinlock implementation.

Spinlock Example There are two important uses of acquire/release accesses: in implementing locks, and in message passing. Relaxed accesses are useful in cases of optimistic reads, where the value read, if it is of interest to the algorithm, will be read again by an acquire read, or an acquire fence will be issued. For example, in the simple CAS-based spinlock implementation shown in Figure 1, $\mathit{lock}(x)$ performs an acquire-on-success CAS and $\mathit{unlock}(x)$ does a release write. The optimistic $\mathit{spin}(x)$ loop that waits for the lock to become free, in contrast, does relaxed reads. The combined release-acquire CAS is supposed to be used for operations that atomically release one lock and acquire another—this is possible, for example, if the locks are represented as different bits of the same word. Further examples can be found in McKenney and Garst (2011).

3. The C11 Memory Model

The C11 memory model is defined axiomatically in terms of program executions. A program *execution* consists of a set of actions and several binary relations over them. Actions describe the memory operations performed by the program, and are labelled with information about the memory order of the operation, the address accessed and the values read and/or written.

$$
\begin{aligned}
\mathsf{Act} ::= {} & \mathrm{skip} \mid \mathrm{W}_{(\mathrm{sc|rel|rlx|na})}(\ell, v) \mid \mathrm{R}_{(\mathrm{sc|acq|rlx|na})}(\ell, v) \\
& \mid \mathrm{RMW}_{(\mathrm{sc|rel\_acq|acq|rel|rlx})}(\ell, v, v') \mid \mathrm{A}(\ell)
\end{aligned}
$$

In summary, we have a no-op action; SC, release, relaxed and non-atomic writes; SC, acquire, relaxed, and non-atomic reads; atomic read-modify-write actions; and allocations. The no-op (skip) action represents local computations, thread forks and joins.

For the subset of C11 we consider, an execution contains the following relations:[1]

- *Sequenced-before* (sb) relates actions of the same thread that follow one another in control flow. We have $\mathsf{sb}(a, b)$ if a and b belong to the same thread and a *immediately* precedes b in the thread's control flow, or a is a fork action and b the first action of the forked thread, or b is a join action and a the last action of the joined thread.
- The *reads-from* map (rf) maps every read action r to the write action w that wrote the value read by r.
- The *memory-order* relation (mo) is a total order on the store actions writing to the same atomic location.
- The *sequential-consistency* order (sc) is a total order over all SC-atomic actions.

Formally, let AName be a countably infinite of action names. Then, an execution, $\mathcal{X}$, is represented as a tuple, $\langle \mathcal{A}, \mathsf{lab}, \mathsf{sb}, \mathsf{rf}, \mathsf{mo}, \mathsf{sc} \rangle$, where $\mathcal{A} \subseteq_{\mathrm{fin}} \mathsf{AName}$ is the set of action names included in the execution, $\mathsf{lab} \in \mathcal{A} \to \mathsf{Act}$ maps every action identifier to its label, $\mathsf{rf} \in \mathcal{A} \rightharpoonup \mathcal{A}$ is the reads-from map, and $\mathsf{sb}, \mathsf{mo}, \mathsf{sc} \in \mathcal{P}(\mathcal{A} \times \mathcal{A})$ are the sequenced-before relation, the memory order, and the sequential consistency order respectively.

From these relations, C11 defines a number of derived relations, the most important of which are: the *synchronizes-with* relation and the *happens-before* order.

- *Synchronizes-with* (sw) relates acquire reads with the release writes that precede in mo order the write whose value was read by the acquire read provided that all the writes between these two writes belong to the same thread or are RMW operations.
- *Happens-before* (hb) is a partial order on actions formalizing the intuition that one action was completed before the other. In the C11 subset we consider, $\mathsf{hb} = (\mathsf{sb} \cup \mathsf{sw})^+$.

The semantics of a program is given by the set of *consistent* executions. An execution is said to be consistent if it satisfies the axioms of the memory model, which will be presented shortly. If, however, any of these consistent executions contains a data race on non-atomic actions, i.e. events generated from two conflicting operations on the same non-atomic location not ordered by hb in either direction, then the program is deemed to have arbitrary semantics. Thus, any sound program logic for C11 concurrency must ensure its specifications imply race-freedom for non-atomic actions.

Expression Semantics Let CExp denote closed expressions (i.e., ones with no free variables). The semantics of such closed expressions, $[\![E]\!]$, is given in Figure 2 as a set of tuples $\langle \mathit{res}, \mathcal{A}, \mathsf{lab}, \mathsf{sb}, \mathit{fst}, \mathit{lst} \rangle$. These tuples represent finite complete executions as well as finite incomplete execution prefixes (used to model infinite executions), where:

(1) res is the result of evaluating the expression or $\bot$ if the execution is incomplete;
(2) $\mathcal{A}$ is the set of all actions contained in the execution;
(3) lab labels the actions with the corresponding operations;
(4) sb represents the sequenced-before relation; and
(5) fst and lst are the first and last actions in the sb-order.

For uniformity, we record the last action even in incomplete executions. In the parallel composition case, the auxiliary function $\mathsf{combine}(\mathit{res}_1, \mathit{res}_2)$ returns res_1 if $\mathit{res}_2 \neq \bot$ and $\bot$ otherwise. In the $[\![E]\!]$ semantics, allocations can return an arbitrary new location, and reads can read an arbitrary value. These will later be constrained by the consistency axioms.

[1] The full model includes two additional relations, dd (data dependency) and dob (dependency ordered before), used to define the happens-before relation for consume reads.

$$[\![-]\!] \;:\; \mathsf{CExp} \to \mathbb{P}(\langle res : \mathsf{Val} \cup \{\bot\}, \mathcal{A} : \mathbb{P}(\mathsf{AName}), \mathsf{lab} : \mathcal{A} \to \mathsf{Act}, \mathsf{sb} : \mathbb{P}(\mathcal{A} \times \mathcal{A}), fst : \mathcal{A}, lst : \mathcal{A}\rangle)$$

$$[\![v]\!] \stackrel{\text{def}}{=} \{\langle v, \{a\}, \mathsf{lab}, \emptyset, a, a\rangle \mid a \in \mathsf{AName} \wedge \mathsf{lab}(a) = \mathrm{skip}\}$$

$$[\![\mathbf{alloc}()]\!] \stackrel{\text{def}}{=} \{\langle \ell, \{a\}, \mathsf{lab}, \emptyset, a, a\rangle \mid a \in \mathsf{AName} \wedge \ell \in \mathsf{Loc} \wedge \mathsf{lab}(a) = \mathrm{A}(\ell)\}$$

$$[\![[v]_Z := v']\!] \stackrel{\text{def}}{=} \{\langle v', \{a\}, \mathsf{lab}, \emptyset, a, a\rangle \mid a \in \mathsf{AName} \wedge \mathsf{lab}(a) = \mathrm{W}_Z(v, v')\}$$

$$[\![[v]_Z]\!] \stackrel{\text{def}}{=} \{\langle v', \{a\}, \mathsf{lab}, \emptyset, a, a\rangle \mid a \in \mathsf{AName} \wedge v' \in \mathsf{Val} \wedge \mathsf{lab}(a) = \mathrm{R}_Z(v, v')\}$$

$$\begin{aligned}[\![\mathbf{CAS}_{X,Y}(v, v_o, v_n)]\!] \stackrel{\text{def}}{=} {} & \{\langle v', \{a\}, \mathsf{lab}, \emptyset, a, a\rangle \mid a \in \mathsf{AName} \wedge v' \in \mathsf{Val} \wedge v' \neq v_o \wedge \mathsf{lab}(a) = \mathrm{R}_Y(v, v')\} \\ \cup {} & \{\langle v_o, \{a\}, \mathsf{lab}, \emptyset, a, a\rangle \mid a \in \mathsf{AName} \wedge \mathsf{lab}(a) = \mathrm{RMW}_X(v, v_o, v_n)\}\end{aligned}$$

$$\begin{aligned}[\![\mathbf{let}\ x = E_1\ \mathbf{in}\ E_2]\!] \stackrel{\text{def}}{=} {} & \{\langle \bot, \mathcal{A}_1, \mathsf{lab}_1, \mathsf{sb}_1, fst_1, lst_1\rangle \mid \langle \bot, \mathcal{A}_1, \mathsf{lab}_1, \mathsf{sb}_1, fst_1, lst_1\rangle \in [\![E_1]\!]\} \\ \cup {} & \{\langle res_2, \mathcal{A}_1 \uplus \mathcal{A}_2, \mathsf{lab}_1 \cup \mathsf{lab}_2, \mathsf{sb}_1 \cup \mathsf{sb}_2 \cup \{(lst_1, fst_2)\}, fst_1, lst_2\rangle \mid \\ & \quad \langle v_1, \mathcal{A}_1, \mathsf{lab}_1, \mathsf{sb}_1, fst_1, lst_1\rangle \in [\![E_1]\!] \wedge \langle res_2, \mathcal{A}_2, \mathsf{lab}_2, \mathsf{sb}_2, fst_2, lst_2\rangle \in [\![E_2[v_1/x]]\!]\}\end{aligned}$$

$$\begin{aligned}[\![\mathbf{repeat}\ E\ \mathbf{end}]\!] \stackrel{\text{def}}{=} {} & \{\langle res_N, \biguplus_{i\in[1..N]} \mathcal{A}_i, \bigcup_{i\in[1..N]} \mathsf{lab}_i, \bigcup_{i\in[1..N]} \mathsf{sb}_i \cup \{(lst_1, fst_2), \ldots, (lst_{N-1}, fst_N)\}, fst_1, lst_N\rangle \mid \\ & \quad \forall i.\ \langle res_i, \mathcal{A}_i, \mathsf{lab}_i, \mathsf{sb}_i, fst_i, lst_i\rangle \in [\![E]\!] \wedge (i \neq N \implies res_i = 0) \wedge res_N \neq 0\}\end{aligned}$$

$$\begin{aligned}[\![E_1 \| E_2]\!] \stackrel{\text{def}}{=} {} & \{\langle \mathsf{combine}(res_1, res_2), \mathcal{A}_1 \uplus \mathcal{A}_2 \uplus \{a_{\text{fork}}, a_{\text{join}}\}, \mathsf{lab}_1 \cup \mathsf{lab}_2 \cup \{a_{\text{fork}} \mapsto \mathrm{skip}, a_{\text{join}} \mapsto \mathrm{skip}\}, \\ & \quad \mathsf{sb}_1 \cup \mathsf{sb}_2 \cup \{(a_{\text{fork}}, fst_1), (a_{\text{fork}}, fst_2), (lst_1, a_{\text{join}}), (lst_2, a_{\text{join}})\}, a_{\text{fork}}, a_{\text{join}}\rangle \mid \\ & \quad \langle res_1, \mathcal{A}_1, \mathsf{sb}_1, fst_1, lst_1\rangle \in [\![E_1]\!] \wedge \langle res_2, \mathcal{A}_2, \mathsf{sb}_2, fst_2, lst_2\rangle \in [\![E_2]\!] \wedge a_{\text{fork}}, a_{\text{join}} \in \mathsf{AName}\}\end{aligned}$$

Figure 2. Semantics of closed program expressions.

$$\nexists x.\ \mathsf{hb}(x, x) \qquad \text{(IrreflexiveHB)}$$

$$\forall \ell.\ \mathsf{totalorder}(\{a \in \mathcal{A} \mid \mathsf{iswrite}_\ell(a)\}, \mathsf{mo}) \wedge \mathsf{hb}_\ell \subseteq \mathsf{mo} \qquad \text{(ConsistentMO)}$$

$$\mathsf{totalorder}(\{a \in \mathcal{A} \mid \mathsf{isSeqCst}(a)\}, \mathsf{sc}) \wedge \mathsf{hb}_{\mathsf{SeqCst}} \subseteq \mathsf{sc} \wedge \mathsf{mo}_{\mathsf{SeqCst}} \subseteq \mathsf{sc} \qquad \text{(ConsistentSC)}$$

$$\forall b.\ \mathsf{rf}(b) \neq \bot \iff \exists \ell, a.\ \mathsf{iswrite}_\ell(a) \wedge \mathsf{isread}_\ell(b) \wedge \mathsf{hb}(a, b) \qquad \text{(ConsistentRFdom)}$$

$$\forall a, b.\ \mathsf{rf}(b) = a \implies \exists \ell, v.\ \mathsf{iswrite}_{\ell,v}(a) \wedge \mathsf{isread}_{\ell,v}(b) \wedge \neg\mathsf{hb}(b, a) \qquad \text{(ConsistentRF)}$$

$$\forall a, b.\ \mathsf{rf}(b) = a \wedge (\mathsf{mode}(a) = \mathrm{na} \vee \mathsf{mode}(b) = \mathrm{na}) \implies \mathsf{hb}(a, b) \qquad \text{(ConsistentRFna)}$$

$$\forall a, b.\ \mathsf{rf}(b) = a \wedge \mathsf{isSeqCst}(b) \implies \mathsf{isc}(a, b) \vee \neg\mathsf{isSeqCst}(a) \wedge (\forall x.\ \mathsf{isc}(x, b) \Rightarrow \neg\mathsf{hb}(a, x)) \qquad \text{(RestrSCReads)}$$

$$\nexists a, b.\ \mathsf{hb}(a, b) \wedge \mathsf{mo}(\mathsf{rf}(b), \mathsf{rf}(a)) \wedge \mathsf{locs}(a) = \mathsf{locs}(b) \qquad \text{(CoherentRR)}$$

$$\nexists a, b.\ \mathsf{hb}(a, b) \wedge \mathsf{mo}(\mathsf{rf}(b), a) \wedge \mathsf{iswrite}(a) \wedge \mathsf{locs}(a) = \mathsf{locs}(b) \qquad \text{(CoherentWR)}$$

$$\nexists a, b.\ \mathsf{hb}(a, b) \wedge \mathsf{mo}(b, \mathsf{rf}(a)) \wedge \mathsf{iswrite}(b) \wedge \mathsf{locs}(a) = \mathsf{locs}(b) \qquad \text{(CoherentRW)}$$

$$\forall a.\ \mathsf{isrmw}(a) \wedge \mathsf{rf}(a) \neq \bot \implies \mathsf{mo}(\mathsf{rf}(a), a) \wedge \nexists c.\ \mathsf{mo}(\mathsf{rf}(a), c) \wedge \mathsf{mo}(c, a) \qquad \text{(AtomicRMW)}$$

$$\forall a, b, \ell.\ \mathsf{lab}(a) = \mathsf{lab}(b) = \mathrm{A}(\ell) \implies a = b \qquad \text{(ConsistentAlloc)}$$

where $\mathsf{iswrite}_{\ell,v}(a) \stackrel{\text{def}}{=} \exists X, v_{\text{old}}.\ \mathsf{lab}(a) \in \{\mathrm{W}_X(\ell, v), \mathrm{RMW}_X(\ell, v_{\text{old}}, v)\}$ $\quad \mathsf{iswrite}_\ell(a) \stackrel{\text{def}}{=} \exists v.\ \mathsf{iswrite}_{\ell,v}(a)$

$\mathsf{isread}_{\ell,v}(a) \stackrel{\text{def}}{=} \exists X, v_{\text{new}}.\ \mathsf{lab}(a) \in \{\mathrm{R}_X(\ell, v), \mathrm{RMW}_X(\ell, v, v_{\text{new}})\}$ $\quad$ etc.

$$\mathsf{rsElem}(a, b) \stackrel{\text{def}}{=} \mathsf{sameThread}(a, b) \vee \mathsf{isrmw}(b)$$

$$\mathsf{rseq}(a) \stackrel{\text{def}}{=} \{a\} \cup \{b \mid \mathsf{rsElem}(a, b) \wedge \mathsf{mo}(a, b) \wedge (\forall c.\ \mathsf{mo}(a, c) \wedge \mathsf{mo}(c, b) \Rightarrow \mathsf{rsElem}(a, c))\}$$

$$\mathsf{sw} \stackrel{\text{def}}{=} \{(a, b) \mid \mathsf{mode}(a) \in \{\mathrm{rel}, \mathrm{rel\_acq}, \mathrm{sc}\} \wedge \mathsf{mode}(b) \in \{\mathrm{acq}, \mathrm{rel\_acq}, \mathrm{sc}\} \wedge \mathsf{rf}(b) \in \mathsf{rseq}(a)\}$$

$$\mathsf{hb} \stackrel{\text{def}}{=} (\mathsf{sb} \cup \mathsf{sw})^+$$

$$\mathsf{hb}_\ell \stackrel{\text{def}}{=} \{(a, b) \in \mathsf{hb} \mid \mathsf{iswrite}_\ell(a) \wedge \mathsf{iswrite}_\ell(b)\}$$

$$X_{\mathsf{SeqCst}} \stackrel{\text{def}}{=} \{(a, b) \in X \mid \mathsf{isSeqCst}(a) \wedge \mathsf{isSeqCst}(b)\}$$

$$\mathsf{isc}(a, b) \stackrel{\text{def}}{=} \mathsf{iswrite}_{\mathsf{locs}(b)}(a) \wedge \mathsf{sc}(a, b) \wedge \nexists c.\ \mathsf{sc}(a, c) \wedge \mathsf{sc}(c, b) \wedge \mathsf{iswrite}_{\mathsf{locs}(b)}(c)$$

Figure 3. Axioms satisfied by consistent C11 executions, $\mathsf{Consistent}(\mathcal{A}, \mathsf{lab}, \mathsf{sb}, \mathsf{rf}, \mathsf{mo}, \mathsf{sc})$.

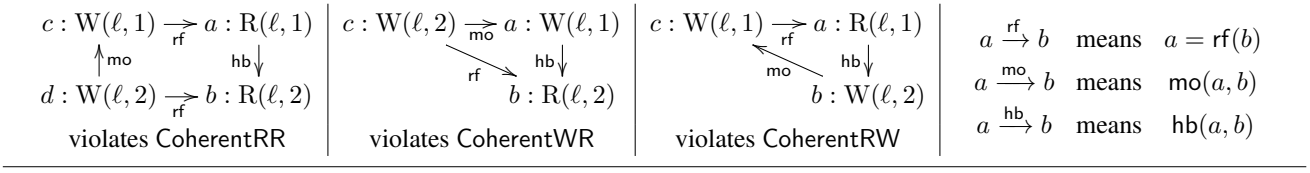


Figure 4. Sample executions violating coherency conditions (Batty et al. 2011).

Consistent Executions According to the C11 model, an execution is consistent, $\mathsf{Consistent}(\mathcal{A}, \mathsf{lab}, \mathsf{sb}, \mathsf{rf}, \mathsf{mo}, \mathsf{sc})$, if all of the properties shown in Figure 3 hold.

(IrreflexiveHB) The happens-before order, hb, must be irreflexive: an action cannot happen before itself.

(ConsistentMO) All write actions on an atomic location ℓ must be totally ordered by mo, and be consistently ordered by hb (restricted to the location ℓ).

(ConsistentSC) The sc relation must be a total order and include both hb and mo restricted to SC actions. This in effect means that SC actions are globally synchronized.

(ConsistentRFdom) The reads-from map, rf, is defined for those read (or RMW) actions for which the execution contains an earlier write (or RMW) to the same location.

(ConsistentRF) Each entry in the reads-from map, rf, should map a read to an earlier or concurrent write to the same location and with the same value.

(ConsistentRFna) Further, if a read reads from a write and either the read or write are non-atomic, then the write must have happened before the read. Batty et al. (2011) also require the write to be *visible*: i.e. not to have been overwritten by another write that happened before the read. This extra condition is unnecessary, as it follows from CoherentWR.

(RestrSCReads) SC reads are further restricted to read only from the immediately preceding SC write to the same location in sc order or from a non-SC write that has not happened before that immediately preceding SC write.

(CoherentRR,CoherentWR,CoherentRW) Next, we have three per-location coherence properties relating mo, hb, and rf. These properties require that mo never contradicts hb or the observed read order, and that rf never reads values that have been overwritten by more recent actions that happened before the read. These coherence properties are depicted in Figure 4.

(AtomicRMW) Each read-modify-write action should execute atomically: it should read from the immediately preceding write in mo.

(ConsistentAlloc) Finally, the same location cannot be allocated twice by different allocation actions.[2]

Remark Our model differs in a few minor ways from that of Batty et al. (2011, 2012). First, we have incorporated the C standard's "additional synchronized with" (asw) relation in sb rather than in sw, because it describes synchronization induced by control flow rather than by data flow.

Second, our sw-relation also relates acquire reads with release writes (whenever the read returns a value written by or after the release write), even if the two actions belong to the same thread (and are thus sb-related), whereas Batty et al. (2012) do not add any sb-related actions to the sw-relation. Since relating such actions also by sw does not affect execution consistency, we do so for uniformity, which eases the definition of validity of Hoare triples in §7.

[2] This axiom suffices, because we do not support deallocation. Had we included deallocation, we would instead require there to be a deallocation actions between any two allocation actions of the same location. The formalized C11 model by Batty et al. (2011, 2012) does not model allocation.

$$
\begin{array}{c}
\textbf{let } a = \mathbf{alloc}() \textbf{ in} \\
\textbf{let } c = \mathbf{alloc}() \textbf{ in} \\
[c]_{\mathrm{rlx}} := 0; \\
\left(\begin{array}{l} [a]_{\mathrm{na}} := 7; \\ [c]_{\mathrm{rel}} := 1 \end{array} \;\middle\|\; \begin{array}{l} \textbf{repeat } [c]_{\mathrm{acq}} \textbf{ end}; \\ [a]_{\mathrm{na}} := [a]_{\mathrm{na}} + 1 \end{array} \right)
\end{array}
$$

Figure 5. Message passing example showing transfer of ownership of the non-atomic location a.

Finally, in the standard, the sb and sw relations are taken to be strict partial orders, corresponding to the transitive closure of our relations. Conversely, our sb relation can be defined in terms of the sb order from the C and C++ standards as follows, $\mathsf{sb}_{\mathrm{our}} = \{(a,b) \in \mathsf{sb}_{\mathrm{std}} \cup \mathsf{asw}_{\mathrm{std}} \mid \nexists c.\ (a,c) \in \mathsf{sb}_{\mathrm{std}} \cup \mathsf{asw}_{\mathrm{std}} \wedge (c,b) \in \mathsf{sb}_{\mathrm{std}} \cup \mathsf{asw}_{\mathrm{std}}\}$. Again, we found the non-transitive versions slightly more convenient when defining the meaning of Hoare triples.

4. Relaxed Separation Logic

To motivate RSL, consider the message passing program shown in Figure 5. The thread on the left updates some data structure using non-atomic memory accesses (here, the location a), and then signals to other threads that the data structure has been updated by performing a release write to c. The thread on the right repeatedly performs acquire reads until it notices that $[c] \neq 0$. Then, it can conclude that the thread on the left has finished its work, and so may safely access the data structure without interfering with it.

This message passing idiom is correct (i.e., race-free) because whenever an acquire read sees the value written by a release write, the write "synchronizes with" the acquire read. Thus, as hb is transitive, any event that happened before the write (e.g., by being sequenced before it), also happens before the read. This, in turn, justifies the ownership transfer from the writing thread to the reading thread.

To model such ownership transfers, RSL extends the grammar of separation logic assertions, P, with three new assertion forms, $\mathsf{Rel}(e, \mathcal{Q})$, $\mathsf{Acq}(e, \mathcal{Q})$, and $\mathsf{RMWAcq}(e, \mathcal{Q})$, where $\mathcal{Q}$ ranges over functions from values to assertions. Formally, RSL assertions are given by following grammar:

$$
\begin{array}{rl}
P, Q ::= & \mathsf{false} \mid P \Rightarrow Q \mid \forall x.\ P \mid \mathsf{emp} \mid e \overset{k}{\mapsto} e' \mid P * Q \\
& \mid \mathsf{Rel}(e, \mathcal{Q}) \mid \mathsf{Acq}(e, \mathcal{Q}) \mid \mathsf{RMWAcq}(e, \mathcal{Q}) \\
& \mid \mathsf{Init}(e) \mid \mathsf{Uninit}(e)
\end{array}
$$

where k ranges over fractional permissions ($\mathsf{Perm} = (0, 1]$, see Boyland 2003). We have the usual classical first order logic constructs (the three primitive ones and the derived: true, $\wedge$, $\vee$, $\neg$, $\exists$), the three assertions forms pertinent to separation logic (empty heap, a single memory cell with

$$\frac{}{\{P\}\, e\, \{y.\ P \wedge y = e\}} \qquad \frac{\{P\}\, E\, \{y.\ Q\}}{\{P * R\}\, E\, \{y.\ Q * R\}}$$

$$\frac{\{P\}\, E_1\, \{x.\ Q\} \quad \forall x.\, \{Q\}\, E_2\, \{y.\ R\}}{\{P\}\, \mathbf{let}\ x = E_1\ \mathbf{in}\ E_2\, \{y.\ R\}} \qquad \frac{\{P\}\, E\, \{y.\ Q\} \quad P' \Rightarrow P \quad \forall y.\, Q \Rightarrow Q'}{\{P'\}\, E\, \{y.\ Q'\}}$$

$$\frac{\{P \wedge b\}\, E_1\, \{y.\ Q\} \quad \{P \wedge \neg b\}\, E_2\, \{y.\ Q\}}{\{P\}\, \mathbf{if}\ b\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2\, \{y.\ Q\}} \qquad \frac{\{P\}\, E\, \{y.\ Q\} \quad \{P'\}\, E\, \{y.\ Q'\}}{\{P \vee P'\}\, E\, \{y.\ Q \vee Q'\}}$$

$$\frac{\{P\}\, E\, \{y.\ Q\} \quad Q[0/y] \Rightarrow P}{\{P\}\, \mathbf{repeat}\ E\ \mathbf{end}\, \{y.\ Q \wedge y \neq 0\}} \qquad \frac{\{P\}\, E\, \{y.\ Q\}}{\{\exists x.\ P\}\, E\, \{y.\ \exists x.\ Q\}}$$

$$\frac{\{P_1\}\, E_1\, \{y.\ Q_1\} \quad \{P_2\}\, E_2\, \{Q_2\}}{\{P_1 * P_2\}\, E_1 \| E_2\, \{y.\ Q_1 * Q_2\}}$$

Figure 6. Standard proof rules supported by RSL.

fractional permission k, and separating conjunction), and five new forms, which we will explain shortly.

RSL judgements are of the form $\{P\}\, E\, \{y.\ Q\}$, where P and Q are assertions respectively denoting the precondition and the postcondition of the expression E. The postcondition, Q, also describes the return value of the expression E, which is bound by the variable y. In cases where the postcondition does not describe the return value, we often omit the y binder. With this setup, we support all the standard rules from Hoare and separation logic (see Figure 6) including the so-called 'structural' rules: the frame, consequence, disjunction, and existential rules.

Another generic rule we support is the RELAX rule below. Generally, when reasoning about a program E, we are always allowed to reason about a relaxation of the program $E' \sqsubseteq E$, which is identical to E except on the atomic access annotations, which may be weaker than those of E according to the partial order: $\mathrm{rlx} \sqsubseteq \mathrm{rel} \sqsubseteq \mathrm{sc}, \mathrm{rlx} \sqsubseteq \mathrm{acq} \sqsubseteq \mathrm{sc}$.

$$\frac{\{P\}\, E'\, \{y.\ Q\} \qquad E' \sqsubseteq E}{\{P\}\, E\, \{y.\ Q\}} \tag{RELAX}$$

Atomic Writes We return to the treatment of atomic memory accesses and the new assertion forms. The first one, $\mathsf{Rel}(\ell, \mathcal{Q})$, represents a permission to write any value v to location ℓ, provided the assertion $\mathcal{Q}(v)$ holds separately. Performing the write consumes the $\mathcal{Q}(v)$ assertion so that it can be transferred to the reader(s).

$$\left\{\begin{array}{l}\mathcal{Q}(v) * \\ \mathsf{Rel}(\ell, \mathcal{Q})\end{array}\right\} [\ell]_{\mathrm{rel}} := v \left\{\begin{array}{l}\mathsf{Init}(\ell) * \\ \mathsf{Rel}(\ell, \mathcal{Q})\end{array}\right\} \tag{W-REL}$$

In order for the ownership transfer to be valid, the writer must synchronize with the reader(s), which means that the write must be at least of release kind (or stronger, namely SC). Besides the ownership transfer, the write also initializes the location ℓ. Keeping track of initialized locations is necessary for subsequent proof rules. In the special case when no ownership transfer occurs (i.e., when $P = \mathsf{emp}$), intuitively we can also use a relaxed write as in the following rule.

$$\left\{\mathsf{Rel}(\ell, v, \mathsf{emp})\right\} [\ell]_{\mathrm{rlx}} := v \left\{\mathsf{Init}(\ell)\right\} \tag{W-RLX*}$$

In this rule, we used the following shorthand notation

$$\mathsf{Rel}(\ell, v, P) \stackrel{\mathrm{def}}{=} \mathsf{Rel}(\ell, \lambda x.\ \mathbf{if}\ x = v\ \mathbf{then}\ P\ \mathbf{else}\ \mathsf{false})$$

for representing the permission to write only the value v and release ownership of P (in this case emp). Intuitive though this rule is, it is unfortunately unsound in C11, as we will explain in Section 6, where we also show that we can restore its soundness by mildly strengthening the model.

In RSL, we allow multiple concurrent writes to the same atomic location by making the permission to perform an atomic write splittable as follows:

$$\begin{array}{l}\mathsf{Rel}(\ell, \mathcal{Q}_1) * \mathsf{Rel}(\ell, \mathcal{Q}_2) \\ \iff \mathsf{Rel}(\ell, \lambda v.\ \mathcal{Q}_1(v) \vee \mathcal{Q}_2(v))\end{array} \tag{REL-SPLIT}$$

Of course, programs that perform multiple concurrent writes to same location and transfer away ownership may leak memory, as some of the writes may be overwritten and thus never read. In this paper, however, we do not regard such memory leaks as an error. If desired, the programmer may explicitly count the number of allocations and deallocations in order to prove that the program has no memory leaks.

Similar to write permissions, the fact that a location has been initialized—captured by $\mathsf{Init}(\ell)$—can be freely duplicated. Once a location is initialized, it remains initialized: it cannot be de-initialized.

$$\mathsf{Init}(\ell) \iff \mathsf{Init}(\ell) * \mathsf{Init}(\ell) \tag{INIT-SPLIT}$$

Atomic Reads The second assertion form, $\mathsf{Acq}(\ell, \mathcal{Q})$, denotes a permission to perform an acquire read of location ℓ and obtain ownership of $\mathcal{Q}(v)$, where v is the value read.

$$\frac{\forall x.\ \mathsf{precise}(\mathcal{Q}(x))}{\left\{\begin{array}{l}\mathsf{Init}(\ell) * \\ \mathsf{Acq}(\ell, \mathcal{Q})\end{array}\right\} [\ell]_{\mathrm{acq}} \left\{\begin{array}{l}v.\ \mathcal{Q}(v) * \\ \quad \mathsf{Acq}(\ell, \mathcal{Q}[v{:=}\mathsf{emp}])\end{array}\right\}} \tag{R-ACQ}$$

The premise of the rule (that $\mathcal{Q}$ should be precise) is a technical requirement that will be explained in Section 7 and may be ignored for the time being. As a precondition, we require not only the permission to perform an acquire read from ℓ, but also the knowledge that the location has been initialized. The latter is needed because reading from uninitialized locations may return any arbitrary value and thus we cannot ensure that $\mathcal{Q}(v)$ was ever established. When reading a value, we acquire $\mathcal{Q}(v)$ and give up the permission to read the same value again with ownership transfer, because otherwise it would have been possible to acquire the same $\mathcal{Q}(v)$ multiple times. Therefore, in the postcondition the assertion attached to the acquire predicate becomes

$$\mathcal{Q}[v{:=}\mathsf{emp}] \stackrel{\mathrm{def}}{=} \lambda y.\ \mathbf{if}\ y{=}v\ \mathbf{then}\ \mathsf{emp}\ \mathbf{else}\ \mathcal{Q}(y)\,.$$

This allows further reads of the same value, but consequent reads will simply not gain any ownership. At any point, it is also possible to do a relaxed read and acquire no ownership.

$$\left\{\mathsf{Init}(\ell) * \mathsf{Acq}(\ell, \mathcal{Q})\right\} [\ell]_{\mathrm{rlx}} \left\{\mathsf{Acq}(\ell, \mathcal{Q})\right\} \tag{R-RLX}$$

Note that this rule does not assert anything about the value read. A more useful rule is the following, which asserts that the value read must be one that may have been written.

$$\left\{\begin{array}{l}\mathsf{Init}(\ell) * \\ \mathsf{Acq}(\ell, \mathcal{Q})\end{array}\right\} [\ell]_{\mathrm{rlx}} \left\{\begin{array}{l}v.\ \mathsf{Acq}(\ell, \mathcal{Q}) \wedge \\ \quad (\mathcal{Q}(v) \neq \mathsf{false})\end{array}\right\} \qquad \text{(R-RLX*)}$$

Similar to W-RLX*, this latter rule is not sound in C11, but is so in the strengthened model of Section 6.

In RSL, we permit multiple readers to read the value written by a single release write. Concretely, consider the scenario where thread A initializes two data structures and signals by a release write that it has finished its work. Then thread B can do an acquire read and notice that A has finished its initialization and then access the first data structure non-atomically. Likewise, thread C can do an acquire read and access the second data structure non-atomically. Such an execution does not have data races and should therefore be permitted. In terms of our program logic, this means that acquire read permissions should be splittable and joinable as follows:

$$\begin{array}{l}\mathsf{Acq}(\ell, \mathcal{Q}_1) * \mathsf{Acq}(\ell, \mathcal{Q}_2) \\ \iff \mathsf{Acq}(\ell, \lambda v.\ \mathcal{Q}_1(v) * \mathcal{Q}_2(v))\end{array} \qquad \text{(ACQ-SPLIT)}$$

Read-Modify-Write Instructions The next new assertion form, $\mathsf{RMWAcq}(\ell, \mathcal{Q})$, is used in the following proof rule for atomic compare-and-swaps.

$$\frac{\begin{array}{r}P \Rightarrow \mathsf{Init}(\ell) * \mathsf{RMWAcq}(\ell, \mathcal{Q}) * \mathsf{true} \\ P * \mathcal{Q}(v) \Rightarrow \mathsf{Rel}(\ell, \mathcal{Q}') * \mathcal{Q}'(v') * R[v/y] \\ X \in \{\mathrm{rel}, \mathrm{rlx}\} \Rightarrow \mathcal{Q}(v) = \mathsf{emp} \\ X \in \{\mathrm{acq}, \mathrm{rlx}\} \Rightarrow \mathcal{Q}'(v') = \mathsf{emp} \\ \left\{P\right\} [\ell]_Y \left\{y.\ y \neq v \Rightarrow R\right\}\end{array}}{\left\{P\right\} \mathbf{CAS}_{X,Y}(\ell, v, v') \left\{y.\ R\right\}} \qquad \text{(CAS*)}$$

The rule has five premises. First, the precondition must ensure that we have permission to do a RMW-read from ℓ and acquire ownership of $\mathcal{Q}(v)$. Second, we require the update performed by the successful **CAS** to be valid: that is, to have the necessary release permission, to satisfy $\mathcal{Q}'(v')$, the assertion that is to be transferred away, and to separately also satisfy the postcondition. As a precondition for this update, we get to assume not only that the initial precondition holds, but also that we have access to the state acquired by ownership transfer, $\mathcal{Q}(v)$.

The next two premises take the access modes into account, suitably restricting the ownership that can be acquired or released. If the successful **CAS** is of release or relaxed kind, then it does not synchronize with the write whose value it read, so it should not acquire any ownership. This is ensured by demanding that $\mathcal{Q}(v) = \mathsf{emp}$. Symmetrically, if the successful **CAS** is of acquire or relaxed kind, it does not synchronize with the reads seeing the value it produced, so it should not release any ownership. This is ensured by demanding that $\mathcal{Q}'(v') = \mathsf{emp}$.

Finally, we require that failed **CAS**s also satisfy the postcondition, R, under the assumption that a value different from the expected one was read.

In its general form, the CAS* rule is sound in the strengthened model of Section 6. In the standard model, it is sound only when $X \in \{\mathrm{rel\_acq}, \mathrm{sc}\}$.

Unlike multiple normal reads, multiple successful **CAS** instructions cannot all read from (and therefore potentially synchronize with) the same write. This follows from the AtomicRMW axiom, which requires RMW actions to read from the immediately preceding write in mo-order. Therefore, it is sound to duplicate the RMW-acquire permission,

$$\begin{array}{l}\mathsf{RMWAcq}(\ell, \mathcal{Q}) \\ \iff \mathsf{RMWAcq}(\ell, \mathcal{Q}) * \mathsf{RMWAcq}(\ell, \mathcal{Q})\end{array} \qquad \text{(RMW-SPLIT)}$$

because the semantics ensures that at most one process will effectively be able to use this permission at any given instant.

In order to be able to prove the last premise of the CAS* rule, we also support the following rule, allowing us to carve out a plain acquire permission from an RMW-acquire one.

$$\frac{\forall v.\ \mathcal{Q}'(v) = \mathsf{emp} \vee \mathcal{Q}(v) = \mathcal{Q}'(v) = \mathsf{false}}{\mathsf{RMWAcq}(\ell, \mathcal{Q}) \iff \mathsf{RMWAcq}(\ell, \mathcal{Q}) * \mathsf{Acq}(\ell, \mathcal{Q}')} \qquad \text{(RMW-ACQ-SPLIT)}$$

The premise of RMW-ACQ-SPLIT ensures that the assertion that we have carved out for plain reads is empty, except perhaps for the values where $\mathcal{Q}(v)$ is false, in which case $\mathcal{Q}'(v)$ may also be false.

Allocation of Atomic Locations Whenever a new atomic location is allocated, the verifier is free to choose a suitable ownership assertion $\mathcal{Q}$ and attach it to the newly allocated location, and moreover to choose whether the ownership of $\mathcal{Q}$ will be acquired using plain reads or using successful **CAS**s. We thus have the following two rules.

$$\left\{\mathsf{emp}\right\} \mathbf{alloc}() \left\{\ell.\ \mathsf{Rel}(\ell, \mathcal{Q}) * \mathsf{Acq}(\ell, \mathcal{Q})\right\} \qquad \text{(A-R)}$$

$$\left\{\mathsf{emp}\right\} \mathbf{alloc}() \left\{\ell.\ \mathsf{Rel}(\ell, \mathcal{Q}) * \mathsf{RMWAcq}(\ell, \mathcal{Q})\right\} \qquad \text{(A-M)}$$

Following the C standard, newly allocated locations are not initialized, and thus do not generate the $\mathsf{Init}(\ell)$ permission required for reading them. To enable reading from these locations, the programmer must first initialize them by performing a plain write as we have already seen.

Non-Atomic Locations Finally, the rules for non-atomic accesses are exactly as in concurrent separation logic. Allocation returns an uninitialized new cell with full permission; writing requires full permission of a location (whether initialized or not), whereas reading works also with partial permission but requires the location to be initialized.

$$\left\{\mathsf{emp}\right\} \mathbf{alloc}() \left\{x.\ \mathsf{Uninit}(x)\right\} \qquad \text{(A-NA)}$$

$$\left\{\ell \overset{1}{\mapsto} \_ \vee \mathsf{Uninit}(\ell)\right\} [\ell]_{\mathrm{na}} := v \left\{\ell \overset{1}{\mapsto} v\right\} \qquad \text{(W-NA)}$$

$$\left\{\ell \overset{k}{\mapsto} v\right\} [\ell]_{\mathrm{na}} \left\{x.\ x = v \wedge \ell \overset{k}{\mapsto} v\right\} \qquad \text{(R-NA)}$$

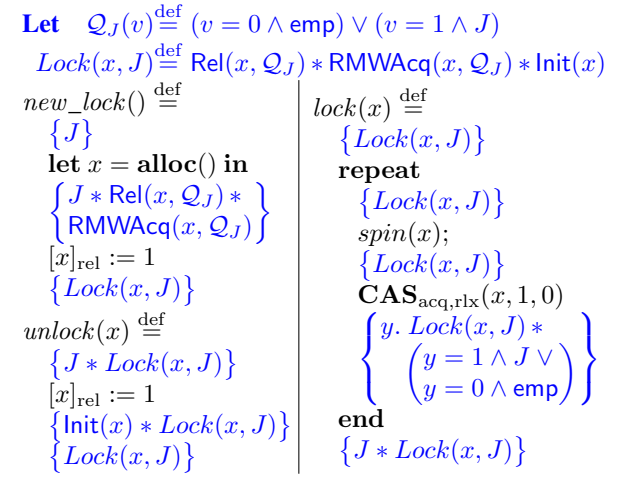


Figure 7. Verification of the lock module.

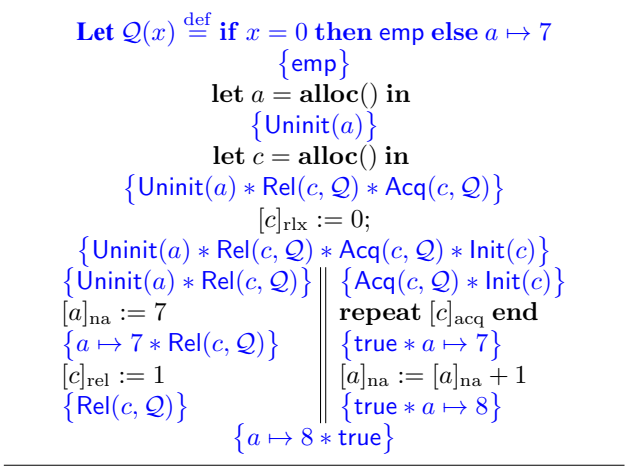


Figure 8. Verification of the message passing example.

These rules ensure that all accessed location have been allocated and there are no races on non-atomic memory locations, and moreover that only initialized locations are read.

5. Examples

We now illustrate RSL by proving simple race-free programs involving release-acquire synchronization patterns. Ownership transfer along those release-acquire synchronizations is necessary to prove them correct, that is, to show that they are memory safe and do not contain data races. To make the proof outlines more concise, we define the following shorthand notations.

$$\mathsf{Emp} \stackrel{\text{def}}{=} \lambda v.\ \mathsf{emp}$$
$$\mathsf{IAcq}(\ell, v, P) \stackrel{\text{def}}{=} \mathsf{Init}(\ell) * \mathsf{Acq}(\ell, \mathsf{Emp}[v := P])$$
$$\mathsf{IRMWAcq}(\ell, v, P) \stackrel{\text{def}}{=} \mathsf{Init}(\ell) * \mathsf{RMWAcq}(\ell, \mathsf{Emp}[v := P])$$

Figure 7: Lock Module As our first example, we consider the lock module introduced in Figure 1. Here we show that any invariant J may be attached to a lock so that we get the same specifications as in concurrent separation logic:

$$\{J\}\ new\_lock()\ \{x.\ Lock(x, J)\}$$
$$\{Lock(x, J)\}\ lock(x)\ \{J * Lock(x, J)\}$$
$$\{J * Lock(x, J)\}\ unlock(x)\ \{Lock(x, J)\}$$
$$Lock(x, J) \iff Lock(x, J) * Lock(x, J)$$

As expected, creating a lock requires the invariant J to hold initially and returns a token confirming that the lock exists and protects the invariant J. Acquiring the lock requires this token and obtains ownership of the invariant. Conversely, releasing the lock requires the invariant to hold and transfers it away. Finally, the token saying that x is a lock protecting resource J can be freely duplicated.

To derive this specification, we define the predicates:

$$\mathcal{Q}_J(v) \stackrel{\text{def}}{=} (v = 0 \land \mathsf{emp}) \lor (v = 1 \land J)$$
$$Lock(x, J) \stackrel{\text{def}}{=} \mathsf{Rel}(x, \mathcal{Q}_J) * \mathsf{RMWAcq}(x, \mathcal{Q}_J) * \mathsf{Init}(x)$$

The $\mathcal{Q}_J(v)$ predicate describes the invariant associated with the location x implementing the lock. It assigns empty ownership when the lock is held ($v = 0 \land \mathsf{emp}$), and ownership of the invariant, J, when the lock is free ($v = 1 \land J$). The $Lock(x, J)$ predicate contains permissions to access the lock by performing release-writes and acquire-RMWs. It also contains the knowledge that the lock is initialized.

In $new\_lock()$, we use the A-M and W-REL rules to initialize the location and transfer away the ownership of J. Similarly, in $unlock(x)$, we use the W-REL to transfer away the ownership of J and then the INIT-SPLIT rule to remove the duplicate $\mathsf{Init}(x)$ fact. In $lock(x)$, we use the R-RLX rule for the relaxed optimistic read in the $spin(x)$ loop, and then the CAS* and the R-RLX* rules to deal with the **CAS**. Finally, the fact that the $Lock(x, J)$ predicate can be freely duplicated follows immediately from REL-SPLIT, RMW-SPLIT, and INIT-SPLIT.

Figure 8: Message Passing As our second example, we consider the message passing idiom of Figure 5. Here, by constructing a proof, we conclude that the program has no data races and moreover, when both threads terminate, we have $[a] = 8$. The proof illustrates the use of the $\mathsf{Acq}(-, -)$ predicate, and the rules A-NA, A-R, W-RLX*, W-NA, W-REL, R-ACQ, and R-NA.

Figure 9: Partial Ownership Transfer Our next example is a variant of the message passing program we have just seen, where after the synchronization between the two threads, both threads read from a. This is valid because two concurrent read accesses do not count as a data race.

In order to verify this program, we use fractional permissions and transfer the partial ownership of the non-atomic location a from the first to the second thread. The first thread writes to a, and then performs a release write to x, giving

$$\left\{a \overset{1}{\mapsto} \_ * \mathsf{Rel}(lock, 0, a \overset{1}{\mapsto} 2 \vee a \overset{1}{\mapsto} 3) * \mathsf{Rel}(lock, 1, \mathsf{emp}) * \mathsf{IRMWAcq}(lock, 0, a \overset{1}{\mapsto} 2 \vee a \overset{1}{\mapsto} 3)\right\}$$

$$\left\{a \overset{1}{\mapsto} \_ * \mathsf{Rel}(lock, 0, a \overset{1}{\mapsto} 2 \vee a \overset{1}{\mapsto} 3)\right\}$$
$[a]_{\mathrm{na}} := 2;$
$$\left\{a \overset{1}{\mapsto} 2 * \mathsf{Rel}(lock, 0, a \overset{1}{\mapsto} 2 \vee a \overset{1}{\mapsto} 3)\right\}$$
$[lock]_{\mathrm{rel}} := 0;$
$\{\mathsf{emp}\}$

$$\left\{\begin{array}{l}\mathsf{Rel}(lock, 1, \mathsf{emp}) * \\ \mathsf{IRMWAcq}(lock, 0, a \overset{1}{\mapsto} 2 \vee a \overset{1}{\mapsto} 3)\end{array}\right\}$$
if $(\mathbf{CAS}_{\mathrm{acq,rlx}}(lock, 0, 1) = 0)$ **then**
$$\left\{\begin{array}{l}(a \overset{1}{\mapsto} 2 \vee a \overset{1}{\mapsto} 3) * \mathsf{IRMWAcq}(\ldots) \\ * \mathsf{Rel}(lock, 1, a \overset{1}{\mapsto} 2 \vee a \overset{1}{\mapsto} 3)\end{array}\right\}$$
$[a]_{\mathrm{na}} := 3$
$$\left\{\begin{array}{l}a \overset{1}{\mapsto} 3 * \mathsf{IRMWAcq}(lock, 0, \ldots) \\ * \mathsf{Rel}(lock, 1, a \overset{1}{\mapsto} 2 \vee a \overset{1}{\mapsto} 3)\end{array}\right\}$$
$[lock]_{\mathrm{rel}} := 0;$
$\{\mathsf{IRMWAcq}(lock, 0, \ldots)\}$
endif

$$\left\{\begin{array}{l}\mathsf{Rel}(lock, 1, \mathsf{emp}) * \\ \mathsf{IRMWAcq}(lock, 0, a \overset{1}{\mapsto} 2 \vee a \overset{1}{\mapsto} 3)\end{array}\right\}$$
if $(\mathbf{CAS}_{\mathrm{acq,rlx}}(lock, 0, 1) = 0)$ **then**
$$\left\{\begin{array}{l}(a \overset{1}{\mapsto} 2 \vee a \overset{1}{\mapsto} 3) * \mathsf{IRMWAcq}(\ldots) \\ * \mathsf{Rel}(lock, 1, a \overset{1}{\mapsto} 2 \vee a \overset{1}{\mapsto} 3)\end{array}\right\}$$
$[a]_{\mathrm{na}} := 2$
$$\left\{\begin{array}{l}a \overset{1}{\mapsto} 2 * \mathsf{IRMWAcq}(lock, 0, \ldots) \\ * \mathsf{Rel}(lock, 1, a \overset{1}{\mapsto} 2 \vee a \overset{1}{\mapsto} 3)\end{array}\right\}$$
$[lock]_{\mathrm{rel}} := 0;$
$\{\mathsf{IRMWAcq}(lock, 0, \ldots)\}$
endif

Figure 10. Example illustrating the use of **CAS** to implement a lock.

$$\left\{a \overset{1}{\mapsto} \_ * \mathsf{Rel}(x, 1, a \overset{0.6}{\mapsto} 2) * \mathsf{IAcq}(x, 1, a \overset{0.6}{\mapsto} 2)\right\}$$

$$\left\{a \overset{1}{\mapsto} \_ * \mathsf{Rel}(x, 1, a \overset{0.6}{\mapsto} 2)\right\}$$
$[a]_{\mathrm{na}} := 2;$
$$\left\{a \overset{1}{\mapsto} 2 * \mathsf{Rel}(x, 1, a \overset{0.6}{\mapsto} 2)\right\}$$
$[x]_{\mathrm{rel}} := 1;$
$$\left\{a \overset{0.4}{\mapsto} 2\right\}$$
$[a]_{\mathrm{na}}$
$$\left\{r.\ r = 2 \wedge a \overset{0.4}{\mapsto} 2\right\}$$

$$\left\{\mathsf{IAcq}(x, 1, a \overset{0.6}{\mapsto} 2)\right\}$$
let $y = [x]_{\mathrm{acq}}$ **in**
$$\left\{(y{=}1)\ ?\ a \overset{0.6}{\mapsto} 2 : \mathsf{emp}\right\}$$
assume$(y = 1)$;
$$\left\{a \overset{0.6}{\mapsto} 2\right\}$$
$[a]_{\mathrm{na}}$
$$\left\{r.\ r = 2 \wedge a \overset{0.6}{\mapsto} 2\right\}$$

Figure 9. Example illustrating fractional ownership and transfer thereof.

$$\begin{array}{c}\textbf{let } a = \textbf{alloc}() \textbf{ in } [a]_{\mathrm{rlx}} := 0; \\ \textbf{let } b = \textbf{alloc}() \textbf{ in } [b]_{\mathrm{rlx}} := 0; \\ \left(\begin{array}{c}\textbf{if } 1 = [a]_{\mathrm{rlx}} \textbf{ then} \\ [b]_{\mathrm{rlx}} := 1\end{array}\right) \parallel \left(\begin{array}{c}\textbf{if } 1 = [b]_{\mathrm{rlx}} \textbf{ then} \\ [a]_{\mathrm{rlx}} := 1\end{array}\right)\end{array}$$

Figure 11. Program with a possible dependency cycle.

[*Initialization actions not shown*]

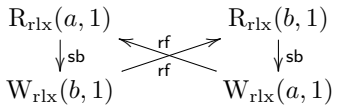


Figure 12. Execution exhibiting the dependency cycle.

away the partial permission $a \overset{0.6}{\mapsto} 2$ (using the W-REL rule). With its remaining $a \overset{0.4}{\mapsto} 2$ permission, it then reads a using the R-NA rule. The second thread synchronizes with the write to x and gets the $a \overset{0.6}{\mapsto} 2$ permission (using the R-ACQ rule), after which it reads a and also gets the value 2 (using the R-NA rule).

Figure 10: Transfer of Permission in Both Directions Our next example demonstrates the use of **CAS** directly to implement a simple mutual exclusion lock. (We could of course use the lock module verified previously, but we include this example in order to demonstrate the CAS* rule again.) Here, for a change, we implement a non-blocking "`tryLock`" command using a conditional, rather than a blocking locking command using a loop.

The lock is implemented by a single location, $lock$, storing 0 if the lock is free and 1 if it is held. (This is opposite to the covention of the earlier lock module.) The lock protects a resource invariant describing the memory cell a. Initially, the first thread starts with the lock acquired and owning a: it establishes the resource invariant and releases the lock. The other two threads start without knowing that the lock was initially held; they both have the permission to write the value 1 to the lock without releasing any ownership, $\mathsf{Rel}(lock, 1, \mathsf{emp})$. By itself, this permission is pretty useless: the threads can do a blind relaxed-atomic write to $lock$ setting its value to 1 (acquired) but without gaining any information. What makes this permission useful, is its combination with the other permission they have, namely to read the state of an unacquired lock with a **CAS** and get ownership of the resource invariant. Successfully performing the **CAS** enables them to later release the lock with the same resource invariant.

6. Dealing with Relaxed Memory Accesses

A serious deficiency of the C11 memory model is that it allows "out of thin air" reads, as illustrated by the program in Figure 11, adapted from Batty et al. (2013).

In this program, two locations are initialized with the value 0, and then two threads are forked, each writing 1 to the one location provided the other already has the value 1. Intuitively, one would expect that the writes would never be executed, but actually the C11 concurrency model permits this outcome. The questionable execution, depicted in Figure 12, is consistent as the two reads can get the value 1 by reading from the corresponding conditional stores.

This counterintuitive behaviour is extremely problematic for formal reasoning as it inhibits even the simplest form

$$\begin{array}{c}\mathbf{let}\ a = \mathbf{alloc}()\ \mathbf{in}\ [a]_{\mathrm{rlx}} := 0;\\ \mathbf{let}\ b = \mathbf{alloc}()\ \mathbf{in}\ [b]_{\mathrm{rlx}} := 1;\\ \left([b]_{\mathrm{rlx}} := [a]_{\mathrm{rlx}} \,\middle\|\, [a]_{\mathrm{rlx}} := [b]_{\mathrm{rlx}}\right);\\ \mathbf{if}\ [a]_{\mathrm{rlx}} < 20\ \mathbf{then}\ \mathbf{print}\ [a]_{\mathrm{rlx}}\end{array}$$

Figure 13. Program showing that range analysis is unsound under C11.

of thread-local reasoning, that of non-relational conjunctive invariants (i.e., invariants where each conjunct describes a property of only one variable). Intuitively, in the previous program, one would expect the invariant

$$[a]_{\mathrm{rlx}} = 0 \wedge [b]_{\mathrm{rlx}} = 0$$

to hold throughout the parallel composition since it holds initially and is preserved by every 'reachable' atomic statement of the program, arguing that the conditional stores are not reachable because the conditions are unsatisfiable according to the invariant. This kind of reasoning is performed by standard compiler optimizations such as "sparse conditional constant propagation" (Wegman and Zadeck 1991).

Note that with the W-RLX* and R-RLX* rules, we can easily prove that if we were to read $[a]$ at the end of the program, we would get 0. (To do so, pick $\mathcal{Q} := (\lambda x.\ x = 0)$ in the allocation rule for both locations.) This shows that these two rules are unsound under C11.

Observe that the same problematic execution remains consistent even if we strengthen either the relaxed reads to acquire/SC reads or (exclusively) the relaxed writes to release/SC writes. To make this execution inconsistent, we have to strengthen both the reads and the writes except at most one access. This means that even adding one of the W-RLX* and R-RLX* rules is unsound.

Global Range Analysis A concrete optimization that is unsound under C11 is global range analysis. Consider the program in Figure 13. An optimizing compiler may argue that the test $[a]_{\mathrm{rlx}} < 20$ will always succeed because $[a]$ and $[b]$ store either 0 or 1, and therefore replace the conditional expression by the **then** branch. Somewhat surprisingly, under C11, this transformation introduces new behaviour and is therefore unsound. Because of the causal dependency cycle, the $[a]_{\mathrm{rlx}}$ read can return any arbitrary value. Therefore, the transformed program can print any arbitrary value, while the original one could only print values less than 20.

A Crude Fix to the Model Since even this very basic reasoning is unsound for relaxed accesses, we decided to strengthen the C11 concurrency model with the following axiom stating that $\mathsf{hb} \cup \mathsf{rf}$ must be acyclic (i.e., its transitive closure must be irreflexive).

$$\mathsf{acyclic}(\mathsf{hb} \cup \{(\mathsf{rf}(a), a) \mid a \in \mathcal{A}\}) \quad \text{(StrongAcyclicHB)}$$

where $\mathsf{acyclic}(R) \stackrel{\text{def}}{=} \nexists x \in \mathcal{A}.\ R^+(x, x)$.

$$\begin{array}{c}\mathbf{let}\ a = \mathbf{alloc}()\ \mathbf{in}\ [a]_{\mathrm{rlx}} := 0;\\ \mathbf{let}\ b = \mathbf{alloc}()\ \mathbf{in}\ [b]_{\mathrm{rlx}} := 0;\\ \begin{pmatrix}\mathbf{let}\ x = [a]_{\mathrm{rlx}}\ \mathbf{in}\\ [b]_{\mathrm{rlx}} := 1\end{pmatrix} \,\middle\|\, \begin{pmatrix}\mathbf{let}\ y = [b]_{\mathrm{rlx}}\ \mathbf{in}\\ [a]_{\mathrm{rlx}} := 1\end{pmatrix}\end{array}$$

Figure 14. Program without a dependency cycle.

With this additional axiom, we can also show the soundness of the "starred" rules for relaxed memory accesses presented in the previous section. In contrast, the soundness of the other rules does not depend on this axiom.

Notice that when adding this strong acyclicity condition, we can drop the strictly weaker IrreflexiveHB axiom, as well as the $\neg\mathsf{hb}(b, a)$ conjunct from the ConsistentRF axiom. We can further drop the slightly awkward ConsistentRFna axiom, and still have the soundness proof go through, because all the proof really needs to know is that the write precedes the read in some well-founded order. In the absence of causal cycles, this order need not be hb: we can instead take it to be $\mathsf{hb} \cup \mathsf{rf}$.

Simple though our proposed fix might seem, it is not perfect. Alas, the StrongAcyclicHB consistency axiom precludes the reordering of independent instructions, a transformation that compilers and processors with out-of-order execution units frequently perform. To illustrate the problem, consider the program in Figure 14, a slight variant of the program in Figure 11, where the writes to $[b]$ and $[a]$ are now independent of the earlier reads from $[a]$ and $[b]$ respectively. The problem is that when operating at the level of single executions, one cannot distinguish whether the $\mathsf{hb} \cup \mathsf{rf}$ cycle in the execution shown in Figure 12 constitutes a dependency cycle or not. If the execution comes from the program in Figure 11, the cycle should clearly be outlawed, but if it comes from the program of Figure 14, the cycle is harmless and should be allowed. Distinguishing these two cases is not easy and seems to require a radical change to the C11 model. Clearly, this lies beyond the scope of this paper.

7. Semantics and Soundness

In this section, we define the semantics of assertions and Hoare triples, and prove that our logic is sound with respect to the C11 memory model.

7.1 Semantics of Assertions

To define the meaning of separation logic assertions, we need an underlying separation algebra, i.e. a commutative partial monoid. To interpret the Acq and Rel assertions, our model of heaps will have to store assertions, which in turn represent sets of heaps. If we naively write down the domain equation, we will get an equation of the form,

$$\mathsf{Heap}_{\mathrm{spec}} \stackrel{?}{\cong} \mathsf{Loc} \rightharpoonup (\ldots + (\ldots \times \mathbb{P}(\mathsf{Heap}_{\mathrm{spec}}))),$$

which does not have a solution in Set. Therefore, we either have to move to a more advanced category such as bounded

$$\mathcal{Q}_1 \oplus_{\mathrm{acq}}^{b_1,b_2} \mathcal{Q}_2 \stackrel{\mathrm{def}}{=} \begin{cases} \mathcal{Q}_1 & \text{if } b_1 \wedge b_2 \text{ and } \mathcal{Q}_1 = \mathcal{Q}_2 \\ \lambda v.\ \mathcal{Q}_1(v) * \mathcal{Q}_2(v) & \text{if } (\neg b_1 \vee \neg b_2) \text{ and } \mathbf{adef}(b_1, \mathcal{Q}_1, b_2, \mathcal{Q}_2) \\ \text{undef} & \text{if } \neg\mathbf{adef}(b_1, \mathcal{Q}_1, b_2, \mathcal{Q}_2) \end{cases}$$

$$\mathbf{rval}(b, \mathcal{Q}) \stackrel{\mathrm{def}}{=} \mathbf{if}\ b\ \mathbf{then}\ \mathcal{Q}\ \mathbf{else}\ \lambda v.\, \mathsf{emp}$$

$$\mathbf{adef}(b_1, \mathcal{Q}_1, b_2, \mathcal{Q}_2) \stackrel{\mathrm{def}}{=} \forall v.\ \mathbf{rval}(b_2, \mathcal{Q}_1)(v) = \mathbf{rval}(b_1, \mathcal{Q}_2)(v) \vee \mathcal{Q}_1(v) = \mathcal{Q}_2(v) = \mathsf{false}$$

$$h_1 \oplus' h_2 \stackrel{\mathrm{def}}{=} \lambda\ell. \begin{cases} h_1(\ell) & \text{if } \ell \in \mathsf{dom}(h_1) \setminus \mathsf{dom}(h_2) \\ h_2(\ell) & \text{if } \ell \in \mathsf{dom}(h_2) \setminus \mathsf{dom}(h_1) \\ \mathsf{NA}[v, k_1 + k_2] & \text{if } h_i(\ell) = \mathsf{NA}[v, k_i] \text{ for } i = 1, 2 \text{ and } k_1 + k_2 \leq 1 \\ \mathsf{Atom}[\lambda v.\ \mathcal{R}_1(v) \vee \mathcal{R}_2(v), \mathcal{Q}_1 \oplus_{\mathrm{acq}}^{b_1,b_2} \mathcal{Q}_2, b_1 \vee b_2, b_1' \vee b_2'] & \text{if } h_i(\ell) = \mathsf{Atom}[\mathcal{R}_i, \mathcal{Q}_i, b_i, b_i'] \text{ for } i = 1, 2 \\ \text{undef} & \text{otherwise} \end{cases}$$

$$h_1 \oplus h_2 \stackrel{\mathrm{def}}{=} \begin{cases} h_1 \oplus' h_2 & \text{if } \mathsf{dom}(h_1 \oplus' h_2) = \mathsf{dom}(h_1) \cup \mathsf{dom}(h_2) \\ \text{undef} & \text{otherwise} \end{cases}$$

Figure 15. Definition of heap composition, $h_1 \oplus h_2$.

ultrametric spaces (Birkedal et al. 2010), or change the equation to avoid the problematic recursion.

Here, for simplicity, we do the latter and cut the cycle by storing syntactic assertions, Assn, instead of semantic assertions, $\mathbb{P}(\mathsf{Heap}_{\mathrm{spec}})$, within heaps. Simply storing syntactic assertions is, however, insufficient because we want the heap model to form a separation algebra and to support the conversions rules REL-SPLIT and ACQ-SPLIT. To allow these conversions, we therefore have to store syntactic assertions up to associativity and commutativity of $*$ and $\vee$ and their units. Furthermore, to support RMW-ACQ-SPLIT, we also need to equate $\mathsf{false} * \mathsf{false}$ and false. Formally, we define $\sim$ to be the smallest equivalence relation on syntactic assertions, equating the following assertions:

(S1) $\forall P, Q \in \mathsf{Assn}.\ P * Q \sim Q * P$,
(S2) $\forall P, Q, R \in \mathsf{Assn}.\ P * (Q * R) \sim (P * Q) * R$,
(S3) $\forall P \in \mathsf{Assn}.\ P * \mathsf{emp} \sim P$,
(S4) $\mathsf{false} * \mathsf{false} \sim \mathsf{false}$,
(S5) $\forall P, Q \in \mathsf{Assn}.\ P \vee Q \sim Q \vee P$,
(S6) $\forall P, Q, R \in \mathsf{Assn}.\ P \vee (Q \vee R) \sim (P \vee Q) \vee R$, and
(S7) $\forall P \in \mathsf{Assn}.\ (P \vee \mathsf{false}) \sim (P \vee P) \sim P$.

where, for convenience, we have also included idempotence for disjunction. The model of heaps, $\mathsf{Heap}_{\mathrm{spec}}$, therefore is:

$$\mathsf{Perm} \stackrel{\mathrm{def}}{=} (0, 1] \qquad \mathbb{B} \stackrel{\mathrm{def}}{=} \{\mathsf{true}, \mathsf{false}\}$$
$$\mathcal{M} \stackrel{\mathrm{def}}{=} \mathsf{Val} \to \mathsf{Assn}/\!\sim$$
$$\mathsf{Heap}_{\mathrm{spec}} \stackrel{\mathrm{def}}{=} \mathsf{Loc} \rightharpoonup \begin{pmatrix} \mathsf{NA}[\mathbb{U} + (\mathsf{Val} \times \mathsf{Perm})] \\ +\mathsf{Atom}[\mathcal{M} \times \mathcal{M} \times \mathbb{B} \times \mathbb{B}] \end{pmatrix}$$

Each allocated location is either non-atomic or atomic. Non-atomic locations can either be uninitialized (represented by special symbol $\mathbb{U}$) or contain a value and a permission. Atomic locations contain two maps from values to syntactic assertions modulo $\sim$ and two Boolean flags. The two maps represent the release and the acquire maps used to interpret the three assertion forms pertinent to RSL: $\mathsf{Rel}(\ell, \mathcal{Q})$, $\mathsf{Acq}(\ell, Q)$, and $\mathsf{RMWAcq}(\ell, \mathcal{Q})$, with the first Boolean flag indicating whether the second map acts as a plain acquire map or as an RMW-acquire map. The second Boolean flag records whether the location has been initialized or not.

Figure 15 defines the composition of two *logically disjoint* heaps, $h_1 \oplus h_2$. Note that two logically disjoint heaps can share some locations, provided that they store compatible information about them. For non-atomic locations, they should be initialized and have compatible permissions (i.e., whose sum does not exceed the full permission, 1). For atomic locations, the two heaps must contain compatible acquire maps, represented by the predicate $\mathbf{adef}(b_1, \mathcal{Q}_1, b_2, \mathcal{Q}_2)$. This predicate is somewhat complex because acquire maps represent plain acquire or RMW-acquire permissions depending on the relevant Boolean flag. The cases are:

(Case $b_1 \wedge b_2$) we must have $\mathcal{Q}_1 = \mathcal{Q}_2$;
(Case $b_1 \wedge \neg b_2$) we require that for all v, either $\mathcal{Q}_2(v) = \mathsf{emp}$ or $\mathcal{Q}_1(v) = \mathcal{Q}_2(v) = \mathsf{false}$;
(Case $\neg b_1 \wedge b_2$) symmetrically to the previous case; and
(Case $\neg b_1 \wedge \neg b_2$) no conditions.

Given these definitions, we can show that $(\mathsf{Heap}_{\mathrm{spec}}, \oplus, \emptyset)$ forms a separation algebra, which in turn means that it is a good model for separation logic assertions.

Lemma 1. *$(\mathsf{Heap}_{\mathrm{spec}}, \oplus, \emptyset)$ forms a separation algebra. That is, $\oplus$ is associative, commutative, and has $\emptyset$ as its identity element.*

In the proof of this lemma, property S4 is required to show associativity; replacing S4 with the more general property $\forall P \in \mathsf{Assn}.\ P * \mathsf{false} \sim \mathsf{false}$ breaks associativity.

We remark that in contrast to most models for separation logic, our $\oplus$ is not cancellative. For example, consider the heap $h_{\mathrm{I}} = \{\ell \mapsto \mathsf{Atom}[\mathsf{False}, \mathsf{Emp}, \mathsf{false}, \mathsf{true}]\}$. Clearly, $h_{\mathrm{I}} \neq \emptyset$ and yet $h_{\mathrm{I}} \oplus h_{\mathrm{I}} = h_{\mathrm{I}} = h_{\mathrm{I}} \oplus \emptyset$. In practice, the lack of cancellativity does not affect reasoning about RSL assertions. It also does not mean that the heap model contains 'junk' information. Indeed, the heap h_{I} is used to model the assertion $\mathsf{Init}(\ell)$, and we want $h_{\mathrm{I}} \oplus h_{\mathrm{I}} = h_{\mathrm{I}}$ to validate INIT-SPLIT.

Definition 1 (Assertion Semantics)**.**
Let $[\![-]\!] : \mathsf{Assn} \to \mathbb{P}(\mathsf{Heap}_{\mathrm{spec}})$ *be:*

$$
\begin{aligned}
[\![\mathsf{false}]\!] &\stackrel{\mathrm{def}}{=} \emptyset \\
[\![P \Rightarrow Q]\!] &\stackrel{\mathrm{def}}{=} \{h \mid h \in [\![P]\!] \implies h \in [\![Q]\!]\} \\
[\![\forall x.\ P]\!] &\stackrel{\mathrm{def}}{=} \{h \mid \forall v.\ h \in [\![P[v/x]]\!]\} \\
[\![\mathsf{emp}]\!] &\stackrel{\mathrm{def}}{=} \{\emptyset\} \\
[\![P * Q]\!] &\stackrel{\mathrm{def}}{=} \{h_1 \oplus h_2 \mid h_1 \in [\![P]\!] \wedge h_2 \in [\![Q]\!]\} \\
[\![\mathsf{Uninit}(\ell)]\!] &\stackrel{\mathrm{def}}{=} \{\{\ell \mapsto \mathsf{NA}[\mathbb{U}]\}\} \\
[\![\ell \overset{k}{\mapsto} v]\!] &\stackrel{\mathrm{def}}{=} \{\{\ell \mapsto \mathsf{NA}[v,k]\}\} \\
[\![\mathsf{Init}(\ell)]\!] &\stackrel{\mathrm{def}}{=} \{\{\ell \mapsto \mathsf{Atom}[\mathsf{False},\mathsf{Emp},\mathsf{false},\mathsf{true}]\}\} \\
[\![\mathsf{Rel}(\ell,\mathcal{Q})]\!] &\stackrel{\mathrm{def}}{=} \{\{\ell \mapsto \mathsf{Atom}[\mathcal{Q},\mathsf{Emp},\mathsf{false},\mathsf{false}]\}\} \\
[\![\mathsf{Acq}(\ell,\mathcal{Q})]\!] &\stackrel{\mathrm{def}}{=} \{\{\ell \mapsto \mathsf{Atom}[\mathsf{False},\mathcal{Q},\mathsf{false},\mathsf{false}]\}\} \\
[\![\mathsf{RMWAcq}(\ell,\mathcal{Q})]\!] &\stackrel{\mathrm{def}}{=} \{\{\ell \mapsto \mathsf{Atom}[\mathsf{False},\mathcal{Q},\mathsf{true},\mathsf{false}]\}\}
\end{aligned}
$$

where $\mathsf{False} \stackrel{\mathrm{def}}{=} \lambda v.\ \mathsf{false}$ *and* $\mathsf{Emp} \stackrel{\mathrm{def}}{=} \lambda v.\ \mathsf{emp}$.

Figure 16. Definition of the semantics of assertions.

Equipped with specification heaps, $\mathsf{Heap}_{\mathrm{spec}}$, we proceed to the semantics of assertions. These are given as a function $[\![-]\!] : \mathsf{Assn} \to \mathbb{P}(\mathsf{Heap}_{\mathrm{spec}})$ in Figure 16.

A basic property of the assertion semantics, that justifies treating stored assertions up to $\sim$, is that $\sim$-related assertions have the same semantics.

Lemma 2. *If* $P \sim Q$*, then* $[\![P]\!] = [\![Q]\!]$.

Moreover, we can easily show that our model validates the logical entailments of Section 4.

Lemma 3. *The properties* REL-SPLIT, ACQ-SPLIT, RMW-SPLIT, RMW-ACQ-SPLIT, *and* INIT-SPLIT *hold universally.*

Finally, we say that an assertion is *precise* if and only if it uniquely determines a subheap where it holds. The definition is standard (O'Hearn 2007), but due of the lack of cancellativity of $\oplus$ we require both the heaps satisfying the assertion to be equal ($h_1 = h_1'$) as well as their remainders ($h_2 = h_2'$). If $\oplus$ were cancellative, then either of the equalities would suffice as it would imply the other.

Definition 2 (Precision)**.** *An assertion is* precise*, denoted* $\mathsf{precise}(P)$*, if and only if for all* h_1*,* h_1'*,* h_2*,* h_2'*, if* $h_1 \in [\![P]\!]$ *and* $h_2 \in [\![P]\!]$ *and* $h_1 \oplus h_1' = h_2 \oplus h_2' \neq \mathrm{undef}$*, then* $h_1 = h_1'$ *and* $h_2 = h_2'$.

7.2 Semantics of Hoare triples

We move on to the meaning of RSL triples, $\{P\}\ E\ \{y.\ Q\}$. To handle both models—the C11 standard one and the strengthened one of Section 6—we parametrize the definitions of the semantics of triples and all auxiliary definitions with respect to the model. For notational simplicity, however, we will present the definitions only for the strengthened model and we will note in text any differences for the standard C11 model.

Given an execution $\mathcal{X} = \langle \mathcal{A}, \mathsf{lab}, \mathsf{sb}, \mathsf{rf}, \mathsf{mo}, \mathsf{sc} \rangle$, we define the helper functions: $\mathsf{SBin}_{\mathcal{X}}(a)$, $\mathsf{SBout}_{\mathcal{X}}(a)$, $\mathsf{SWin}_{\mathcal{X}}(a)$, and $\mathsf{SWout}_{\mathcal{X}}(a)$, to get the set of sb/sw incoming/outgoing edges of an action $a \in \mathcal{A}$. Given also a set of actions, $V \subseteq \mathcal{A}$, we denote the set of its hb and rf predecessors as $\mathsf{Pre}_{\mathcal{X}}(V)$.

$$\mathsf{Pre}_{\mathcal{X}}(V) \stackrel{\mathrm{def}}{=} \{a \mid \exists b \in V.\ \mathsf{hb}(a,b) \vee a = \mathsf{rf}(b)\}$$

This definition is very useful because we will generally be considering sets of actions V that are prefix-closed, namely $\mathsf{Pre}_{\mathcal{X}}(V) \subseteq V$, and we will be growing such sets by adding one action at a time while maintaining prefix-closure. Doing so is always possible for consistent executions because of the StrongAcyclicHB axiom. In the standard C11 model, we have to resort to a stronger definition of $\mathsf{Pre}_{\mathcal{X}}(V)$ that includes only the hb edges, not arbitrary rf edges as well.

$$\mathsf{Pre}^{\mathsf{standard\_C11}}_{\mathcal{X}}(V) \stackrel{\mathrm{def}}{=} \{a \mid \exists b \in V.\ \mathsf{hb}(a,b)\}$$

To define the meaning of RSL triples, we will generally be annotating hb-edges with appropriate heaps. When doing so, however, it will be important to distinguish between happens-before edges that occur because of an sb-edge and those that occur because of an sw-edge. We therefore introduce the following definition that tags them accordingly.

Definition 3 (Tagged Happens Before)**.** *Given an execution* $\mathcal{X}$*, let* $\mathsf{thb}_{\mathcal{X}}$ *be a tagged union of* $\mathsf{sb}_{\mathcal{X}}$ *and* $\mathsf{sw}_{\mathcal{X}}$*, constructed as follows*

$$
\begin{aligned}
\mathsf{thb}_{\mathcal{X}} \stackrel{\mathrm{def}}{=}\ & \{(\text{“}\mathsf{sb}\text{”}, a, b) \mid (a,b) \in \mathsf{sb}_{\mathcal{X}}\} \\
& \cup \{(\text{“}\mathsf{sw}\text{”}, a, b) \mid (a,b) \in \mathsf{sw}_{\mathcal{X}}\}
\end{aligned}
$$

For a program expression, E, we denote $\mathcal{C}[\![E]\!]$ as the set of its *consistent contextual executions*. These executions are obtained by plugging in an execution of E in some arbitrary execution context, such that the whole execution is consistent, as follows.

$$
\begin{aligned}
\mathcal{C}[\![E]\!] \stackrel{\mathrm{def}}{=} \{\ & \langle res, \mathcal{A}_{\mathrm{ctx}}, \mathcal{A}_{\mathrm{prg}}, \mathcal{X}, fst, lst\rangle \mid \exists \mathsf{lab}_{\mathrm{ctx}}, \mathsf{lab}_{\mathrm{prg}}. \\
& \exists \mathsf{sb}.\ \exists \mathsf{sb}_{\mathrm{prg}} = \mathsf{sb} \cap (\mathcal{A}_{\mathrm{prg}} \times \mathcal{A}_{\mathrm{prg}}). \\
& \mathcal{X} = \langle \mathcal{A}_{\mathrm{ctx}} \uplus \mathcal{A}_{\mathrm{prg}}, \mathsf{lab}_{\mathrm{ctx}} \cup \mathsf{lab}_{\mathrm{prg}}, \mathsf{sb}, \_, \_, \_\rangle \\
& \wedge \langle res, \mathcal{A}_{\mathrm{prg}}, \mathsf{lab}_{\mathrm{prg}}, \mathsf{sb}_{\mathrm{prg}}, fst, lst\rangle \in [\![E]\!] \\
& \wedge (\exists! a.\ \mathsf{sb}(a, fst)) \wedge (\exists! b.\ \mathsf{sb}(lst, b)) \\
& \wedge \mathsf{Consistent}(\mathcal{X})\ \}
\end{aligned}
$$

In the definition of $\mathcal{C}[\![E]\!]$, we require that (1) the part of the execution corresponding to the expression matches its semantics, (2) fst has a unique sb-predecessor, (3) lst has a unique sb-successor, and (4) the entire execution is consistent. The requirements about the unique predecessor of fst and the unique successor of lst will be used for selecting unique edges responsible for carrying the expression's precondition and postcondition.

To define the meaning of RSL triples, we will annotate the thb-edges of consistent contextual executions with heaps. We will call such functions, $hmap : \mathsf{thb}_{\mathcal{X}} \rightharpoonup \mathsf{Heap}_{\mathrm{spec}}$,

Definition 4 (Local annotation validity)**.** *Given an execution, $\mathcal{X} = \langle \mathcal{A}, \mathsf{lab}, \mathsf{sb}, \mathsf{rf}, \mathsf{mo}, \mathsf{sc} \rangle$, a heap map, $hmap : \mathsf{thb}_{\mathcal{X}} \rightharpoonup \mathsf{Heap}_{\mathrm{spec}}$, and a set of actions $V \subseteq \mathcal{A}$, the predicate $\mathsf{Valid}(hmap, V)$ holds if and only if for all actions $a \in V$, there exist $\ell, v, \mathcal{Q}, \mathcal{Q}', \mathcal{Q}'', Z, h_1, h_1', h_2, h_{\mathrm{F}}, h_{\mathrm{sink}}$ such that*

$$
\begin{pmatrix} \mathsf{lab}(a) = \mathrm{skip} \\ \wedge\ hmap(\mathsf{SBin}_{\mathcal{X}}(a)) = hmap(\mathsf{SBout}_{\mathcal{X}}(a)) \oplus h_{\mathrm{sink}} \end{pmatrix}
$$

$$
\vee \begin{pmatrix} \mathsf{lab}(a) = \mathrm{W}_Z(\ell, v) \wedge Z \in \{\mathrm{rlx}, \mathrm{rel}, \mathrm{sc}\} \\ \wedge\ h_1 = \{\ell \mapsto \mathsf{Atom}[\mathcal{Q}, \mathsf{Emp}, \mathsf{false}, \_]\} \\ \wedge\ h_1' = \{\ell \mapsto \mathsf{Atom}[\mathcal{Q}, \mathsf{Emp}, \mathsf{false}, \mathsf{true}]\} \\ \wedge\ h_2 = hmap(\mathsf{SWout}_{\mathcal{X}}(a)) \oplus h_{\mathrm{sink}} \wedge h_2 \in [\![\mathcal{Q}(v)]\!] \\ \wedge\ hmap(\mathsf{SBin}_{\mathcal{X}}(a)) = h_1 \oplus h_2 \oplus h_{\mathrm{F}} \\ \wedge\ hmap(\mathsf{SBout}_{\mathcal{X}}(a)) = h_1' \oplus h_{\mathrm{F}} \\ \wedge\ (Z = \mathrm{rlx} \implies \mathcal{Q}(v) = \mathsf{emp}) \end{pmatrix}
$$

$$
\vee \begin{pmatrix} \mathsf{lab}(a) = \mathrm{RMW}_Z(\ell, v, v') \wedge Z \neq \mathrm{na} \\ \wedge\ hmap(\mathsf{SBin}_{\mathcal{X}}(a))(\ell) = \mathsf{Atom}[\_, \mathcal{Q}, \mathsf{true}, \mathsf{true}] \\ \wedge\ hmap(\mathsf{SBin}_{\mathcal{X}}(a)) \oplus hmap(\mathsf{SWin}_{\mathcal{X}}(a)) \\ \quad = \{\ell \mapsto \mathsf{Atom}[\mathcal{Q}', \mathsf{Emp}, \mathsf{false}, \mathsf{false}]\} \oplus \\ \quad\ hmap(\mathsf{SBout}_{\mathcal{X}}(a)) \oplus hmap(\mathsf{SWout}_{\mathcal{X}}(a)) \oplus h_{\mathrm{sink}} \\ \wedge\ hmap(\mathsf{SWin}_{\mathcal{X}}(a)) \in [\![\mathcal{Q}(v)]\!] \\ \wedge\ (hmap(\mathsf{SWout}_{\mathcal{X}}(a)) \oplus h_{\mathrm{sink}}) \in [\![\mathcal{Q}'(v')]\!] \\ \wedge\ (Z \in \{\mathrm{rlx}, \mathrm{rel}\} \implies \mathcal{Q}(v) = \mathsf{emp}) \\ \wedge\ (Z \in \{\mathrm{rlx}, \mathrm{acq}\} \implies \mathcal{Q}'(v') = \mathsf{emp}) \end{pmatrix}
$$

$$
\vee \begin{pmatrix} \mathsf{lab}(a) = \mathrm{A}(\ell) \\ \wedge\ hmap(\mathsf{SBout}_{\mathcal{X}}(a)) = hmap(\mathsf{SBin}_{\mathcal{X}}(a)) \oplus \\ \qquad \{\ell \mapsto \mathsf{Atom}[\mathcal{Q}, \mathcal{Q}, \_, \mathsf{false}]\} \end{pmatrix}
$$

$$
\vee \begin{pmatrix} \mathsf{lab}(a) = \mathrm{A}(\ell) \\ \wedge\ hmap(\mathsf{SBout}_{\mathcal{X}}(a)) = hmap(\mathsf{SBin}_{\mathcal{X}}(a)) \oplus \{\ell \mapsto \mathsf{NA}[\mathbb{U}]\} \end{pmatrix}
$$

$$
\vee \begin{pmatrix} \mathsf{lab}(a) = \mathrm{W}_{\mathrm{na}}(\ell, v) \\ \wedge\ hmap(\mathsf{SBin}_{\mathcal{X}}(a))(\ell) \in \{\mathsf{NA}[\mathbb{U}], \mathsf{NA}[\_, 1]\} \\ \wedge\ hmap(\mathsf{SBout}_{\mathcal{X}}(a)) = hmap(\mathsf{SBin}_{\mathcal{X}}(a))[\ell \mapsto \mathsf{NA}[v, 1]] \end{pmatrix}
$$

$$
\vee \begin{pmatrix} \mathsf{lab}(a) = \mathrm{R}_{\mathrm{na}}(\ell, v) \\ \wedge\ hmap(\mathsf{SBin}_{\mathcal{X}}(a))(\ell) = \mathsf{NA}[v, \_] \\ \wedge\ hmap(\mathsf{SBin}_{\mathcal{X}}(a)) = hmap(\mathsf{SBout}_{\mathcal{X}}(a)) \end{pmatrix}
$$

$$
\vee \begin{pmatrix} \mathsf{lab}(a) \in \{\mathrm{R}_{\mathrm{rlx}}(\ell, v), \mathrm{R}_{\mathrm{acq}}(\ell, v), \mathrm{R}_{\mathrm{sc}}(\ell, v)\} \\ \wedge\ hmap(\mathsf{SWin}_{\mathcal{X}}(a)) \in [\![\mathcal{Q}(v)]\!] \wedge \mathsf{precise}(\mathcal{Q}(v)) \\ \wedge\ hmap(\mathsf{SBin}_{\mathcal{X}}(a)) = \{\ell \mapsto \mathsf{Atom}[\mathsf{False}, \mathcal{Q}, \mathsf{false}, \mathsf{true}]\} \oplus h_{\mathrm{F}} \\ \wedge\ hmap(\mathsf{SBout}_{\mathcal{X}}(a)) = hmap(\mathsf{SWin}_{\mathcal{X}}(a)) \oplus h_{\mathrm{F}} \oplus \\ \qquad \{\ell \mapsto \mathsf{Atom}[\mathsf{False}, \mathcal{Q}[v := \mathsf{emp}], \mathsf{false}, \mathsf{true}]\} \end{pmatrix}
$$

Definition 5 (Configuration safety)**.** *Given sets of actions, $\mathcal{A}_{\mathrm{ctx}}$ and $\mathcal{A}_{\mathrm{prg}}$, an execution $\mathcal{X} = \langle \mathcal{A}_{\mathrm{prg}} \uplus \mathcal{A}_{\mathrm{ctx}}, \mathsf{lab}, \mathsf{sb}, \mathsf{rf}, \mathsf{sc} \rangle$, a natural number, $n \in \mathbb{N}$, a set of actions, V, a heap map, $hmap : \mathsf{Resp}_{\mathcal{X}}(V) \to \mathsf{Heap}_{\mathrm{spec}}$, a distinguished final action, $lst \in \mathcal{A}_{\mathrm{prg}}$, and a set of heaps, Q, we define $\mathsf{safe}^n_{\mathcal{X}}(V, hmap, \mathcal{A}_{\mathrm{ctx}}, \mathcal{A}_{\mathrm{prg}}, lst, Q)$ by structural recursion on n as follows:*
$\mathsf{safe}^0_{\mathcal{X}}(V, hmap, \mathcal{A}_{\mathrm{ctx}}, \mathcal{A}_{\mathrm{prg}}, lst, Q)$ holds always.
$\mathsf{safe}^{n+1}_{\mathcal{X}}(V, hmap, \mathcal{A}_{\mathrm{ctx}}, \mathcal{A}_{\mathrm{prg}}, lst, Q)$ holds if and only if the following conditions all hold:

- *If $lst \in V$, then $hmap(\mathsf{SBout}_{\mathcal{X}}(lst)) \in Q$; and*
- *For all $a \in \mathcal{A}_{\mathrm{prg}} \setminus V$ such that $\mathsf{Pre}_{\mathcal{X}}(\{a\}) \subseteq V$, there exists $hmap' : \mathsf{Resp}_{\mathcal{X}}(\{a\}) \to \mathsf{Heap}_{\mathrm{spec}}$ such that $\mathsf{Valid}(hmap \cup hmap', V \cup \{a\})$ and $\mathsf{safe}^n_{\mathcal{X}}(V \cup \{a\}, hmap \cup hmap', \mathcal{A}_{\mathrm{ctx}}, \mathcal{A}_{\mathrm{prg}}, lst, Q)$; and*
- *For all $a \in \mathcal{A}_{\mathrm{ctx}} \setminus V$ such that $\mathsf{Pre}_{\mathcal{X}}(\{a\}) \subseteq V$, and all $hmap' : \mathsf{Resp}_{\mathcal{X}}(\{a\}) \to \mathsf{Heap}_{\mathrm{spec}}$, if $\mathsf{Valid}(hmap \cup hmap', V \cup \{a\})$, then $\mathsf{safe}^n_{\mathcal{X}}(V \cup \{a\}, hmap \cup hmap', \mathcal{A}_{\mathrm{ctx}}, \mathcal{A}_{\mathrm{prg}}, lst, Q)$.*

Figure 17. Definitions of annotation validity and configuration safety.

heap annotations or heap maps. For each thb-edge, it is important to decide who is responsible for choosing a heap to annotate that edge with: is it the program itself or is it its environment? Therefore, given a set of program actions $A \subseteq \mathcal{A}$, we define the set, $\mathsf{Resp}_{\mathcal{X}}(A)$ of edges whose annotation is the responsibility of the program.

$$
\mathsf{Resp}_{\mathcal{X}}(A) \stackrel{\mathrm{def}}{=} \bigcup_{a \in A} (\mathsf{SBout}_{\mathcal{X}}(a) \cup \mathsf{SWin}_{\mathcal{X}}(a))
$$

This definition deserves some explanation.

First, as expected, the program is responsible for correctly annotating its outgoing sequenced-before edges. Conversely, it can assume that incoming sb-edges are correctly annotated. This part is consistent with the usual semantics of Hoare triples: the program may assume the precondition holds when starting its execution, and must establish the postcondition when returning.

What is perhaps a bit unusual is that the program is also responsible for the annotations on the *incoming* synchronization edges, and not the outgoing ones. This is because when an acquire read synchronizes with a release write, it is the reader that 'knows' how much state ownership is to be transferred along the sw-edge. The writer simply knows how much total ownership is to be transferred away from itself, but not how this is to be distributed to the various readers that synchronize with the write.

In a slight abuse of notation, given a heap annotation, $hmap$, and a set of context edges, $S \subseteq \mathsf{thb}_{\mathcal{X}}$, we will let $hmap(S) \stackrel{\mathrm{def}}{=} \bigoplus_{x \in S \cap \mathsf{dom}(hmap)} hmap(x)$.

Annotation Validity and Configuration Safety Figure 17 contains two important auxiliary definitions. First, we have annotation validity (Definition 4). A heap map, $hmap$ is *valid* up to a set of actions V, if and only if for every action $a \in V$, the annotation is locally valid around that action: basically the sum of the annotated heaps on the incoming edges should equal the sum of the annotated heaps on the outgoing edges, modulo the effect of action a.

Second, we have configuration safety (Definition 5), defined in the style of Vafeiadis (2011). Here, a configuration is a set of *visited* actions, $V \subseteq (\mathcal{A}_{\text{ctx}} \cup \mathcal{A}_{\text{prg}})$, and heap annotation, $hmap$, annotating precisely the thb-edges for which V is responsible. $\mathsf{safe}^n_{\mathcal{X}}(V, hmap, \ldots)$ asserts that such a configuration is safe for at least n further actions. Unless $n = 0$, a safe configuration must:

- Annotate the (unique) sb-outgoing edge from the command with a heap satisfying the postcondition in case the last action of the command is in V;
- For any "ready-to-execute" action a of the command, it must be possible to extend the heap map so that it is safe also up to a for $n-1$ actions; and
- For any "ready-to-execute" action of the context, any valid extension of the heap map should be safe for $n-1$ actions.

The informal notion of action a being "ready-to-execute" is captured by the constraint that a has not yet been visited whereas all its predecessors have: $a \notin V \wedge \mathsf{Pre}(\{a\}) \subseteq V$.

With these auxiliary definitions, we define the meaning of RSL triples as follows:

Definition 6 (Meaning of RSL triples)**.**
The Hoare triple, $\{P\}\, E\, \{y.\, Q\}$*, holds if and only if*
for all $\langle res, \mathcal{A}_{\text{ctx}}, \mathcal{A}_{\text{prg}}, \mathcal{X}, fst, lst\rangle \in \mathcal{C}[\![E]\!]$*,*
for all $V \subseteq \mathcal{A}_{\text{ctx}}$ *such that* $\mathsf{Pre}_{\mathcal{X}}(V) \subseteq V$*,*
for all $hmap \in \mathsf{Resp}_{\mathcal{X}}(V) \to \mathsf{Heap}_{\text{spec}}$*, for all* $R \in \mathsf{Assn}$*,*
if $hmap(\mathsf{SBin}_{\mathcal{X}}(fst)) \in [\![P * R]\!]$ *and* $\mathsf{Valid}(hmap, V)$*, then*
for all $n \in \mathbb{N}$*,* $\mathsf{safe}^n_{\mathcal{X}}(V, hmap, \mathcal{A}_{\text{ctx}}, \mathcal{A}_{\text{prg}}, lst, Post)$*,*
where $Post = \begin{cases} [\![Q[v/y] * R]\!] & \text{if } res = v \\ \mathsf{Heap}_{\text{spec}} & \text{if } res = \bot \end{cases}$.

The definition says that for any consistent contextual execution of E, all valid configurations annotating only the context edges and satisfying the precondition on the (unique) incoming sb-edge to the program, are safe for any number of steps. As is common in the definitions of the meaning of separation logic triples, the definition bakes in the frame rule—that is, it quantifies over all assertions R and star-conjoins R to the precondition and the postcondition.

7.3 Memory Safety and Race Freedom

The soundness proof of RSL consists of two parts. First, we have to show that every proof rule of §4 is a valid entailment according to the semantics of Hoare triples in Definition 6. Second, we have to show that RSL triples denote something useful for program executions, for example that they do not contain any data races nor any dangling reads.

We start with the second task as it is somewhat simpler. More specifically, we shall show that any consistent execution of a verified program under the true precondition is (a) memory safe, (b) has no uninitialized reads, (c) has no data races, and (d) if the program terminates, its postcondition is satisfiable. By memory safety, we mean that allocation of a location must *happen before* any action reading or writing that location.

Definition 7. *An execution is* memory safe *if and only if* $\forall a \in \mathcal{A}.\ \mathsf{isaccess}_{\ell}(a) \implies \exists b.\ \mathsf{hb}(b, a) \wedge \mathsf{lab}(b) = \mathrm{A}(\ell)$.

Given a validly annotated execution by the heap map $hmap$, observe the following: (1) any action, a, accessing the location ℓ must have $\ell \in \mathsf{dom}(hmap(\mathsf{SBin}(a)))$; and (2) whenever a location is in the domain of the annotation of an edge leading to some action b, (i.e., when $\ell \in \mathsf{dom}(hmap(\_, \_, b))$), then there must be an hb-earlier allocation action for that location. Putting these two together, we get memory safety for validly annotated executions.

Absence of reads from uninitialized locations follows by a similar argument. First, we say that a read action, a, reads from an uninitialized location if $\mathsf{rf}(a) = \bot$, which from (ConsistentRFdom) means that there must be no previous write to that location. We can, however, observe that the annotation validity for read actions, a, requires that $hmap(\mathsf{SBin}(a))(\ell) = \mathsf{Atom}[\_, \_, \_, \mathsf{true}]$ (for atomic locations) or $hmap(\mathsf{SBin}(a))(\ell) = \mathsf{NA}[v, \_]$ (for non-atomic locations). But, in order to get one of these heaps in a valid annotation, it must be the case that there was an hb-earlier write to the same location.

Proving race-freedom is slightly more involved. First, let us formalize exactly what race-freedom is. We say that two actions are *conflicting* if both access the same location, at least one of them is a write, and at least one of the accesses is non-atomic (i.e., atomic accesses do not conflict with one another). An execution is race-free is all conflicting actions are ordered by hb.

Definition 8. *Two actions* $a \neq b$ *are* conflicting *if there exists a location* ℓ *such that* $\mathsf{isaccess}_{\ell}(a)$ *and* $\mathsf{isaccess}_{\ell}(b)$ *and* $\mathsf{iswrite}(a) \vee \mathsf{iswrite}(b)$*, and* $\mathsf{mode}(a) = \mathrm{na} \vee \mathsf{mode}(b) = \mathrm{na}$.

Definition 9. *An execution is* race-free *if and only if for all conflicting actions* $a, b \in \mathcal{A}$*, we have* $\mathsf{hb}(a, b) \vee \mathsf{hb}(b, a)$.

To prove race-freedom, we need the notion of a set of transitions, $\mathcal{T}$, being pairwise independent. We say that $\mathcal{T}$ is pairwise independent, if there exists no pair of transitions in $\mathcal{T}$ such that one happens before the other.

Definition 10 (Independent Edges)**.** *In an execution,* $\mathcal{X}$*, a set of transitions* $\mathcal{T} \subseteq \mathsf{thb}_{\mathcal{X}}$ *is* pairwise independent*, denoted* $\mathsf{PairIndep}(\mathcal{T})$*, if and only if for all* $(\_, a, a'), (\_, b, b') \in \mathcal{T}$*, we have* $\neg \mathsf{hb}_{\mathcal{X}}(a', b)$.

The crux of the race freedom proof is the following independent heap compatibility lemma, which states that in every validly annotated execution, the heaps annotated at independent edges are $\oplus$-compatible.

Lemma 4 (Independent Heap Compatibility)**.** *For every consistent execution,* $\mathcal{X}$*, heap map,* $hmap : \mathsf{Resp}_{\mathcal{X}}(\mathcal{A}_{\mathcal{X}}) \to \mathsf{Heap}_{\text{spec}}$*, and pairwise independent set of transitions,* $\mathcal{T}$*, if* $\mathsf{Valid}(hmap, \mathcal{A}_{\mathcal{X}})$ *holds, then* $\bigoplus_{x \in \mathcal{T}} hmap(x)$ *is defined.*

To prove this lemma, we need the notion of the depth of a set of actions, which we take to be the number of its elements and its predecessors.

Definition 11 (Action Depth)**.** *Given an execution, $\mathcal{X}$, the depth of a set of actions, $A \subseteq \mathcal{A}_\mathcal{X}$, which we denote as $\mathbf{D}_\mathcal{X}(A)$, is the number of actions in the set or that have happened before it,* $|\bigcup_{n\geq 0} \underbrace{\mathsf{Pre}_\mathcal{X}(\cdots \mathsf{Pre}_\mathcal{X}(A)\cdots)}_{n}|$.

The depth of actions satisfies this important property:

Lemma 5. *If $\mathsf{hb}_\mathcal{X}(a,b)$, then $\mathbf{D}_\mathcal{X}(\{a\}) < \mathbf{D}_\mathcal{X}(\{b\})$.*

Lemma 4 is then proved by induction using the metric $\mathbf{D}_\mathcal{X}(\{a \mid (\_, a, \_) \in \mathcal{T}\})$, followed by case analysis on the action in $\mathcal{T}$ with largest $\mathbf{D}_\mathcal{X}(\{-\})$ value.

Putting everything together, our main soundness theorem is stated in terms of complete consistent executions:

$$\begin{aligned}\mathcal{CC}[\![E]\!] \stackrel{\text{def}}{=} \{\ \langle res, \mathcal{X}\rangle \mid \exists a, b.\ & \\ & a \neq b \wedge \mathsf{lab}_\mathcal{X}(a) = \mathsf{lab}_\mathcal{X}(b) = \text{skip} \\ & \wedge \langle res, \{a,b\}, \_, \mathcal{X}, \_, \_\rangle \in \mathcal{C}[\![E]\!]\ \}\end{aligned}$$

where the program expression is put inside the trivial context providing it with an incoming sb-edge from a skip action and an outgoing sb-edge to a skip action. Here it is:

Theorem 1 (Adequacy)**.** *Let $\{\mathsf{true}\}\ E\ \{y.\ Q\}$. For every execution $\langle res, \mathcal{X}\rangle \in \mathcal{CC}[\![E]\!]$, $\mathcal{X}$ is memory safe, has no reads from uninitialized locations and no races. Moreover, if the execution is terminating, then Q holds of the result.*

7.4 Soundness of the Proof Rules

We move on to the proofs of soundness of the individual rules. For each rule, we have to prove that it is a valid entailment given the meaning of RSL triples (Definition 6). With the exception of R-ACQ and CAS*, these proofs are relatively straightforward because the conditions imposed by local validity are almost directly enforced by the proof rules.

The proofs of R-ACQ and CAS* are more complex because we also have to annotate the incoming sw-edges correctly and show that the annotation is valid not only for the program action under consideration, but also for the context actions at the other end—that is, for the write or RMW action with which the read or CAS synchronizes.

We start with the R-ACQ rule. Consider a consistent contextual execution where $\mathcal{A}_{\text{prg}} = \{a\}$ and $\mathsf{lab}(a) = \text{R}_{\text{acq}}(\ell, v)$. We proceed with a case split. If $\mathcal{Q}(v) = \mathsf{emp}$, we can simply annotate any incoming sw-edges with the empty heap and set $hmap'(\mathsf{SBout}_\mathcal{X}(a)) = hmap(\mathsf{SBin}_\mathcal{X}(a))$, which trivially preserves validity. When, however, $\mathcal{Q}(v) \neq \mathsf{emp}$, the situation is much more difficult, because it is not immediately obvious that there is an incoming sw-edge that can be annotated in a way that satisfies the local validity conditions of both the acquire-read and the release-write (or RMW) at the other end. For this case, our proof works as follows.

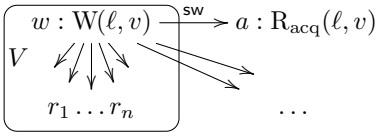
First, as the precondition includes $\mathsf{Init}(\ell)$, we know that there exists a write to ℓ that happens before the acquire read.

Lemma 6 (Init)**.** *If $\mathsf{Valid}(V, hmap)$ and $\mathsf{Pre}_\mathcal{X}(V) \subseteq V$ and $hmap(\_,\_,a)(\ell) = \mathsf{Atom}[\_,\_,\_,\mathsf{true}]$, then there exists $c \in V$ such that $\mathsf{lab}_\mathcal{X}(c) = \text{W}_\_(\ell, \_)$ and $(c,a) \in \mathsf{hb}_\mathcal{X}$.*

Therefore, from the consistency axiom ConsistentRF, we get that $\exists w.\ \mathsf{rf}(a) = w$. As $\mathsf{Pre}_\mathcal{X}(\{a\}) \subseteq V$, we also know $w \in V$.

Next, we will show that w must be a plain atomic write that synchronizes with a. To see why this holds, observe that $\ell \in \mathsf{dom}(hmap(\mathsf{SBin}_\mathcal{X}(w)))$ holds as $hmap$ is locally valid at $w \in V$. Now, informally, we can trace back through the $\mathsf{thb}_\mathcal{X}$ edges to the point where for some node $c \in V$ such that $\mathsf{hb}_\mathcal{X}(c, w)$, we have $\ell \notin \mathsf{dom}(hmap(\mathsf{SBin}_\mathcal{X}(c)))$ and yet $\ell \in \mathsf{dom}(hmap(\mathsf{SBout}_\mathcal{X}(c)))$. Since $hmap$ is locally valid, the only way for this to happen is if $\mathsf{lab}_\mathcal{X}(c) = \text{A}(\ell)$. Similarly, we can follow $\mathsf{thb}_\mathcal{X}$ edges backwards from a and find a node $d \in V$ such that $\mathsf{hb}_\mathcal{X}(d, a)$ and $\ell \notin \mathsf{dom}(hmap(\mathsf{SBin}_\mathcal{X}(d)))$ and $\ell \in \mathsf{dom}(hmap(\mathsf{SBout}_\mathcal{X}(d)))$. Again, since $hmap$ is locally valid, $\mathsf{lab}_\mathcal{X}(d) = \text{A}(\ell)$, and so from the consistency axiom ConsistentAlloc, we obtain that $c = d$. When tracing back from a, at each step we can show that there exist $\mathcal{Q}'$ and b such that $hmap(t_{n+1})(\ell) = \mathsf{Atom}[\_, \mathcal{Q}', b, \_]$ and either $b = \mathsf{false} \wedge \mathcal{Q}'(v) \neq \mathsf{emp}$ or $\mathcal{Q}'(v) = \mathsf{false}$. So, in total, we get $hmap(\mathsf{SBout}_\mathcal{X}(c))(\ell) = \mathsf{Atom}[\mathcal{Q}', \mathcal{Q}', b, \_]$ and either $b = \mathsf{false} \wedge \mathcal{Q}'(v) \neq \mathsf{emp}$ or $\mathcal{Q}'(v) = \mathsf{false}$. Similarly, when tracing back from b, at each step we can show that whenever $hmap(t_{n+1})(\ell) = \mathsf{Atom}[\mathcal{Q}', \_, b, \_]$, then there exist $\mathcal{Q}''$ and b' such that $hmap(t_n)(\ell) = \mathsf{Atom}[\mathcal{Q}'', \_, b', \_]$ and $\mathcal{Q}''(v) \Rightarrow \mathcal{Q}(v)$ and $b' \Rightarrow b$. So, in total we get that there exist $\mathcal{Q}''$ and b' such that $hmap(\mathsf{SBin}_\mathcal{X}(w))(\ell) = \mathsf{Atom}[\mathcal{Q}'', \_, b', \_]$, and $\mathcal{Q}''(v) \Rightarrow \mathcal{Q}'(v)$ and $b' \Rightarrow b$. Since $hmap$ is locally valid at w, $\exists h \in [\![\mathcal{Q}''(v)]\!]$; thus, $b' = b = \mathsf{false}$ and $\mathcal{Q}''(v) \neq \mathsf{emp}$, which means that w is a write that synchronizes with a.

We have the following picture: w synchronizes with a, but possibly also with some other reads $r_1, \ldots, r_n \in V$ and perhaps even some reads not in V.



From the local validity of $hmap$ at $w \in V$, we know that $(h_1 \oplus \cdots \oplus h_n \oplus h_{\text{sink}}) \in [\![\mathcal{Q}''(v)]\!]$, where each h_i is the heap annotated on the sw-edge from w to r_i. What remains to be shown is that we can split h_{sink} further; that is, we can find h', h'_{sink} such that $h_{\text{sink}} = h' \oplus h'_{\text{sink}}$ and $h' \in [\![\mathcal{Q}(v)]\!]$. Then, we annotate the $(\text{“sw”}, w, a)$ edge with h' and $\mathsf{SBout}_\mathcal{X}(a)$ with $h' \oplus \mathsf{SBin}_\mathcal{X}(a)$, thereby ensuring local validity at both a and w. To find such a split, we rely on the following lemma.

Lemma 7 (Well-formedness)**.** *Given a consistent execution $\mathcal{X}$, a prefix-closed set of actions, $V \subseteq \mathcal{A}_\mathcal{X}$ with $\mathsf{Pre}_\mathcal{X}(V) \subseteq V$, a heap map, $hmap \in \mathsf{Resp}_\mathcal{X}(V) \to \mathsf{Heap}_{\text{spec}}$, that is locally valid with respect to V, $\mathsf{Valid}(hmap, V)$, a pairwise*

independent set of transitions $\mathcal{T}$, such that $\{a \mid (\_, a, \_) \in \mathcal{T}\} \subseteq V$ *and* $hmap(\mathcal{T})(\ell) = \mathsf{Atom}[\_, \mathcal{Q}, \_, \_]$*, an action,* w*, such that* $\mathsf{lab}(w) = \mathrm{W}(\ell, v)$ *and* $hmap(\mathsf{SBin}_{\mathcal{X}}(w))(\ell) = \mathsf{Atom}[\mathcal{Q}'', \_, \_, \_]$*, and a partial map* $\mathcal{R} : V \rightharpoonup \mathsf{Assn}$*, such that for all* $a \in \mathsf{dom}(\mathcal{R})$*,* a *is a read action that synchronizes with* w *and acquires ownership of* $\mathcal{R}(a)$*, if moreover,* $\{a \mid \exists(\_, \_, b) \in \mathcal{T} \wedge (a = b \vee \mathsf{hb}(a, b))\} \cap \mathsf{dom}(\mathcal{R}) = \emptyset$*, then,* $\mathcal{Q}''(v) \Rightarrow \mathcal{Q}(v) * \circledast_{r \in \mathsf{dom}(\mathcal{R})} \mathcal{R}(r) * \mathsf{true}$.

The proof of this lemma is rather technical and can be found in the Coq formalization. At a high level, however, it is similar to the proofs already described, using the depth metric to trace back the $\mathcal{T} \cup \{(\text{“sw”}, w, r) \mid r \in \mathsf{dom}(\mathcal{R})\}$ edges until we reach c, the action that allocated ℓ.

Applying this lemma, we get that $(h_1 \oplus \cdots \oplus h_n \oplus h_{\text{sink}}) \in [\![\mathcal{Q}(v) * \mathcal{R}(r_1) * \ldots * \mathcal{R}(r_n) * \mathsf{true}]\!]$, and since for all i, we also know that $\mathsf{precise}(\mathcal{R}(r_i))$ and $h_i \in [\![\mathcal{R}(r_i)]\!]$, we obtain $h_{\text{sink}} \in [\![\mathcal{Q}(v) * \mathsf{true}]\!]$, as required.

The proof of **CAS** is actually much simpler because there cannot be any resource-acquiring reads that synchronize with the write/RMW whence the **CAS** reads from. Details of this proof can be found in the Coq formalization.

7.5 The Coq Formalization

Our Coq development covers the entire soundness proof outlined in this section and follows the LaTeX presentation very closely. To avoid excessive proof duplication, the definitions of configuration safety and triple validity are parametrized with respect to the memory model; that is, either the standard model or the one with the StrongAcyclicHB condition.

One notable difference is that in Coq we represent finite sets of actions, $\mathcal{A}$, as lists, and domain-restricted functions as functions over the full domain. For example, instead of $\mathsf{lab} \in \mathcal{A} \to \mathsf{Act}$, in Coq we have $\mathsf{lab} : \mathsf{AName} \to \mathsf{Act}$, and add a consistency axiom stating that $\forall x \notin \mathcal{A}.\ \mathsf{lab}(x) = \text{skip}$. Similarly, we define $hmap : \mathsf{thb}(\mathsf{AName} \times \mathsf{AName}, \mathsf{AName} \times \mathsf{AName}) \to \mathsf{Heap}_{\text{spec}}$, and in the definition of configuration safety, instead of saying that there exists a $hmap'$ such that the configuration with $hmap \uplus hmap'$ is safe, we say that there exists $hmap''$ such that $\forall e \in \mathsf{Resp}_{\mathcal{X}}(V).\ hmap''(e) = hmap(e)$ holds and the configuration with $hmap''$ is safe.

Another difference is that in Coq the treatment of assertions up to $\sim$ is achieved by defining a syntactic assertion normalization function, $norm$, with the property that $P \sim Q \iff norm(P) = norm(Q)$. Then, we represent $\mathsf{Assn}/\!\sim$ as $\{P \in \mathsf{Assn} \mid norm(P) = P\}$.

Finally, following Nanevski et al. (2010), we represent heaps as the option type $\mathsf{Heap}_{\text{spec}} \cup \{\bot\}$, with $\bot$ representing undefined heaps. This removes the ‘definedness’ side-conditions from the statements of commutativity and associativity of heap composition. In effect, we move the definedness checks to the semantics of assertions, where we ensure that $\bot \notin [\![P]\!]$ for any assertion P.

The formal development excluding standard libraries consists of about 3000 lines of definitions and statements of lemmas and theorems, 5500 lines of proof, 500 lines of comments, and took the first author about two months to complete. It is worth pointing out that the formal proof revealed that a bug that we had missed in our earlier paper proofs: namely, the requirement that the ownership transfer governed by the R-ACQ rule to be precise. While we do not think that this side condition is strictly necessary for soundness in the absence of the conjunction rule, the current proof style fundamentally requires it.

8. Related Work and Conclusion

This paper introduced relaxed separation logic, a moderate extension of concurrent separation logic (O’Hearn 2007) with special primitives for handling C11’s acquire and release atomic accesses.

8.1 Related Work

About the C11 Model The C11 concurrency model is part of the C and C++ 2011 standards (ISO/IEC 9899:2011; ISO/IEC 14882:2011), and has been formalized by Batty et al. (2011). In a subsequent paper, Batty et al. (2012) simplified the C11 model in the absence of consume reads. It is this simplified model that we used in this paper.

In a recent paper, Batty et al. (2013) considered the notion of library atomicity in the context of the C11 memory model. While this work is largely orthogonal to our defining a program logic about C11, we expect that combining the two approaches will be fruitful as program logics are often the means for proving atomicity, at least in the SC setting.

Logics for Other Weak Memory Models We are aware of only three lines of work that define program logics over a relaxed memory model, none of which handles the C11 memory model.

- Ferreira et al. (2010) proved the soundness of concurrent separation logic (CSL) over a class of relaxed memory models, all satisfying the DRF-guarantee. In hindsight, their result is not surprising as the soundness of CSL over SC (sequential consistency) ensures that CSL-verified programs do not contain any data races, and hence whether the soundness proof is done over SC or over the relaxed memory model is irrelevant.
- Ridge (2010) developed a rely-guarantee proof system over x86-TSO and used it to verify a x86-TSO version of Simpson’s four slot algorithm, with all the results mechanized in the HOL theorem prover.
- Wehrman and Berdine (2011) proposed a variant of separation logic for x86-TSO featuring primitive assertions for modelling the state of the TSO buffers and both temporal and spatial separating conjunctions.

Besides obviously handling different memory models and being quite different program logics, there is a fundamental difference between the current work and these earlier pa-

pers on program logics for relaxed memory models. In this work, we define the meaning of Hoare triples directly over an axiomatic partial order semantics for concurrent programs, whereas the earlier works used an operational or an operationally flavoured trace semantics, very much like the traditional soundness proofs over SC. As a result, our soundness proof is completely different from the soundness proofs of the aforementioned papers.

8.2 Possible Future Research Directions

Being the first program logic for C11 concurrency, there are numerous opportunities for extending RSL, for example to deal with more advanced features of the C11 memory model, such as consume reads and memory fences. Similarly, one can also try to adapt to C11 setting more advanced program logics, such as RGSep (Vafeiadis and Parkinson 2007), concurrent abstract predicates (Dinsdale-Young et al. 2010), or CaReSL (Turon et al. 2013).

Initialization of Atomics For simplicity, RSL tags locations as atomic or non-atomic and permits atomic accesses only on atomic locations and non-atomic accesses only on non-atomic locations. As a result of this choice, initialization writes to atomic accesses in our model also have to be atomic, whereas the C11 standard also allows non-atomic initialization writes to atomic location. To enable the verification of such programs, we should somehow allow the following 'conversion' rule

$$\frac{\mathcal{Q}(v) = \mathsf{emp}}{\left\{\ell \mapsto v\right\}\, 0\, \left\{\mathsf{Rel}(\ell, \mathcal{Q}) * \mathsf{Acq}(\ell, \mathcal{Q}) * \mathsf{Init}(\ell)\right\}}$$

Automation Another important research direction would be to develop techniques and tools for automating RSL proofs by adapting some of the work that has been done in automating standard separation logic (Distefano et al. 2006; Calcagno et al. 2009; Dudka et al. 2011).

Acknowledgments

We would like to thank Lars Birkedal, Arthur Charguéraud, Mike Dodds, Alexey Gotsman, Matthew Parkinson, Aaron Turon, John Wickerson, and the anonymous OOPSLA 2013 reviewers for their very useful comments that improved the content of this paper. The research was supported by the EC FP7 FET Young Explorers scheme via the project ADVENT.

References

M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL 2011*, pages 55–66. ACM, 2011.

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*, pages 509–520. ACM, 2012.

M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency. In *POPL 2013*, pages 235–248. ACM, 2013.

L. Birkedal, K. Støvring, and J. Thamsborg. The category-theoretic solution of recursive metric-space equations. *Theoretical Computer Science*, 411(47):4102–4122, 2010.

J. Boyland. Checking interference with fractional permissions. In *SAS 2003*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.

C. Calcagno, D. Distefano, and V. Vafeiadis. Bi-abductive resource invariant synthesis. In *APLAS*, volume 5904 of *LNCS*, pages 259–274. Springer, 2009.

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

D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, volume 3920 of *LNCS*, pages 287–302. Springer, 2006.

K. Dudka, P. Peringer, and T. Vojnar. Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In *CAV*, volume 6806 of *LNCS*, pages 372–378. Springer, 2011.

R. Ferreira, X. Feng, and Z. Shao. Parameterized memory models and concurrent separation logic. In *ESOP 2010*, volume 6012 of *LNCS*, pages 267–286. Springer, 2010.

C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI 1993*, pages 237–247. ACM, 1993.

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

ISO/IEC 9899:2011. Programming language C, 2011.

P. E. McKenney and B. Garst. N1525: Memory-order rationale, 2011. Available at `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1525.htm`.

A. Nanevski, V. Vafeiadis, and J. Berdine. Structuring the verification of heap-manipulating programs. In *POPL*, pages 261–274. ACM, 2010.

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

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

S. Sarkar, K. Memarian, S. Owens, M. Batty, P. Sewell, L. Maranget, J. Alglave, and D. Williams. Synchronising C/C++ and POWER. In *PLDI 2012*, pages 311–322. ACM, 2012.

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

V. Vafeiadis. Concurrent separation logic and operational semantics. In *MFPS 2011*, volume 276 of *ENTCS*, pages 335–351. Elsevier, 2011.

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

M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2): 181–210, Apr. 1991.

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

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

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 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]:

$$\begin{array}{l} [x]_{\texttt{at}} := 1 \\ \texttt{if } [y]_{\texttt{at}} == 0 \texttt{ then} \\ \quad \textit{/* crit. section */} \end{array} \quad \Big\| \quad \begin{array}{l} [y]_{\texttt{at}} := 1 \\ \texttt{if } [x]_{\texttt{at}} == 0 \texttt{ then} \\ \quad \textit{/* crit. section */} \end{array}$$

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):

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

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

In general, then, release-acquire in C11 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

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

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

$$\frac{K[e] \xrightarrow{\alpha} K[e'] \quad \text{if} \quad e \xrightarrow{\alpha} e'}{}$$

$$\texttt{alloc}(n) \xrightarrow{\mathbb{A}(\ell..\ell+n-1)} \ell$$

$$[\ell]_O \xrightarrow{\mathbb{R}(\ell,V,O)} V$$

$$[\ell]_O := V \xrightarrow{\mathbb{W}(\ell,V,O)} 0$$

$$\texttt{CAS}(\ell, V_o, V_n) \xrightarrow{\mathbb{U}(\ell,V_o,V_n)} 1$$

$$\texttt{CAS}(\ell, V_o, V_n) \xrightarrow{\mathbb{R}(\ell,V',\texttt{at})} 0 \quad \text{if} \quad V' \neq V_o$$

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

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

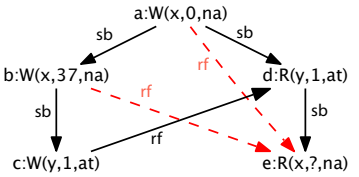
$$\langle T \uplus [i \mapsto (a, K[\texttt{fork}\ e])]; G\rangle \longrightarrow \langle T \uplus [i \mapsto (a, K[0])] \uplus [j \mapsto (a, e)]; G\rangle$$

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:

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

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.,* let $x = V$ 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 $[\![e]\!] = \mathbf{err}$; the C11 semantics leaves such programs undefined. Any expression verified by GPS is guaranteed to be free of data races on non-atomic locations and memory errors.

3. GPS: 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.

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

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

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

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

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

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

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

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

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

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:

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

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

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

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:

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

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:

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

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 $\mathbf{LP}$ 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:

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

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:

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

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

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

Finally, we have a rule for atomic writes:

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

Writes are surprisingly subtle. Prior to writing, our thread knows some lower bound s on the protocol state. But because the write may be racing with unknown other writes (or `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.

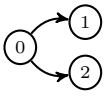
In GPS terms, we again imagine some resource invariant P to which the critical sections provide access—but we need some way to reflect the logical synchronization condition of the algorithm. More generally, we need a way for threads to gain ownership of resources not because they won a physical, `CAS`-mediated 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:

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

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):



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

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:

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

Here we have elided the protocol and ghost types, which are $\mathbf{Choice}$ 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):

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

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

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

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 + `buf`), together with a reader index (at q + `ri`) specifying the array offset of the next item to be consumed, and a writer index (at q + `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:

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

The spec is parameterized over a predicate P that should hold of all the elements in the buffer; it guarantees that $P(x)$ 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 $\Box$ 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 $i < j_0 + N \le j + N$, the escrows we just learned about in the previous step include $\mathbf{PE}(\gamma, q, i)$, which we need later.

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

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

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

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

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

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

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

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

5. 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.

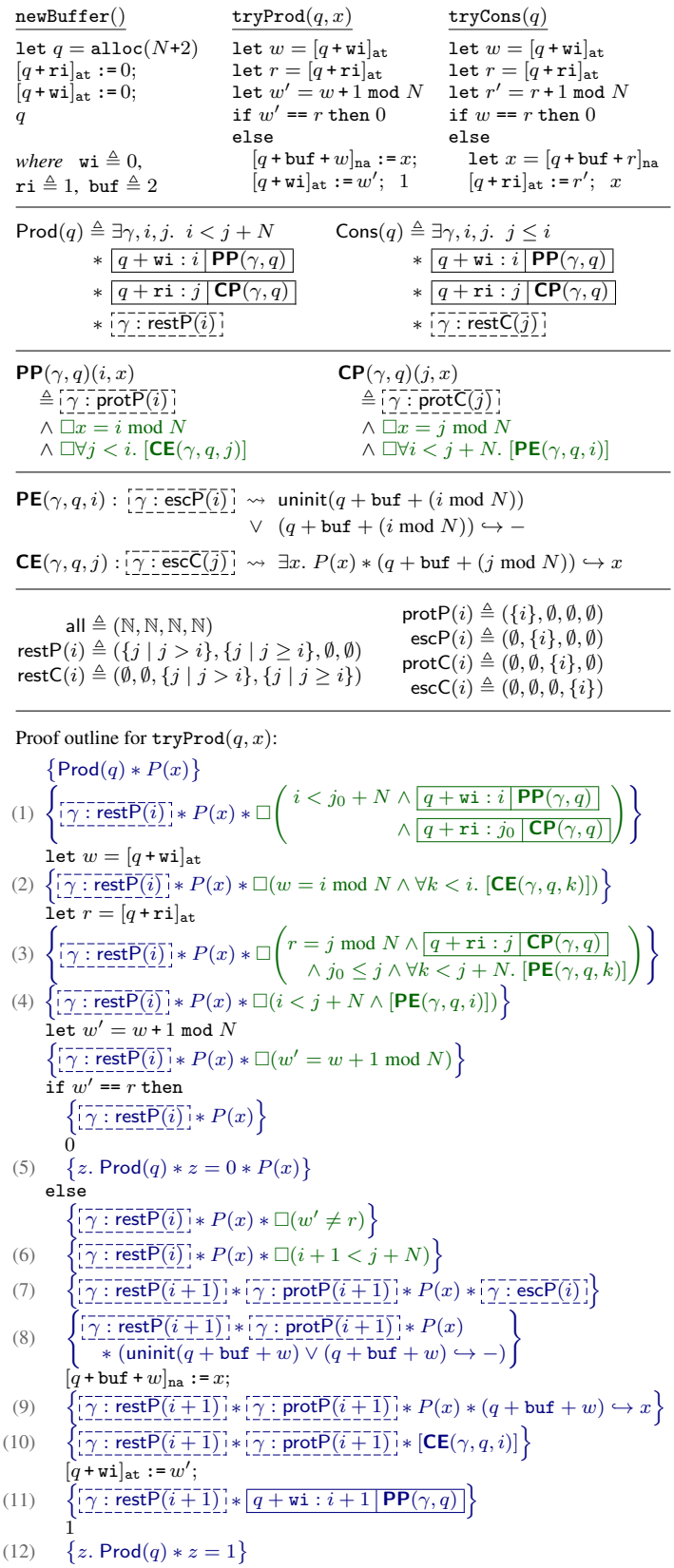


Figure 3. Proof excerpt for the circular buffer case study

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

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

A.2 Semantics

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

$$
\begin{array}{rcll}
n \texttt{ + } m & \xrightarrow{\mathbb{S}} & k & k = n + m \\
n \texttt{ mod } m & \xrightarrow{\mathbb{S}} & k & k = n \bmod m \\
n \texttt{ == } m & \xrightarrow{\mathbb{S}} & 1 & n = m \\
n \texttt{ == } m & \xrightarrow{\mathbb{S}} & 0 & n \neq m \\
\texttt{let } x = V \texttt{ in } e & \xrightarrow{\mathbb{S}} & e[V/x] & \\
\texttt{repeat } e \texttt{ end} & \xrightarrow{\mathbb{S}} & \texttt{let } x = e \texttt{ in if } x \texttt{ then } x \texttt{ else repeat } e \texttt{ end} & \\
\texttt{if } V \texttt{ then } e_1 \texttt{ else } e_2 & \xrightarrow{\mathbb{S}} & e_1 & V \neq 0 \\
\texttt{if } V \texttt{ then } e_1 \texttt{ else } e_2 & \xrightarrow{\mathbb{S}} & e_2 & V = 0
\end{array}
$$

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

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

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

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

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

$$
\begin{array}{rcl}
\mathsf{execs}(e) & \triangleq & \{\ (e', G) \mid \langle [i \mapsto (\mathsf{start}, e)]; ([\mathsf{start} \mapsto \mathbb{S}], \emptyset, \emptyset, \emptyset)\rangle \longrightarrow^* \langle [i \mapsto (\_, e')] \uplus T; G\rangle\ \} \\
[\![e]\!] & \triangleq & \begin{cases} \mathbf{err} & \exists(\_, G) \in \mathsf{execs}(e).\ \mathsf{dataRace}(G) \vee \mathsf{memoryError}(G) \\ \{\ V \mid (V, \_) \in \mathsf{execs}(e)\ \} & \text{otherwise} \end{cases}
\end{array}
$$

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.\ \mathrm{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)} \\
\text{where } \mathsf{hb} \triangleq (\mathsf{sb} \cup \mathsf{sw})^+ & \\
\qquad \mathsf{sw} \triangleq \{(a, b) \mid \mathsf{rf}(a) = b,\ \mathsf{isAtomic}(a),\ \mathsf{isAtomic}(b)\} & \\
\qquad \mathsf{reads}(a, \ell, V) \triangleq A(a) \in \{\mathbb{R}(\ell, V, -), \mathbb{U}(\ell, V, -)\} & \\
\qquad \mathsf{writes}(a, \ell, V) \triangleq A(a) \in \{\mathbb{W}(\ell, V, -), \mathbb{U}(\ell, -, V)\} & \\
\qquad \mathrm{strictTotalOrder}(S, R) \triangleq (\nexists a.\ R(a, a)), & \\
\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:

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

- 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

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

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

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

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

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

B.2.2 Rely/guarantee

α	r'	$r' \in \mathsf{rely}(r, \alpha)$ if
$\mathbb{R}(\ell, V, \mathtt{na})$	r	$r[\ell] = \mathtt{na}(V') \implies V = V'$
$\mathbb{R}(\ell, V, \mathtt{at})$	$r \oplus \lvert r_{\mathsf{rf}} \rvert$	$r[\ell] = \mathtt{at}(-) \implies r_{\mathsf{rf}} \in \mathsf{envMove}(r, \ell, V)$
$\mathbb{U}(\ell, V, V')$	$r \oplus r_{\mathsf{rf}}$	$r[\ell] = \mathtt{at}(-) \implies r_{\mathsf{rf}} \in \mathsf{envMove}(r, \ell, V)$
$\mathbb{W}(\ell, V, \mathtt{at})$	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

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

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

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

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

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

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

where

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

B.3.4 Concurrent compatibility

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

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

where $N \triangleq \mathrm{dom}(G.A)$

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

$$
\begin{array}{lrcl}
\text{Sort} & \theta & ::= & \mathsf{Val} \mid \mathsf{State} \mid \mathsf{PCM}_\mu \\
\text{Var} & X & ::= & \ell \mid x \mid s \\
\text{Term} & t & ::= & X \mid n \mid \varepsilon_\mu \mid t \cdot_\mu t \mid \cdots \\
\text{Proposition} & P & ::= & t = t \mid P \wedge P \mid P \vee P \mid P \Rightarrow P \mid \forall X : \theta.\, P \mid \exists X : \theta.\, P \\
& & \mid & \Box P \mid P * P \mid \mathsf{uninit}(t) \mid t \hookrightarrow t \mid \boxed{t : t \mid \tau} \mid t \sqsubseteq_\tau t \mid \boxed{\begin{array}{c|c}\multicolumn{2}{c}{t : t} \\ \hline \multicolumn{2}{c}{\mu}\end{array}} \mid [\sigma]
\end{array}
$$

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
$\Box P$	$\lvert r \rvert \in [\![P]\!]^\rho$
$P_1 * P_2$	$r \in [\![P_1]\!]^\rho * [\![P_2]\!]^\rho$
$\mathsf{uninit}(t)$	$r.\Pi([\![t]\!]^\rho) = \mathsf{uninit}$
$t \hookrightarrow t'$	$r.\Pi([\![t]\!]^\rho) = \mathtt{na}([\![t']\!]^\rho)$
$\boxed{t : t' \mid \tau}$	$r.\Pi([\![t]\!]^\rho) \geq \mathtt{at}(\tau, \{[\![t']\!]^\rho\})$
$\boxed{\begin{array}{c\|c}\multicolumn{2}{c}{t : t'} \\ \hline \multicolumn{2}{c}{\mu}\end{array}}$	$r.g(\mu)([\![t]\!]^\rho) \geq [\![t']\!]^\rho$
$[\sigma]$	$\sigma \in r.\Sigma$

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

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

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

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

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

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

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

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

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

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

then $\exists a$.

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

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

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

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

then

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

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

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

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

then $\exists a, S'$.

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

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

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

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

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

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

C.3.4 Protocol equivalence for writes

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

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

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

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

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

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

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

C.3.6 Step preparation

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

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

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

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

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

C.3.7 Instrumented execution and adequacy

We define functions $\mathsf{erase} : \mathit{IThreadMap} \to \mathit{ThreadMap}$ and $\mathsf{post} : \mathit{IThreadMap} \to \mathbb{N} \xrightarrow{\text{fin}} \mathit{Val} \to \mathit{Prop}$ 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

$\texttt{let}\ x = \texttt{alloc}(1)\ \texttt{in}$
$\texttt{let}\ y = \texttt{alloc}(1)\ \texttt{in}$
$[x]_{\texttt{na}} \texttt{:=} 0;$
$[y]_{\texttt{at}} \texttt{:=} 0;$
$\texttt{fork}\ [x]_{\texttt{na}} \texttt{:=} 1; [y]_{\texttt{at}} \texttt{:=} 1;$
$\texttt{repeat}\ [y]_{\texttt{at}}\ \texttt{end};$
$[x]_{\texttt{na}}$

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:

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

D.1.3 Verification

$\left\{\mathsf{true}\right\}$
$\left\{\exists\gamma.\ \boxed{\gamma : \diamond}^{\dagger}\right\}$
$\texttt{let}\ x = \texttt{alloc}(1)\ \texttt{in}$
$\left\{\boxed{\gamma : \diamond}^{\dagger} * \mathsf{uninit}(x)\right\}$
$\texttt{let}\ y = \texttt{alloc}(1)\ \texttt{in}$
$\left\{\boxed{\gamma : \diamond}^{\dagger} * \mathsf{uninit}(x) * \mathsf{uninit}(y)\right\}$
$[x]_{\texttt{na}} := 0;$
$\left\{\boxed{\gamma : \diamond}^{\dagger} * x \hookrightarrow 0 * \mathsf{uninit}(y)\right\}$
$[y]_{\texttt{at}} := 0;$
$\left\{\boxed{\gamma : \diamond}^{\dagger} * x \hookrightarrow 0 * \boxed{y : 0 \mid \mathsf{YP}(\gamma, x)}\right\}$
$\texttt{fork}$
$\quad\left\{x \hookrightarrow 0 * \boxed{y : 0 \mid \mathsf{YP}(\gamma, x)}\right\}$
$\quad[x]_{\texttt{na}} := 1;\ \left\{x \hookrightarrow 1 * \boxed{y : 0 \mid \mathsf{YP}(\gamma, x)}\right\}$
$\quad\left\{[\mathsf{XE}(\gamma, x)] * \boxed{y : 0 \mid \mathsf{YP}(\gamma, x)}\right\}$
$\quad[y]_{\texttt{at}} := 1;$
$\quad\left\{\boxed{y : 1 \mid \mathsf{YP}(\gamma, x)}\right\}$
$\left\{\boxed{\gamma : \diamond}^{\dagger} * \boxed{y : 0 \mid \mathsf{YP}(\gamma, x)}\right\}$
$\texttt{repeat}\ [y]_{\texttt{at}}\ \texttt{end};$
$\left\{\boxed{\gamma : \diamond}^{\dagger} * \boxed{y : 1 \mid \mathsf{YP}(\gamma, x)} * [\mathsf{XE}(\gamma, x)]\right\}$
$\left\{\boxed{y : 1 \mid \mathsf{YP}(\gamma, x)} * x \hookrightarrow 1\right\}$
$[x]_{\texttt{na}}$
$\left\{z.\ z = 1\right\}$

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

D.2.3 Proof setup

Top-level spec

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

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`

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

D.2.5 Verification of `spin`

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

D.2.6 Verification of `lock`

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

D.2.7 Verification of `unlock`

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

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

D.3 Ernie Cohen's lock example

D.3.1 Parameters

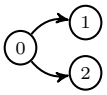
Fix some assertion P for the resources to be raced for.

D.3.2 Code

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

D.3.3 Proof setup

Protocols



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

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

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

D.3.4 Proof

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

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

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

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 `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:

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

We also have two protocols, $\mathbf{Head}$ and $\mathbf{Tail}$, each with a single state called Inv, interpreted as:

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

High-level predicates

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

D.4.4 Verification of `newBuffer`

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

D.4.5 Verification of `findTail`

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

D.4.6 Verification of `tryEnq`

$\left\{P(x) * \square(\mathsf{Queue}(q))\right\}$
$\texttt{let } n = \texttt{alloc}(2);$
$\left\{P(x) * \mathsf{uninit}(n + \texttt{data}) * \mathsf{uninit}(n + \texttt{link}) * \square(n \neq 0)\right\}$
$[n + \texttt{data}]_{\texttt{na}} := x;$
$\left\{P(x) * (n + \texttt{data}) \hookrightarrow x * \mathsf{uninit}(n + \texttt{link})\right\}$
$\left\{P(x) * (n + \texttt{data}) \hookrightarrow x * \mathsf{uninit}(n + \texttt{link}) * \exists \gamma'.\ \boxed{\gamma' : \diamond}^{\square}\right\}$
$[n + \texttt{link}]_{\texttt{at}} := 0;$
$\left\{P(x) * (n + \texttt{data}) \hookrightarrow x * \boxed{n + \texttt{link} : \mathsf{Null} \mid \mathsf{Link}(\gamma')} * \boxed{\gamma' : \diamond}^{\square}\right\}$
$\texttt{let } t = \texttt{repeat findTail}(q) \texttt{ end}$
$\left\{P(x) * (n + \texttt{data}) \hookrightarrow x * \boxed{n + \texttt{link} : \mathsf{Null} \mid \mathsf{Link}(\gamma')} * \boxed{\gamma' : \diamond}^{\square} * \exists \gamma.\ \boxed{t + \texttt{link} : \mathsf{Null} \mid \mathsf{Link}(\gamma)}\right\}$
$\texttt{if CAS}(t + \texttt{link}, 0, n) \texttt{ then}$
  $\left\{\boxed{t + \texttt{link} : \mathsf{Linked}(n) \mid \mathsf{Link}(\gamma)} * \boxed{n + \texttt{link} : \mathsf{Null} \mid \mathsf{Link}(\gamma')}\right\}$
  $\left\{\boxed{n + \texttt{link} : \mathsf{Null} \mid \mathsf{Link}(\gamma')} * \boxed{q + \texttt{tail} : \mathsf{Inv} \mid \mathsf{Tail}}\right\}$
  $[q + \texttt{tail}]_{\texttt{at}} := n;$
  $\left\{\mathsf{true}\right\}$
  1
  $\left\{z.\ z = 1\right\}$
$\texttt{else}$
  $\left\{P(x) * \boxed{t + \texttt{link} : \mathsf{Null} \mid \mathsf{Link}(\gamma)}\right\}$
  0
  $\left\{z.\ z = 0 * P(x)\right\}$

D.4.7 Verification of `tryDeq`

$\{\mathsf{Queue}(q)\}$
$\{\Box(\boxed{q + \texttt{head} : \mathsf{Inv} \mid \mathbf{Head}})\}$
`let` $s = [q + \texttt{head}]_{\mathtt{at}}$
$\{\exists \gamma.\ \boxed{s + \texttt{link} : \mathsf{Null} \mid \mathbf{Link}(\gamma)}\}$
`let` $n = [s + \texttt{link}]_{\mathtt{at}}$
$\{n \neq 0 \Rightarrow \exists \gamma'.\ [\mathbf{DEQ}(n + \texttt{data}, \gamma, \gamma')] * \boxed{n + \texttt{link} : \mathsf{Null} \mid \mathbf{Link}(\gamma')}\}$
`if` n `==` 0 `then`
  $\{\mathsf{true}\}$
  0
  $\{x.\ x = 0\}$
`else`
  $\{\boxed{q + \texttt{head} : \mathsf{Inv} \mid \mathbf{Head}} * [\mathbf{DEQ}(n + \texttt{data}, \gamma, \gamma')] * \boxed{n + \texttt{link} : \mathsf{Null} \mid \mathbf{Link}(\gamma')}\}$
  `if` $\texttt{CAS}(q + \texttt{head}, s, n)$ `then`
    $\{\boxed{q + \texttt{head} : \mathsf{Inv} \mid \mathbf{Head}} * \exists x.\ n + \texttt{data} \hookrightarrow x * P(x)\}$
    $[n + \texttt{data}]_{\mathtt{na}}$
    $\{x.\ P(x)\}$
  `else`
    $\{\boxed{q + \texttt{head} : \mathsf{Inv} \mid \mathbf{Head}}\}$
    0
    $\{x.\ x = 0\}$

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

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

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:

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

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

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

and state interpretations

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

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`

$\{\mathsf{true}\}$
$\{\exists\gamma.\ \boxed{\gamma : \mathsf{all}}\}$
$\texttt{let}\ q = \texttt{alloc}(N + 2)$
$\{\boxed{\gamma : \mathsf{all}} * \mathsf{uninit}(q) * \cdots * \mathsf{uninit}(q + N + 1)\}$
$\{\boxed{\gamma : \mathsf{all}} * \mathsf{uninit}(q) * \cdots * \mathsf{uninit}(q + N + 1)\}$
$\{\boxed{\gamma : \mathsf{all}} * \mathsf{uninit}(q + \texttt{wi}) * \mathsf{uninit}(q + \texttt{ri}) * [\mathbf{PE}(\gamma, q, 0)] * \cdots * [\mathbf{PE}(\gamma, q, N - 1)]\}$
$[q + \texttt{ri}]_{\texttt{at}} := 0;$
$\{\boxed{q + \texttt{ri} : 0 \mid \mathbf{CP}(\gamma, q)} * \boxed{\gamma : \mathsf{restC}(0)} * \boxed{\gamma : \mathsf{protP}(0)} * \boxed{\gamma : \mathsf{restP}(0)} * \mathsf{uninit}(q + \texttt{wi})\}$
$[q + \texttt{wi}]_{\texttt{at}} := 0;$
$\{\boxed{q + \texttt{ri} : 0 \mid \mathbf{CP}(\gamma, q)} * \boxed{q + \texttt{wi} : 0 \mid \mathbf{PP}(\gamma, q)} * \boxed{\gamma : \mathsf{restC}(0)} * \boxed{\gamma : \mathsf{restP}(0)}\}$
q
$\{\mathsf{Prod}(q) * \mathsf{Cons}(q)\}$

D.5.5 Verification of `tryProd`

$\left\{\mathsf{Prod}(q) * P(x)\right\}$
$\left\{\boxed{\gamma : \mathsf{restP}(i)} * P(x) * \square\left(i < j_0 + N \wedge \boxed{q + \texttt{wi} : i \mid \mathbf{PP}(\gamma, q)} \wedge \boxed{q + \texttt{ri} : j_0 \mid \mathbf{CP}(\gamma, q)}\right)\right\}$
`let` $w = [q + \texttt{wi}]_{\texttt{at}}$
$\left\{\boxed{\gamma : \mathsf{restP}(i)} * P(x) * \square(w = i \bmod N \wedge \forall k < i.\ [\mathbf{CE}(\gamma, q, k)])\right\}$
`let` $r = [q + \texttt{ri}]_{\texttt{at}}$
$\left\{\boxed{\gamma : \mathsf{restP}(i)} * P(x) * \square\left(r = j \bmod N \wedge \forall k < j + N.\ [\mathbf{PE}(\gamma, q, k)] \wedge j_0 \le j \wedge \boxed{q + \texttt{ri} : j \mid \mathbf{CP}(\gamma, q)}\right)\right\}$
$\left\{\boxed{\gamma : \mathsf{restP}(i)} * P(x) * \square(i < j + N \wedge [\mathbf{PE}(\gamma, q, i)])\right\}$
`let` $w' = w + 1$ `mod` N
$\left\{\boxed{\gamma : \mathsf{restP}(i)} * P(x) * \square(w' = w + 1 \bmod N)\right\}$
`if` w' `==` r `then`
  $\left\{\boxed{\gamma : \mathsf{restP}(i)} * P(x)\right\}$
  0
  $\left\{z.\ \mathsf{Prod}(q) * z = 0 * P(x)\right\}$
`else`
  $\left\{\boxed{\gamma : \mathsf{restP}(i)} * P(x) * \square(w' \neq r)\right\}$
  $\left\{\boxed{\gamma : \mathsf{restP}(i)} * P(x) * \square(i + 1 < j + N)\right\}$
  $\left\{\boxed{\gamma : \mathsf{restP}(i+1)} * \boxed{\gamma : \mathsf{protP}(i+1)} * P(x) * \boxed{\gamma : \mathsf{escP}(i)}\right\}$
  $\left\{\boxed{\gamma : \mathsf{restP}(i+1)} * \boxed{\gamma : \mathsf{protP}(i+1)} * P(x) * (\mathsf{uninit}(q + \texttt{buf} + w) \vee (q + \texttt{buf} + w) \hookrightarrow -)\right\}$
  $[q + \texttt{buf} + w]_{\texttt{na}} := x;$
  $\left\{\boxed{\gamma : \mathsf{restP}(i+1)} * \boxed{\gamma : \mathsf{protP}(i+1)} * P(x) * (q + \texttt{buf} + w) \hookrightarrow x\right\}$
  $\left\{\boxed{\gamma : \mathsf{restP}(i+1)} * \boxed{\gamma : \mathsf{protP}(i+1)} * [\mathbf{CE}(\gamma, q, i)]\right\}$
  $[q + \texttt{wi}]_{\texttt{at}} := w';$
  $\left\{\boxed{\gamma : \mathsf{restP}(i+1)} * \boxed{q + \texttt{wi} : i + 1 \mid \mathbf{PP}(\gamma, q)}\right\}$
  1
  $\left\{z.\ \mathsf{Prod}(q) * z = 1\right\}$

D.5.6 Verification of `tryCons`

$\left\{\mathsf{Cons}(q)\right\}$
$\left\{\boxed{\gamma : \mathsf{restC}(j)} * \Box\left(j \le i_0 \land \boxed{q + \texttt{wi} : i_0 \mid \mathbf{PP}(\gamma, q)} \land \boxed{q + \texttt{ri} : j \mid \mathbf{CP}(\gamma, q)}\right)\right\}$
`let` $w = [q + \texttt{wi}]_{\texttt{at}}$
$\left\{\boxed{\gamma : \mathsf{restC}(j)} * \Box\left(w = i \bmod N \land \forall k < i.\ [\mathbf{CE}(\gamma, q, k)] \land i_0 \le i \land \boxed{q + \texttt{wi} : i \mid \mathbf{PP}(\gamma, q)}\right)\right\}$
$\left\{\boxed{\gamma : \mathsf{restC}(j)} * \Box(j \le i)\right\}$
`let` $r = [q + \texttt{ri}]_{\texttt{at}}$
$\left\{\boxed{\gamma : \mathsf{restC}(j)} * \Box(r = j \bmod N \land \forall k < j + N.\ [\mathbf{PE}(\gamma, q, k)])\right\}$
`if` w `==` r `then`
  $\left\{\boxed{\gamma : \mathsf{restC}(j)}\right\}$ 0
  $\left\{x.\ \mathsf{Cons}(q) * x = 0\right\}$
`else`
  $\left\{\boxed{\gamma : \mathsf{restC}(j)} * w \ne r\right\}$
  $\left\{\boxed{\gamma : \mathsf{restC}(j)} * \Box(j < i)\right\}$
  $\left\{\boxed{\gamma : \mathsf{restC}(j+1)} * \boxed{\gamma : \mathsf{protC}(j+1)} * \boxed{\gamma : \mathsf{escC}(j)}\right\}$
  $\left\{\boxed{\gamma : \mathsf{restC}(j+1)} * \boxed{\gamma : \mathsf{protC}(j+1)} * \exists x.\ P(x) * (q + \texttt{buf} + r) \hookrightarrow x\right\}$
  `let` $x = [q + \texttt{buf} + r]_{\texttt{na}}$
  $\left\{\boxed{\gamma : \mathsf{restC}(j+1)} * \boxed{\gamma : \mathsf{protC}(j+1)} * P(x) * (q + \texttt{buf} + r) \hookrightarrow x\right\}$
  $\left\{\boxed{\gamma : \mathsf{restC}(j+1)} * P(x) * \boxed{\gamma : \mathsf{protC}(j+1)} * [\mathbf{PE}(\gamma, q, j + N)]\right\}$
  $[q + \texttt{ri}]_{\texttt{at}}$ `:=` r `+` 1 `mod` N;
  $\left\{\boxed{\gamma : \mathsf{restC}(j+1)} * P(x) * \boxed{q + \texttt{ri} : j + 1 \mid \mathbf{CP}(\gamma, q)}\right\}$
  x
  $\left\{x.\ \mathsf{Cons}(q) * P(x)\right\}$

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

$\mathtt{newLock}() \triangleq$
  $\mathtt{let}\ x = \mathtt{alloc}(2)$
  $[x + \mathtt{ns}]_{\mathtt{at}} := 0;$
  $[x + \mathtt{tc}]_{\mathtt{at}} := 0;$
  x

$\mathtt{lock}(x) \triangleq$
  $\mathtt{let}\ y = \mathtt{FAI}(x + \mathtt{tc})$
  $\mathtt{repeat}$
    $\mathtt{let}\ z = [x + \mathtt{ns}]_{\mathtt{at}}$
    $y == z$
  $\mathtt{end}$

$\mathtt{unlock}(x) \triangleq$
  $\mathtt{let}\ z = [x + \mathtt{ns}]_{\mathtt{at}}$
  $[x + \mathtt{ns}]_{\mathtt{at}} := (z + 1)\ \mathtt{mod}\ \mathsf{C}$

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 $\mathsf{MayAcquire}$ 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:

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

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

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

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.

$$\mathsf{Perms}^\gamma_{\geq}(t) \Rrightarrow \mathsf{LkPerm}^\gamma(t) * \mathsf{UnPerm}^\gamma(t) * \mathsf{Perms}^\gamma_{\geq}(t+1) \quad \text{(GetPerms)}$$

$$\mathsf{LkPerm}^\gamma(t) * \mathsf{LkPerm}^\gamma(t) \Rrightarrow \bot \quad \text{(LkPermExclusive)}$$

$$\mathsf{UsedUP}^\gamma_{<}(t) * \mathsf{UnPerm}^\gamma(t') \Rrightarrow t \leq t' \quad \text{(UnusedUnPerms)}$$

$$\mathsf{UsedUP}^\gamma_{<}(t) * \mathsf{UnPerm}^\gamma(t) \Rrightarrow \mathsf{UsedUP}^\gamma_{<}(t+1) \quad \text{(UseUnPerm)}$$

$$\mathsf{UsedUP}^\gamma_{<}(t) \Rrightarrow \Box(\mathsf{UsedUP}^\gamma_{<}(t)) \quad \text{(UsedPermsPure)}$$

$$\mathsf{MyTkts}^\gamma(i,T) * \mathsf{AllTkts}^\gamma(M) \Rrightarrow (\forall t.\ M(t) = i \Leftrightarrow t \in T) \quad \text{(MyAllCoherence)}$$

$$\mathsf{MyTkts}^\gamma(i,T) * \mathsf{AllTkts}^\gamma(M) * t = |\mathrm{dom}(M)| \Rrightarrow \mathsf{MyTkts}^\gamma(i, T \uplus \{t\}) * \mathsf{AllTkts}^\gamma(M \uplus [t \mapsto i]) \quad \text{(GetTicket)}$$

$$\mathsf{true} \Rrightarrow \exists \gamma.\ \mathsf{AllTkts}^\gamma(\emptyset) * \left(\mathop{\Large *}_{i<\mathbf{C}} \mathsf{MyTkts}^\gamma(i,\emptyset)\right) * \mathsf{Perms}^\gamma_{\geq}(0) * \mathsf{UsedUP}^\gamma_{<}(0) \quad \text{(NewGhost)}$$

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 $\mathbf{C}$.

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

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

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

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

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

D.6.4 Verification of `newLock`

$\{P\}$

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

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

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

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

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

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

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

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

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

x

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

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

D.6.5 Verification of `lock`

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

D.6.6 Verification of `unlock`

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

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:

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

Verifying TSO programs

Bart Jacobs

Report CW 660, May 2014

KU Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Verifying TSO programs

Bart Jacobs

Report CW 660, May 2014

Department of Computer Science, KU Leuven

Abstract

TSO (Total Store Order) is the memory consistency model implemented by the x86 and x64 architectures. While for data-race-free programs the stronger SC (Sequential Consistency) memory consistency model can be assumed, some programs escape from the SC constraints for performance reasons. In this document we propose an approach for verifying programs under the TSO memory consistency model.

Verifying TSO programs

Bart Jacobs
iMinds-DistriNet, Dept. Comp. Sci., KU Leuven, Belgium
bart.jacobs@cs.kuleuven.be

Abstract

TSO (Total Store Order) is the memory consistency model implemented by the x86 and x64 architectures. While for data-race-free programs the stronger SC (Sequential Consistency) memory consistency model can be assumed, some programs escape from the SC constraints for performance reasons. In this document we propose an approach for verifying programs under the TSO memory consistency model.

1 Introduction

Memory Consistency. In the past, most verification approaches for concurrent programs have assumed that memory behaves as if all threads' memory accesses are executed in an interleaved fashion directly on a single global memory, such that there is a total order on all memory accesses and each read operation on a memory location yields the value of the most recent preceding write operation to that memory location in that total order. This memory consistency model is known as *sequential consistency* (SC).

However, real programming platforms (hardware architectures and compiled or interpreted programming languages) do not offer simple sequential consistency: these platforms' memory access primitives have weaker memory consistency semantics, in order to allow for important performance optimizations at the level of the memory hierarchy, the processor, and any compilation steps, such as caching, pipelining, and common subexpression elimination.

Still, most programming languages guarantee that programs for which all sequentially consistent executions are *data-race-free*, in some sense, have only sequentially consistent executions. Data-race-freedom usually means that all conflicting memory accesses are ordered by a *happens-before* relation induced by program order and by synchronization constructs such as mutual exclusion locks. As a result, most programmers can indeed safely assume sequential consistency and use the program verification techniques which assume sequential consistency and which verify data-race-freedom.

Nonetheless, in some cases it is necessary to consider non-data-race-free code, in order to avoid the performance overhead of synchronization. For such programs, reasoning must occur directly in terms of the weaker memory consistency semantics offered by the platform.

One of the most important programming platforms today is the x86/x64 family of processor architectures. The memory consistency model offered by these architectures is the x86-TSO (Total Store Order) memory model. In

the TSO memory model, each thread has a *write buffer*. Write operations by a thread are enqueued at the end of its write buffer; the memory subsystem decides arbitrarily when to dequeue a write operation from the front of a thread's write buffer and apply it to main memory. Main memory itself is sequentially consistent. Read operations by a thread are satisfied from the thread's write buffer, if possible, or else from main memory.

The Approach. In this document, we propose a verification approach for programs that use memory accesses with TSO semantics. To motivate and illustrate the approach, we will use the running example of a Java virtual machine implementation for a TSO platform. Such an implementation must implement Java field accesses efficiently. Assume it implements them as plain TSO read and write operations. A challenge is the issue of object initialization: threads may allocate Java objects concurrently, and leak references to those objects into shared fields without any synchronization. A thread that accesses an object expects the object to contain a valid pointer to a virtual method dispatch table (*vtable*), even if the object was obtained through a race. On a TSO platform, this is easy to achieve by making sure that an object's vtable pointer is initialized before references to the object are stored into fields of existing objects in the heap. The FIFO nature of the write buffers then ensures that if a thread sees an object reference, it also sees the properly initialized vtable pointer.

Notice that this program is not data-race-free, and therefore program verification approaches that verify data-race-freedom are not applicable. Instead, we propose an extension of separation logic with *TSO spaces*. TSO spaces are similar to the shared resources of Concurrent Separation Logic (CSL), except that memory owned by a TSO space may be accessed through TSO operations (memory accesses with TSO semantics) rather than classical critical sections. Also, where CSL associates a resource invariant with each resource, we associate an *abstract state space* with each TSO space, as well as an *abstraction predicate* that associates each abstract state with a corresponding separation logic assertion. The abstract state space is equipped with an *abstract reachability* pre-order $\preceq$.

Knowledge about the state of a TSO space in the proof of a thread is represented as an abstract state, representing the thread's view of the state of the TSO space. A thread's view is a lower bound (under the abstract reachability pre-order) on the actual state of the TSO space.

Each TSO write operation is associated with an *abstract state transition function*, mapping an abstract pre-state to an abstract post-state. These abstract state transition functions must respect the abstract reachability relation and properly abstract the concrete behavior of the TSO operation. Also, crucially, to account for TSO's relaxed behavior, the abstract state transition functions must be *monotonic*: they must respect reachability and be sound with respect to concrete behavior not only in the current abstract state, but also in all reachable future abstract states. In a thread's proof, when the thread performs a TSO write operation, its local view of the TSO space is updated per the abstract state transition function.

Each TSO read operation is associated with a function f mapping result values to new abstract states. This function must satisfy the property that for any result value v, and for any future abstract state α', if the target location

may have value v in this abstract state, then $f(v) \preceq \alpha'$. In a thread's proof, when the thread performs a TSO read operation, its local view of the TSO space is replaced with the (hopefully more precise) lower bound given by function f.

Verifying the Example. For example, to verify the Java virtual machine implementation using the proposed logic, we put the heap in a TSO space. As the abstract state space, we adopt the powerset of addresses in this heap, each abstract state representing the set of the addresses of the currently allocated and initialized objects. The subset relation serves as the reachability order. The abstraction predicate states 1) that allocated objects occupy disjoint heap space, 2) that they properly point to an existing virtual method dispatch table, and 3) that their fields point to allocated objects. Each thread is aware of the addresses of objects it allocated itself, as well as addresses read from fields of known objects. That is, reading a field updates the thread's view by inserting the newly discovered object address into the abstract state. After a thread allocates an object and initializes its run-time type information, it performs a no-op operation on the TSO space to update its local view of the set of allocated objects, inserting the newly allocated object into the abstract state. Writing the value of a local variable to a field corresponds to the identity function at the abstract level, since all object references a thread holds in local variables are already in the thread's local view.

The remainder of this document is structured as follows. In Section 2, we present the basic idea of the approach in the context of a simplified programming language without pointers. In Sec 3, we extend the programming language with locked instructions. In Sections 5 and 6 we integrate our approach into separation logic. In Section 7 we report on a preliminary encoding of the approach into the logic of the VeriFast program verifier. We offer a conclusion in Section 8.

2 The Basic Idea

2.1 Program Syntax

We consider a simple programming language with threads and shared global variables $g \in G$. There are no local variables.

$$\begin{array}{rcl} \mathit{Heaps} & = & G \to \mathbb{Z} \\ \delta \in \Delta & = & \mathit{Heaps} \to \mathit{Heaps} \\ c \in \mathit{Cmds} & ::= & \delta; c \mid c(g) \mid \textbf{done} \mid \textbf{fail} \\ \mathit{prog} & ::= & c \mid\mid \cdots \mid\mid c \end{array}$$

The heaps $h \in \mathit{Heaps}$ are the maps from variable names to values. A *command* is either an *update* δ (a function from heaps to heaps) followed by another command; or a *read operation* $c(g)$ consisting of the variable g to be read, and a function $c(\cdot)$ from values to commands; or the operation **done** indicating the end of the thread; or the operation **fail** indicating a failure. A program is a parallel composition of commands.

We consider generic updates instead of writes of individual variables, to allow for the instrumentation of writes with *ghost updates*.

2.2 Small-Step Semantics

The semantics of the programming language is defined as a small-step relation $\rightsquigarrow$ between machine configurations $\gamma \in \mathsf{Configs}$. A machine configuration consists of a heap and a multiset of thread configurations $\theta \in \mathit{ThreadConfigs}$. (A multiset over elements of a set X can be modeled as a function that maps each $x \in X$ to the number of times it occurs in the multiset.) A thread configuration consists of a write buffer (a sequence of updates) and a command.

$$\begin{array}{rcl} \mathit{Buffers} & = & \Delta^* \\ \theta \in \mathit{ThreadConfigs} & = & \mathit{Buffers} \times \mathit{Cmds} \\ \gamma \in \mathit{Configs} & = & \mathit{Heaps} \times (\mathit{ThreadConfigs} \rightarrow_{\text{fin}} \mathbb{N}) \\ \rightsquigarrow & \subseteq & \mathit{Configs} \times \mathit{Configs} \end{array}$$

There are only three kinds of steps: an Enqueue step enqueues an update; a Read step reduces a read operation; and a Dequeue step applies an update to the heap.

$$\textsc{Enqueue}\quad \frac{}{(h, \{\!|(\overline{\delta}, \delta; c)|\!\} \uplus \Theta) \rightsquigarrow (h, \{\!|(\overline{\delta} \cdot \delta, c)|\!\} \uplus \Theta)}$$

$$\textsc{Read}\quad \frac{}{(h, \{\!|(\overline{\delta}, c(g))|\!\} \uplus \Theta) \rightsquigarrow (h, \{\!|(\overline{\delta}, c(\overline{\delta}(h)(g)))|\!\} \uplus \Theta)}$$

$$\textsc{Dequeue}\quad \frac{}{(h, \{\!|(\delta \cdot \overline{\delta}, c)|\!\} \uplus \Theta) \rightsquigarrow (\delta(h), \{\!|(\overline{\delta}, c)|\!\} \uplus \Theta)}$$

Definition 1 (Failing Configuration)**.** *We say a machine configuration γ is* failing*, denoted $\gamma \in \mathit{Fail}$, if there exists $h, \overline{\delta}, \Theta$ such that $\gamma = (h, \{\!|(\overline{\delta}, \mathbf{fail})|\!\} \uplus \Theta)$.*

Definition 2 (Program Safety)**.** *A program $c_1 \mid\mid \cdots \mid\mid c_n$ is* safe *when started from an initial heap h_0 if $\forall \gamma'.\ (h, \{\!|(\epsilon, c_1)|\!\} \uplus \cdots \uplus \{\!|(\epsilon, c_n)|\!\}) \rightsquigarrow^* \gamma' \Rightarrow \gamma' \notin \mathit{Fail}$.*

2.3 Proof System

We assume a set $A \subseteq 2^{\mathit{Heap}}$ of heap predicates, *the abstract state space*, to be chosen by the proof developer. We use α to range over A. We assume an *abstract reachability pre-order* $\preceq$ on A.

We assume a partitioning of the variables $g \in G$ into the shared variables G_{s} and the *owned variables* of thread i G_i, for each i. We assume that only thread i updates the variables owned by thread i; however, reading is not restricted. A heap predicate $L \subseteq \mathit{Heaps}$ is *local to thread i* if it can be invalidated only by updates to variables owned by thread i: $\mathit{Local}_i = \{L \mid \forall h, h'.\ h \in L \wedge h|_{G_i} = h'|_{G_i} \Rightarrow h' \in L\}$.

We define a validity condition on commands, $\mathsf{valid}_i(\alpha, L, c)$, where $\alpha \in A$ is thread i's local view of the heap (including the shared variables), $L \in \mathit{Local}_i$ is information about thread i's owned variables, and $c \in \mathit{Cmds}$ is the command being executed by the thread.

$$
\begin{array}{l}
\mathsf{valid}_i(\alpha, L, \delta; c) = \\
\quad \exists f \in A \to A, L' \in \mathit{Local}_i. \\
\qquad (\forall \alpha' \succeq \alpha.\ f(\alpha') \succeq \alpha') \wedge \\
\qquad (\forall \alpha' \succeq \alpha, \alpha'' \succeq \alpha'.\ f(\alpha'') \succeq f(\alpha')) \wedge \\
\qquad (\forall \alpha' \succeq \alpha, h \in \alpha' \cap L.\ \delta(h) \in f(\alpha') \cap L') \wedge \\
\qquad \mathsf{valid}_i(f(\alpha), L', c) \\
\mathsf{valid}_i(\alpha, L, c(g)) = \\
\quad \forall v.\ \exists \alpha'. \\
\qquad (\forall \alpha'' \succeq \alpha, h \in \alpha'' \cap L.\ h(g) = v \Rightarrow \alpha'' \succeq \alpha') \wedge \\
\qquad \mathsf{valid}_i(\alpha', L, c(v)) \\
\mathsf{valid}_i(\alpha, L, \mathbf{done}) = \mathsf{True} \\
\mathsf{valid}_i(\alpha, L, \mathbf{fail}) = \forall \alpha' \succeq \alpha, h \in \alpha' \cap L.\ \mathsf{False}
\end{array}
$$

An update is valid if there exists an abstract version f of the update and a local postcondition L' such that in all abstract states α' reachable from the thread's view α, f respects abstract reachability (first conjunct), f is monotonic (second conjunct), and f soundly abstracts the update (third conjunct) given the local precondition L and the local postcondition L'. Furthermore, the continuation must be valid under the post-abstract state $f(\alpha)$ and the local postcondition L'.

As will become clear from the soundness proof, the monotonicity requirement ensures that the current thread's proof outline remains valid when updates of other threads are dequeued while the current thread's write buffer is nonempty.

A read operation is valid if for each possible result of the read operation, there exists a new view α' such that all abstract states α'' reachable from the current view where the result is possible (given local information L) are reachable from α', and furthermore the continuation is valid under this updated view α'.

A **done** operation is always valid. A **fail** operation is valid provided no abstract state reachable from the current view is feasible.

Definition 3 (Program Validity)**.** *We say a program $c_1 \mid\mid \cdots \mid\mid c_n$, started from initial heap h_0, is* valid *if there exists an initial abstract state $\alpha \in A$ and local preconditions $L_i \in \mathit{Local}_i$ such that $h \in \alpha \cap L_1 \cap \cdots \cap L_n$ and $\mathsf{valid}_i(\alpha, L_i, c_i)$, for all i.*

2.4 Soundness

The target soundness property is that valid programs are safe. In the remainder of this subsection, we sketch a proof of this property.

We define validity of a thread configuration $\mathsf{valid\_tcfg}_i(\alpha, L, \theta)$ as follows:

$$\mathsf{valid\_tcfg}_i(\alpha, L, (\delta_1 \cdots \delta_n, c)) = \mathsf{valid}_i(\alpha, L, \delta_1; \cdots; \delta_n; c)$$

We define validity of a machine configuration $\mathsf{valid\_cfg}(\gamma)$ as follows:

$$
\begin{array}{l}
\mathsf{valid\_cfg}((h, \{\!|\theta_1, \ldots, \theta_n|\!\})) = \\
\quad \exists \alpha, L_1, \ldots, L_n.\ h \in \alpha \cap L_1 \cap \cdots \cap L_n \wedge \forall i.\ \mathsf{valid\_tcfg}_i(\alpha, L_i, \theta_i)
\end{array}
$$

where $L_i \in \mathit{Local}_i$.

Lemma 1. *If $\mathsf{valid}_i(\alpha, L, c)$ and $\alpha' \succeq \alpha$ then $\mathsf{valid}_i(\alpha', L, c)$.*

Proof. By induction on c. □

Lemma 2. *If* $\mathsf{valid\_cfg}(\gamma)$ *and* $\gamma \rightsquigarrow \gamma'$*, then* $\mathsf{valid\_cfg}(\gamma')$.

Proof. By case analysis on the step rule.

- **Case** ENQUEUE. Trivial.
- **Case** READ. Assume $\gamma = (h, \{\!\{(\delta_1 \cdot \dots \cdot \delta_m, c(g))\}\!\} \uplus \Theta)$ and $h'(g) = v$ with $h' = (\delta_m \circ \dots \circ \delta_1)(h)(g) = v$ and $\gamma' = (h, \{\!\{(\delta_1 \cdot \dots \cdot \delta_m, c(v))\}\!\} \uplus \Theta)$. By the first premise, there exists an α and an L_i^0 such that $h \in \alpha \cap L_i^0$ and $\mathsf{valid}_i(\alpha, L_i^0, \delta_1; \dots; \delta_m; c(g))$. By definition of valid_i, there exist f_j and L_i^j, for $j \in \{1, \dots, m\}$, such that $\mathsf{valid}_i(\alpha'', L_i^j, c(g))$ with $\alpha'' = (f_m \circ \dots \circ f_1)(\alpha)$. Note that $\alpha'' \succeq \alpha$ (by the first conjunct of validity of updates) and $h' \in \alpha''$ (by the third conjunct of validity of updates) and $h''(g) = v$. Therefore, by validity of reads, there exists an $\alpha' \preceq \alpha''$ such that $\mathsf{valid}_i(\alpha', L_i^m, c(v))$, and thus, by Lemma 1, $\mathsf{valid}_i(\alpha'', L_i^m, c(v))$.
- **Case** DEQUEUE. We need to prove three things: that the new heap satisfies the new abstract state and all local preconditions; that the new configuration of the thread whose write was dequeued is valid; and that all other threads' configurations remain valid. First of all: we know $h \in \alpha \cap L_i$ and therefore $\delta(h) \in f(\alpha) \cap L_i'$. It follows directly that the new configuration of the active thread is valid. The other threads' local preconditions are preserved, by their locality. The other threads' configurations remain valid by Lemma 1.

□

Lemma 3. *If* $\mathsf{valid\_cfg}(\gamma)$ *then* $\gamma \notin \mathit{Fail}$.

Proof. Assume $h \in \alpha \cap L_1 \cap \dots \cap L_n \wedge \forall i.\ \mathsf{valid\_tcfg}_i(\alpha, L_i, \theta_i)$. Assume $\theta_i = (\overline{\delta}, \mathbf{fail})$ for some i. Then by $\mathsf{valid\_tcfg}_i(\alpha, L_i, \theta_i)$ we have $\forall \alpha' \succeq \alpha, h \in \alpha' \cap L_i.\ \mathsf{False}$. By taking $\alpha' = \alpha$ we obtain a contradiction. □

2.5 Examples

Virtual Machine. We encode a simplified version of the virtual machine example into our formal programming language.

Firstly, we encode addressable memory into the language by choosing an indexed set of variables: let $G = \{\mathsf{m}_0, \dots, \mathsf{m}_{9999}\}$.

We consider "objects" consisting just of a single field; we do not consider vtable pointers in this example. Each field of each allocated object must point to an allocated object. In the initial heap, there is a single object at address 0, and its field points to itself. All other memory locations hold the value -1, which is not an address: $h_0 = \{\mathsf{m}_0 \mapsto 0, \mathsf{m}_{1 \dots 9999} \mapsto -1\}$.

The program consists of two threads. The first thread initializes an object at location 1, by storing a reference to itself into its field, and then publishes this object by storing a reference to the object into the field of the initially allocated object at location 0. The second thread reads the field of the initial object and asserts that the value is an address. It then reads the field of the object at this address and asserts that the resulting value is again an address.

$c_1 = \langle \mathsf{m}_1 := 1 \rangle; \langle \mathsf{m}_0 := 1 \rangle; \mathbf{done}$
$c_2 = \ell \leftarrow \mathsf{m}_0; \mathsf{assert}(0 \leq \ell \leq 9999); \ell' \leftarrow \mathsf{m}_\ell; \mathsf{assert}(0 \leq \ell' \leq 9999); \mathbf{done}$

Notice that this program relies on the FIFO nature of the write buffers in the TSO memory model.

We can verify this program in the approach of this section as follows. As the abstract state space, we take the predicates α_Λ where Λ is a set of integers, and α_Λ is satisfied by a heap h if each value in Λ is an address that is mapped by h to a value in Λ:

$$A = \{\alpha_\Lambda \mid \alpha_\Lambda = \alpha_\Lambda = \{h \mid \forall \ell \in \Lambda.\ \ell \in \mathit{Addresses} \wedge h(\mathsf{m}_\ell) \in \Lambda\}\}$$

where $\mathit{Addresses} = \{0, \ldots, 9999\}$.

Abstract reachability corresponds to the subset relation on the indices: $\alpha_\Lambda \preceq \alpha_{\Lambda'} \Leftrightarrow \Lambda \subseteq \Lambda'$.

In this proof, we do not use the local predicates. That is, we take $L = \mathit{Heaps}$ for all local preconditions and postconditions L.

Our initial abstract state is $\alpha_{\{0\}}$.

The proof outline for c_1 is as follows:[1]

$\alpha_{\{0\}}$
$\langle \mathsf{m}_1 := 1 \rangle;$ $\quad f(\alpha_\Lambda) = \alpha_{\Lambda \cup \{1\}}$
$\alpha_{\{0,1\}}$
$\langle \mathsf{m}_0 := 1 \rangle;$ $\quad f(\alpha_\Lambda) = \alpha_\Lambda$
$\alpha_{\{0,1\}}$
done

That is, the first update is associated with a function that adds the value 1 to Λ. The second update is associated with the identity function. It's easy to see that this proof outline is valid.

The proof outline for c_2 is as follows. (Notice that $\Lambda \not\subseteq \mathit{Addresses} \Rightarrow \alpha_\Lambda = \emptyset$; i.e., if an abstract state's index contains values that are not addresses, the abstract state is infeasible.)

$\alpha_{\{0\}}$
$\ell \leftarrow \mathsf{m}_0;$ $\quad \alpha' = \alpha_{\{0,\ell\}}$
$\alpha_{\{0,\ell\}}$
$\mathsf{assert}(0 \leq \ell \leq 9999);$
$\alpha_{\{0,\ell\}}$
$\ell' \leftarrow \mathsf{m}_\ell;$ $\quad \alpha' = \alpha_{\{0,\ell,\ell'\}}$
$\alpha_{\{0,\ell,\ell'\}}$
$\mathsf{assert}(0 \leq \ell' \leq 9999);$
$\alpha_{\{0,\ell,\ell'\}}$
done

[1] In these proof outlines, we specify the abstract state before and after each program command; furthermore, we annotate each update with abstract version (the f function), and each read operation with its new abstract state α' (which may depend on the result of the read operation).

Producer-Consumer. For this example, consider two threads that communicate via a single shared variable b. This variable (the *buffer*) is either *empty*, if its value is 0, or *full* otherwise. The producer thread puts the integers from k down to 1 into the buffer. Putting a value into the buffer means writing it into the buffer and then waiting for the buffer to become empty. The consumer thread repeatedly takes a value from the buffer and asserts that it is the next lower integer. Taking a value from the buffer means reading the buffer until a nonzero value is read, and then writing zero.

$$\mathsf{prod}_0 = \textbf{done}$$
$$\mathsf{prod}_{k+1} = \langle \mathsf{b} := k+1 \rangle; v \leftarrow \mathsf{b}; \textbf{if } v = 0 \textbf{ then } \mathsf{prod}_k \textbf{ else done}$$
$$\mathsf{cons}_0 = \textbf{done}$$
$$\mathsf{cons}_{k+1} = v \leftarrow \mathsf{b}; \textbf{if } v = 0 \textbf{ then done else } (\textbf{assert } v = k+1; \langle \mathsf{b} := 0 \rangle; \mathsf{cons}_k)$$

Our verification goal is to prove that $\forall k.\ \mathsf{prod}_k \mid\mid \mathsf{cons}_k$ is safe when started in heap $\{\mathsf{b} \mapsto 0\}$.

We introduce two ghost variables: p and c, both initially equal to $k+1$. p is owned by the producer, and c is owned by the consumer.

We instrument the program with ghost updates:

$$\mathsf{prod'}_0 = \textbf{done}$$
$$\mathsf{prod'}_{k+1} = \langle \mathsf{b} := k+1; \mathsf{p} := k+1 \rangle; v \leftarrow \mathsf{b}; \textbf{if } v = 0 \textbf{ then } \mathsf{prod'}_k \textbf{ else done}$$
$$\mathsf{cons'}_0 = \textbf{done}$$
$$\mathsf{cons'}_{k+1} =$$
$$\quad v \leftarrow \mathsf{b};$$
$$\quad \textbf{if } v = 0 \textbf{ then done else } (\textbf{assert } v = k+1; \langle \mathsf{b} := 0; \mathsf{c} := k+1 \rangle; \mathsf{cons'}_k)$$

For the proof, we define the abstract state space as $A = \{\alpha_{p,c} \mid \alpha_{p,c} = \{\{\mathsf{p} \mapsto p, \mathsf{c} \mapsto c, \mathsf{b} \mapsto b \mid p = c \wedge b = 0 \vee p = c - 1 \wedge b = p\}\}\}$. That is, each abstract state $\alpha_{p,c}$ is either a singleton (if $p \le c \le p+1$) or the empty set (otherwise).

We define the abstract reachability relation $\alpha_{p,c} \preceq \alpha_{p',c'}$ as $p > p' \vee p = p' \wedge c \ge c'$.

The initial abstract state is $\alpha_{k+1,k+1}$.

We use the local predicates $P_k = \{h \mid h(\mathsf{p}) = k\}$ and $C_k = \{h \mid h(\mathsf{c}) = h\}$.

We prove by induction on k that $\mathsf{valid}_1(\alpha_{k+1,k+1}, P_{k+1}, \mathsf{prod}_k)$. Case $k = 0$ is trivial. Assume $k > 0$.

$$\begin{array}{l}
\alpha_{k+1,k+1}, P_{k+1} \\
\langle \mathsf{b} := k; \mathsf{p} := k \rangle; \qquad f(\alpha_{p,c}) = \alpha_{p-1,c} \\
\alpha_{k,k+1}, P_k \\
v \leftarrow \mathsf{b}; \qquad \alpha' = (v = 0 \ ?\ \alpha_{k,k} \ :\ \alpha_{k,k+1}) \\
v = 0 \ ?\ \alpha_{k,k}, P_k \ :\ \alpha_{k,k+1}, P_k \\
\textbf{if } v = 0 \textbf{ then } \mathsf{prod'}_{k-1} \textbf{ else done}
\end{array}$$

We prove by induction on k that $\mathsf{valid}_2(\alpha_{k+1,k+1}, C_{k+1}, \mathsf{cons}_k)$. Case $k = 0$

is trivial. Assume $k > 0$.

$\alpha_{k+1,k+1}, C_{k+1}$
$v \leftarrow \mathsf{b}; \qquad \alpha' = (v = 0 \ ? \ \alpha_{k+1,k+1} \ : \ \alpha_{k,k+1})$
$v = 0 \ ? \ \alpha_{k+1,k+1}, C_{k+1} \ : \ \alpha_{v,v+1}, C_{k+1}$
if $v = 0$ **then done else** (
  $\alpha_{v,v+1}, C_{k+1}$
  assert $v = k$;
  $\alpha_{k,k+1}, C_{k+1}$
  $\langle \mathsf{b} := 0; \mathsf{c} := k \rangle; \qquad f(\alpha_{p,c}) = \alpha_{p,c-1}$
  $\alpha_{k,k}, C_k$
  cons'_{k-1}
)

3 Locked Instructions

The x86 architecture supports *locked instructions*, i.e. machine instructions prefixed by the `LOCK` prefix. The semantics of the `LOCK` prefix is that during execution of the prefixed instruction, main memory is locked, so no other threads' write operations are dequeued from their write buffers; furthermore, after the instruction is finished but before the lock is released, the current thread's write buffer is flushed.

We extend our programming language with locked operations:

$$c ::= \cdots \mid \textbf{locked}\ \delta; c(\mathsf{heap})$$

A locked operation consists of an update δ and a continuation c parameterized by a heap (i.e. a function from heaps to commands).

We extend the program semantics with a single step rule:

LOCKED

$$\frac{}{(h, \{\!\{(\epsilon, \textbf{locked}\ \delta; c(\mathsf{heap}))\}\!\} \uplus \Theta) \rightsquigarrow (\delta(h), \{\!\{(\epsilon, c(h))\}\!\} \uplus \Theta)}$$

We define validity of a locked operation:

$$\begin{array}{l} \mathsf{valid}_i(\alpha, L, \textbf{locked}\ \delta; c(\mathsf{heap})) = \\ \quad \forall \alpha' \succeq \alpha, h \in \alpha' \cap L.\ \exists \alpha'' \succeq \alpha', L' \in \mathit{Local}_i. \\ \quad\quad \delta(h) \in \alpha'' \cap L' \wedge \mathsf{valid}_i(\alpha'', L', c(h)) \end{array}$$

We extend the soundness proof.

Theorem 1. *The extended system is sound.*

Proof. Lemma 1 is preserved trivially. For Lemma 2, the proof is analogous to case DEQUEUE. □

Example: Mutex. Consider a program that protects a shared resource using a mutual exclusion lock implemented using a Compare-And-Set (CAS) instruction. The mutex is implemented as a shared variable m whose value is zero when the mutex is not held, and one when the mutex is held. The shared resource is a shared variable r whose value should always be 0 except during an operation on the resource.

Each thread attempts to acquire the resource by performing a CAS instruction on m, attempting to set it from 0 to 1. If successful, it asserts that r equals 0, then sets it to 1, then resets it to 0, and finally releases the mutex by setting m to 0.

The command executed by each thread is as follows:

$$
\begin{array}{l}
\mathsf{thread}_i = \\
\quad h \leftarrow \textbf{locked}\ \langle \textbf{if}\ \mathsf{m} = 0\ \textbf{then}\ \mathsf{m} := 1 \rangle; \\
\quad \textbf{assume}\ h(\mathsf{m}) = 0; \\
\quad r \leftarrow \mathsf{r}; \textbf{assert}\ r = 0; \\
\quad \langle \mathsf{r} := 1 \rangle; \langle \mathsf{r} := 0 \rangle; \\
\quad \langle \mathsf{m} := 0 \rangle
\end{array}
$$

The initial heap h_0 is $\{\mathsf{m} \mapsto 0, \mathsf{r} \mapsto 0\}$.

We introduce one shared ghost variable u_0, and one ghost variable u_i for each thread i, owned by thread i, with $i > 0$. Initially u_0 has value 1 and all other u_i have value 0.

We instrument the program with ghost updates as follows:

$$
\begin{array}{l}
\mathsf{thread'}_i = \\
\quad h \leftarrow \textbf{locked}\ \langle \textbf{if}\ \mathsf{m} = 0\ \textbf{then}\ (\mathsf{m} := 1; \mathsf{u}_i, \mathsf{u}_0 := \mathsf{u}_0, 0) \rangle; \\
\quad \textbf{assume}\ h(\mathsf{m}) = 0; \\
\quad r \leftarrow \mathsf{r}; \textbf{assert}\ r = 0; \\
\quad \langle \mathsf{r} := 1; \mathsf{u}_i{+}{+} \rangle; \langle \mathsf{r} := 0; \mathsf{u}_i{+}{+} \rangle; \\
\quad \langle \mathsf{m} := 0; \mathsf{u}_0, \mathsf{u}_i := \mathsf{u}_i, 0 \rangle
\end{array}
$$

We take as the abstract state space the set $A = \{\alpha_{u,r} \mid \alpha_{u,r} = \{h \mid h(\mathsf{r}) = r \wedge \exists i.\ (i \neq 0 \wedge h(\mathsf{m}) = 1 \vee i = 0 \wedge r = 0 \wedge h(\mathsf{m}) = 0) \wedge \forall j.\ j = i \wedge h(\mathsf{u}_j) = u \vee j \neq i \wedge h(\mathsf{u}_j) = 0\}\}$. In each abstract state, the current update count u is stored either in u_0, indicating that the mutex is not held by any thread, or in u_i with $i > 0$, indicating that the mutex is held by thread i.

We define the abstract reachability relation $\alpha_{u,r} \preceq \alpha_{u',r'}$ as $u < u' \vee (u, r) = (u', r')$.

We define local predicates $L_{i,u} = \{h \mid h(\mathsf{u}_i) = u\}$ and $L_\top = \mathit{Heaps}$.

The initial abstract state is $\alpha_{1,0}$.

Proof outline:

$$
\begin{array}{l}
\alpha_{1,0}, L_\top \\
h \leftarrow \textbf{locked}\ \langle \textbf{if}\ \mathsf{m} = 0\ \textbf{then}\ (\mathsf{m} := 1; \mathsf{u}_i, \mathsf{u}_0 := \mathsf{u}_0, 0) \rangle; \\
h(\mathsf{m}) = 0\ ?\ \alpha_{h(\mathsf{u}_0),0}, L_{i,h(\mathsf{u}_0)}\ :\ \alpha_{1,0} \\
\textbf{assume}\ h(\mathsf{m}) = 0; \\
\alpha_{h(\mathsf{u}_0),0}, L_{i,h(\mathsf{u}_0)} \\
r \leftarrow \mathsf{r}; \qquad \alpha' = (r = 0\ ?\ \alpha_{h(\mathsf{u}_0),0}\ :\ \alpha_{h(\mathsf{u}_0)+1,r}) \\
\textbf{assert}\ r = 0; \\
\alpha_{h(\mathsf{u}_0),0}, L_{i,h(\mathsf{u}_0)} \\
\langle \mathsf{r} := 1; \mathsf{u}_i{+}{+} \rangle; \qquad f(\alpha_{u,r}) = \alpha_{u+1,1} \\
\alpha_{h(\mathsf{u}_0)+1,1}, L_{i,h(\mathsf{u}_0)+1} \\
\langle \mathsf{r} := 0; \mathsf{u}_i{+}{+} \rangle; \qquad f(\alpha_{u,r}) = \alpha_{u+1,0} \\
\alpha_{h(\mathsf{u}_0)+2,0}, L_{i,h(\mathsf{u}_0)+2} \\
\langle \mathsf{m} := 0; \mathsf{u}_0, \mathsf{u}_i := \mathsf{u}_i, 0 \rangle \qquad f(\alpha_{u,r}) = \alpha_{u,r} \\
\alpha_{h(\mathsf{u}_0)+2,0}, L_\top
\end{array}
$$

4 Marriage of TSO and Separation Logic

In the next two sections, we will marry our approach with ownership and the assertion logic of separation logic on the one hand, and with the Hoare logic-style proofs approach on the other hand.

5 Marriage of TSO, Ownership, and Separation Assertions

In the preceding sections, we assumed a static partitioning of the set of variables into shared variables and variables owned by particular threads. In this section, we adopt a more dynamic approach by considering *fractional heaps*. The fractional heaps $H \in \mathit{FracHeaps}$ are defined as follows:

$$\mathit{FracHeaps} = G \rightharpoonup \mathbb{Z} \times (0, 1]$$

For $H, H_1, H_2 \in \mathit{FracHeaps}$, we say $H = H_1 \bullet H_2$ iff

$$\forall g, v.\ H(g, v) = H_1(g, v) + H_2(g, v)$$

where

$$H(g, v) = \begin{cases} \pi & \text{if } H(g) = (v, \pi) \\ 0 & \text{if } \nexists \pi.\ H(g) = (v, \pi) \end{cases}$$

We reinterpret updates of the programming language as follows:

$$\delta(H) = H' \Leftrightarrow \forall H_0, h.\ H \bullet H_0 = h \Rightarrow \delta(h) = H' \bullet H_0$$

We now pick both the abstract states and the local predicates from the powerset of fractional heaps: $A \subseteq 2^{\mathit{FracHeaps}}, L_i \subseteq \mathit{FracHeaps}$.

Validity of commands is updated as follows:

$$\begin{array}{l}
\mathsf{valid}(\alpha, L, \delta; c) = \\
\quad \exists f \in A \to A, L' \subseteq \mathit{FracHeaps}. \\
\qquad (\forall \alpha' \succeq \alpha.\ f(\alpha') \succeq \alpha') \wedge \\
\qquad (\forall \alpha' \succeq \alpha, \alpha'' \succeq \alpha'.\ f(\alpha'') \succeq f(\alpha')) \wedge \\
\qquad (\forall \alpha' \succeq \alpha, H \in \alpha' \bullet L.\ \delta(H) \in f(\alpha') \bullet L') \wedge \\
\qquad \mathsf{valid}(f(\alpha), L', c) \\
\mathsf{valid}(\alpha, L, c(g)) = \\
\quad (\forall v, H \in L.\ g \in \mathrm{dom}(H) \wedge (H(g) = (v, _) \Rightarrow \mathsf{valid}(\alpha, L, c(v)))) \vee \\
\quad \forall v.\ \exists \alpha'. \\
\qquad \forall \alpha'' \succeq \alpha, H \in \alpha'' \bullet L.\ g \in \mathrm{dom}(H) \wedge (H(g) = (v, _) \Rightarrow \alpha'' \succeq \alpha') \wedge \\
\qquad \mathsf{valid}(\alpha', L, c(v)) \\
\mathsf{valid}(\alpha, L, \mathbf{done}) = \mathsf{True} \\
\mathsf{valid}(\alpha, L, \mathbf{fail}) = \forall \alpha' \succeq \alpha, h \in \alpha' \bullet L.\ \mathsf{False} \\
\mathsf{valid}(\alpha, L, \mathbf{locked}\ \delta; c(\mathsf{heap})) = \\
\quad \forall \alpha' \succeq \alpha, H \in \alpha' \bullet L.\ \exists \alpha'' \succeq \alpha', L'. \\
\qquad \delta(H) \in \alpha'' \bullet L' \wedge \forall H_0, h.\ h = H \bullet H_0 \Rightarrow \mathsf{valid}(\alpha'', L', c(h))
\end{array}$$

Notice that mostly we simply replaced occurrences of $\alpha \cap L$ by $\alpha \bullet L$ to reflect the fact that α and L are now predicates over fractional heaps. We also added

an alternative clause for validity of read operations, allowing for slightly simpler proofs in the case where certain result values can be excluded based on local knowledge and no refinement of the abstract state is required.

Validity of a configuration is now defined as:

$$\begin{aligned}&\mathsf{valid\_cfg}((h, \{\!|\theta_1, \ldots, \theta_n|\!\})) = \\ &\quad \exists \alpha, L_1, \ldots, L_n, h \in \alpha \bullet L_1 \bullet \cdots \bullet L_n.\ \forall i.\ \mathsf{valid}(\alpha, L_i, \theta_i)\end{aligned}$$

where $P \bullet Q = \{H \mid \exists H_1 \in P, H_2 \in Q.\ H = H_1 \bullet H_2\}$.

Theorem 2. *This approach is sound.*

Proof. Monotonicity is preserved. Preservation of configuration validity under the step relation is entirely analogous; the only difference now is that, when an update of thread i is dequeued, preservation of L_j for $j \neq i$ is argued based on the fact that $\delta(H)$ is well-defined and therefore $\delta(H \bullet H_{\mathrm{L}}) = \delta(H) \bullet H_{\mathrm{L}}$. The case where the step is a read operation and the new clause for validity of read operations is used, is discharged easily. □

Example. Exploiting separation logic, we can simplify the proof of the mutex example of the preceding section.

We recall that the command executed by each thread is as follows:

$$\begin{aligned}&\mathsf{thread}_i = \\ &\quad h \leftarrow \textbf{locked}\ \langle \textbf{if}\ \mathsf{m} = 0\ \textbf{then}\ \mathsf{m} := 1\rangle; \\ &\quad \textbf{assume}\ h(\mathsf{m}) = 0; \\ &\quad r \leftarrow \mathsf{r}; \textbf{assert}\ r = 0; \\ &\quad \langle \mathsf{r} := 1\rangle; \langle \mathsf{r} := 0\rangle; \\ &\quad \langle \mathsf{m} := 0\rangle\end{aligned}$$

For this version of the proof, we do not need to introduce any ghost variables. Indeed, in this version the *permissions* play the corresponding role.

The abstract state space is a singleton: $A = \{\alpha\}$; $\alpha = \{H \mid H \vDash \mathsf{m} \mapsto 0 * \mathsf{r} \mapsto 0 \vee \mathsf{m} \mapsto 1\}$.

For the local predicates, we use separation logic assertions.

Proof outline:

$$\begin{aligned}&\alpha, \textbf{emp} \\ &h \leftarrow \textbf{locked}\ \langle \textbf{if}\ \mathsf{m} = 0\ \textbf{then}\ \mathsf{m} := 1\rangle; \\ &h(\mathsf{m}) = 0\ ?\ \alpha, \mathsf{r} \mapsto 0\ :\ \alpha, \textbf{emp} \\ &\textbf{assume}\ h(\mathsf{m}) = 0; \\ &\alpha, \mathsf{r} \mapsto 0 \\ &r \leftarrow \mathsf{r}; \\ &\alpha, \mathsf{r} \mapsto 0 \wedge r = 0 \\ &\textbf{assert}\ r = 0; \\ &\alpha, \mathsf{r} \mapsto 0 \\ &\langle \mathsf{r} := 1\rangle; \qquad f(\alpha) = \alpha \\ &\alpha, \mathsf{r} \mapsto 1 \\ &\langle \mathsf{r} := 0\rangle; \qquad f(\alpha) = \alpha \\ &\alpha, \mathsf{r} \mapsto 0 \\ &\langle \mathsf{m} := 0\rangle \qquad f(\alpha) = \alpha \\ &\alpha, \textbf{emp}\end{aligned}$$

6 A Hoare Logic for TSO

In this section we adopt a more realistic programming language, and we define a Hoare logic for it, based on separation logic.

6.1 Syntax and Semantics of Programs

The syntax of the programming language is as follows:

$$\begin{array}{rl} e ::= & x \mid z \mid e + e \mid e - e \\ b ::= & e = e \mid e < e \mid \neg b \\ u ::= & x := [e] \mid [e] := e' \mid \textbf{if } b \textbf{ then } u \textbf{ else } u \mid u; u \\ c ::= & x := e \mid c; c \mid \textbf{if } b \textbf{ then } c \textbf{ else } c \mid \textbf{while } b \textbf{ do } c \\ & \mid x := [e] \mid \langle u \rangle \mid \textbf{locked } u \mid \textbf{fork } c \mid \textbf{fail} \end{array}$$

The semantics of updates u is as follows:

$$(h, s, x := [e]) \Downarrow (h, s[x := h(s(e))]) \qquad (h, s, [e] := e') \Downarrow (h[s(e) := s(e')], s)$$

$$\frac{s(b) \qquad (h, s, u) \Downarrow (h', s')}{(h, s, \textbf{if } b \textbf{ then } u \textbf{ else } u') \Downarrow (h', s')} \qquad \frac{\neg s(b) \qquad (h, s, u') \Downarrow (h', s')}{(h, s, \textbf{if } b \textbf{ then } u \textbf{ else } u') \Downarrow (h', s')}$$

$$\frac{(h, s, u) \Downarrow (h', s') \qquad (h', s', u') \Downarrow (h'', s'')}{(h, s, u; u') \Downarrow (h'', s'')}$$

We define $[\![u]\!](s)(h) = h' \Leftrightarrow \exists s'.\ (h, s, u) \Downarrow (h', s')$.
The small-step relation $\rightsquigarrow$ on machine configurations is defined in Figure 1.

6.2 Proof System

In this logic, we support multiple TSO regions, and we allow abstract states to talk about TSO regions. Let $\mathcal{A}$ be a set of *abstract state space names*. For each $A \in \mathcal{A}$, let $\mathsf{St}(A)$ be a set of *abstract state names*, with an abstract reachability order $\preceq_A$ defined on it.

Let $\mathcal{R}$ be a set of *TSO region names*. An *abstract superstate* $\tilde{\alpha} \in \tilde{A}$ is a finite partial function from $\mathcal{R}$ to $\bigcup_{A \in \mathcal{A}} \mathsf{St}(A)$. It specifies the set of allocated TSO region names, and their current abstract state. We define $\tilde{\alpha} \preceq \tilde{\alpha}'$ as $\forall (r, \alpha) \in \tilde{\alpha}.\ \exists \alpha' \succeq \alpha.\ (r, \alpha') \in \tilde{\alpha}'$. Let sat_A be a function from $\mathsf{St}(A)$ to *semantic assertions*, i.e. sets of pairs of fractional heaps and abstract superstates. We require that all semantic assertions X be *upward-closed*: $(H, \tilde{\alpha}) \in X \wedge \tilde{\alpha}' \succeq \tilde{\alpha} \Rightarrow (H, \tilde{\alpha}') \in X$.

The syntax of assertions P is separation logic with fractional permissions, plus the $\mathsf{tso}_r^A(\alpha)$ assertion which states that the TSO region r has been allocated with abstract state space A and its abstract state is at least α. tso assertions may appear only in positive positions. Assertion expressions E are like program expressions e, except that they may mention logical variables X.

$$\begin{array}{l} E ::= z \mid x \mid X \mid E + E \mid E - E \\ P ::= E = E \mid E < E \mid E \overset{\pi}{\mapsto} E \mid \exists X.\ P \mid P \Rightarrow P \mid P * P \mid \textbf{emp} \mid \mathsf{tso}_r^A(\alpha) \end{array}$$

$$\textsc{Assign}\quad (h, (\overline{\delta}, s, x := e; \kappa) \cdot \Theta) \rightsquigarrow (h, (\overline{\delta}, s[x := s(e)], \kappa) \cdot \Theta)$$

$$\textsc{Seq}\quad (h, (\overline{\delta}, s, (c; c'); \kappa) \cdot \Theta) \rightsquigarrow (h, (\overline{\delta}, s, c; c'; \kappa) \cdot \Theta)$$

$$\textsc{IfTrue}\quad \frac{s(b)}{(h, (\overline{\delta}, s, \textbf{if } b \textbf{ then } c \textbf{ else } c'; \kappa) \cdot \Theta) \rightsquigarrow (h, (\overline{\delta}, s, c; \kappa) \cdot \Theta)}$$

$$\textsc{IfFalse}\quad \frac{\neg s(b)}{(h, (\overline{\delta}, s, \textbf{if } b \textbf{ then } c \textbf{ else } c'; \kappa) \cdot \Theta) \rightsquigarrow (h, (\overline{\delta}, s, c'; \kappa) \cdot \Theta)}$$

$$\textsc{WhileTrue}\quad \frac{s(b)}{(h, (\overline{\delta}, s, \textbf{while } b \textbf{ do } c; \kappa) \cdot \Theta) \rightsquigarrow (h, (\overline{\delta}, s, c; \textbf{while } b \textbf{ do } c; \kappa) \cdot \Theta)}$$

$$\textsc{WhileFalse}\quad \frac{\neg s(b)}{(h, (\overline{\delta}, s, \textbf{while } b \textbf{ do } c; \kappa) \cdot \Theta) \rightsquigarrow (h, (\overline{\delta}, s, \kappa) \cdot \Theta)}$$

$$\textsc{Read}\quad (h, (\overline{\delta}, s, x := [e]; \kappa) \cdot \Theta) \rightsquigarrow (h, (\overline{\delta}, s[x := \overline{\delta}(h)(s(e))], \kappa) \cdot \Theta)$$

$$\textsc{Enqueue}\quad (h, (\overline{\delta}, s, \langle u \rangle; \kappa) \cdot \Theta) \rightsquigarrow (h, (\overline{\delta} \cdot [\![u]\!](s), s, \kappa) \cdot \Theta)$$

$$\textsc{Locked}\quad \frac{(h, s, u) \Downarrow (h', s')}{(h, (\epsilon, s, \textbf{locked } u; \kappa) \cdot \Theta) \rightsquigarrow (h', (\epsilon, s', \kappa) \cdot \Theta)}$$

$$\textsc{Fork}\quad (h, (\epsilon, s, \textbf{fork } c; \kappa) \cdot \Theta) \rightsquigarrow (h, (\epsilon, s, \kappa) \cdot (\epsilon, s, c; \textbf{done}) \cdot \Theta)$$

$$\textsc{Dequeue}\quad (h, (\delta \cdot \overline{\delta}, s, \kappa) \cdot \Theta) \rightsquigarrow (\delta(h), (\overline{\delta}, s, \kappa) \cdot \Theta)$$

Figure 1: Operational semantics of the realistic programming language

The proof rules are the standard rules of separation logic (where heap mutation is encoded as a simple update $\langle [e] := e' \rangle$), except that we add some rules and we remove some rules.

We add the following additional rules for updates and reads; these are useful for the case where insufficient permission is available locally, so a TSO space must be accessed.

UPDATE

$$\frac{\begin{array}{c}\forall \alpha' \succeq \alpha.\ f(\alpha) \succeq \alpha \qquad \forall \alpha' \succeq \alpha, \alpha'' \succeq \alpha'.\ f(\alpha'') \succeq f(\alpha') \\ \forall \alpha' \succeq \alpha, H, \tilde{\alpha}, s.\ H, \tilde{\alpha} \in L(s) \bullet \mathsf{sat}_A(\alpha') \Rightarrow u(H, s), \tilde{\alpha} \in L'(s) \bullet \mathsf{sat}_A(f(\alpha'))\end{array}}{\{L \wedge \mathsf{tso}_r^A(\alpha)\}\ \langle u \rangle\ \{L' \wedge \mathsf{tso}_r^A(f(\alpha))\}}$$

READ

$$\frac{\forall \alpha'' \succeq \alpha, s, H, \tilde{\alpha}.\ H, \tilde{\alpha} \in L(s) \bullet \mathsf{sat}_A(\alpha'') \Rightarrow s(e) \in \mathrm{dom}(H) \wedge \forall v.\ H(s(e)) = (v, \_) \Rightarrow \alpha'' \succeq \alpha'(v)}{\{L \wedge x = v_0 \wedge \mathsf{tso}_r^A(\alpha)\}\ x := [e]\ \{L[v_0/x] \wedge \mathsf{tso}_r^A(\alpha'(x))\}}$$

We remove the disjunction rule and, correspondingly, the rule for existential quantification. However, we include restricted versions of these rules that allow case splitting on the value of a local variable.

6.3 Unsoundness of the disjunction rule

Including these rules would be unsound, since they would allow associating different abstract operations and different local postconditions with an update and different lower bounds with a read operation depending on the value of a ghost variable. That would allow one to prove that the program $\langle [30] := 2 \rangle; \mathbf{fork}\ (\langle [10] := 1 \rangle; r_1 := [20]; \langle [30] := r_1 \rangle); \langle [20] := 1 \rangle; r_2 := [10]; r_3 := [30]; \mathbf{assert}\ \neg(r_3 = 0 \wedge r_2 = 0)$ (an encoding of the classic $((x := 1; r_1 := y)\ ||\ (y := 1; r_2 := x)); \mathbf{assert}\ \neg(r_1 = 0 \wedge r_2 = 0)$ program) is safe, which is false. Indeed, introduce a ghost variable [40] owned by the forked thread that records the value of [20] at the time of the update of [10], and a ghost variable [50] owned by the main thread that records the value of [10] at the time of the update of [20], both initially 2. We have two abstract states: a state α_1 that states the invariant $([40] \neq 2 \Rightarrow [10] = 1) \wedge ([50] \neq 2 \Rightarrow [20] = 1) \wedge \neg([40] = 0 \wedge [50] = 0) \wedge ([30] = 0 \Rightarrow [40] = 0)$, and a state α_2 that is absurd, with $\alpha_1 \preceq \alpha_2$. For the read of [20], we pick the lower bound α_2 if $[40] = 1$ and the result of the read operation is 0, and α_1 otherwise. Therefore, after the read operation we have that $r_1 = 0 \Rightarrow [40] = 0$, which allows us to prove that the update of [30] preserves the invariant. Similarly, in the main thread, for the read of [10], we pick the lower bound α_2 if $[50] = 1$ and the result of the read operation is 0, and α_1 otherwise. Therefore, after the read operation we have that $r_2 = 0 \Rightarrow [50] = 0$. Finally, for the read of [30], we pick the lower bound α_2 if $r_2 = 0$ and the result of the read operation is 0, and α_1 otherwise.

This shows that it is unsound to allow the lower bound of a read operation to depend on ghost variables. Similarly, it is unsound to allow the abstract update or the local postcondition of an update to depend on ghost variables. To show this, introduce the additional abstract states $\alpha_3 = (\alpha_1 \wedge [40] = 0)$ and $\alpha_4 = (\alpha_1 \wedge [50] = 0)$, with $\alpha_1 \preceq \alpha_3 \preceq \alpha_2$ and $\alpha_1 \preceq \alpha_4 \preceq \alpha_2$. The abstract update for the update of [10] and [20] remains the identity function. However, after the update of [10], insert a no-op update that updates the abstract state to α_3 if $[40] = 0$ and α_1 otherwise. For the read of [20], pick lower bound α_3

for result 0, and α_1 otherwise. Similarly, after the update of [20], insert a no-op update that updates the abstract state to α_4 if $[50] = 0$ and α_1 otherwise. For the read of [10], pick lower bound α_4 if the result is zero, and α_1 otherwise. For the write of [30], the case of $[40] = 1 \wedge r_1 = 0$ is contradictory and therefore can be ignored. For the read of [30], pick lower bound α_2 if $r_2 = 0 \wedge r_3 = 0$, and α_1 otherwise.

The restricted rules that allow case splitting on a local variable do not suffer from this issue since local variables do not depend on ghost variables. (To understand this, note that the only way for information to flow from the heap to the store is through read operations, and read operations read only real variables.)

6.4 Soundness

First, an auxiliary definition: a fractional heap H and abstract superstate $\tilde{\alpha}'$ satisfy an abstract superstate $\tilde{\alpha}$ if they satisfy the separating conjunction of the abstract states of the allocated regions.

$$H, \tilde{\alpha}' \in \tilde{\alpha} \Leftrightarrow H, \tilde{\alpha}' \in \Pi_{r \in \mathrm{dom}(\tilde{\alpha})}.\ \mathsf{sat}(\tilde{\alpha}(r))$$

We prove soundness via the intermediary of a notion of validity of a machine configuration, defined analogously to Sec 5 as follows:

$\mathsf{valid\_cfg}((h, \Theta_{1..n})) \Leftrightarrow \exists \tilde{\alpha}, L_{1..n}.\ (h, \tilde{\alpha}) \in \tilde{\alpha} \bullet \Pi_i L_i \wedge \mathsf{valid\_tcfg}(\tilde{\alpha}, L_i, \Theta_i)$
$\mathsf{valid\_tcfg}(\tilde{\alpha}, L, \delta \cdot \overline{\delta}, s, \kappa) =$
  $\exists f : \tilde{A} \to \tilde{A}, L'.$
    $(\forall \tilde{\alpha}' \succeq \tilde{\alpha}.\ f(\tilde{\alpha}') \succeq \tilde{\alpha}') \wedge$
    $(\forall \tilde{\alpha}' \succeq \tilde{\alpha}, \tilde{\alpha}'' \succeq \tilde{\alpha}'.\ f(\tilde{\alpha}'') \succeq f(\tilde{\alpha}')) \wedge$
    $(\forall \tilde{\alpha}' \succeq \tilde{\alpha}, H.\ (H, \tilde{\alpha}') \in \tilde{\alpha}' \bullet L \Rightarrow (\delta(H), f(\tilde{\alpha}')) \in f(\tilde{\alpha}') \bullet L') \wedge$
    $\mathsf{valid\_updates}(f(\tilde{\alpha}), L', \overline{\delta}, s, \kappa)$
$\mathsf{valid\_tcfg}(\tilde{\alpha}, L, \epsilon, s, \langle u \rangle; \kappa) = \mathsf{valid\_tcfg}(\tilde{\alpha}, L, [\![u]\!](s), s, \kappa)$
$\mathsf{valid\_tcfg}(\tilde{\alpha}, L, \epsilon, s, x := [e]; \kappa) =$
  $\forall v.$
    $(\forall \tilde{\alpha}'' \succeq \tilde{\alpha}, H.\ (\tilde{\alpha}'', H) \in \tilde{\alpha}'' \bullet L \Rightarrow$
      $s(e) \in \mathrm{dom}(H) \wedge (H(s(e)) = (v, \_) \Rightarrow \mathsf{valid\_tcfg}(\tilde{\alpha}, L, \epsilon, s[x := v], \kappa))) \vee$
    $\exists \tilde{\alpha}'.$
      $(\forall \tilde{\alpha}'' \succeq \tilde{\alpha}, H.\ (H, \tilde{\alpha}'') \in \tilde{\alpha}'' \bullet L \Rightarrow s(e) \in \mathrm{dom}(H) \wedge (H(s(e)) = (v, \_) \Rightarrow \tilde{\alpha}'' \succeq \tilde{\alpha}')) \wedge$
      $\mathsf{valid\_tcfg}(\tilde{\alpha}', L, \epsilon, s[x := v], \kappa)$

This notion of validity is sound. The proofs are analogous to those of Section 5.

Lemma 4. *If* $\gamma \rightsquigarrow \gamma'$ *and* $\mathsf{valid\_cfg}(\gamma)$ *then* $\mathsf{valid\_cfg}(\gamma')$.

Lemma 5. *If* $\mathsf{valid\_cfg}(\gamma)$ *then* $\gamma \not\rightsquigarrow^* \mathit{Fail}$.

We prove a correspondence between the Hoare rules and the notion of validity.

Lemma 6. *If* $\vdash \{P\}\ c\ \{Q\}$ *and* $\forall s'.\ \exists \tilde{\alpha}'.\ (\forall (H, \tilde{\alpha}'') \in [\![Q]\!]_{I,s'}.\ \tilde{\alpha}'' \succeq \tilde{\alpha}') \wedge \mathsf{valid}(\tilde{\alpha}', [\![Q]\!]_{I,s'}, s', \kappa)$ *and* $\forall (H, \tilde{\alpha}') \in L.\ \tilde{\alpha}' \succeq \tilde{\alpha} \Rightarrow (H, \tilde{\alpha}') \in [\![P]\!]_{I,s}$ *then* $\mathsf{valid}(\tilde{\alpha}, L, s, c; \kappa)$.

Proof. By induction on the derivation of the Hoare triple.

- **Case** ASSIGN. OK.
- **Case** MUTATE. Take $f = \lambda\tilde{\alpha}.\ \tilde{\alpha}$.
- **Case** LOOKUP. OK.
- **Case** UPDATE. OK.
- **Case** READ. OK.
- **Case** SEQ. OK.
- **Case** CONSEQ. OK.

□

We can now prove the soundness of our Hoare logic with respect to the operational semantics of the programming language.

Theorem 3. *If* $\{\mathbf{emp}\}\ c\ \{\mathbf{true}\}$ *then* $(\emptyset, \{\!|(\epsilon, \emptyset, c; \mathbf{done})|\!\}) \not\rightsquigarrow^* \mathit{Fail}$.

Proof. Apply Lemma 6 with $\tilde{\alpha} = \tilde{\alpha}' = \emptyset$ and $L = \{(\emptyset, \_)\}$ to obtain $\mathsf{valid}(\emptyset, L, \emptyset, c; \mathbf{done})$ and therefore $\mathsf{valid\_cfg}(\emptyset, \{\!|(\epsilon, \emptyset, c; \mathbf{done})|\!\})$. We obtain the goal by Lemma 5. □

7 Tool support

We developed a preliminary encoding of the proof rules for TSO memory accesses of the preceding section into our VeriFast sound modular static verification tool for C programs, and we checked versions of the examples (VM, lock, producer-consumer) in the C programming language using this encoding. These examples are included with the latest VeriFast distribution in the directory `examples/tso`.

In the development of our encoding, we needed to take special care to make sure that the abstract updates associated with TSO updates and the abstract lower bounds associated with TSO reads do not depend on ghost variables. Indeed, the naive approach of specifying the abstract updates and abstract lower bounds as ghost arguments of the TSO operations is unsound, since VeriFast allows ghost arguments to depend on ghost variables.

In our current encoding, we work around this issue by requiring the list of all abstract updates and abstract lower bounds to be used by operations on a given TSO space to be specified when the TSO space is created. The specific abstract update or lower bound for a particular operation is then selected by passing an index into this list as a non-ghost argument to the TSO operation (which appears in the program as a C function call). Furthermore, to allow the abstract updates and lower bounds to depend on the (non-ghost) state of the thread, we allow a variable number of additional non-ghost arguments to be passed to the TSO operations. These are passed on as extra arguments to the abstract updates and lower bounds.

This encoding is sound but it has the downside that it requires modifications to the C program: extra arguments to the TSO operations, and extra variables to track the thread state.

A better approach which we envision for future work is to extend VeriFast with support for additional ghost-levels of code and variables, beyond the level of reality (the lowest ghost-level) and the single existing ghost- level. Information flow from higher to lower ghost-levels would be disallowed. For the TSO encoding, we would use three ghost-levels: the real level, the semi-ghost level, and the full ghost level. Full ghost variables can be modified as part of TSO updates; abstract updates and lower bounds are specified as semi-ghost arguments.

8 Conclusion

We presented an approach for the modular formal verification of programs that use memory accesses with x86-TSO semantics.

A comparison with related work is future work.

Acknowledgments

The author thanks Ernie Cohen for very helpful comments and discussions. This work was partially funded by EU FP7 FET-Open project ADVENT (grant number 308830).