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.

$$
\begin{aligned}
[\![-]\!] \;:\; & \mathsf{CExp} \to \mathbb{P}(\langle \mathit{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}), \mathit{fst} : \mathcal{A}, \mathit{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')\} \\
[\![\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')\} \\
&\quad \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)\} \\
[\![\mathbf{let}\ x = E_1\ \mathbf{in}\ E_2]\!] &\stackrel{\text{def}}{=} \{\langle \bot, \mathcal{A}_1, \mathsf{lab}_1, \mathsf{sb}_1, \mathit{fst}_1, \mathit{lst}_1\rangle \mid \langle \bot, \mathcal{A}_1, \mathsf{lab}_1, \mathsf{sb}_1, \mathit{fst}_1, \mathit{lst}_1\rangle \in [\![E_1]\!]\} \\
&\quad \cup \{\langle \mathit{res}_2, \mathcal{A}_1 \uplus \mathcal{A}_2, \mathsf{lab}_1 \cup \mathsf{lab}_2, \mathsf{sb}_1 \cup \mathsf{sb}_2 \cup \{(\mathit{lst}_1, \mathit{fst}_2)\}, \mathit{fst}_1, \mathit{lst}_2\rangle \mid \\
&\qquad \langle v_1, \mathcal{A}_1, \mathsf{lab}_1, \mathsf{sb}_1, \mathit{fst}_1, \mathit{lst}_1\rangle \in [\![E_1]\!] \wedge \langle \mathit{res}_2, \mathcal{A}_2, \mathsf{lab}_2, \mathsf{sb}_2, \mathit{fst}_2, \mathit{lst}_2\rangle \in [\![E_2[v_1/x]]\!]\} \\
[\![\mathbf{repeat}\ E\ \mathbf{end}]\!] &\stackrel{\text{def}}{=} \{\langle \mathit{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 \{(\mathit{lst}_1, \mathit{fst}_2), \ldots, (\mathit{lst}_{N-1}, \mathit{fst}_N)\}, \mathit{fst}_1, \mathit{lst}_N\rangle \mid \\
&\qquad \forall i.\ \langle \mathit{res}_i, \mathcal{A}_i, \mathsf{lab}_i, \mathsf{sb}_i, \mathit{fst}_i, \mathit{lst}_i\rangle \in [\![E]\!] \wedge (i \neq N \implies \mathit{res}_i = 0) \wedge \mathit{res}_N \neq 0\} \\
[\![E_1 \| E_2]\!] &\stackrel{\text{def}}{=} \{\langle \mathsf{combine}(\mathit{res}_1, \mathit{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}\}, \\
&\qquad \mathsf{sb}_1 \cup \mathsf{sb}_2 \cup \{(a_{\text{fork}}, \mathit{fst}_1), (a_{\text{fork}}, \mathit{fst}_2), (\mathit{lst}_1, a_{\text{join}}), (\mathit{lst}_2, a_{\text{join}})\}, a_{\text{fork}}, a_{\text{join}}\rangle \mid \\
&\qquad \langle \mathit{res}_1, \mathcal{A}_1, \mathsf{sb}_1, \mathit{fst}_1, \mathit{lst}_1\rangle \in [\![E_1]\!] \wedge \langle \mathit{res}_2, \mathcal{A}_2, \mathsf{sb}_2, \mathit{fst}_2, \mathit{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.

$$
\begin{aligned}
&\nexists x.\ \mathsf{hb}(x,x) && \text{(IrreflexiveHB)} \\
&\forall \ell.\ \mathsf{totalorder}(\{a \in \mathcal{A} \mid \mathsf{iswrite}_\ell(a)\}, \mathsf{mo}) \wedge \mathsf{hb}_\ell \subseteq \mathsf{mo} && \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} && \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) && \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) && \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) && \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)) && \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) && \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) && \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) && \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) && \text{(AtomicRMW)} \\
&\forall a, b, \ell.\ \mathsf{lab}(a) = \mathsf{lab}(b) = \mathrm{A}(\ell) \implies a = b && \text{(ConsistentAlloc)}
\end{aligned}
$$

$$
\begin{aligned}
&\text{where } \mathsf{iswrite}_{\ell,v}(a) \stackrel{\text{def}}{=} \exists X, v_{\text{old}}.\ \mathsf{lab}(a) \in \{\mathrm{W}_X(\ell, v), \mathrm{RMW}_X(\ell, v_{\text{old}}, v)\} \qquad \mathsf{iswrite}_\ell(a) \stackrel{\text{def}}{=} \exists v.\ \mathsf{iswrite}_{\ell,v}(a) \\
&\qquad \mathsf{isread}_{\ell,v}(a) \stackrel{\text{def}}{=} \exists X, v_{\text{new}}.\ \mathsf{lab}(a) \in \{\mathrm{R}_X(\ell, v), \mathrm{RMW}_X(\ell, v, v_{\text{new}})\} \qquad \text{etc.} \\
&\qquad \mathsf{rsElem}(a,b) \stackrel{\text{def}}{=} \mathsf{sameThread}(a,b) \vee \mathsf{isrmw}(b) \\
&\qquad \mathsf{rseq}(a) \stackrel{\text{def}}{=} \{a\} \cup \{b \mid \mathsf{rsElem}(a,b) \wedge \mathsf{mo}(a,b) \wedge (\forall c.\ \mathsf{mo}(a,c) \wedge \mathsf{mo}(c,b) \Rightarrow \mathsf{rsElem}(a,c))\} \\
&\qquad \mathsf{sw} \stackrel{\text{def}}{=} \{(a,b) \mid \mathsf{mode}(a) \in \{\mathrm{rel}, \mathrm{rel\_acq}, \mathrm{sc}\} \wedge \mathsf{mode}(b) \in \{\mathrm{acq}, \mathrm{rel\_acq}, \mathrm{sc}\} \wedge \mathsf{rf}(b) \in \mathsf{rseq}(a)\} \\
&\qquad \mathsf{hb} \stackrel{\text{def}}{=} (\mathsf{sb} \cup \mathsf{sw})^+ \\
&\qquad \mathsf{hb}_\ell \stackrel{\text{def}}{=} \{(a,b) \in \mathsf{hb} \mid \mathsf{iswrite}_\ell(a) \wedge \mathsf{iswrite}_\ell(b)\} \\
&\qquad X_{\mathsf{SeqCst}} \stackrel{\text{def}}{=} \{(a,b) \in X \mid \mathsf{isSeqCst}(a) \wedge \mathsf{isSeqCst}(b)\} \\
&\qquad \mathsf{isc}(a,b) \stackrel{\text{def}}{=} \mathsf{iswrite}_{\mathsf{locs}(b)}(a) \wedge \mathsf{sc}(a,b) \wedge \nexists c.\ \mathsf{sc}(a,c) \wedge \mathsf{sc}(c,b) \wedge \mathsf{iswrite}_{\mathsf{locs}(b)}(c)
\end{aligned}
$$

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

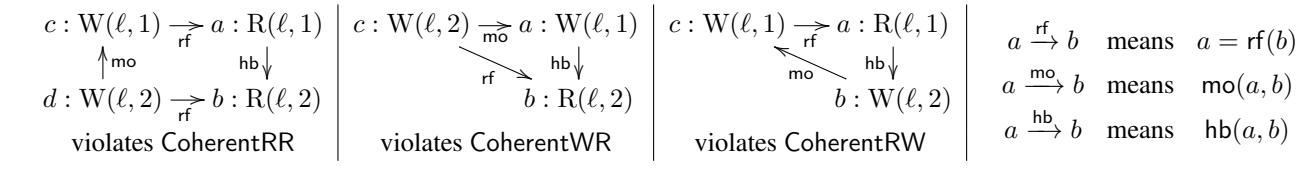


Figure 4. Sample executions violating coherency conditions (Batty et al. 2011).

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.

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

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).

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

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:

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

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.

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

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 $\mathbf{CAS}$ 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 $\mathbf{CAS}$ 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 $\mathbf{CAS}$ 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 $\mathbf{CAS}$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 $\mathbf{CAS}$ 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 $\mathbf{CAS}$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.

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

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

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

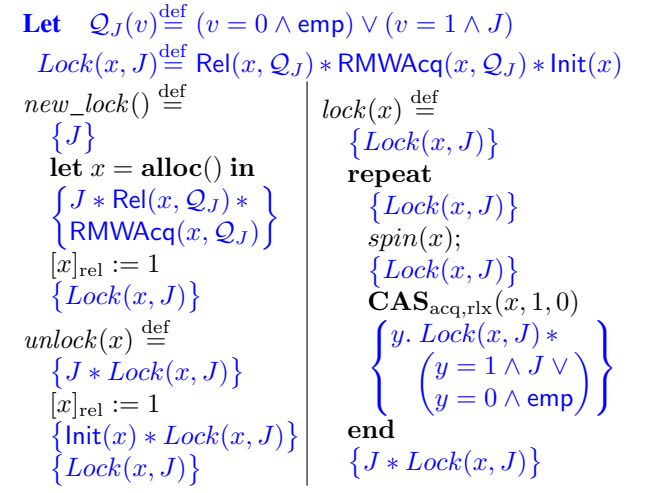


Figure 7. Verification of the lock module.

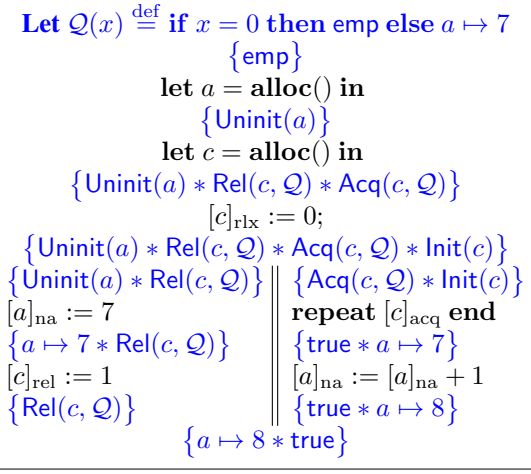


Figure 8. Verification of the message passing example.

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 $\mathbf{CAS}$. 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

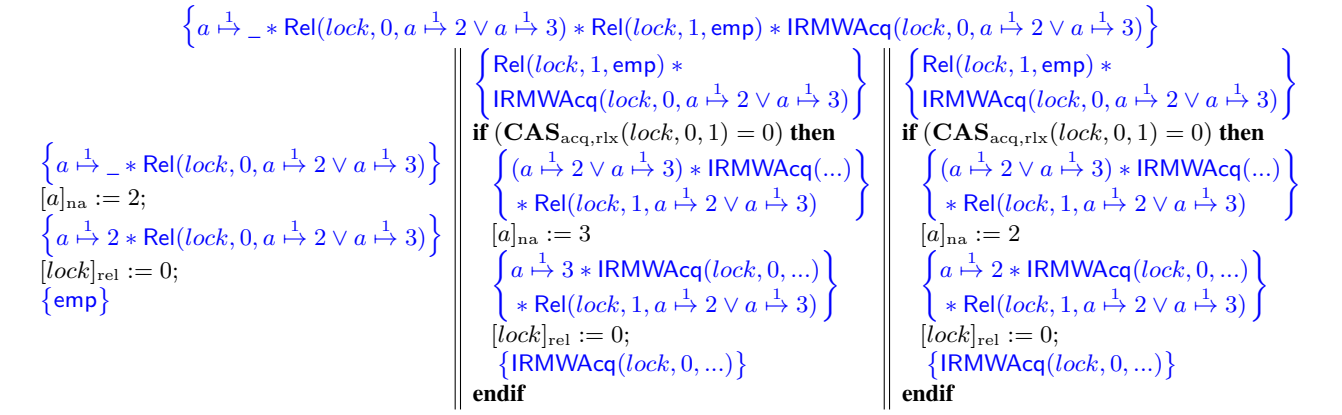


Figure 10. Example illustrating the use of **CAS** to implement a lock.

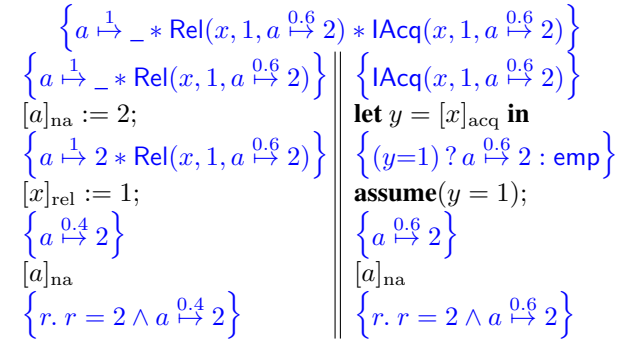


Figure 9. Example illustrating fractional ownership and transfer thereof.

Figure 11. Program with a possible dependency cycle.

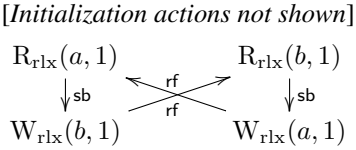


Figure 12. Execution exhibiting the dependency cycle.

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

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

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

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

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

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

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

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

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, $\mathbb{P}(\mathsf{Heap}_{\text{spec}})$, 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, $\mathsf{Heap}_{\text{spec}}$, therefore is:

$$\mathsf{Perm} \stackrel{\text{def}}{=} (0, 1] \qquad \mathbb{B} \stackrel{\text{def}}{=} \{\mathsf{true}, \mathsf{false}\}$$

$$\mathcal{M} \stackrel{\text{def}}{=} \mathsf{Val} \to \mathsf{Assn}/\!\sim$$

$$\mathsf{Heap}_{\text{spec}} \stackrel{\text{def}}{=} \mathsf{Loc} \rightharpoonup \begin{pmatrix} \mathsf{NA}[\mathbb{U} + (\mathsf{Val} \times \mathsf{Perm})] \\ +\mathsf{Atom}[\mathcal{M} \times \mathcal{M} \times \mathbb{B} \times \mathbb{B}] \end{pmatrix}$$

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 $(\mathsf{Heap}_{\text{spec}}, \oplus, \emptyset)$ forms a separation algebra, which in turn means that it is a good model for separation logic assertions.

Lemma 1. *$(\mathsf{Heap}_{\text{spec}}, \oplus, \emptyset)$ 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 $h_{\text{I}} = \{\ell \mapsto \mathsf{Atom}[\mathsf{False}, \mathsf{Emp}, \mathsf{false}, \mathsf{true}]\}$. Clearly, $h_{\text{I}} \neq \emptyset$ and yet $h_{\text{I}} \oplus h_{\text{I}} = h_{\text{I}} = h_{\text{I}} \oplus \emptyset$. 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 $h_{\text{I}} \oplus h_{\text{I}} = h_{\text{I}}$ 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.

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

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*

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

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

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

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

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

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

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

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

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 $\mathsf{lab}(a) = \mathrm{R}_{\text{acq}}(\ell, v)$. 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.

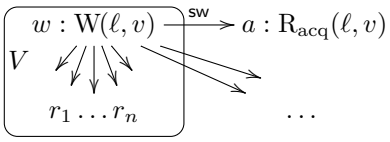
First, as the precondition includes $\mathsf{Init}(\ell)$, we know that there exists a write to ℓ that happens before the acquire read.

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 $\mathsf{lab}_\mathcal{X}(c) = \mathrm{W}\_(\ell,\_)$ 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 $\mathsf{lab}_\mathcal{X}(c) = \mathrm{A}(\ell)$. 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, $\mathsf{lab}_\mathcal{X}(d) = \mathrm{A}(\ell)$, 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 ("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 $\mathcal{T} \cup \{(\text{“}\mathsf{sw}\text{”}, w, r) \mid r \in \mathsf{dom}(\mathcal{R})\}$ 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 $\forall x \notin \mathcal{A}.\ \mathsf{lab}(x) = \mathrm{skip}$. 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.