Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 276 (2011) 335–351

www.elsevier.com/locate/entcs

Concurrent Separation Logic and Operational Semantics

Viktor Vafeiadis

Max Planck Institute for Software Systems (MPI-SWS), Germany

Abstract

This paper presents a new soundness proof for concurrent separation logic (CSL) in terms of a standard operational semantics. The proof gives a direct meaning to CSL judgments, which can easily be adapted to accommodate extensions of CSL, such as permissions and storable locks, as well as more advanced program logics, such as RGSep. Further, it explains clearly why resource invariants should be 'precise' in proofs using the conjunction rule.

Keywords: Separation logic; concurrency; soundness; race condition

1 Introduction

Concurrent separation logic [15] (CSL) is a concurrent program logic, a formal system for proving certain correctness properties of concurrent programs. It is based on the notion of resource ownership, where the resource typically is dynamically allocated memory (i.e., the heap). Since its inception by O'Hearn, it has become quite popular, because it permits elegant correctness proofs of some complex concurrent pointer programs that keep track of their memory consumption and explicitly deallocate any unused memory. Its popularity is evident by the number of extensions to CSL (e.g., permissions [2,1], locks in the heap [9,13], variables as resource [16], re-entrant locks [11]).

Besides having many extensions, CSL also has many soundness proofs. Some proofs [3,12,10] are about plain CSL, some [5,9,13,11] are about a particular extension, while others [6,4] are abstract.

Following Brookes's original proof [3], several proofs [13,6,4] give the semantics of triples in terms of a non-standard 'intermediate' semantics that keeps explicit track of resource ownership during execution. In such semantics, acquiring and releasing a lock, operations that normally update a single bit, instead allocate or deallocate part of the heap (receiving it from or sending to a shared resource). The

doi:10.1016/j.entcs.2011.09.029

adequacy of the intermediate semantics is usually justified by an 'erasure' theorem stating that the intermediate semantics simulates a standard semantics.

Some other proofs [11,14,9,10] instead are completely syntactic: they never define the meaning of CSL judgments, but rather establish a global invariant that ensures data-race freedom and that is preserved under execution steps. This proof technique is similar to the "progress and preservation" strategy that is common in soundness proofs of type systems and is rather fragile. If, for instance, a new construct were to be added to the language, the soundness of the existing rules would have to be reproved. Moreover, as the meaning of CSL judgments is never defined except perhaps for closed 'top-level' programs, it is never clear what program specifications actually mean.

In this paper, we take a direct approach to proving soundness of CSL. We define the meaning of CSL judgments directly in terms of a standard concrete operational semantics for the programming language. Our definition is concise and results in a relatively simple soundness proof, which we have formalised in Isabelle/HOL. [1] Our soundness statement has three important benefits:

(i) It encompasses the framing aspect of separation logic. As a result, the proof does not technically require that the operational semantics satisfies the "safety monotonicity" and the "frame" properties [20].

(ii) It does not insist on resource invariants being precise. Similar to Gotsman et al. [10], we prove (a) that CSL with possibly imprecise resource invariants and without the conjunction rule is sound, and (b) that the conjunction rule is sound provided that the resource invariants in scope are precise. Both proofs use the same semantics for CSL judgments.

(iii) It can easily be adapted to cover CSL extensions, such as permissions [2,1], RGSep [19], deny/guarantee, and concurrent abstract predicates [7].

Paper Outline. For pedagogic reasons, we will first focus on a cut down version of CSL where the only construct for synchronisation is an atomic block executing in one atomic step (§2). We shall give the syntax and semantics of the programming language and of separation logic assertions, as well as the CSL proof rules. We shall then define carefully the semantics of the CSL judgments (§3) and prove that the proof rules are sound (§4).

Later, in §5, we shall consider O'Hearn's original setting with multiple named conditional critical regions that execute non-atomically, but in mutual exclusion, and prove CSL's data race freedom result. Finally, we will adapt our correctness statements to handle extensions of CSL, such as permissions (§6) and RGSep (§7).

[1] The proof scripts are available at `http://www.mpi-sws.org/~viktor/cslsound` [18].

$$\frac{}{(\mathbf{skip}; C_2), \sigma \to C_2, \sigma} \quad (\textsc{Seq1})$$

$$\frac{C_1, \sigma \to C_1', \sigma'}{(C_1; C_2), \sigma \to (C_1'; C_2), \sigma'} \quad (\textsc{Seq2})$$

$$\frac{C_1, \sigma \to \mathbf{abort}}{(C_1; C_2), \sigma \to \mathbf{abort}} \quad (\textsc{SeqA})$$

$$\frac{\sigma = (s,h) \quad [\![B]\!](s)}{(\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2), \sigma \to C_1, \sigma} \quad (\textsc{If1})$$

$$\frac{\sigma = (s,h) \quad \neg[\![B]\!](s)}{(\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2), \sigma \to C_2, \sigma} \quad (\textsc{If2})$$

$$\frac{C, \sigma \to^* \mathbf{abort}}{\mathbf{atomic}\ C, \sigma \to \mathbf{abort}} \quad (\textsc{AtomA})$$

$$\frac{}{(\mathbf{while}\ B\ \mathbf{do}\ C), \sigma \to (\mathbf{if}\ B\ \mathbf{then}\ (C; \mathbf{while}\ B\ \mathbf{do}\ C)\ \mathbf{else}\ \mathbf{skip}), \sigma} \quad (\textsc{Loop})$$

$$\frac{C, \sigma \to^* \mathbf{skip}, \sigma'}{(\mathbf{atomic}\ C), \sigma \to \mathbf{skip}, \sigma'} \quad (\textsc{Atom})$$

$$\frac{C_1, \sigma \to C_1', \sigma'}{(C_1 \| C_2), \sigma \to (C_1' \| C_2), \sigma'} \quad (\textsc{Par1})$$

$$\frac{C_2, \sigma \to C_2', \sigma'}{(C_1 \| C_2), \sigma \to (C_1 \| C_2'), \sigma'} \quad (\textsc{Par2})$$

$$\frac{}{(\mathbf{skip} \| \mathbf{skip}), \sigma \to \mathbf{skip}, \sigma} \quad (\textsc{Par3})$$

$$\frac{C_1, \sigma \to \mathbf{abort}}{(C_1 \| C_2), \sigma \to \mathbf{abort}} \quad (\textsc{ParA1})$$

$$\frac{C_2, \sigma \to \mathbf{abort}}{(C_1 \| C_2), \sigma \to \mathbf{abort}} \quad (\textsc{ParA2})$$

(Assign)	$x := E, (s,h) \to \mathbf{skip}, (s[x := [\![E]\!](s)], h)$	
(Read)	$x := [E], (s,h) \to \mathbf{skip}, (s[x := v], h)$	if $h([\![E]\!](s)) = v$
(ReadA)	$x := [E], (s,h) \to \mathbf{abort}$	if $[\![E]\!](s) \notin \mathbf{dom}(h)$
(Wri)	$[E] := E', (s,h) \to \mathbf{skip}, (s, h[[\![E]\!](s) := [\![E']\!](s)])$	if $[\![E]\!](s) \in \mathbf{dom}(h)$
(WriA)	$[E] := E', (s,h) \to \mathbf{abort}$	if $[\![E]\!](s) \notin \mathbf{dom}(h)$
(All)	$x := \mathbf{alloc}(E), (s,h) \to \mathbf{skip}, (s[x := \ell], h[\ell := [\![E]\!](s)])$	where $\ell \notin \mathbf{dom}(h)$
(Free)	$\mathbf{dispose}(E), (s,h) \to \mathbf{skip}, (s, h[[\![E]\!](s) := \bot])$	if $[\![E]\!](s) \in \mathbf{dom}(h)$
(FreeA)	$\mathbf{dispose}(E), (s,h) \to \mathbf{abort}$	if $[\![E]\!](s) \notin \mathbf{dom}(h)$

Fig. 1. Small-step operational semantics for commands.

2 Concurrent Separation Logic

Consider the following simple language of commands:

$$\begin{aligned} E &::= x \mid n \mid E + E \mid E - E \mid \dots \\ B &::= B \wedge B \mid \neg B \mid E = E \mid E \leq E \mid \dots \\ C &::= \mathbf{skip} \mid x := E \mid x := [E] \mid [E] := E \mid x := \mathbf{alloc}(E) \mid \mathbf{dispose}(E) \\ &\quad \mid C_1; C_2 \mid C_1 \| C_2 \mid \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 \mid \mathbf{while}\ B\ \mathbf{do}\ C \mid \mathbf{atomic}\ C \end{aligned}$$

Arithmetic expressions, E, consist of program variables, integer constants, and arithmetic operations. Boolean expressions, B, consist of arithmetic equalities and inequalities and Boolean operations. Commands, C, include the empty command, variable assignments, memory reads, writes, allocations and deallocations, sequential composition, parallel composition, conditionals, loops, and atomic commands.

We assume a domain of variable names ($\mathsf{VarName}$), a domain of memory locations (Loc) and a domain of values (Val) that includes memory locations and define the following composite domains:

$s \in \mathsf{Stack} \stackrel{\text{def}}{=} \mathsf{VarName} \to \mathsf{Val}$	stacks (interpretations for variables)
$h \in \mathsf{Heap} \stackrel{\text{def}}{=} \mathsf{Loc} \rightharpoonup_{\text{fin}} \mathsf{Val}$	heaps (dynamically allocated memory)
$\sigma \in \mathsf{State} \stackrel{\text{def}}{=} \mathsf{Stack} \times \mathsf{Heap}$	program states

Arithmetic and Boolean expressions are interpreted denotationally as total functions from stacks to values or Boolean values respectively:

$$\begin{array}{l} [\![\_]\!] : \mathsf{Exp} \to \mathsf{Stack} \to \mathsf{Val} \\ [\![x]\!](s) \stackrel{\text{def}}{=} s(x) \\ [\![E_1 + E_2]\!](s) \stackrel{\text{def}}{=} [\![E_1]\!](s) + [\![E_2]\!](s) \end{array} \quad \left| \quad \begin{array}{l} [\![\_]\!] : \mathsf{BoolExp} \to \mathsf{Stack} \to \{\mathsf{true}, \mathsf{false}\} \\ [\![B_1 \wedge B_2]\!](s) \stackrel{\text{def}}{=} [\![B_1]\!](s) \wedge [\![B_2]\!](s) \\ [\![E_1 \leq E_2]\!](s) \stackrel{\text{def}}{=} [\![E_1]\!](s) \leq [\![E_2]\!](s) \end{array} \right.$$

Commands are given a small-step operational semantics in Figure 1. Configurations are pairs (C, σ) of a command and a state. There are transitions from one configuration to another as well as transitions from a configuration to **abort** denoting execution errors such as accessing an unallocated memory location. Parallel composition interleaves executions of its two components, while atomic commands execute their body, C, in one transition. In the premise of Atom, $\to^*$ stands for zero or more $\to$ transitions. [2]

Separation logic assertions include Boolean expressions, all the classical connectives, first order quantification, and five assertions pertinent to separation logic. These are the empty heap assertion (**emp**), the points-to assertion ($E_1 \mapsto E_2$) indicating that the heap consists of a single memory cell with address E_1 and contents E_2, separating conjunction ($*$), separating implication ($-\!\!*$), and an iterative version of separating conjunction ($\circledast$):

$$\begin{array}{rl} P, Q, R, J ::= & B \mid P \vee Q \mid P \wedge Q \mid \neg P \mid P \Rightarrow Q \mid \forall x.\ P \mid \exists x.\ P \\ & \mid \mathbf{emp} \mid E_1 \mapsto E_2 \mid P * Q \mid P -\!\!* Q \mid \circledast_{i \in \mathcal{I}} P_i \end{array}$$

Assertions denote sets of states. Their semantics is given as a modelling relation, $s, h \models P$, stating that the state (s, h) satisfies the assertion P.

$$\begin{array}{lcl} s, h \models \mathbf{emp} & \stackrel{\text{def}}{\iff} & \mathbf{dom}(h) = \emptyset \\ s, h \models E \mapsto E' & \stackrel{\text{def}}{\iff} & \mathbf{dom}(h) = [\![E]\!](s) \wedge h([\![E]\!](s)) = [\![E']\!](s) \\ s, h \models P * Q & \stackrel{\text{def}}{\iff} & \exists h_1, h_2.\ h = h_1 \uplus h_2 \wedge (s, h_1 \models P) \wedge (s, h_2 \models Q) \\ s, h \models P -\!\!* Q & \stackrel{\text{def}}{\iff} & \forall h_1.\ \mathsf{def}(h \uplus h_1) \wedge (s, h_1 \models P) \implies (s, h \uplus h_1 \models Q) \end{array}$$

Here, $h_1 \uplus h_2$ stands for the union of the two heaps h_1 and h_2 and is undefined unless $\mathbf{dom}(h_1) \cap \mathbf{dom}(h_2) = \emptyset$. We write $\mathsf{def}(X)$ to say that X is defined. The other assertions are interpreted classically. Finally, we write $E \mapsto -$ as a shorthand for $\exists v.\ E \mapsto v$ where $v \notin \mathsf{fv}(E)$.

An important class of assertions are the so-called *precise* assertions, which are assertions satisfied by at most one subheap of any given heap. Formally, if there are satisfied by two such heaps, h_1 and h_1', the two must be equal:

Definition 2.1 An assertion, P, is *precise* iff for all h_1, h_2, h_1', and h_2', if $\mathsf{def}(h_1 \uplus h_2)$ and $h_1 \uplus h_2 = h_1' \uplus h_2'$ and $s, h_1 \models P$ and $s, h_1' \models P$, then $h_1 = h_1'$.

[2] Normally, in addition to Atom, there should be another rule for infinite executions for the body of atomic blocks. For simplicity, we omit such a rule. In §5, we will present a different semantics that does not involve $\to^*$ and does not suffer from this problem.

$$\frac{}{J \vdash \{P\}\ \mathbf{skip}\ \{P\}}\ (\textsc{Skip})$$

$$\frac{x \notin \mathsf{fv}(J)}{J \vdash \{[E/x]P\}\ x := E\ \{P\}}\ (\textsc{Assign})$$

$$\frac{x \notin \mathsf{fv}(E, E', J)}{J \vdash \{E \mapsto E'\}\ x := [E]\ \{E \mapsto E' \wedge x = E'\}}\ (\textsc{Read})$$

$$\frac{}{J \vdash \{E \mapsto -\}\ [E] := E'\ \{E \mapsto E'\}}\ (\textsc{Write})$$

$$\frac{x \notin \mathsf{fv}(E, J)}{J \vdash \{\mathbf{emp}\}\ x{:=}\mathbf{alloc}(E)\ \{x \mapsto E\}}\ (\textsc{Alloc})$$

$$\frac{}{J \vdash \{E \mapsto -\}\ \mathbf{dispose}(E)\ \{\mathbf{emp}\}}\ (\textsc{Free})$$

$$\frac{J \vdash \{P\}\ C_1\ \{Q\} \quad J \vdash \{Q\}\ C_2\ \{R\}}{J \vdash \{P\}\ C_1; C_2\ \{R\}}\ (\textsc{Seq})$$

$$\frac{J \vdash \{P \wedge B\}\ C_1\ \{Q\} \quad J \vdash \{P \wedge \neg B\}\ C_2\ \{Q\}}{J \vdash \{P\}\ \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \{Q\}}\ (\textsc{If})$$

$$\frac{J \vdash \{P \wedge B\}\ C\ \{P\}}{J \vdash \{P\}\ \mathbf{while}\ B\ \mathbf{do}\ C\ \{P \wedge \neg B\}}\ (\textsc{While})$$

$$\frac{\mathbf{emp} \vdash \{P * J\}\ C\ \{Q * J\}}{J \vdash \{P\}\ \mathbf{atomic}\ C\ \{Q\}}\ (\textsc{Atom})$$

$$\frac{J \vdash \{P_1\}\ C_1\ \{Q_1\} \quad J \vdash \{P_2\}\ C_2\ \{Q_2\} \quad \mathsf{fv}(J, P_1, C_1, Q_1) \cap \mathsf{wr}(C_2) = \emptyset \quad \mathsf{fv}(J, P_2, C_2, Q_2) \cap \mathsf{wr}(C_1) = \emptyset}{J \vdash \{P_1 * P_2\}\ C_1 \| C_2\ \{Q_1 * Q_2\}}\ (\textsc{Par})$$

$$\frac{J * R \vdash \{P\}\ C\ \{Q\}}{J \vdash \{P * R\}\ C\ \{Q * R\}}\ (\textsc{Share})$$

$$\frac{J \vdash \{P\}\ C\ \{Q\} \quad \mathsf{fv}(R) \cap \mathsf{wr}(C) = \emptyset}{J \vdash \{P * R\}\ C\ \{Q * R\}}\ (\textsc{Frame})$$

$$\frac{J \vdash \{P\}\ C\ \{Q\} \quad P' \Rightarrow P \quad Q \Rightarrow Q'}{J \vdash \{P'\}\ C\ \{Q'\}}\ (\textsc{Conseq})$$

$$\frac{J \vdash \{P_1\}\ C\ \{Q_1\} \quad J \vdash \{P_2\}\ C\ \{Q_2\}}{J \vdash \{P_1 \vee P_2\}\ C\ \{Q_1 \vee Q_2\}}\ (\textsc{Disj})$$

$$\frac{J \vdash \{P\}\ C\ \{Q\} \quad x \notin \mathsf{fv}(C)}{J \vdash \{\exists x.\ P\}\ C\ \{\exists x.\ Q\}}\ (\textsc{Ex})$$

$$\frac{J \vdash \{P_1\}\ C\ \{Q_1\} \quad J \vdash \{P_2\}\ C\ \{Q_2\} \quad J \text{ precise}}{J \vdash \{P_1 \wedge P_2\}\ C\ \{Q_1 \wedge Q_2\}}\ (\textsc{Conj})$$

Fig. 2. Concurrent separation logic proof rules.

CSL judgments are of the form, $J \vdash \{P\}\ C\ \{Q\}$, where J is known as the resource invariant, P as the precondition, and Q as the postcondition. Informally, these specifications say that if C is executed from an initial state satisfying $P * J$, then J will be satisfied throughout execution and the final state (if the command terminates) will satisfy $Q * J$. There is also an ownership reading attached to the specifications saying that the command 'owns' the state described by its precondition: the command can change it and can assume that no other parallel thread can change it. In contrast, the state described by J can be changed by other concurrently executing threads. The only guarantee is that it will always satisfy the resource invariant, J.

The proof rules are shown in Figure 2. Among the proof rules, some are particularly noteworthy. Read and Write both require that the memory cell accessed is part of the precondition: this ensures that the cell is allocated (and hence, the access will be safe) and that no other thread is accessing it concurrently. Atom allows the body of atomic blocks to use the resource invariant, J, and requires them to re-establish it at the postcondition. Par allows us to compose two threads in parallel if and only if their preconditions describe disjoint parts of the heap. This prevents data races on memory locations. The side-conditions ensure that there are also no data races on program variables—here, fv returns the set of free variables of a command or an assertion, whereas $\mathsf{wr}(C)$ returns the set of variables being as-

signed to by the command C.[3] SHARE allows us at any time to extend the resource invariant by separatingly conjoining part of the local state, R. FRAME allows us to ignore part of the local state, the *frame* R, which is not used by the command, ensuring that R is still true at the postcondition.

Finally, the conjunction rule, CONJ, has a perhaps surprising side-condition. This side-condition is necessary for soundness as illustrated by Reynolds's counterexample [15, §11]. Most presentations require precise J's in all judgments. This, however, is unnecessary: only CONJ needs precision.

3 The Meaning of CSL Judgments

We define the semantics of CSL judgments in terms of an auxiliary predicate, $\mathsf{safe}_n(C, s, h, J, Q)$, stating that the command C executing with a stack, s, and a *local* heap, h, is safe with respect to the resource invariant J and the post-condition Q for up to n execution steps. A CSL judgment, $J \models \{P\}\ C\ \{Q\}$, simply says that the program C is safe with respect to J and Q for every initial local state satisfying the precondition, P, and for any number of steps:

Definition 3.1 (Configuration Safety)

- $\mathsf{safe}_0(C, s, h, J, Q)$ holds always.
- $\mathsf{safe}_{n+1}(C, s, h, J, Q)$ holds if and only if

(i) if $C = \mathbf{skip}$, then $s, h \models Q$; and

(ii) for all h_J and h_F, if $s, h_J \models J$ and $(h \uplus h_J \uplus h_\mathrm{F})$ is defined, then $C, (s, h \uplus h_J \uplus h_\mathrm{F}) \not\to \mathbf{abort}$; and

(iii) for all C', h_J, h_F, h' and s', if $s, h_J \models J$, and $(h \uplus h_J \uplus h_\mathrm{F})$ is defined, and $C, (s, h \uplus h_J \uplus h_\mathrm{F}) \to C', (s', h')$, then there exist h'' and h'_J such that $h' = h'' \uplus h'_J \uplus h_\mathrm{F}$ and $s', h'_J \models J$ and $\mathsf{safe}_n(C', s', h'', J, Q)$.

Definition 3.2 $J \models \{P\}\ C\ \{Q\}$ if and only if for all n, s, and h, if $s, h \models P$, then $\mathsf{safe}_n(C, s, h, J, Q)$.

Intuitively, any configuration is safe for zero steps. For $n+1$ steps, it must (i) satisfy the postcondition if it is a terminal configuration, (ii) not abort, and (iii) after any step, re-establish the resource invariant and be safe for another n steps. The number of steps merely ensures the definition is structurally decreasing.

In more detail, h is the part of the heap that is 'owned' by the command: the command can update h and no other command can access it in parallel. In conditions (ii) and (iii), h_J represents the part of the heap that is shared among threads, and must hence satisfy the resource invariant. So, condition (iii) ensures that after the transition a new such component, h'_J, can be found. Finally, h_F represents the remaining part of the heap owned by the rest of the system. In condition (ii), the command must not abort regardless of what that remaining part

[3] For simplicity, we impose draconian variable side-conditions. In effect, only heap cells may be shared among threads, as J cannot mention any updateable variables.

is. In condition (*iii*), the command must not change any part of the heap that could be owned by another thread. Therefore, h_F must be a subheap of the new heap h'.

Safety Monotonicity & Frame Property. The purpose of the h_F quantifications is to admit the frame rule. In condition (*ii*), h_F essentially plays the role of "safety monotonicity" [20], which requires that if (C, h) is safe (i.e., does not abort), then $(C, h \uplus h_F)$ is also safe.

Similarly, in condition (*iii*), h_F plays the role of the "frame property" [20], which requires that whenever (C, h) is safe and $C, (s, h \uplus h_F) \to C', (s', h')$, then there exists h'' such that $C, (s, h) \to C', (s', h'')$ and $h' = h'' \uplus h_F$. Condition (*iii*) does not quite imply the frame property, as it does not require that $C, (s, h) \to C', (s', h'')$. It rather takes the transition $C, (s, h) \to C', (s', h'')$ into account even though it might not be present.

The difference is quite subtle. In particular, if the operational semantics satisfies the safety monotonicity and frame properties (which it does in our case), we can drop the h_F quantification. (See [18] for a proof.) Having the quantification, however, is crucial for some of the CSL extensions (see §6) and even simplifies some of the proofs for the normal CSL (Par and Frame).

Discussion. A nice aspect of Definition 3.1 is that the straightforward lemmas about safety of compound commands are usually already inductive, thereby rendering the otherwise most challenging part of soundness proofs trivial. The only exception is Lemma 5.3 about the resource declaration rule (for an extension of Definition 3.1 to handle multiple named CCRs), which was arguably the most intellectually challenging part of the proof.

A second benefit is that we do not strictly require an abort semantics to prove the soundness of CSL: if we drop condition (*ii*) from Definition 3.1, we can still prove the soundness of CSL without ever referring to an abort semantics. In contrast, proofs relying on the safety monotonicity and frame properties heavily depend on an abort semantics (e.g., [3,4,5,6,9,10]).

4 Soundness Proof

We start with some basic –but important– properties of the semantics. In the following, let $[s \sim s']^X$ stand for $\forall x \in X.\ s(x) = s'(x)$ and $\overline{X}$ for the complement of set X.

Proposition 4.1 *If $C, (s, h) \to C', (s', h')$, then $\mathsf{fv}(C') \subseteq \mathsf{fv}(C)$, $\mathsf{wr}(C') \subseteq \mathsf{wr}(C)$, and $[s \sim s']^{\overline{\mathsf{wr}(C)}}$.*

Proposition 4.2 *(i) If $[s \sim s']^{\mathsf{fv}(E)}$, then $[\![E]\!](s) = [\![E]\!](s')$.*

(ii) If $[s \sim s']^{\mathsf{fv}(B)}$, then $[\![B]\!](s) = [\![B]\!](s')$.

(iii) If $[s \sim s']^{\mathsf{fv}(P)}$, then $s, h \models P$ if and only if $s', h \models P$.

(iv) If $[s \sim s']^{\mathsf{fv}(C)}$ and $C, s \to \mathbf{abort}$, then $C, s' \to \mathbf{abort}$.

(v) If $X \supseteq \mathsf{fv}(C)$ and $[s \sim s']^X$ and $C, s \to C_1, s_1$, then there exist s'_1 such that

$C, s' \to C'_1, s'_1$ *and* $[s_1 \sim s'_1]^X$.

Now, consider Definition 3.1. By construction, safe is monotonic with respect to n: if a configuration is safe for a number of steps, n, it is also safe for a smaller number of steps, m. (This is proved by induction on m.)

Lemma 4.3 *If* $\mathsf{safe}_n(C, s, h, J, Q)$ *and* $m \leq n$, *then* $\mathsf{safe}_m(C, s, h, J, Q)$.

Further, as a corollary of Proposition 4.2, $\mathsf{safe}_n(C, s, h, J, Q)$ depends only on the values of variables that are mentioned in C, J, Q.

Lemma 4.4 *If* $\mathsf{safe}_n(C, s, h, J, Q)$ *and* $[s \sim s']^{\mathsf{fv}(C,J,Q)}$, *then* $\mathsf{safe}_n(C, s', h, J, Q)$.

The soundness theorem for CSL is the following:

Theorem 4.5 (CSL Soundness) *If* $J \vdash \{P\}\ C\ \{Q\}$, *then* $J \models \{P\}\ C\ \{Q\}$.

Our proof strategy is to prove that each proof rule is a sound implication if we replace all the $\vdash$ by $\models$. Then, the theorem follows by a straightforward rule induction. For brevity, we only show the proofs of the most interesting rules.

(Skip) The rule for **skip** follows immediately from the following lemma, whose proof is trivial because there are no transitions from **skip**.

Lemma 4.6 *For all* n, s, h, J, *and* Q, *if* $s, h \models Q$, *then* $\mathsf{safe}_n(\mathbf{skip}, s, h, J, Q)$.

(Atom) We need an auxiliary lemma for code executing in atomic blocks:

Lemma 4.7 *If* $\forall n.\ \mathsf{safe}_n(C, s, h, \mathbf{emp}, Q)$ *and* $\mathsf{def}(h \uplus h_{\mathrm{F}})$, *then*
(i) $\neg(C, (s, h \uplus h_{\mathrm{F}}) \to^* \mathbf{abort})$; *and*
(ii) if, moreover, $C, (s, h \uplus h_{\mathrm{F}}) \to^* \mathbf{skip}, (s', h')$, *then there exists* h'' *such that* $h' = h'' \uplus h_{\mathrm{F}}$ *and* $s', h'' \models Q$.

This lemma is proved by an induction on the length of the $\to^*$ traces, noting that when $J = \mathbf{emp}$, the second clause of Definition 3.1 simplifies to $\mathsf{safe}_{n+1}(C, s, h, \mathbf{emp}, Q)$ if and only if

(*i*) if $C = \mathbf{skip}$, then $s, h \models Q$; and

(*ii*) for all h_{F}, if $\mathsf{def}(h \uplus h_{\mathrm{F}})$, then $C, (s, h \uplus h_{\mathrm{F}}) \not\to \mathbf{abort}$; and

(*iii*) for all h_{F}, C', s', h', if $C, (s, h \uplus h_{\mathrm{F}}) \to C', (s', h')$, then there exists h'' such that $h' = h'' \uplus h_{\mathrm{F}}$ and $\mathsf{safe}_n(C', s', h'', J, Q)$.

The main lemma for atomic commands is as follows:

Lemma 4.8 *If* $\mathbf{emp} \models \{P * J\}\ C\ \{Q * J\}$, *then* $J \models \{P\}\ \mathbf{atomic}\ C\ \{Q\}$.

Proof. Assume (*) $\mathbf{emp} \models \{P * J\}\ C\ \{Q * J\}$, and pick arbitrary $s, h \models P$ and n. We have to show that $\mathsf{safe}_n(\mathbf{atomic}\ C, s, h, J, Q)$. If $n = 0$, this is trivial; so consider $n = m + 1$. Condition (*i*) is trivial as $\mathbf{atomic}\ C \neq \mathbf{skip}$.
(*ii*) If $\mathbf{atomic}\ C, (s, h \uplus h_J \uplus h_{\mathrm{F}}) \to \mathbf{abort}$, then from the operational semantics $C, (s, h \uplus h_J \uplus h_{\mathrm{F}}) \to^* \mathbf{abort}$, which with Lemma 4.7 contradicts (*).

(*iii*) The only way for **atomic** $C, (s, h \uplus h_J \uplus h_F) \to C', (s', h')$ is if $C' = \textbf{skip}$ and $C, (s, h \uplus h_J \uplus h_F) \to^* \textbf{skip}, (s', h')$. Hence, from assumption (*) and Lemma 4.7, there exists h'' such that $h' = h'' \uplus h_F$ and $s', h'' \models Q * J$. So, there exist h''' and h'_J such that $h'' = h''' \uplus h'_J$, $s', h''' \models Q$, and $s', h'_J \models J$. Finally, from Lemma 4.6, we get $\mathsf{safe}_m(\textbf{skip}, s', h''', J, Q)$. □

(Par) For parallel composition, we need the following auxiliary lemma:

Lemma 4.9 *If* $\mathsf{safe}_n(C_1, s, h_1, J, Q_1)$, $\mathsf{safe}_n(C_2, s, h_2, J, Q_1)$, $h_1 \uplus h_2$ *is defined,* $\mathsf{fv}(J, C_1, Q_1) \cap \mathsf{wr}(C_2) = \emptyset$, *and* $\mathsf{fv}(J, C_2, Q_2) \cap \mathsf{wr}(C_1) = \emptyset$, *then* $\mathsf{safe}_n(C_1 \| C_2, s, h_1 \uplus h_2, J, Q_1 * Q_2)$.

Proof. By induction on n. In the inductive step, we know $IH(n) \stackrel{\text{def}}{=}$

$$\begin{aligned}\forall C_1, h_1, C_2, h_2.\ &\mathsf{safe}_n(C_1, s, h_1, J, Q_1) \wedge \mathsf{safe}_n(C_2, s, h_2, J, Q_1) \wedge \mathsf{def}(h_1 \uplus h_2) \\ &\wedge \mathsf{fv}(J, C_1, Q_1) \cap \mathsf{wr}(C_2) = \emptyset \wedge \mathsf{fv}(J, C_2, Q_2) \cap \mathsf{wr}(C_1) = \emptyset \\ \implies &\mathsf{safe}_n(C_1 \| C_2, s, h_1 \uplus h_2, J, Q_1 * Q_2)\end{aligned}$$

and we have to show $IH(n+1)$. So, pick arbitrary C_1, h_1, C_2, h_2 and assume (1) $\mathsf{safe}_{n+1}(C_1, s, h_1, J, Q_1)$, (2) $\mathsf{safe}_{n+1}(C_2, s, h_2, J, Q_2)$, (3) $\mathsf{def}(h_1 \uplus h_2)$ and (4) the variable side-conditions, and try to show $\mathsf{safe}_{n+1}(C_1 \| C_2, s, h_1 \uplus h_2, J, Q_1 * Q_2)$. Condition (*i*) is trivial.
(*ii*) If $C_1 \| C_2, (s, h_1 \uplus h_2 \uplus h_J \uplus h_F) \to \textbf{abort}$, then according to the operational semantics $C_1, (s, h_1 \uplus h_2 \uplus h_J \uplus h_F) \to \textbf{abort}$ or $C_2, (s, h_1 \uplus h_2 \uplus h_J \uplus h_F) \to \textbf{abort}$, contradicting our assumptions (1) and (2).
(*iii*) Pick arbitrary C', h_J, h_F, s', h' such that $s, h_J \models J$, $(h_1 \uplus h_2 \uplus h_J \uplus h_F)$ is defined, and $C_1 \| C_2, (s, h_1 \uplus h_2 \uplus h_J \uplus h_F) \to (C', s', h')$. The operational semantics has three possible transitions for $C_1 \| C_2$.
Case (Par1). $C' = C'_1 \| C_2$ and $C_1, (s, h_1 \uplus h_2 \uplus h_J \uplus h_F) \to C'_1, (s', h')$.
From (1), there exist h'_1 and h'_J such that $h' = h'_1 \uplus h'_J \uplus (h_2 \uplus h_F)$, $s', h'_J \models J$, and $\mathsf{safe}_n(C'_1, s', h'_1, J, Q_1)$.

From (2) and Proposition 4.3, we have $\mathsf{safe}_n(C_2, s, h_2, J, Q_2)$. Then, from Propositions 4.4 and 4.1, and assumption (4), we have $\mathsf{safe}_n(C_2, s', h_2, J, Q_2)$. Also, from Proposition 4.1 and (4), $\mathsf{fv}(C'_1, Q_1) \cap \mathsf{wr}(C_2) = \emptyset$ and $\mathsf{fv}(C_2, Q_2) \cap \mathsf{wr}(C'_1) = \emptyset$, and hence from $IH(n)$, $\mathsf{safe}_n(C'_1 \| C_2, s', h'_1 \uplus h_2, J, Q_1 * Q_2)$.
Case (Par2). This case is completely symmetric.
Case (Par3). $C_1 = C_2 = C' = \textbf{skip}$, $h' = h_1 \uplus h_2 \uplus h_J \uplus h_F$. From (1) and (2), unfolding the definition of safe, we have that $s, h_1 \models Q_1$ and $s, h_2 \models Q_2$. So, $s, h_1 \uplus h_2 \models Q_1 * Q_2$, and, from Lemma 4.6, $\mathsf{safe}_n(\textbf{skip}, s, h_1 \uplus h_2)$. □

(Frame) The frame rule is a cut-down version of the parallel composition rule. It follows directly from the following lemma:

Lemma 4.10 *If* $\mathsf{safe}_n(C, s, h, J, Q)$, $\mathsf{fv}(R) \cap \mathsf{wr}(C) = \emptyset$, $h \uplus h_R$ *is defined, and* $s, h_R \models R$, *then* $\mathsf{safe}_n(C, s, h \uplus h_R, J, Q * R)$.

Proof. By induction on n. The base case is trivial. For the inductive step, assume

(*) $\mathsf{safe}_{n+1}(C, s, h, J, Q)$, (†) $\mathsf{fv}(R) \cap \mathsf{wr}(C) = \emptyset$, and (‡) $s, h_R \models R$.
Now, we have to prove $\mathsf{safe}_{n+1}(C, s, h \uplus h_R, J, Q * R)$.
(*i*) From (*), we get $s, h \models Q$ and so, using (‡), $s, h \uplus h_R \models Q * R$.
(*ii*) Pick h_J and h_F. Then, from (*), $C, (s, h \uplus h_R \uplus h_J \uplus h_\mathrm{F}) \not\to \mathbf{abort}$.
(*iii*) If $C, (s, h \uplus h_R \uplus h_J \uplus h_\mathrm{F}) \to C', (s', h')$, then from (*), there exist h'', h'_J such that $h' = h'' \uplus h'_J \uplus (h_R \uplus h_\mathrm{F})$ and $s', h'_J \models J$ and $\mathsf{safe}_n(C', s', h'', J, Q)$. Now, from (†), (‡), Prop. 4.1 and 4.2, we get $s', h_R \models R$ and $\mathsf{fv}(R) \cap \mathsf{wr}(C') = \emptyset$. Therefore, from the induction hypothesis, $\mathsf{safe}_n(C', s', h' \uplus h_R, J, Q * R)$. □

(Share) We need the following lemma, which is similar to the previous one.

Lemma 4.11 *If* $\mathsf{safe}_n(C, s, h, J * R, Q)$, $h \uplus h_R$ *is defined, and* $s, h_R \models R$, *then* $\mathsf{safe}_n(C, s, h \uplus h_R, J, Q * R)$.

Proof. By induction on n. For the inductive step,
(*i*) From our assumptions, $s, h \models Q$ and $s, h_R \models R$, and so $s, h \uplus h_R \models Q * R$.
(*ii*) $C, (s, h \uplus h_R \uplus h_J \uplus h_\mathrm{F}) \not\to \mathbf{abort}$ follows directly from our assumptions.
(*iii*) If $C, (s, h \uplus h_R \uplus h_J \uplus h_\mathrm{F}) \to C', (s', h')$, then from our assumptions, there exist h'', h'_{JR} such that $h' = h'' \uplus h'_{JR} \uplus h_\mathrm{F}$ and $s', h'_{JR} \models J * R$ and $\mathsf{safe}_n(C', s', h'', J * R, Q)$. From the definition of $*$, there exist h'_J and h'_R such that $h'_{JR} = h'_J \uplus h'_R$ and $s', h'_J \models J$ and $s', h'_R \models R$. Therefore, from the induction hypothesis, $\mathsf{safe}_n(C', s', h'' \uplus h'_R, J, Q * R)$, as required. □

(Conj) Now consider the conjunction rule. Its soundness rests upon the validity of the following implication:

$$\mathsf{safe}_n(C, s, h, J, Q_1) \wedge \mathsf{safe}_n(C, s, h, J, Q_2) \implies \mathsf{safe}_n(C, s, h, J, Q_1 \wedge Q_2)\,.$$

Naturally, one would expect to prove this implication by induction on n with an induction hypothesis quantifying over all C and h. The base case is trivial; so consider the $n + 1$ case. The first two subcases are easy; so consider subcase (*iii*). From the first assumption, we know that there exist h^1 and h^1_J such that $h' = h^1 \uplus h^1_J$ and $h^1_J \models J$ and $\mathsf{safe}_n(C', h^1, J, Q_1)$. Similarly, from the first assumption, there exist h^2 and h^2_J such that $h' = h^2 \uplus h^2_J$ and $h^2_J \models J$ and $\mathsf{safe}_n(C', h^2, J, Q_2)$, but, in general, we do not know that $h^1 = h^2$ which would allow us to complete the proof. Since, however, J must be precise, then (from Definition 2.1) $h^1_J = h^2_J$, and since $\uplus$ is cancellative, we also have $h^1 = h^2$ and the result follows by applying the induction hypothesis. □

5 Multiple Resources & Data Race Freedom

In this section, we consider the programming language used by O'Hearn [15] and Brookes [3], which has multiple named resources and permits the execution of critical regions acting on different resources to go on in parallel. The programming language replaces atomic commands, **atomic** C, with two new constructs and an intermediate

$$\frac{(C_1,\sigma)\to(C_1',\sigma') \quad \mathsf{locked}(C_1')\cap\mathsf{locked}(C_2)=\emptyset}{(C_1\|C_2,\sigma)\to(C_1'\|C_2,\sigma')}\ (\textsc{Par1}) \qquad \frac{(C_2,\sigma)\to(C_2',\sigma') \quad \mathsf{locked}(C_1)\cap\mathsf{locked}(C_2')=\emptyset}{(C_1\|C_2,\sigma)\to(C_1\|C_2',\sigma')}\ (\textsc{Par2})$$

$$\frac{(\mathsf{accesses}(C_1,s)\cap\mathsf{writes}(C_2,s))\cup(\mathsf{accesses}(C_2,s)\cap\mathsf{writes}(C_1,s))\neq\emptyset}{(C_1\|C_2,(s,h))\to\mathbf{abort}}\ (\textsc{RaceDetect})$$

(Res1)	$\mathbf{resource}\ r\ \mathbf{in}\ C,\sigma \to \mathbf{resource}\ r\ \mathbf{in}\ C',\sigma'$	if $C,\sigma\to C',\sigma'$
(Res2)	$\mathbf{resource}\ r\ \mathbf{in}\ \mathbf{skip},\sigma \to \mathbf{skip},\sigma$	
(With1)	$\mathbf{with}\ r\ \mathbf{when}\ B\ \mathbf{do}\ C,\sigma \to \mathbf{within}\ r\ \mathbf{do}\ C,\sigma$	if $\sigma=(s,h)$ and $[\![B]\!](s)$
(With2)	$\mathbf{within}\ r\ \mathbf{do}\ C,\sigma \to \mathbf{within}\ r\ \mathbf{do}\ C',\sigma'$	if $C,\sigma\to C',\sigma'$
(With3)	$\mathbf{within}\ r\ \mathbf{do}\ \mathbf{skip},\sigma \to \mathbf{skip},\sigma$	
(ResA)	$\mathbf{resource}\ r\ \mathbf{in}\ C,\sigma \to \mathbf{abort}$	if $C,\sigma\to\mathbf{abort}$
(WithA)	$\mathbf{within}\ r\ \mathbf{do}\ C,\sigma \to \mathbf{abort}$	if $C,\sigma\to\mathbf{abort}$

Fig. 3. Operational semantics for CCRs.

command form:

$$C ::= \ldots \mid \mathbf{resource}\ r\ \mathbf{in}\ C \mid \mathbf{with}\ r\ \mathbf{when}\ B\ \mathbf{do}\ C \mid \mathbf{within}\ r\ \mathbf{do}\ C$$

The first declares a new mutual exclusion lock, r, known as a *resource* or a *resource bundle* in CSL terminology. The second construct denotes a conditional critical region (CCR) which runs in isolation with respect to any other CCRs with the same lock. Executing a CCR blocks until the resource is available and the condition B is true, and then executes the body C in isolation to other CCRs acting on the same resource. This is achieved by holding a lock for the duration of testing whether B is satisfied and the execution of its body. Finally, $\mathbf{within}\ r\ \mathbf{do}\ C$ represents a partially executed CCR: one that has acquired the lock, tested the condition, and still has to execute C. We define $\mathsf{locked}(C)$ to be the set of regions syntactically locked by C: those r for which C contains a $\mathbf{within}\ r\ \mathbf{do}\ C'$ subterm.

The operational semantics is given by the rules of Figure 1 (excluding Par1, Par2, Atom, AtomA) and the new rules shown in Figure 3. The reduction rules for parallel composition (Par1, Par2) have been adapted to check that two threads do not hold the same lock at the same time. This was unnecessary in the simpler setting because atomic blocks executed in one step.

To show absence of data races, we have added a rule (RaceDetect) that aborts whenever a data race is observed. Here, the functions $\mathsf{accesses}(C,s)$ and $\mathsf{writes}(C,s)$ return the set of heap locations accessed or modified by C respectively. Their formal definitions can be found in [18].

CSL judgments for multiple resources are of the form $\Gamma\vdash\{P\}\ C\ \{Q\}$, where Γ is a mapping from resource names, r, to their corresponding resource invariants, which are normal assertions. We have the proof rules from Figure 2 –except (Atom) and (Share)– uniformly replacing J by Γ. In addition, we have the following two rules concerning resource declarations and CCRs:

$$\frac{\Gamma,r:J\vdash\{P\}\ C\ \{Q\}}{\Gamma\vdash\{P*J\}\ \mathbf{resource}\ r\ \mathbf{in}\ C\ \{Q*J\}} \qquad \frac{\Gamma\vdash\{(P*J)\wedge B\}\ C\ \{Q*J\}}{\Gamma,r:J\vdash\{P\}\ \mathbf{with}\ r\ \mathbf{when}\ B\ \mathbf{do}\ C\ \{Q\}}$$

The first rule is analogous to the Share rule: it allows us to declare a new

resource bundle, r, and associate a resource invariant with it. The second rule is analogous to Atom, allowing the verifier to assume that the relevant resource invariant holds separately at the beginning of the CCR and requiring him to be re-establish it at the end of the CCR.

The definition of configuration safety is adapted as follows:

Definition 5.1

- $\mathsf{safe}_0(C, s, h, \Gamma, Q)$ holds always.
- $\mathsf{safe}_{n+1}(C, s, h, \Gamma, Q)$ if and only if

(*i*) if $C = \mathbf{skip}$, then $s, h \models Q$; and
(*ii*) for all h_F, if $(h \uplus h_\mathrm{F})$ is defined, then $C, (s, h \uplus h_\mathrm{F}) \not\to \mathbf{abort}$; and
(*iii*) $\mathsf{accesses}(C, s) \subseteq \mathbf{dom}(h)$; and
(*iv*) for all C', h_Γ, h_F, s', h', L', if $s, h_\Gamma \models \circledast_{r \in \mathsf{locked}(C') \setminus \mathsf{locked}(C)} \Gamma(r)$, and $(C, (s, h \uplus h_\Gamma \uplus h_\mathrm{F}), L) \to (C', (s', h'), L')$, then there exist h'' and h'_Γ such that $h' = h'' \uplus h'_\Gamma \uplus h_\mathrm{F}$ and $s', h'_\Gamma \models \circledast_{r \in \mathsf{locked}(C) \setminus \mathsf{locked}(C')} \Gamma(r)$ and $\mathsf{safe}_n(C', s', h'', \Gamma, Q)$.

Similar to Definition 3.1, here h is the part of the heap owned by the command; h_Γ is the part that belongs to definitely unacquired resources (since any memory cells belonging to a currently acquired resource are part of the local heap of the thread that holds the lock for that resource); and h_F represents the *frame*, namely memory cells belonging to other parts of the system. The set $\mathsf{locked}(C') \setminus \mathsf{locked}(C)$ represents the set of locks that have been acquired by the transition from C to C': for all of those, we assume that the resource invariant holds. Conversely, $\mathsf{locked}(C') \setminus \mathsf{locked}(C)$ is the set of locks released by the transition: for all those, we check that the resource invariant is established. Finally, the new conjunct $\mathsf{accesses}(C, s) \subseteq \mathbf{dom}(h)$ is included so that we can show that safe programs do not have any data races.

As before, the semantics of triples is given in terms of the safe predicate:

Definition 5.2 $\Gamma \models \{P\}\ C\ \{Q\}$ if and only if for all n, s, h, if $s, h \models P$ then $\mathsf{safe}_n(C, L, s, h, \Gamma, Q)$.

The proof of soundness proceeds as before and has been fully formalised in Isabelle/HOL. See [18] for details. We say that a command is well-formed if and only if it does not have two different subcommands simultaneously having acquired the same CCR lock, as this cannot occur in a normal execution. To prove the soundness of the two new rules, we use the following lemmas:

Lemma 5.3 *If* $\mathsf{safe}_n(C, s, h, (\Gamma, r : R), Q)$ *and* C *is well-formed and* $\mathsf{fv}(R) \cap \mathsf{wr}(C) = \emptyset$*, then*

(i) if $r \notin \mathsf{locked}(C)$*, then for all* h_R*, if* $\mathbf{dom}(h) \cap \mathbf{dom}(h_R) = \emptyset$ *and* $s, h_R \models R$*, then* $\mathsf{safe}_n(\mathbf{resource}\ r\ \mathbf{in}\ C, s, h \uplus h_R, \Gamma, Q * R)$*; and*

(ii) if $r \in \mathsf{locked}(C)$*, then* $\mathsf{safe}_n(\mathbf{resource}\ r\ \mathbf{in}\ C, s, h, \Gamma, Q * R)$.

Lemma 5.4 *If* $\mathsf{safe}_n(C, s, h, \Gamma, Q * R)$ *and* $\mathbf{within}\ r\ \mathbf{do}\ C$ *is well-formed, then* $\mathsf{safe}_n(\mathbf{within}\ r\ \mathbf{do}\ C, s, h, (\Gamma, r : R), Q)$.

The proofs of these lemmas can be found in [18]. Our formalisation also covers local variable declarations as well as the auxiliary variable elimination rule as in Brookes's original proof [3].

6 Permissions

Permissions [2,1] are an extension to the standard heap model that enables read-sharing between parallel threads. Consider, for example, the Hoare triple: $\{10 \mapsto -\}\ x := [10] \parallel y := [10]\ \{10 \mapsto -\}$. Standard CSL cannot verify that the program satisfies its specification because to read from [10] both threads must know that the cell is allocated (i.e., have $10 \mapsto -$ as a precondition), but the assertion $10 \mapsto - * 10 \mapsto -$ (required by the parallel composition rule) is unsatisfiable. With permissions, one can instead split $10 \mapsto -$ into two half permissions, $(10 \overset{0.5}{\mapsto} -) * (10 \overset{0.5}{\mapsto} -)$, and give one to each thread. The idea then is such partial permissions are read-only: they allow the cell to be read, but not updated. This is captured by the following new proof rule:

$$\frac{x \notin \mathsf{fv}(E, E', E'', J)}{J \vdash \{E \overset{E'}{\mapsto} E''\}\ x := [E]\ \{E \overset{E'}{\mapsto} E'' \wedge x = E''\}} \qquad (\textsc{Read2})$$

At the postcondition, the two half permissions are collected and joined to give back $10 \mapsto -$, which is just shorthand notation for $10 \overset{1}{\mapsto} -$.

Permission models are sets, K, with a distinguished element, $\top \in K$, called *full* permission, and a commutative and associative partial operator, $\oplus$, denoting addition of two permissions, satisfying the following properties:

$$\forall k \in K.\ \neg\mathsf{def}(\top \oplus k) \quad \text{and} \quad \forall k \in K \setminus \{\top\}.\ \exists k' \in K.\ k \oplus k' = \top$$

The first equation says that $\top$ is the greatest permission, as it cannot be combined with any other permission. The second equation says that every non-full permission has a complement permission which when added to it gives full permission. The model we saw previously is known as *fractional permissions*. K is the set of numbers in the range $(0, 1]$, $\oplus$ is ordinary addition and is undefined when the result falls out of the range, and $\top = 1$. The complement of fractional permission k is simply $1 - k$.

To model a heap with permissions, we extend $\oplus$ to act on permission-value pairs as follows:

$$(k_1, v_1) \oplus (k_2, v_2) \overset{\text{def}}{=} \begin{cases} (k_1 \oplus k_2, v_1) & \text{if } v_1 = v_2 \text{ and } \mathsf{def}(k_1 \oplus k_2) \\ \textit{undefined} & \text{otherwise} \end{cases}$$

We also extend $\oplus$ to act on permission-heaps, $\mathsf{PH} \overset{\text{def}}{=} \mathsf{Loc} \rightharpoonup (\mathsf{Perm} \times \mathsf{Val})$, as follows. We take $h_1 \oplus h_2$ to be defined if and only if $h_1(a) \oplus h_2(a)$ is defined for all $a \in (\mathbf{dom}(h_1) \cap \mathbf{dom}(h_2))$. If $h_1 \oplus h_2$ is defined, it has domain $\mathbf{dom}(h_1) \cup \mathbf{dom}(h_2)$

with the following values:

$$(h_1 \oplus h_2)(a) \stackrel{\text{def}}{=} \begin{cases} h_1(a) \oplus h_2(a) & \text{if } a \in (\mathbf{dom}(h_1) \cap \mathbf{dom}(h_2)) \\ h_1(a) & \text{if } a \in (\mathbf{dom}(h_1) \setminus \mathbf{dom}(h_2)) \\ h_2(a) & \text{if } a \in (\mathbf{dom}(h_2) \setminus \mathbf{dom}(h_1)) \end{cases}$$

As expected, adding two permission-heaps is defined whenever for each location in their overlap, the heaps store the same value and permissions that can be added together. The result is a permission-heap whose permissions for the location in the overlap is just the sum of the individual heaps.

Assertions are now modelled by permission-heaps, PH. The new assertion form $E_1 \overset{E_2}{\mapsto} E_3$ has the following semantics:

$$s, h \models E_1 \overset{E_2}{\mapsto} E_3 \iff \mathbf{dom}(h) = \{[\![E_1]\!](s)\} \wedge h([\![E_1]\!](s)) = ([\![E_2]\!](s), [\![E_3]\!](s))$$

We can consider the set of concrete heaps as being a subset of that of permission-heaps by equating a concrete heap, h, with the permission-heap, h', which has the same domain as h and for each location $\ell \in \mathbf{dom}(h)$, $h'(\ell) = (\top, h(\ell))$. In other words, h' has full permission and the same values for every location in h, and no permission for any other location. Observe that every permission-heap can be extended to a normal heap:

$$\forall h \in \mathsf{PH}.\ \exists h_{\mathrm{F}} \in \mathsf{PH}.\ (h \oplus h_{\mathrm{F}}) \in \mathsf{Heap}\,.$$

This allows us to use the same definitions for safe predicate as we have seen already, uniformly replacing $\uplus$ with $\oplus$ and having the h, h_{F}, etc. range over permission-heaps rather than normal heaps. The definition is a bit subtle: as the operational semantics is defined over normal heaps, $(C, h \oplus h_J \oplus h_{\mathrm{F}}) \to \ldots$ makes sense only when $h \oplus h_J \oplus h_{\mathrm{F}}$ is a normal heap, a condition that is always possible to achieve as h_{F} is universally quantified.

The check in the safe definition that h_{F} does not change by transitions ensures that programs update the values only of heap locations they have full permission to, but allows threads to access any memory they partially own.

The soundness proof carries over to permission-heaps with no difficulty. See the machine-checked proof [18] for details.

7 RGSep

RGSep [19] is a more radical extension to CSL replacing resource invariants by two binary predicates, R and G, known as the rely and the guarantee respectively. As in CSL, the heap is logically divided into parts owned by threads and other parts owned by resources (and hence shared among threads, but accessed only within an atomic commands). The rely, R, describes the changes made to the resource-owned states by the environment (i.e., every other thread in the system that could execute

concurrently with the current command), whereas the guarantee, G, describes the changes made by the command itself.

Preconditions and postconditions are also changed into binary predicates describing both the local (thread-owned) and the shared (resource-owned) components of the state. We shall use the notation $s, (h_1, h_2) \models P$ to denote that the stack s and the heaps h_1 and h_2 satisfy the binary assertion P, whether it is a pre-, a post-, a rely or a guarantee condition. [4]

The $\mathsf{safe}^{\mathrm{RG}}$ predicate records not only the local heap, h_L, but also the shared heap, h_S, as this is needed for R and G:

Definition 7.1

- $\mathsf{safe}_0^{\mathrm{RG}}(C, s, h_L, h_S, R, G, Q)$ holds always.
- $\mathsf{safe}_{n+1}^{\mathrm{RG}}(C, s, h_L, h_S, R, G, Q)$ if and only if

(i) if $C = \mathbf{skip}$, then $s, (h_L, h_S) \models Q$; and
(ii) for all h_{F}, $C, (s, h_L \uplus h_S \uplus h_{\mathrm{F}}) \not\to \mathbf{abort}$; and
(iii) whenever $C, (s, h_L \uplus h_S \uplus h_{\mathrm{F}}) \to C', (s', h')$, then there exist h'_L and h'_S such that $h' = h'_L \uplus h'_S \uplus h_{\mathrm{F}}$ and $s, (h_S, h'_S) \models G$ and $\mathsf{safe}_n^{\mathrm{RG}}(C', s', h'_L, h'_S, R, G, Q)$;
(iv) whenever $s, (h_S, h'_S) \models R$ and $\mathsf{def}(h_L \uplus h'_S)$, then $\mathsf{safe}_n^{\mathrm{RG}}(C, s, h_L, h'_S, R, G, Q)$.

A configuration is safe for $n+1$ steps if (i) whenever it is a terminal configuration, it satisfies the postcondition; and (ii) it does not abort; and (iii) whenever it performs a transition, its change to the shared state satisfies the guarantee and the new configuration remains safe for n steps; and finally (iv) whenever the environment changes the shared state according to the rely, the resulting configuration remains safe for another n steps.

The semantics of RGSep judgments is defined in terms of $\mathsf{safe}^{\mathrm{RG}}$ in the standard way:

Definition 7.2 $R; G \models_{\mathrm{RGSep}} \{P\}\ C\ \{Q\}$ if and only if for all s, h_L, h_S, and n, if $s, (h_L, h_S) \models P$, then $\mathsf{safe}_n^{\mathrm{RG}}(C, s, h_L, h_S, R, G, Q)$.

Note that the RGSep definitions use exactly the same operational semantics for commands as the CSL definitions: we did not have to come up with a new special semantics. As we did earlier with CSL, it is possible to extend the RGSep definitions to multiple shared regions. The soundness proof goes through in pretty much the same way as in §4 and in [17].

8 Conclusion

The paper has presented a concise soundness proof of CSL and related program logics that does not involve any intermediate instrumented semantics, unlike most proofs in the literature (e.g., [3,5,9,13,6]). We have shown that inventing elaborate semantics is unnecessary and have argued that it is also harmful because it

[4] RGSep uses different syntax to denote pre- and postconditions than the one used to denote rely and guarantee conditions. In this paper, however, we shall not go into the syntax of RGSep assertions, and so we overlook such syntactic differences.

obscures the soundness argument. This becomes increasingly problematic as one moves towards larger languages and more complicated concurrent program logics.

As mentioned already, there exist several soundness proofs for concurrent separation logic, while even the first proof by Brookes [3] came 3-4 years after the CSL proof rules were conceived. This is partly due to the intricacy of the soundness resulting from imprecise assertions (cf., Reynolds's counterexample [15, §11]) and partly due to the numerous extensions to CSL that came along (e.g., permissions [2,1], "variables as resource" [16], "locks-in-the-heap" [9,13]) for which existing proofs required adaptation (e.g. [5]) or new proofs were developed [9,11,13].

A partial solution to the plethora of adapted proofs was given by Calcagno et al. [6] with *abstract separation logic*, a soundness proof of CSL with respect to an abstract operational semantics to commands that could be instantiated to the various permission and variables-as-resource models. This unifying approach, unfortunately, has a significant drawback: the soundness of any particular instance of the logic (e.g., CSL with fractional permissions) tells us nothing about how verified programs behave when executed by the hardware. This is because the instantiated abstract semantics bears little resemblance to the 'machine semantics.' To get a meaningful correspondence, one would have to relate the two semantics, a task that is most likely non-trivial. This is why our proof is instead based on a concrete semantics.

The style of semantic definitions presented in this paper has also been used to justify the soundness of more advanced program logics, such as the *concurrent abstract predicates* of Dinsdale-Young et al. [7]. So far, however, we have used this style of semantic definitions to justify the correctness only of program logics about partial correctness. It is quite possible to extend these definitions in order to capture certain kinds of liveness properties. For example, we can define the meaning of a Hoare triple for obstruction-freedom by changing $\mathsf{safe}_0(C, h, ...)$ instead of always being true to require that C terminates under no environment interference. In the future, I would like to explore this direction further.

Acknowledgement

I would like to thank Stephen Brookes, Matthew Parkinson, Peter O'Hearn, and Glynn Winskel, who encouraged me to write this paper, and also John Wickerson and the anonymous reviewers for their valuable comments.

References

[1] Bornat, R., Calcagno, C., O'Hearn, P. W., Parkinson, M. J., *Permission accounting in separation logic*, in: *POPL* (2005), pp. 259–270.

[2] Boyland, J., *Checking interference with fractional permissions*, in: *10th SAS*, LNCS **2694** (2003), pp. 55–72.

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

[4] Brookes, S., *Fairness, resources, and separation*, Electr. Notes Theor. Comput. Sci. **265** (2010), pp. 177–195.

[5] Brookes, S., *Variables as resource for shared-memory programs: Semantics and soundness*, Electr. Notes Theor. Comput. Sci. **158** (2006), pp. 123–150.

[6] Calcagno, C., O'Hearn, P. W., Yang, H., *Local action and abstract separation logic*, in: *LICS* (2007), pp. 366–378.

[7] Dinsdale-Young, T., Dodds, M., Parkinson, M., Gardner, P., Vafeiadis, V., *Concurrent abstract predicates*, in: *ECOOP*, LNCS **6183** (2010), pp. 504–528.

[8] Gotsman, A., "Logics and analyses for concurrent heap-manipulating programs," Ph.D. dissertation, University of Cambridge Computer Laboratory (2009), also available as Technical Report UCAM-CL-TR-758.

[9] Gotsman, A., Berdine, J., Cook, B., Rinetzky, N., Sagiv, M., *Local reasoning for storable locks and threads*, in: Shao, Z., editor, *APLAS*, LNCS **4807** (2007), pp. 19–37.

[10] Gotsman, A., Berdine, J., Cook, B., *Precision and the conjunction rule in concurrent separation logic*, in: *MFPS*, (20011)

[11] Haack, C., Huisman, M., Hurlin, C., *Reasoning about Java's reentrant locks*, in: Ramalingam, G., editor, *APLAS*, LNCS **5356** (2008), pp. 171–187.

[12] Hayman, J., Winskel, G., *Independence and concurrent separation logic*, in: *LICS* (2006), pp. 147–156.

[13] Hobor, A., Appel, A. W., Zappa Nardelli, F., *Oracle semantics for concurrent separation logic*, in: S. Drossopoulou, editor, *ESOP*, LNCS **4960** (2008), pp. 353–367.

[14] Jacobs, B., Piessens, F., *Expressive modular fine-grained concurrency specification*, in: *POPL* (2011).

[15] O'Hearn, P. W., *Resources, concurrency and local reasoning*, Theor. Comput. Sci. **375** (2007), pp. 271–307.

[16] Parkinson, M. J., Bornat, R., Calcagno, C., *Variables as resource in Hoare logics*, in: *LICS* (2006), pp. 137–146.

[17] Vafeiadis, V., "Fine-grained concurrency verification," Ph.D. dissertation, University of Cambridge Computer Laboratory (2007), available as Technical Report UCAM-CL-TR-726.

[18] Vafeiadis, V., *Concurrent separation logic and operational semantics (Isabelle proof)* (2011), http://www.mpi-sws.org/~viktor/cslsound/.

[19] Vafeiadis, V., Parkinson, M., *A marriage of rely/guarantee and separation logic*, in: Caires, L., Vasconcelos, V. T., editors, *CONCUR*, LNCS **4703** (2007), pp. 256–271.

[20] Yang, H., O'Hearn, P. W., *A semantic basis for local reasoning*, in: Nielsen, M., Engberg, U., editors, *FoSSaCS*, LNCS **2303** (2002), pp. 402–416.