A Proposal for Weak-Memory Local Reasoning

Ian Wehrman* and Josh Berdine†

Abstract

Program logics are formal systems for specifying and reasoning about software programs. Most program logics make the strong assumption that all threads agree on the value of shared memory at all times. This assumption can be unsound though for programs with races, like many concurrent data structures. Verification of these difficult programs must take into account the weaker models of memory provided by the architectures on which they execute. In this paper, we describe progress toward a program logic for local reasoning about racy concurrent programs executing on a weak, x86-like memory model.

1 Introduction

Most concurrent software verification techniques rely on a surprisingly strong assumption: namely, that all processes agree on the value of shared memory at all times. This is, of course, not generally true, but it is often a *safe* assumption because of implicit guarantees provided by modern computer architectures, which guarantee that programs without races will not observe such inconsistencies. The soundness of most concurrent software verification techniques therefore relies on race-freedom of the program under study. This is not considered a major shortcoming though because races usually indicate a program error.

There are, however, useful and interesting programs for which races do not indicate an error. For example, many concurrent data structures, which optimize for speed and throughput by using locks and fence instructions sparingly, are often racy by design. Their correctness is demonstrated by relating the executions of their relatively daring implementations to those of their simpler, sequential counterparts. Constructing such a relation therefore requires a technique that is tolerant of races. But that requirement comes with a significant consequence: any technique that tolerates races soundly must also admit that processes may observe inconsistencies in the value of shared memory that result from the peculiarities of the architecture's memory model.

The verification literature offers little insight into the problem of verifying concurrent data structures and other inherently racy programs against realistic

*iwehrman@cs.utexas.edu, University of Texas at Austin. Supported in part by the National Science Foundation under grant CNS-0910913.

†jjb@microsoft.com, Microsoft Research.

memory models. This is partly because these models adds serious complication to an already difficult problem, but also because, until recently, formal specifications of common architectures' memory models did not exist publicly. Fortunately, the latter problem has been alleviated with recent specification efforts, notably for the x86 memory model [8]. So, for that model, the path toward a solution to the correctness problem of concurrent data structures and other important programs now lies essentially unimpeded.

In this paper, we describe work-in-progress toward a solution for the partial-correctness problem for concurrent shared-memory programs with semantics based on the x86-TSO multiprocessor memory model. We sketch a new Hoare-style logic designed for C-like programs with load, store and fence instructions, pointers and pointer arithmetic, and dynamic memory allocation and disposal. Crucially, the logic embodies an x86-specific principle of local reasoning, allowing proofs to be constructed compositionally using frame rules, as in separation logic. (Some familiarity with separation logic and, in particular, concurrent separation logic is assumed.)

The ideas presented here are informal and incomplete. In particular, parts of the model are not fully specified, and the logic does not yet have a proof of soundness. The eventual goal is to be able to use the logic to reason soundly about the behavior of racy concurrent data structures on x86-like machines, but this remains future work. The goal of this paper is to share some of our ideas with a broader audience, with the hope of generating discussion and gathering feedback.

2 Sequential Reasoning

We begin with a program logic for reasoning about sequential programs, executing on single-processor x86-like machines. In Section 3, we extend the logic to handle the parallel execution of threads on multiple, independent x86-like processors. This weak-memory logic differs from Hoare logic, separation logic and other strong-memory program logics in that it is expressive enough to distinguish between temporarily buffered writes (that result from incomplete store operations) and those that reside in shared memory. The programming language consequently incorporates appropriately modified semantics for load and store commands, as well as fencing commands.

An earlier version of the sequential fragment of the logic is described in complete detail, including a soundness proof, in a technical note [12].

2.1 Single-processor States

A program's state in Hoare's original program logic is described by a variable valuation, sometimes also referred to as a *store*. This structure proved insufficient for representing and reasoning about dynamically allocated memory and pointers, and so, with separation logic, was augmented with a *heap* data structure, a partial function that associates some memory locations with a runtime

value. But, in a concurrent system interacting with a relaxed memory model, threads may not always agree on a unique value for every location in memory. Consequently, we must again elaborate the structure used to describe program states.

The x86-TSO memory model [8] is described operationally with *write buffers*—per-processor FIFO queues of location-value pairs, which represent pending writes to shared memory. The semantics of load and store in this model are adjusted to first consult the processor's write buffer instead of directly interacting with the memory: a store command enqueues a new write into the buffer; a load command returns the most recent value in the buffer for the specified location, only if none exists does it return the value from memory. Buffered writes are committed to memory and made visible to other processors in the order in which they were enqueued, but otherwise nondeterministically. Additionally, fence commands are used to delay a thread's progress until all buffered writes at the executing processor are committed to memory.

To capture this behavior, we augment states to include, along with a shared store and heap, an array of location-value queues, which model per-processor write buffers. We begin though with just single-processor system states that consist of a store, a heap and a single queue:

$$\mathit{SPStates} = \mathit{Stores} \times \mathit{Heaps} \times \mathit{Queues}.$$

For a state σ, we write $s(\sigma)$ for the store component, $h(\sigma)$ for the heap component and $q(\sigma)$ for the queue component.

To support additional logical and operational features, we will continue to augment the notion of states throughout the paper. In particular, in Section 3, we describe the extension to multi-processor system states.

2.2 Predicates

For the sake of developing a Hoare-style logic for program reasoning, we wish to define predicates as sets of x86-TSO states. But because the x86-TSO memory model allows buffered writes to be committed nondeterministically we focus on sets of states that includes all intermediate, partially committed states. For example, a predicate with a state that describes two buffered writes also include the state in which the first write has committed, and also the state in which both writes have committed. Technically, we define a partial order $\leq$ on states, called the *flushing order*, such that $\sigma_1 \leq \sigma_2$ iff σ_1 is the result of committing to memory some part of the queue of σ_2, and we define predicates as sets of states that are down-closed w.r.t. this order.

2.3 Assertions

We borrow from separation logic the *empty* and *points-to* atomic formulas, **emp** and $e \mapsto e'$, which informally denote empty and single-point heaps, respectively, as well as, in this logic, the empty queue. Below, we write $\mathcal{E}[\![e]\!]$ for the function

that evaluates an expression to a value in the context of some store:

$$\begin{array}{lcll}(s,h,q) & \models & \mathbf{emp} & \iff h = \emptyset \wedge q = \epsilon \\ (s,h,q) & \models & e \mapsto e' & \iff h = \{(\mathcal{E}[\![e]\!]s, \mathcal{E}[\![e']\!]s)\} \wedge q = \epsilon\end{array}$$

To these we add a new *leads-to* atomic formula, $e \rightsquigarrow e'$, which denotes the empty heap and the queue with a single write at location e with value e':

$$\sigma \models e \rightsquigarrow e' \iff \exists (s,h,q)\,.\,\sigma \leq (s,h,q) \wedge h = \emptyset \wedge q = [(\mathcal{E}[\![e]\!]s, \mathcal{E}[\![e']\!]s)]$$

Note that the meaning of $e \rightsquigarrow e'$ includes also states in which the buffered write has committed to memory, yielding the following logical implication:

$$e \mapsto e' \models e \rightsquigarrow e'. \tag{1}$$

We make use of two separating conjunctions in this logic: the *interleaving* conjunction, $(P \ast Q)$, which describes the disjoint union of heaps and all possible interleavings of the write buffers; and the *sequential* conjunction, $(P \;;\; Q)$, which describes states in which the buffered writes of P precede the buffered writes of Q. To make this more precise, we first introduce a few auxiliary definitions. For a state σ, its set of *allocated locations*, written $\mathit{alloc}(\sigma)$, is given by $\mathrm{dom}(h(\sigma)) \cup \mathrm{dom}(q(\sigma))$. States are *compatible* when their stores match and their allocated locations are disjoint:

$$\sigma_1 \smile \sigma_2 \iff s(\sigma_1) = s(\sigma_2) \wedge \mathit{alloc}(\sigma_1) \cap \mathit{alloc}(\sigma_2) = \emptyset\,.$$

For compatible states σ_1 and σ_2, we define the following functions:

$$\begin{array}{rcl}\sigma_1 \ast \sigma_2 & = & \{(s,h,q) \mid s = s(\sigma_1) \wedge h = h(\sigma_1) \uplus h(\sigma_2) \wedge q \in q(\sigma_1) \# q(\sigma_2)\} \\ \sigma_1 \;;\; \sigma_2 & = & \begin{cases}(s(\sigma_1),\ h(\sigma_1),\ q(\sigma_1) +\!\!+ q(\sigma_2)) & \text{if } h(\sigma_2) = \emptyset \\ (s(\sigma_1),\ \mathit{flush}(h(\sigma_1), q(\sigma_1)) \uplus h(\sigma_2),\ q(\sigma_2)) & \text{otherwise}\end{cases}\end{array}$$

where $q_1 +\!\!+ q_2$ denotes the concatenation of buffers, and $\mathit{flush}(h,q)$ the heap that results from committing the writes of q in order into heap h. Note that the first function yields a set of states (each of which corresponds to an interleaving of the two buffers) and is symmetric, while the second yields a single state and is asymmetric. We use these semantic functions to define the meaning of the interleaving and sequential conjunctions:

$$\begin{array}{lcll}(s,h,q) & \models & P \ast Q & \iff \exists \sigma_1 \smile \sigma_2\,.\,\sigma_1 \models P \wedge \sigma_2 \models Q \wedge (s,h,q) \in \sigma_1 \ast \sigma_2 \\ (s,h,q) & \models & P \;;\; Q & \iff \exists \sigma_1 \smile \sigma_2\,.\,\sigma_1 \models P \wedge \sigma_2 \models Q \wedge (s,h,q) = \sigma_1 \;;\; \sigma_2\end{array}$$

Algebraically, both conjunctions have **emp** as a left and right unit, both are associative, and the interleaving conjunction is commutative. Additionally, we have that $(P \;;\; Q) \models (P \ast Q)$, and postulate (though have not yet proved) an *exchange law*, ala Hoare [6]:

$$(P \ast Q) \;;\; (R \ast S) \models (P \;;\; R) \ast (Q \;;\; S). \tag{2}$$

Here are a few example assertions and their informal meanings:

- $x \mapsto 1 \ * \ y \mapsto 2$ — two distinct locations in the heap;
- $x \rightsquigarrow 1 \ * \ y \rightsquigarrow 2$ — two writes, possibly buffered, to distinct locations, in either order;
- $x \mapsto 1 \ ; \ y \rightsquigarrow 2$ — two writes to distinct locations, the former in the heap, the latter possibly still buffered;
- $x \rightsquigarrow 1 \ ; \ y \rightsquigarrow 2$ — two writes to distinct locations, in which the x write precedes the y write;
- $x \rightsquigarrow 1 \ ; \ y \mapsto 2$ — two distinct locations, both in the heap.

The final example describes two *heap* locations and no buffered writes because the former write comes before the latter write, which has already committed to the heap. Hence, the former must also have been committed to the heap because writes commit in order. This leads to the following equivalence:

$$e \rightsquigarrow e' \ ; \ f \mapsto f' \ \equiv \ e \mapsto e' \ * \ f \mapsto f'. \tag{3}$$

Note how the different interpretation in the last two examples corresponds to the case split in the definition of the ; operation on states.

It would be useful to be able to describe the effect of committing a write to memory, like e in Eq. 3, without referring to an additional write, like f. We accomplish this with another atomic formula **bar**, which stands for "barrier," and which can be used to logically describe the exact relationship between points-to and leads-to:

$$e \rightsquigarrow e' \ ; \ \mathbf{bar} \ \equiv \ e \mapsto e'. \tag{4}$$

To give meaning to **bar**, we augment states with an additional boolean value, which indicates whether nondeterministic flushing of buffered writes has begun. We require that the value be $\mathbb{T}$ in any state in which the heap is non-empty—heap values are just committed writes that were once buffered. The meaning of **bar** is thus:

$$(s, h, q, b) \models \mathbf{bar} \iff h = \emptyset \wedge q = \epsilon \wedge b = \mathbb{T}.$$

Other semantic definitions are updated accordingly for this augmented notion of state. In particular, the case split in the definition of the semantic function $\sigma_1 \ ; \ \sigma_2$ is w.r.t. the value of the boolean instead of the emptiness of the heap; the boolean is set in models of points-to, and unset in models of the empty formula.

2.4 Asymmetric Counting Permissions

With the assertions given thus far, it is possible to describe successive writes to distinct locations, but not to the same locations; to do so would violate the disjointness requirements of the separating conjunctions. For example,

$$e \mapsto e' \ ; \ e \rightsquigarrow e'' \ \equiv \ \mathbf{false}$$

because no models of the first conjunct are compatible with any models of the second—the allocated location e cannot be separated across the sequential separating conjunction.

It would, of course, be useful to describe successive writes to a single location; in particular to state an axiom for the store command. To solve this problem, we augment states with a notion of sharing. In particular, we associate with each allocated location an element of an *asymmetric counting* permission model, $(\mathbb{Z}, \widehat{+})$. The model is inspired by counting permissions of Bornat et al, [1], but here the operation is partial and asymmetric, and is defined as follows:

$$a \mathbin{\widehat{+}} b = \begin{cases} a + b & \text{if } a < 0 \wedge (b < 0 \vee -b \leq a) \\ \bot & \text{otherwise.} \end{cases}$$

The semantic function $(\sigma_1 \ ; \ \sigma_2)$ is redefined with a relaxed notion of compatibility: locations allocated in both states must have compatible (i.e., not $\bot$) permissions. For now, the semantic function $(\sigma_1 \ * \ \sigma_2)$ is unchanged, requiring disjointness of allocated locations. Syntactically, points-to and leads-to assertions are annotated with integers (e.g., $e \rightsquigarrow_n e'$) that denote their location's permission value.

To understand the intuition behind asymmetric counting permissions, first recall Bornat's original counting permission model. There, negative integer annotations denote read-only permission for the location, and nonnegative integer annotations indicate the number of read-only permissions that have been split off. In particular, a zero annotation indicates full permission. For example,

$$x \mapsto_{-1} 1 \ * \ x \mapsto_{-1} 1 \ * \ x \mapsto_{2} 1,$$

is a consistent formula in Bornat's logic with full permission (-1 + -1 + 2 = 0), in which two read-only assertions have been split off of the original assertion.

The asymmetric model can be derived from Bornat's counting model by replacing the separating conjunction with the sequential conjunction, and requiring that permissions are combined with $\widehat{+}$ in the order shown in the example above, from negative to positive. Then we can interpret a nonnegative annotation as denoting the most-recently written (top-most) value, where the particular value indicates the number of prior values, if any. Such prior values are indicated by negative annotations, and the full permission by zero. For example, the following is a consistent formula that describes two successive writes to location x:

$$x \mapsto_{-1} 1 \ ; \ x \rightsquigarrow_{1} 2,$$

because $-1 \mathbin{\widehat{+}} 1 = 0$. The following, on the other hand, are inconsistent:

$$x \mapsto_{-1} 1 \ ; \ x \mapsto_{-1} 2 \ ; \ x \rightsquigarrow_{1} 3 \quad \text{and} \quad x \mapsto_{0} 1 \ ; \ x \mapsto_{-1} 2$$

because the top-most write, annotated with $n \geq 0$, must succeed no more than n earlier writes ($-2 \mathbin{\widehat{+}} 1 = \bot$), and no writes may follow the top write, annotated with a non-negative integer ($0 \mathbin{\widehat{+}} -1 = \bot$).

In the next section, we leverage the features of this permission model (i.e., consecutive writes to a single location, write counting, indication of the top-most write) to give small axioms and suitably general frame rules for a sequential specification logic. That is, the role played by this asymmetric counting permissions model is to enable assertions to locally state constraints which are potentially global across the entire underlying write buffer.

2.5 Sequential Programs and Specifications

Sequential program specifications are given by Hoare triples:

$$\{P\}\ c\ \{Q\},$$

where P is a precondition assertion for command c and Q is a postcondition assertion. Triples have the fault-avoiding partial-correctness semantics that is standard in separation logic: for every state $\sigma \models P$, executions of c from σ do not fault and either diverge or terminate in a state σ' such that $\sigma' \models Q$.

To derive these triples, the usual inference rules of Hoare logic are available, including, e.g., the rule of consequence. And, as in separation logic, there are frame rules[1] for the separating conjunctions. First, for the interleaving conjunction:

$$\frac{\{P\}\ c\ \{Q\}}{\{R\ *\ P\}\ c\ \{R\ *\ Q\}} \qquad \text{(INT-FRAME)}$$

Intuitively, this captures the fact that a program is not affected by locations outside of its footprint, whether values in the heap or writes arbitrarily interleaved in the write buffer. And second, a frame rule for the sequential conjunction:

$$\frac{\{P\}\ c\ \{Q\}}{\{R\ ;\ P\}\ c\ \{R\ ;\ Q\}} \qquad \text{(SEQ-FRAME)}$$

Note that sequential conjunction is not generally commutative, so this is explicitly a left-side frame rule. Intuitively, this additionally captures the fact that a program is not affected by the presence of additional, previous buffered writes, whether to locations outside or within its footprint, so long as the precondition and frame do not make conflicting assumptions about the top-most write or number of pending writes to individual locations.

We can now give relatively small[2] axioms for the atomic commands of the language. Again, axioms from Hoare logic for non-heap-manipulating commands are completely standard.

[1] Here, as in separation logic, there must be some mechanism for controlling the occurrence of modified variables, but we make no attempt to specify these explicitly here.

[2] The formula-level variables in these axiom schemes can be eliminated for the sake of achieving truly small axioms at the cost of additional semantic and syntactic complexity by the addition of a right-hand sequential frame rule. This complicates the store axiom in particular. Aesthetically, we find these relatively small axioms to be a reasonable tradeoff.

In an x86-like memory model, the load command fetches the most recent value written to a location e, whether buffered or in memory, and assigns that value to variable x. The following axiom schema captures this:

$$\{e \rightsquigarrow_n e' \ ; \ P\} \ x := [e] \ \{e \rightsquigarrow_n e' \ ; \ P \wedge x = e'\}, \qquad \text{(LOAD)}$$

where n is any nonnegative integer. Note that formula P may describe, e.g., writes to distinct locations more recent than the last write to e. If, on the other hand, P describes more recent writes to e, the precondition becomes inconsistent and the specification is vacuously true. Alternately, by instantiating P with **bar**, the axiom specifies loading from shared memory instead of from the write buffer. Using the sequential frame rule, we can conclude that earlier writes are irrelevant to the execution of the load command. Similarly, with the interleaving frame rule, we can conclude that writes to other locations, either before or after the last write to e, are also irrelevant.

A store of value e' to address e results in a new write added to the top of the write buffer, assuming the location has already been allocated:

$$\{e \rightsquigarrow_n v \ ; \ P\} \ [e] := e' \ \{e \rightsquigarrow_{-1} v \ ; \ P \ ; \ e \rightsquigarrow_{n+1} e'\}. \qquad \text{(STORE)}$$

where n is any nonnegative integer. In the precondition, the leads-to formula serves as a witness to the location's allocated status; the sequential frame rule can be used to describe the previous n values of e. In the postcondition, the topmost write from the precondition now becomes one of $n+1$ writes that occurred prior to the new write, with annotated permission value $n+1$. Again, P can be instantiated to describe writes to distinct locations that occurred between the previously most recent write to e and the new write, or **bar** to describe the situation in which the previous write to e is a heap value.

The fence command commits all pending writes to memory. Its axiom simply introduces a **bar** formula:

$$\{\mathbf{emp}\} \ \mathbf{fence} \ \{\mathbf{bar}\}. \qquad \text{(FENCE)}$$

Using the fence axiom, the sequential frame rule and the rule of consequence with Eq. 4, we can prove, for example:

$$\{x \mapsto_{-1} 1 \ ; \ x \rightsquigarrow_1 2\} \ \mathbf{fence} \ \{x \mapsto_0 2\}.$$

Dynamic allocation and disposal are performed by the **new** and **free** commands, respectively. The semantics of these commands are not defined at the level of the memory model, so there is some choice about what operational meaning to give them. In practice, these commands are typically implemented using fence commands to ensure system-wide consistency. In contrast, we have chosen to extricate barriers from their meaning, primarily to explore the circumstances in which barriers are actually needed. (The typical semantics of allocation and disposal can be recovered by explicitly adding fence instructions before and after these commands.)

Absent a succeeding barrier, allocation is perfectly natural; it simply adds a new write to an unallocated location to the top of the write buffer:

$$\{\mathbf{emp}\}\ x := \mathbf{new}(e)\ \{x \rightsquigarrow_0 e\} \qquad \text{(NEW)}$$

Note that the write in the postcondition has full permission, which forces earlier writes, framed from the left, to have distinct locations. Otherwise, this could result in a duplicate allocation.

The meaning of the free command absent a preceding barrier is less clear. Our semantics is conservative: if a location has at most one value in the system, it may be deallocated. In particular, we make no attempt to describe the outcome of deallocation without a barrier when there are multiple pending writes. Perhaps in this case the command should fault, or perhaps all pending writes should be removed from the system, leaving writes to other locations unaffected. In any case, the following axiom[3] describes the conservative semantics and does not allow anything to be proved in the less clear cases:

$$\{e \rightsquigarrow_0 -\}\ \mathbf{free}(e)\ \{\mathbf{emp}\} \qquad \text{(FREE)}$$

Symmetric to the case for allocation, the write in the precondition has full permission, which in this case prevents earlier writes to the same location from being framed on from the left, yielding a double disposal. By using the rule of consequence with Eq. 1, the precondition may be strengthened from a leads-to assertion to a points-to assertion, thus axiomatizing disposal of shared memory.

3 Concurrent Reasoning

In this section, we sketch an approach to concurrent, multi-processor program reasoning w.r.t. an x86-like memory model in the style of concurrent separation logic, where shared regions of memory are described by resource invariants [7].

3.1 Multi-processor States

We begin by generalizing the structures used to represent states for multi-processor systems. Single-processor states were represented by a tuple consisting of a store, a heap (with permissions), a queue, and an additional boolean value indicating whether buffered writes have been committed to memory. For multi-processor states, we replace the single queue with an array of queues—one for each processor—modeled as a function in $(\mathit{Procs} \to \mathit{Queues})$, where $\mathit{Procs} \subseteq \mathbb{N}$ is a fixed set of processor identifiers. Commands are associated with a processor identifier $p \in \mathit{Procs}$ so that, for example, a store command on processor p adds a new write to the pth write buffer. A single store (variable valuation) is sufficient as we assume that no variable is modified in more than one process. A single boolean value for committed writes is sufficient because we update the

[3] We take $e \rightsquigarrow_n -$ as shorthand for $\exists v\,.\, e \rightsquigarrow_n v$, for some $v \notin \mathrm{fv}(e)$.

compatibility relation for sequential composition to require that at most one queue be non-empty—this is elaborated upon in the next section.

3.2 Assertions

Syntactically, assertions are largely the same as for the sequential logic. Because states now have multiple write buffers, we must annotate the leads-to assertion with an additional expression indicating the buffer in which it resides, e.g., $e \rightsquigarrow^p_n e'$. We update the function $(\sigma_1 \ * \ \sigma_2)$ to describe the pointwise interleavings of the states' write buffers. And, as mentioned earlier, we restrict the compatibility relation for $(\sigma_1 \ ; \ \sigma_2)$ so that there exists at most one $p \in \mathit{Procs}$ such that the pth buffer is nonempty in both σ_1 and σ_2. This prevents us from writing (consistent) assertions like $e \rightsquigarrow^p_n e' \ ; \ f \rightsquigarrow^q_m f'$, which do not seem particularly useful. On the other hand, this allows for the simplicity of using a single boolean value for representing the presence of committed writes, and hence also an unannotated **bar** formula with the following useful equivalence:

$$e \rightsquigarrow^p_n e' \ ; \ \mathbf{bar} \ \equiv \ e \rightsquigarrow^q_n e' \ ; \ \mathbf{bar}. \tag{5}$$

Equation 5 justifies the following equivalence, an update of Eq. 4, which intuitively means that it does not matter from which processor's write buffer a heap value originated:

$$e \mapsto_n e' \ \equiv \ e \rightsquigarrow^p_n e' \ ; \ \mathbf{bar}. \tag{6}$$

3.3 Concurrent Programs and Specifications

We assume commands c are annotated with the processor identifier on which they execute, e.g., c^p for a sequential command executing on processor p, or $c_0^p \,||\, c_1^q$ for a parallel composition of c_0 and c_1 executing on processors p and q respectively. When specifying a sequential command, we sometimes also parametrize the turnstile with a processor identifier, taken as short-hand for the uniform annotation of leads-to assertions with that identifier, as well as the command. And, when there are no leads-to assertions or if the command and assertions reference only a single processor, we may omit the processor identifiers altogether.

Concurrent program specifications are given by Hoare triples parametrized by a *shared resource* assertion, which is an invariant for the heap part of shared state:

$$R \vdash \{P\} \ c \ \{Q\}.$$

The meaning of a concurrent specification is roughly as in concurrent separation logic [3]: assuming the parallel environment respects the invariant R, for every state $\sigma \models R \ * \ P$, executions of c from σ do not fault, respect the invariant R, and either diverge or terminate in a state σ' such that $\sigma' \models R \ * \ Q$.

The concurrent logic inherits all of the rules from the sequential logic[4] by uniformly annotating them with a processor identifier and a shared resource assertion. For example, using the aforementioned short-hand, the store axiom schema becomes:

$$R \vdash \{e \rightsquigarrow_n v \ ; \ P\} \ [e] := e' \ \{e \rightsquigarrow_{-1} v \ ; \ P \ ; \ e \rightsquigarrow_{n+1} e'\} \qquad \text{(STORE)}$$

We require commands c that access shared memory locations to be explicitly annotated $\langle c \rangle$ to indicate their atomicity assumptions. Of course, we do not suppose that the underlying x86-like machine supports atomic execution of arbitrary commands. But this allows uniform reasoning about those atomic commands that are supported by the machine, such as atomic increment or compare-and-swap, which can be defined as atomic sequential commands consisting of load, store and barrier commands. For example, atomic increment is defined as $\langle y := [x] \,;\, [x] := y + 1 \,;\, \mathbf{fence} \rangle$.

The atomicity requirement also does not rule out race conditions. For example, in the parallel composition $\langle [x] := 1 \rangle \,||\, \langle y := [x] \rangle$, the execution of the two atomic commands can be assumed not to overlap, but a race on x still exists because the write is buffered and may commit sometime after the store command terminates.

A program satisfies a concurrent specification only if its executions respect the shared resource invariant, which is a heap-only assertion. Atomic commands may assume that the shared state respects the invariant upon entering the atomic section, and must ensure that their terminating state respects the invariant on exit. But terminating states may contain buffered writes, which cannot possibly satisfy a heap-only assertion. This should be considered safe, though, as long as the writes never leave the heap in a state that violates the shared resource invariant as they commit.

Technically, we describe this property as follows. For state σ, we write $\sigma \backslash \epsilon$ for the state that results from replacing each buffer in σ with the empty buffer. Let R be a heap-only assertion. We define the *expansion* of R, written $\overline{R}$, as follows:

$$\sigma \models \overline{R} \iff \forall \sigma' \,.\, \sigma' \leq \sigma \Rightarrow \sigma' \backslash \epsilon \models R.$$

This yields, for example,[5] the following logical implication:

$$x \mapsto_{-1} 1 \ ; \ x \rightsquigarrow_1^p 2 \models \overline{x \mapsto_0 \{1, 2\}}.$$

Hence, the expansion $\overline{R}$ describes all states with buffered writes that, when committed to memory, respect the invariant R. With this syntax, the rule for accessing shared state is as follows:

$$\frac{\mathbf{emp} \vdash \{R \ * \ P\} \ c \ \{\overline{R} \ * \ Q\}}{R \vdash \{P\} \ \langle c \rangle \ \{Q\}} \qquad \text{(ATOMIC)}$$

[4]The possible exception being the rule of conjunction. Its soundness likely depends on a suitable notion of *precision* that is not yet well understood in the context the weak logic [5].

[5]In the sequel, we write $P[S/x]$ as shorthand for $\exists x \in S \,.\, P(x)$.

The parallel composition rule is directly from concurrent separation logic:

$$\frac{R \vdash \{P\}\ c\ \{Q\} \qquad R \vdash \{P'\}\ c'\ \{Q'\}}{R \vdash \{P\ *\ P'\}\ c \,||\, c'\ \{Q\ *\ Q'\}} \quad (\textsc{par})$$

And, finally, the sharing rule allows us to move shared resource assertions to the local specification:

$$\frac{R \vdash \{P\}\ c\ \{Q\}}{\mathbf{emp} \vdash \{\overline{R}\ *\ P\}\ c\ \{\overline{R}\ *\ Q\}} \quad (\textsc{share})$$

In the conclusion, there may still be pending writes into the resource defined by R, and so the most we may conclude is that these pending writes satisfy the relation $\overline{R}$.

With these rules, in contrast to CSL, we can prove *racy* programs. For example:

$$\mathbf{emp} \vdash \{x \mapsto_0 \{1,2\}\}\ \langle [x] := 1\rangle \,||\, \langle y := [x]\rangle\ \{y = 1 \vee y = 2\}.$$

3.3.1 A Stubborn Example

Consider the following message-passing parallel program $c_w \,||\, c_r$, where c_w and c_r are defined as follows:

$$c_w = \langle [\mathit{data}] := 1\rangle\,;\langle [\mathit{ready}] := 1\rangle \quad \text{and} \quad c_r = \langle x := [\mathit{ready}]\rangle\,;\langle y := [\mathit{data}]\rangle.$$

We would like to prove, under some precondition, the following specification:

$$\mathbf{emp} \vdash \{...\}\ c_w \,||\, c_r\ \{x = 1 \Rightarrow y = 1\}.$$

We proceed by defining a resource invariant[6] R for the shared locations:

$$R = \exists d\,.\,\exists r\,.\,\mathit{data} \mapsto d\ *\ \mathit{ready} \mapsto r \wedge (r = 1 \Rightarrow d = 1),$$

and then use sequential reasoning and the rule for atomic sections to show that each component individually respects the invariant. We could then complete the proof as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{R \vdash \{\mathbf{emp}\}\ c_w\ \{\mathbf{emp}\}} \quad \dfrac{\vdots}{R \vdash \{\mathbf{emp}\}\ c_r\ \{\mathbf{emp}\ *\ (x = 1 \Rightarrow y = 1)\}}}{R \vdash \{\mathbf{emp}\}\ c_w \,||\, c_r\ \{\mathbf{emp}\ *\ (x = 1 \Rightarrow y = 1)\}}\ (\textsc{par})}{\mathbf{emp} \vdash \{R\}\ c_w \,||\, c_r\ \{R\ *\ (x = 1 \Rightarrow y = 1)\}}\ (\textsc{share})}{\mathbf{emp} \vdash \{R\}\ c_w \,||\, c_r\ \{x = 1 \Rightarrow y = 1\}}\ (\textsc{cons})$$

[6] Counting permission annotations in resource invariants, implicitly 0, are omitted for readability.

Unfortunately, we are unable to show that the writing thread c_w satisfies its part of the specification, i.e., that $R \vdash \{\mathbf{emp}\}\ c_w\ \{\mathbf{emp}\}$. The first write is not problematic, for we can readily show that setting *data* will never violate the invariant. The second write, on the other hand, is problematic. We need to show in particular that:

$$R\ ;\ \mathit{ready} \rightsquigarrow_1 1 \models \overline{R},$$

but this is impossible because the implication is false. Consider that if both *data* and *ready* are unset—which is possible, according to the invariant—then setting *ready* violates the invariant.

In traditional (strong-memory) concurrent separation logic, we could solve this problem with an auxiliary variable: modify the program to atomically assign to *aux* as it stores to *data*, and strengthen the invariant to require that $\mathit{aux} = 1 \Rightarrow \mathit{data} \mapsto 1$. The second write can then be proved with the precondition $\mathit{aux} = 1$ as part of its local state. But this is insufficient in the weak logic, because we have no way of knowing when the write to *data* will commit, so it is not necessarily the case that $\mathit{aux} = 1 \Rightarrow \mathit{data} \mapsto 1$.

What we do know is that by the time the second write, to *ready*, commits the write to *data* must have committed because writes always commit in order. Unfortunately this relationship is lost in the logic as described thus far; the relative order of writes to shared memory is abstracted away when exiting an atomic section, summarized by the invariant only as a set of possible heap values.

On the other hand, if the writing thread could instead keep ownership of the shared locations in its local state, then it could precisely track the relative order of writes; as per the rule for atomic sections, no summarization is needed for the local post-state. But then these locations would have to be removed from the resource invariant, and hence from the shared state, and so the reading thread would be unable to access them without additional synchronization, which would defeat the purpose of the weak logic entirely.

3.4 Splitting Permissions

This suggests a solution based, yet again, on permission accounting ala Bornat et al, this time with a splitting algebra [1]. In fact, we use a fractional model $(\mathbb{Q}, \widehat{*})$, as initially suggested by Boyland, for sharing read-only resources among threads [2] across the interleaving separating conjunction. For $s_1, s_2 \in \mathbb{Q}$:

$$s_1 \mathbin{\widehat{*}} s_2 = \begin{cases} s_1 + s_2 & \text{if } s_1 > 0/1 \text{ and } s_2 > 0/1 \text{ and } s_1 + s_2 \leq 1/1 \\ \top & \text{otherwise.} \end{cases}$$

The compatibility function for $(\sigma_1 * \sigma_2)$ is modified to require that locations allocated in both heaps have equal values, compatible fractional permissions and that both have full counting permission. The compatibility function for $(\sigma_1\ ;\ \sigma_2)$ is strengthened to require that locations allocated in both states have equal fractional permissions. The $*$ operation on states is constant in the counting permissions, and the ; operation is constant on the fractional permissions.

The points-to and leads-to assertions are now annotated with a pair (s, n), where s is a splitting permission and n is a counting permission.[7] The store axiom is updated to require full splitting permission (i.e., $(1/1, n)$, with $n \geq 0$), while for the load axiom, any splitting permission suffices (i.e., (s, n), with $0/1 < s \leq 1/1$ and $n \geq 0$). The allocation and disposal axioms guarantee resp. require full splitting permission.

This model results in the following basic heap equivalences:

$$s = s_1 \mathbin{\widehat{*}} s_2 \Rightarrow (e \mapsto_{s_1,0} e' \ * \ e \mapsto_{s_2,0} e' \ \equiv \ e \mapsto_{s,0} e') \tag{7}$$

$$n = n_1 \mathbin{\widehat{;}} n_2 \Rightarrow (e \mapsto_{s,n_1} e' \ ; \ e \mapsto_{s,n_2} e' \ \equiv \ e \mapsto_{s,n} e') \tag{8}$$

Additionally, the following equivalence indicates the usefulness of the combined model for our purposes:

$$x \mapsto_{1/1,-1} 1 \ ; \ x \rightsquigarrow_{1/1,1} 2 \ \equiv \ (x \mapsto_{1/2,0} \{1,2\}) \ * \ (x \mapsto_{1/2,-1} 1 \ ; \ x \rightsquigarrow_{1/2,1} 2) \tag{9}$$

Intuitively, this means an assertion with full permission can be split into two half-permission parts, one of which captures just the possible heap values, and one of which captures the ordered writes. This is exactly the sort of sharing needed to complete the message-passing proof: a thread may remember its own writes to a location provided the parallel context only reads from the location.

3.4.1 A Stubborn Example, Revisited

With our combined model, we now return to the message-passing program. First, let us denote by R_x the assertion like R, but in which each points-to assertion has permission x:

$$R_x = \exists d \,.\, \exists r \,.\, \mathit{data} \mapsto_x d \ * \ \mathit{ready} \mapsto_x r \wedge (r = 1 \Rightarrow d = 1).$$

Now, taking the resource invariant to be $R_{1/2}$, we can prove for the reading thread c_r that $x = 1 \Rightarrow y = 1$ as before, and sketch the following proof for the writing thread c_w under precondition P, defined below:

$\{P \ : \ \mathit{data} \mapsto_{1/2,0} - \ * \ \mathit{ready} \mapsto_{1/2,0} -\}$
$\quad \langle [\mathit{data}] := 1 \rangle;$
$\{(\mathit{data} \mapsto_{1/2,-1} - \ * \ \mathit{ready} \mapsto_{1/2,0} -) \ ; \ \mathit{data} \rightsquigarrow_{1/2,1} 1\}$
$\quad \langle [\mathit{ready}] := 1 \rangle$
$\{(\mathit{data} \mapsto_{1/2,-1} - \ * \ \mathit{ready} \mapsto_{1/2,-1} -) \ ; \ \mathit{data} \rightsquigarrow_{1/2,1} 1 \ ; \ \mathit{ready} \rightsquigarrow_{1/2,1} 1\}$
$\{\mathbf{true}\}$

The proof of each triple relies crucially on an equivalence similar to Eq. 9, which allows us to combine the half-permission heap values described by the resource

[7] A simple product suffices to combine splitting and counting models in this setting, in contrast to the necessarily more elaborate models of Parkinson [9] and Dockins, et al [4], because each splitting aspect is effectively isolated to a single conjunction: counting to the sequential conjunction, and splitting to the interleaving conjunction.

invariant, $R_{1/2}$, with the half-permission heap values and writes from the local state, P, into the full, writable permission: $R_{1/2} * P \equiv R_1$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{R_{1/2} \vdash \{P\}\ c_w\ \{\mathbf{true}\}} \quad \dfrac{\vdots}{R_{1/2} \vdash \{\mathbf{emp}\}\ c_r\ \{x = 1 \Rightarrow y = 1\}}}{R_{1/2} \vdash \{P\}\ c_w \,||\, c_r\ \{x = 1 \Rightarrow y = 1\}}\ (\textsc{par})}{\mathbf{emp} \vdash \{R_{1/2} * P\}\ c_w \,||\, c_r\ \{R_{1/2} * (x = 1 \Rightarrow y = 1)\}}\ (\textsc{share})}{\mathbf{emp} \vdash \{R_1\}\ c_w \,||\, c_r\ \{x = 1 \Rightarrow y = 1\}}\ (\textsc{cons})$$

4 Related Work and Conclusion

The only other logic we know for reasoning soundly about racy concurrent programs (without unrealistic strong memory assumptions) is from Ridge, who has encoded a rely-guarantee-style system for x86 assembly programs into [10]. He has given a formal soundness proof w.r.t. the x86-TSO memory model, and used the logic to construct proofs of several interesting concurrent data structures. In contrast to the small axioms and frame rules in our work, his logic does not incorporate any particular mechanism for local reasoning. Much work remains on the weak logic sketched in this paper. A significant shortcoming of the combined permission model is that some assertions do not denote down-closed sets of states, which can lead to violations of soundness. Although we believe it to be relatively straightforward to avoid such assertions, a conservative method of checking this property is needed. (This is similar to stability requirements in, e.g., RG-Sep [11].) Additionally, some of the logical implications from Section 2.3, like the exchange law, fail to hold for assertions with partial permissions. Further investigation of the soundness of the logic is an immediate priority, as well as finding a useful, partial axiomatization of the assertion language. Finally, we hope to be able to apply the logic to some racy concurrent data structures.

References

[1] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In J. Palsberg and M. Abadi, editors, *POPL*, pages 259–270. ACM, 2005.

[2] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *SAS*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003.

[3] S. Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270, 2007.

[4] R. Dockins, A. Hobor, and A. W. Appel. A fresh look at separation algebras and share accounting. In Z. Hu, editor, *APLAS*, volume 5904 of *Lecture Notes in Computer Science*, pages 161–177. Springer, 2009.

[5] A. Gotsman, J. Berdine, and B. Cook. Precision and the conjunction rule in concurrent separation logic. In *Mathematical Foundations of Programming Semantics*, 2011. To appear.

[6] C. A. R. Hoare, B. Möller, G. Struth, and I. Wehrman. Concurrent kleene algebra. In M. Bravetti and G. Zavattaro, editors, *CONCUR*, volume 5710 of *Lecture Notes in Computer Science*, pages 399–414. Springer, 2009.

[7] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

[8] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-tso. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 391–407. Springer, 2009.

[9] M. J. Parkinson. Local reasoning for java. Technical report, University of Cambridge, 2005. Technical Report UCAM-CL-TR-64.

[10] T. Ridge. A rely-guarantee proof system for x86-tso. In G. T. Leavens, P. W. O'Hearn, and S. K. Rajamani, editors, *VSTTE*, volume 6217 of *Lecture Notes in Computer Science*, pages 55–70. Springer, 2010.

[11] V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In L. Caires and V. T. Vasconcelos, editors, *CONCUR*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer, 2007.

[12] I. Wehrman. Semantics and syntax of a weak-memory concurrent separation logic: Sequential fragment. `http://cs.utexas.edu/~iwehrman/x86-logic/sequential.pdf`, 2010.