Local Action and Abstract Separation Logic

Extended version of paper from LICS'07, version of April 30, 2007

Cristiano Calcagno
Imperial College, London

Peter W. O'Hearn
Queen Mary, University of London

Hongseok Yang
Queen Mary, University of London

Abstract

Separation logic is an extension of Hoare's logic which supports a local way of reasoning about programs that mutate memory. We present a study of the semantic structures lying behind the logic. The core idea is of a local action*, a state transformer that mutates the state in a local way. We formulate local actions for a general class of models called* separation algebras*, abstracting from the RAM and other specific concrete models used in work on separation logic. Local actions provide a semantics for a generalized form of (sequential) separation logic. We also show that our conditions on local actions allow a general soundness proof for a separation logic for concurrency, interpreted over arbitrary separation algebras.*

1 Introduction

Separation logic is an extension of Hoare's logic which has been used to attack the old problem of reasoning about the mutation of data structures in memory [32, 16, 21, 33]. Separation logic derives its power from an interplay between the separating conjunction connective $*$ and proof rules for commands that use $*$. Chief among these are the frame rule [16, 21] and the concurrency rule [22].

$$\frac{\{p\}\, C\, \{q\}}{\{p * r\}\, C\, \{q * r\}}\ \mathsf{FrameRule}$$

$$\frac{\{p_1\}\, C_1\, \{q_1\} \quad \{p_2\}\, C_2\, \{q_2\}}{\{p_1 * p_2\}\, C_1 \parallel C_2\, \{q_1 * q_2\}}\ \mathsf{ConcurrencyRule}$$

The frame rule codifies an intuition of local reasoning. The idea is that, if we establish a given Hoare triple, then the precondition contains all the resources that the command will access during computation (other than resources allocated after the command starts). As a consequence, any additional state will remain unchanged; so the invariant assertion R in the rule (the frame axiom), can be freely tacked onto the precondition and the postcondition. Similarly, the concurrency rule states that processes that operate on separate resources can be reasoned about independently.

Syntactically, the concurrency and frame rules are straightforward. But, the reason for their soundness is subtle, and rests on observations about the local way that imperative programs work [37]. Typically, a program accesses a circumscribed collection of resources; for example, the memory cells accessed during execution (the memory footprint). Our purpose in this paper is to describe these semantic assumptions in a general way, for a collection of models that abstract away from the RAM and other concrete models used in work on separation logic.

By isolating the circumscription principles for a class of models, the essential assumptions needed to justify the logic become clearer. In particular, our treatment of concurrency shows that soundness of a concurrent version of separation logic relies only on locality properties of the primitive actions (basic commands) in the programming language. Soundness of the concurrent logic was very difficult to come by, originally, even for a particular separation algebra (the RAM model); it was proven in a remarkable work of Brookes [12]. Our treatment of concurrency builds on Brookes's original insights, but makes several different choices in formulation which allow for a more general proof that applies to arbitrary separation algebras. In high-level terms we show that as long as the primitive commands in a language satisfy the frame rule, one obtains a model of a concurrent logic. This is in contrast to Brookes's original further papers on concurrent separation logics [10, 11, 15, 14], all of which have proven soundness for particular models in a way that relies on very specific interpretations of the primitive actions, rather than for a general class of models.

This paper will not contain any practical uses of the

logic. We refer the reader to, e.g., [36, 27, 5, 4] and their references for examples of manual proofs in separation logic and work on automatic proof tools. Related work will be discussed at the end of the paper.

Warning. In this paper we avoid the traditional Hoare logic punning of program variables as logical variables, to avoid nasty side conditions in the proof rules; see [9, 26] for further discussion. This is for theoretical simplicity; our results can be extended to cover variable alteration, as is done in most separation logic papers, with their associated modifies clauses. (Furthermore, avoiding the pun is more in line with real languages like C or ML, even if it departs slightly from the theoretical tradition in program logic.)

2 Separation Algebras and Predicates

Most papers on separation logic make use of a domain of heaps, which is equipped with a partial operator for gluing together heaps that are separate in some sense (various senses have appeared in the literature). We abstract from this situation with the following definition.

Definition 1 (Separation Algebra) *A* **separation algebra** *is a cancellative, partial commutative monoid* $(\Sigma, \bullet, u)$ *. A partial commutative monoid is given by a partial binary operation where the unity, commutativity and associativity laws hold for the equality that means both sides are defined and equal, or both are undefined. The cancellative property says that for each* $\sigma \in \Sigma$*, the partial function* $\sigma \bullet (\cdot) : \Sigma \rightharpoonup \Sigma$ *is injective. The induced* **separateness** *(*$\#$*) and* **substate** *(*$\preceq$*) relations are given by*

$$\begin{array}{rcl} \sigma_0 \# \sigma_1 & \textit{iff} & \sigma_0 \bullet \sigma_1 \textit{ is defined} \\ \sigma_0 \preceq \sigma_2 & \textit{iff} & \exists \sigma_1 . \, \sigma_2 = \sigma_0 \bullet \sigma_1 . \end{array}$$

Examples of separation algebras:

1. Heaps, as finite partial functions from l-values to r-values

   $$H \;=\; L \rightharpoonup_{\mathit{fin}} RV$$

   where the empty partial function is the unit and where $h_0 \bullet h_1$ takes the union of partial functions when h_0 and h_1 have disjoint domains of definition. $h_0 \bullet h_1$ is undefined when $h_0(l)$ and $h_1(l)$ are both defined for at least one l-value $l \in L$. By taking L to be the set of natural numbers and RV the set of integers we obtain the RAM model [21, 33]. By taking L to be a set of locations and RV to be certain tuples of values or `nil` we obtain models where the heap consists of records [32, 16]. And by taking L to be sequences of field names we obtain a model of hierarchical storage [1].

2. Heaps with permissions [8],

   $$\mathit{HPerm} \;=\; L \rightharpoonup_{\mathit{fin}} RV \times P$$

   where P is a *permission algebra* (i.e., a set with a partial commutative and associative operation $\circ$ satisfying the concellativity condition). $h_0 \bullet h_1$ is again the union when the domains are disjoint. Some overlap is allowed, though: when $h_0(l) = (rv, p_0)$, $h_1(l) = (rv, p_1)$ and $p_0 \circ p_1$ is defined then h_0 and h_1 are compatible at l. When all common l-values are compatible, $h_0 \bullet h_1$ is defined, and $(h_0 \bullet h_1)(l) = (rv, p_0 \circ p_1)$ for all compatible locations l. An example of a permission algebra is the interval $(0, 1]$ of rational numbers, with $\circ$ being addition but undefined when permissions add up to more than 1.

   Another permission algebra is given by the set $\{R, RW\}$ of read and read-write permissions, where $R \circ R = R$ and $RW \circ p$ is undefined. At first sight, it might be thought that this algebra models the idea of many readers and a single writer. But, unfortunately, it does not allow conversion of a total (RW) permission to several read permissions, or vice versa. In contrast, the algebra $(0, 1]$ does allow such conversion using identities such as $1/2 + 1/2 = 1$; see [8].

3. Variables as resource [9, 26] is $S \times H$, where H is as above and

   $$S \;=\; \mathit{Var} \rightharpoonup_{\mathit{fin}} \mathit{Val}$$

   has the "union of disjoint functions" partial monoid structure. Variables-as-resource models can also be mixed with the permission construction.

4. Multisets over a given set of Places, with $\bullet$ as multiset union, which can be used to model states in Petri nets without capacity [31].

5. The monoid $[\mathsf{Places} \rightharpoonup_{\mathit{fin}} \{\mathtt{marked}, \mathtt{unmarked}\}]$ of partial functions, again with union of functions with disjoint domain, can be used to model Petri nets with capacity 1. Note that the $\bullet$ operation in nets with capacity 1 is partial, while in nets without capacity it is total.

Definition 2 *Let* Σ *be a separation algebra. Predicates over* Σ *are just elements of the powerset* $P(\Sigma)$*. It has an ordered total commutative monoid structure* $(*, \mathit{emp})$ *given by*

$$\begin{array}{rcl} p * q & = & \{\sigma_0 \bullet \sigma_1 \mid \sigma_0 \# \sigma_1 \wedge \sigma_0 \in p \wedge \sigma_1 \in q\} \\ \mathit{emp} & = & \{u\} \end{array}$$

$P(\Sigma)$ is in fact a boolean BI algebra, where $p * (\cdot)$ and $p \sqcap (\cdot)$ have right adjoints [29]. Because they are left adjoints they preserve all joins, so we automatically get two distributive laws

$$\bigsqcup X * p = \bigsqcup\{x * p \mid x \in X\}$$
$$\bigsqcup X \sqcap p = \bigsqcup\{x \sqcap p \mid x \in X\}.$$

Similar laws do not hold generally for $\bigsqcap$, but the predicates p that satisfy an analogue of the first law play a crucial role in this work.

Definition 3 (Precise Predicates) *A predicate $p \in P(\Sigma)$ is* **precise** *if for every $\sigma \in \Sigma$, there exists at most one $\sigma_p \preceq \sigma$ such that $\sigma_p \in p$. We let Prec denote the set of precise predicates.*

Examples.

1. In the heap model, if $l \in L$ is an l-value and $rv \in R$ an r-value, then the predicate $l \mapsto rv \in P(\Sigma)$ is the set $\{\sigma\}$ consisting of a single state σ where $\sigma(l) = rv$ and $\sigma(l')$ is undefined for other l-values l'. It is precise.

2. $l_0 \mapsto rv_0 * l_1 \mapsto rv_1$ is the set $\{\sigma\}$ where σ is defined only on locations l_0 and l_1, mapping them to rv_0 and rv_1. Again, this predicate is precise.

3. $l_0 \mapsto rv_0 \sqcup l_1 \mapsto rv_1$ is the disjunction, i.e. the set $\{\sigma_0, \sigma_1\}$ where σ_i maps l_i to rv_i and is undefined elsewhere. $l_0 \mapsto rv_0 \sqcup l_1 \mapsto rv_1$ is not a precise predicate.

Lemma 4 (Precision Characterization) *1. Every singleton predicate $\{\sigma\}$ is precise.*

2. p is precise iff for all $X \subseteq P(\Sigma)$,

$$\bigsqcap X * p = \bigsqcap\{x * p \mid x \in X\}$$

Condition 2 in this lemma can be taken as a basis for a definition of precision in a complete lattice endowed with an ordered commutative monoid (rather than the specific lattices $P(\Sigma)$). Also, the assumption that $\sigma \bullet (\cdot)$ be injective is equivalent to the requirement

$$\bigsqcap X * \{\sigma\} = \bigsqcap\{x * \{\sigma\} \mid x \in X\}.$$

This property is used in the characterization of the lattice structure of local functions, in the proof of completeness for the sequential logic, and again later when we turn to concurrency. In sum, precision plays a greater role here than in previous work, where it arose as a technical reaction to soundness problems in proof rules for information hiding [24, 12, 22].

3 Local Actions

3.1 Conceptual Development

In [37] a soundness proof was given for separation logic in terms of an operations semantics for the RAM (heaps) model. The development there revolved around relations

$$R \subseteq \Sigma \times (\Sigma \cup \{\mathtt{fault}\})$$

$(\sigma, \sigma') \in R$ signifies that the program can deliver final state σ' when started in σ, and $(\sigma, \mathtt{fault}) \in R$ signifies that a memory fault can occur (by dereferencing a dangling pointer). In terms of these relations, two properties were identified that correspond to the frame rule:

1. Safety Monotonicity: $(\sigma, \mathtt{fault}) \notin R$ and $\sigma \preceq \sigma'$ implies $(\sigma', \mathtt{fault}) \notin R$.

2. Frame Property: If $(\sigma_0, \mathtt{fault}) \notin R$ and $\sigma = \sigma_0 \bullet \sigma_1$ and $(\sigma, \sigma') \in R$ then $\exists \sigma_0'.\ \sigma' = \sigma_0' \bullet \sigma_1$ and $(\sigma_0, \sigma_0') \in R$.

The first condition says that if a state has enough resource for safe execution of a command, then so do superstates. The second condition says that if a little state σ_0 has enough resource for the command to execute safely, then execution on any bigger state can be tracked back to the small state.

These two conditions can be shown to be equivalent to the frame rule: a relation R satisfies Safety Monotonicity and the Frame Property iff the frame rule is sound for it. But, the formulation of the second property is unpleasant: it is a tabulation of a true operational fact (as shown in [37]), but in developing our theory we seek a simpler condition.

The first step to this simpler formulation is to make use of the fact that fault trumps; that is, in separation logic Hoare triples are interpreted so that fault falsifies a triple [16, 37, 33]. The result is that Hoare triples cannot distinguish between a command that just faults, and one that nondeterministically chooses between faulting and terminating normally. This suggests that in the semantics we can use functions from states to the powerset $P(\Sigma)$, together with fault, rather than relations as above.

But, order-theoretically, where should fault go? The answer is on the top, as in functions

$$f : \Sigma \to P(\Sigma)^{\top}.$$

Conceptually, faulting is like Scott's top: an inconsistent or over-determined value. Technically, putting fault on the top allows us to characterize the pointwise order in terms of Hoare triples (Proposition 7), which shows that it is the correct order for our purposes.

This much is mostly a standard essay on relations versus functions into powersets. The payoff comes in the treatment

of locality. Using functions into the (topped) powerset, we can work with a much simpler condition:

$$\textbf{locality} : \sigma_1 \# \sigma_2 \text{ implies } f(\sigma_1 \bullet \sigma_2) \sqsubseteq (f\sigma_1) * \{\sigma_2\}.$$

Here, the $*$ operation is extended with $\top$, by $p * \top = \top * p = \top$. It can be verified that a relation R satisfies Safety Monotonicity and Frame Property iff the corresponding function $func(R) : \Sigma \to P(\Sigma)^\top$ satisfies locality.[1]

Aside on total correctness. The use of $P(\Sigma)^\top$ above is strongly oriented to a partial correctness logic, where divergence does not falsify a Hoare triple. The empty set in $P(\Sigma)^\top$ represents divergence. For total correctness we would just have to flip $P(\Sigma)^\top$ upside down, obtaining $(P(\Sigma)^\top)^{op}$. In total correctness divergence and faulting are identified, and both denote $\top$ (which in $(P(\Sigma)^\top)^{op}$ is the least element, traditionally called "chaos".) The empty set is a "miracle" element, included just to make sure we get a complete partial order. In the total correctness semantics we would have to work with monotone rather than continuous functions, to account for infinite nondeterminism (as represented by allocation in separation logic works). $(P(\Sigma)^\top)^{op}$ is just the standard Smyth domain used for demonic nondeterminism, while $P(\Sigma)^\top$ is an unusual angelic domain (the extra $\top$ is usually not present in the angelic domains).

Aside on predicate transformers. If we were to use maps $P(\Sigma) \to P(\Sigma)$ for backwards predicate transformers, or $P(\Sigma)^\top \to P(\Sigma)^\top$ for forwards transformers, then a mathematically simpler expression of locality would be possible. For the forwards transformers it is $F(p * r) \sqsubseteq F(p) * r$ and for the backwards it is the reverse. We do not use predicate transformers as a semantics here, because it would be circular to use them to *justify* the logic:[2] they are essentially rewriting (Hoare) logic in order-theoretic terms. We use state-transformers on separation algebras because they are comparatively removed from the logic, and closer to the way that programs work.

3.2 Technical Development

Definition 5 *$P(\Sigma)^\top$ is obtained by adding a new greatest element to $P(\Sigma)$. It has a total commutative monoid structure, keeping the unit emp the same as in $P(\Sigma)$, and extending $*$ so that $p * \top = \top * p = \top$.*

Definition 6 (Semantic Hoare Triple) *If $p, q \in P(\Sigma)$ and $f : \Sigma \to P(\Sigma)^\top$ then*

$$\langle\!\langle p \rangle\!\rangle\ f\ \langle\!\langle q \rangle\!\rangle \text{ holds iff for all } \sigma \in p.\ f\sigma \sqsubseteq q$$

This is fault-avoiding because the postcondition does not include top. A justification for putting fault on the top is the following.

Proposition 7 (Order Characterization) *$f \sqsubseteq g$ iff for all $p, q \in P(\Sigma)$, $\langle\!\langle p \rangle\!\rangle\ g\ \langle\!\langle q \rangle\!\rangle$ implies $\langle\!\langle p \rangle\!\rangle\ f\ \langle\!\langle q \rangle\!\rangle$*

Definition 8 (Local Action) *Suppose Σ is a separation algebra. A* **local action** *$f : \Sigma \to P(\Sigma)^\top$ is a function satisfying the* **locality condition***:*

$$\sigma_1 \# \sigma_2 \text{ implies } f(\sigma_1 \bullet \sigma_2) \sqsubseteq (f\sigma_1) * \{\sigma_2\}.$$

We let $LocAct$ denote the set of local actions, with pointwise order.

Lemma 9 *$LocAct$ is a complete lattice, with meets and joins defined pointwise (and inherited from the function space $[\Sigma \to P(\Sigma)^\top]$).*

Proof: Assume that $\sigma = \sigma_0 \# \sigma_1$. Then we can show that the pointwise meet is local

$$\begin{array}{lcl} (\bigsqcap F)(\sigma_0 \bullet \sigma_1) & = & \bigsqcap\{f(\sigma_0 \bullet \sigma_1) \mid f \in F\} \\ & \sqsubseteq & \bigsqcap\{f(\sigma_0) * \{\sigma_1\} \mid f \in F\} \\ & = & \big(\bigsqcap\{f(\sigma_0) \mid f \in F\}\big) * \{\sigma_1\} \\ & = & ((\bigsqcap F)\sigma_0) * \{\sigma_1\}. \end{array}$$

The second-last step used that $\{\sigma_1\}$ is precise (Lemma 4).

For pointwise joins,

$$\begin{array}{lcl} (\bigsqcup F)(\sigma_0 \bullet \sigma_1) & = & \bigsqcup\{f(\sigma_0 \bullet \sigma_1) \mid f \in F\} \\ & \sqsubseteq & \bigsqcup\{f(\sigma_0) * \{\sigma_1\} \mid f \in F\} \\ & = & \big(\bigsqcup\{f(\sigma_0) \mid f \in F\}\big) * \{\sigma_1\} \\ & = & ((\bigsqcup F)\sigma_0) * \{\sigma_1\}. \end{array}$$

In the second-last step, this time, we used the distribution of $\bigsqcup$ over $(\cdot)*\{\sigma_1\}$, which in fact holds for arbitrary predicates rather than just singletons. ■

Given any precondition p_1 and postcondition p_2, we can define the best or largest local action satisfying the triple $\langle\!\langle p_1 \rangle\!\rangle\ -\ \langle\!\langle p_2 \rangle\!\rangle$[3].

Definition 10 (Best Local Action) *$bla[p_1, p_2]$ is the function of type $\Sigma \to P(\Sigma)^\top$ defined by*

$$bla[p_1, p_2](\sigma) = \bigsqcap\{p_2 * \{\sigma_0\} \mid \sigma = \sigma_0 \bullet \sigma_1, \sigma_1 \in p_1\}.$$

Lemma 11 *Let $f = bla[p_1, p_2]$. The following hold:*

- *f is a local action;*

[1] $func(R)\sigma$ returns $\top$ when $(\sigma, \texttt{fault}) \in R$, even if we also have $(\sigma, \sigma') \in R$ for some $\sigma' \neq \texttt{fault}$.

[2] This is not to deny their usefulness for other purposes, such as in abstract interpretation.

[3] This is an analogue of the "specification statement" studied in the refinement literature

- $\langle\!\langle p_1 \rangle\!\rangle\ f\ \langle\!\langle p_2 \rangle\!\rangle$;
- *If* $\langle\!\langle p_1 \rangle\!\rangle\ t\ \langle\!\langle p_2 \rangle\!\rangle$ *and* t *is local then* $t \sqsubseteq f$.

Proof: To show that f is local, consider σ', σ'' such that $\sigma' \# \sigma''$. We then calculate

$$\begin{array}{rl} & bla[p_1, p_2](\sigma' \bullet \sigma'') \\ = & \bigsqcap\{p_2 * \{\sigma_0\} \mid \sigma' \bullet \sigma'' = \sigma_0 \bullet \sigma_1, \sigma_1 \in p_1\} \\ \sqsubseteq & \bigsqcap\{p_2 * \{\sigma_0 \bullet \sigma''\} \mid \sigma' = \sigma_0 \bullet \sigma_1, \sigma_1 \in p_1\} \\ = & \bigsqcap\{p_2 * \{\sigma_0\} * \{\sigma''\} \mid \sigma' = \sigma_0 \bullet \sigma_1, \sigma_1 \in p_1\} \\ = & \left(\bigsqcap\{p_2 * \{\sigma_0\} \mid \sigma' = \sigma_0 \bullet \sigma_1, \sigma_1 \in p_1\}\right) * \{\sigma''\} \\ = & bla[p_1, p_2](\sigma') * \{\sigma''\}. \end{array}$$

In the second-last step we used that $\{\sigma''\}$ is precise (Lemma 4).

To show that f satisfies $\langle\!\langle p_1 \rangle\!\rangle\ f\ \langle\!\langle p_2 \rangle\!\rangle$, consider $\sigma \in p_1$. Then $f(\sigma) \sqsubseteq p_2 * emp = p_2$.

For the last point, suppose $\{p_1\}\, t\, \{p_2\}$ and t is local. Then for all $\sigma, \sigma_1, \sigma_2$ such that $\sigma = \sigma_1 \bullet \sigma_2$ and $\sigma_1 \in p_1$,

$$\begin{array}{rcl} t(\sigma) & = & t(\sigma_1 \bullet \sigma_2) \\ & \sqsubseteq & t(\sigma_1) * \{\sigma_2\} \\ & \sqsubseteq & p_2 * \{\sigma_2\}. \end{array}$$

Thus $t(\sigma) \sqsubseteq f(\sigma)$. ■

Local actions form a one-object category (a monoid), where the identity is $bla[emp, emp]$ and the composition $f; g$ functionally composes f with the obvious lifting $g^\dagger : P(\Sigma)^\top \to P(\Sigma)^\top$.[4] This structure is used in the semantics of `skip` and sequential composition in Figure 1.

Examples.

1. For the *Heaps* model, the function f which always returns emp is not local. The function f that sets all allocated locations to some specific r-value (say, 0) is not local. For, f applied to a singleton heap $[l \mapsto 2]$ returns $[l \mapsto 0]$. According to locality, it should not alter any location other than l. But, $f([l \mapsto 2, l' \mapsto 2]) = \{[l \mapsto 0, l' \mapsto 0]\}$. Such a function is not definable in the languages used in separation logic (and is typically not definable in, say, C or Java.). In contrast, the operations of heap mutation and allocation and disposal are local actions (see the next section).

2. Any transition on a Petri net without capacity defines a local action on the multiset monoid (in fact, a deterministic local action). Any transition on a net with capacity 1 determines a local action on the separation algebra $[\mathsf{Places} \rightharpoonup_{fin} \{\mathtt{marked}, \mathtt{unmarked}\}]$. Interestingly, transitions for nets with capacity do not define local actions on the separation algebra which is the set of places with union of disjoint subsets as $\bullet$.

[4] How to make this into a more genuine, many object, category is not completely obvious.

$$\begin{array}{rcl} [\![c]\!]v & = & v(c) \\ [\![\mathtt{skip}]\!]v\sigma & = & \{\sigma\} \quad (= bla[emp, emp]\sigma \\ [\![C_1; C_2]\!]v & = & ([\![C_1]\!]v); ([\![C_2]\!]v) \\ [\![C^\star]\!]v & = & \bigsqcup_n [\![C^{\,n}]\!]v \\ [\![C_1 + C_2]\!]v & = & [\![C_1]\!]v \sqcup [\![C_2]\!]v \end{array}$$

- $v : PrimCommands \to LocAct$
- $[\![C]\!]v \in LocAct$
- $(f; g)\sigma = \begin{cases} \top & \text{if } f\sigma = \top \\ \bigsqcup\{g\sigma' \mid \sigma' \in f\sigma\} & \text{otherwise} \end{cases}$

Figure 1. Denotational Semantics

3. Generally, $bla[p, emp]$ is the local action that disposes of, or annihilates, p. Similarly, $bla[emp, p]$ allocates p, or lets the knowledge of p materialize.

The annihilation $bla[p, emp]$ behaves strangely when p is not precise. For example, when p is $l_0 \mapsto r_0 \sqcup l_1 \mapsto r_1$ for $l_0 \neq l_1$ in the heap model, $bla[p, emp]$ on the heap $[l_0 \mapsto r_0]$ disposes l_0, but diverges on $[l_0 \mapsto r_0, l_1 \mapsto r_1]$. However, in case p is precise, the definition is well-behaved. We set out to analyze this.

We use the following preliminary definition of $\sigma \backslash p$, which removes from σ a substate satisfying p (if it exists).

Definition 12 (Removal) $(\cdot)\backslash p : \Sigma \to P(\Sigma)^\top$ *is the function where*

$$\sigma\backslash p \;=\; \begin{cases} \top & \textit{if } \sigma \notin p * \textit{true} \\ \{\sigma_0 \mid \exists \sigma_1.\, \sigma = \sigma_0 \bullet \sigma_1 \\ \qquad \textit{and } \sigma_1 \in p\} & \textit{otherwise} \end{cases}$$

Here *true* means the set of all states, i.e., Σ. Note that σ_0 in this definition is necessarily unique when p is precise. The following lemma confirms that when p is precise, annihilation successfully deletes the state described by p.

Lemma 13 *If* p *is precise, then* $bla[p, emp]\sigma \;=\; \sigma\backslash p$.

4 Programming Language

The commands of our language are as follows:

$$C \quad ::= \quad c \mid \mathtt{skip} \mid C; C \mid C + C \mid C^\star$$

Here, c ranges over an unspecified collection $PrimCommands$ of primitive commands, $+$ is nondeterministic choice, ; is sequential composition, and $(\cdot)^\star$ is Kleene-star (iterated ;). We use $+$ and $(\cdot)^\star$ instead of

conditionals and while loops for theoretical simplicity: given appropriate primitive actions the conditionals and loops can be encoded, but we do not need to explicitly consider boolean conditions in the abstract theory.

The denotational semantics of commands is given in Figure 1. The meanings of primitive commands are given by a valuation v. The meaning of Kleene-star is a local action because of Lemma 9.

Example Language and Model. We illustrate this definition with a particular concrete model and several primitive commands. As a model we take

$$\Sigma = S \times H \;=\; (Var \rightharpoonup_{fin} RV) \times (L \rightharpoonup_{fin} RV)$$

as in Section 2. We assume further that $L \subseteq RV$.

For a given rv and $x, y \in Var$, we can define

$$\begin{array}{lll} & load(l,x) & \\ = & \bigsqcap_{rv} bla[l \mapsto rv * x \mapsto - \;\;,\;\; l \mapsto rv * x \mapsto rv] & \\ & store(x,l) & \\ = & \bigsqcap_{rv} bla[l \mapsto - * x \mapsto rv \;\;,\;\; l \mapsto rv * x \mapsto rv] & \\ & move(x,y) & \\ = & \bigsqcap_{rv} bla[x \mapsto rv * y \mapsto - \;\;,\;\; x \mapsto rv * y \mapsto rv] & \end{array}$$

Here, $load$ is the analogue of the assembly language instruction that retrieves a value from memory and puts it in a register, while $store$ takes a value from the register bank and puts it into memory, and $move$ copies a value from one register to another. Here the use of $\bigsqcap$ is the meet of local actions (not just assertions), and is essentially being used to model universal quantification outside of a triple to treat rv as a ghost variable (a variable that is not a program variable). [5]

Primitive commands $free(l)$ and $new(x)$ for disposing and allocating heap locations denote the following best local actions.

$$\begin{array}{rcl} free(l) & = & bla[l \mapsto -, emp] \\ new(x) & = & bla[x \mapsto -, \bigsqcup_l x \mapsto l * l \mapsto -] \end{array}$$

So, allocation and deallocation are special cases of the general concepts of materialization and annihilation. In these definitions $l \mapsto -$ is the predicate denoting $\{\sigma\}$ where $\sigma(l)$ is defined and where $\sigma(l_0)$ is undefined for $l_0 \neq l$. In the postcondition for $new(x)$ we are using $\bigsqcup_l$ to play the role of existential quantification in the evident way.[6]

We have used the best local actions $bla[-,-]$ to define these functions, but we could also define them by more explicit reference to states. For example, $load(l,x)$ is

$$\lambda\sigma.\mathit{if}\ (l, x \in dom(\sigma))\ \mathit{then}\ (\sigma | x{:=}\sigma(l))\ \mathit{else}\ \top$$

where we use $(\sigma | x{:=}rv)$ for updating a partial function.

In this model we can have boolean expressions for testing, say, whether two locations have the same value ($[l] == [l']$). Following [16], we call a predicate p *intuitionistic* if $p * true = p$, and define the intuitionistic negation $\neg_i p$ of p to be $\{\sigma \mid \forall \sigma'.\ \sigma \bullet \sigma' \notin p\}$. Generally, we can presume a collection of primitive boolean expressions b, which give rise to primitive commands $\mathtt{assume}(B)$ for some intuitionistic predicate $B \in P(\Sigma)$. Our valuation v has to map $\mathtt{assume}(B)$ to a local action $v(\mathtt{assume}(B)) \in LocAct$ which returns an input state σ if B holds in σ; diverges if $\neg_i B$ holds; faults otherwise. Then, we can encode conditionals and loops as

$$\begin{array}{l} (\mathtt{assume}(B); C_1) + (\mathtt{assume}(\neg_i B)); C_2) \\ (\mathtt{assume}(B); C)^\star; \mathtt{assume}(\neg_i B) \end{array}$$

The point of this is just to make clear that, in the general theory, we do not need to consider boolean expressions explicitly: the `assume` statements can be taken to be given primitive commands, in which case their use in loops and conditionals can be encoded in terms of the more basic nondeterministic choice and Kleene iteration.[7]

5 An Abstract Separation Logic

5.1 Proof Theory

The rules for Abstract Separation Logic are in Figure 2. Note that we have to require that I be nonempty in the conjunction rule because $\{true\}C\{true\}$ does not generally hold in separation logic (because of the fault-avoiding interpretation of triples).

Definition 14 (Axioms) *An axiom set* Ax *is a set of triples*

$$\{p\}c\{q\}$$

for primitive commands c*, where there is at least one axiom for each primitive command.*[8]

[5] We could have defined these operations without using $\bigsqcap$ if we took ghosts seriously in our theory. For instance, $bla[l \mapsto rv * x \mapsto -, l \mapsto rv * x \mapsto rv]$ is a good specification of $load$, as long as we understand that rv is a ghost. But, we can get the same mathematical effect using $\bigsqcap$ outside of triples, which helps to keep the theory simpler by not having a special kind of variable. Put another way, we have all the mathematical structure needed to explain ghosts, but just use that structure directly rather than undertake additional formalization.

[6] The assertion would sometimes be written $\exists l.x \mapsto l * l \mapsto -$, and we are just using the ability of a complete Boolean algebra to interpret quantification.

[7] If we wanted to include booleans explicitly in the general theory we could use "local booleans", functions $\Sigma \to \{t, f\}_\perp$ that are monotone wrt the $\preceq$ order on Σ.

[8] The canonical axiom $\{false\}\, c\, \{false\}$ can be taken when a specific choice is not desired.

STRUCTURAL RULES

$$\frac{\{p\}\,C\,\{q\}}{\{p * r\}\,C\,\{q * r\}} \qquad \frac{p' \sqsubseteq p \quad \{p\}\,C\,\{q\} \quad q \sqsubseteq q'}{\{p'\}\,C\,\{q'\}}$$

$$\frac{\{p_i\}\,C\,\{q_i\},\ \text{all } i \in I}{\{\bigsqcup_{i\in I} p_i\}\,C\,\{\bigsqcup_{i\in I} q_i\}} \qquad \frac{\{p_i\}\,C\,\{q_i\},\ \text{all } i \in I}{\{\bigsqcap_{i\in I} p_i\}\,C\,\{\bigsqcap_{i\in I} q_i\}}\ I \neq \emptyset$$

BASIC CONSTRUCTS

$$\frac{}{\{p\}\,\mathtt{skip}\,\{p\}} \qquad \frac{\{p\}\,C_1\,\{q\} \quad \{q\}\,C_2\,\{r\}}{\{p\}\,C_1; C_2\,\{r\}}$$

$$\frac{\{p\}\,C_1\,\{q\} \quad \{p\}\,C_2\,\{q\}}{\{p\}\,C_1 + C_2\,\{q\}} \qquad \frac{\{p\}\,C\,\{p\}}{\{p\}\,C^\star\,\{p\}}$$

Figure 2. Rules of Abstract Separation Logic

Definition 15 (Proof-theoretic Consequence Relation) *We write*

$$\mathsf{Ax} \vdash \{p\}\,C\,\{q\},$$

to mean that $\{p\}\,C\,\{q\}$ *is derivable from* Ax *using the rules in Figure 2.*

Note that the consequent, but not the antecedent, in $\mathsf{Ax} \vdash \{p\}\,C\,\{q\}$ might involve a composite command.

5.2 Semantics

Definition 16 (Satisfaction) *Suppose that we have a valuation* v *as in Figure 1. We say that* v **satisfies** $\{p\}\,C\,\{q\}$ *just if* $\langle\!\langle p \rangle\!\rangle\ [\![C]\!]v\ \langle\!\langle q \rangle\!\rangle$ *is true according to Definition 6.*

Definition 17 (Semantic Consequence Relation) *We write*

$$\mathsf{Ax} \models \{p\}\,C\,\{q\}$$

to mean that for all valuations v*, if* v *satisfies* Ax *then* v *satisfies* $\{p\}\,C\,\{q\}$*.*

Theorem 18 (Soundness) *All of the proof rules preserve semantic validity* ($\mathsf{Ax} \models \{p\}\,C\,\{q\}$)*. As a result, a proof-theoretic consequence is also a semantic consequence:*

$$\mathsf{Ax} \vdash \{p\}\,C\,\{q\} \qquad \text{implies} \qquad \mathsf{Ax} \models \{p\}\,C\,\{q\}.$$

The soundness result is easy to prove. The soundness of the frame rule follows from the locality condition in Definition 8, and all the other rules are straightforward.

We can also prove the converse of Theorem 18. The key to this is to induce a particular valuation from any set of axioms.

Definition 19 (Canonical Valuation) $v(\mathsf{Ax})$ *is the valuation mapping* c *to*

$$\bigsqcap\{bla[p,q] \mid \{p\}\,c\,\{q\} \in \mathsf{Ax}\}.$$

Lemma 20 $\mathsf{Ax} \models \{p\}\,C\,\{q\}$ *iff* $v(\mathsf{Ax})$ *satisfies* $\{p\}\,C\,\{q\}$

Proof: Suppose that $\mathsf{Ax} \models \{p\}\,C\,\{q\}$. Then, by the definition of $\mathsf{Ax} \models \{p\}\,C\,\{q\}$, every valuation v satisfying Ax also satisfies $\{p\}\,C\,\{q\}$. Since $v(\mathsf{Ax})$ satisfies Ax, we have that $\langle\!\langle p \rangle\!\rangle\ [\![C]\!](v(\mathsf{Ax}))\ \langle\!\langle q \rangle\!\rangle$, as required by the only if direction of the lemma. For the reverse, assume $v(\mathsf{Ax})$ satisfies $\{p\}\,C\,\{q\}$ and that v is some valuation satisfying Ax. Hoare triples are downwards closed, in the sense that if $\langle\!\langle p \rangle\!\rangle\ f\ \langle\!\langle q \rangle\!\rangle$ and $g \sqsubseteq f$ then $\langle\!\langle p \rangle\!\rangle\ g\ \langle\!\langle q \rangle\!\rangle$. It is easy to see that $v(\mathsf{Ax})$ is the greatest valuation satisfying all the triples in Ax, so $v \sqsubseteq v(\mathsf{Ax})$. (That is, where valuations are ordered pointwise.) That v satisfies $\{p\}\,C\,\{q\}$ follows. ■

Theorem 21 (Completeness)

$$\mathsf{Ax} \models \{p\}\,C\,\{q\} \qquad \textit{implies} \qquad \mathsf{Ax} \vdash \{p\}\,C\,\{q\}$$

Proof: Let $v_a = v(\mathsf{Ax})$ be the canonical valuation. We prove that if v_a satisfies $\{p\}\,C\,\{q\}$ then $\mathsf{Ax} \vdash \{p\}\,C\,\{q\}$. The conclusion then follows from Lemma 20. The proof proceeds by induction on the structure of C. First, we note that we can reduce completeness for general p's in the precondition to completeness for singletons $p = \{\sigma\}$. The reason is that we can use an instance of the disjunction rule

$$\frac{\{\{\sigma\}\}\,c\,\{q\},\ \text{all } \sigma \in p}{\{\bigsqcup_{\sigma\in p}\{\sigma\}\}\,c\,\{q\}}$$

to reduce the general case to the case for singleton preconditions. We use this reduction in the first base case below.

Suppose $C = c$ is a primitive command.[9] We assume, wlog, that $p = \{\sigma\}$ is a singleton. Let I be an indexing, and let p_i, q_i be such that

$$\{p_i\}\,c\,\{q_i\}, \quad \text{where} \quad i \in I$$

is the collection of all axioms in Ax that involve c. Let

$$J = \{\sigma_{i1} \mid i \in I \ \wedge\ \exists \sigma_{i0} \in p_i.\ \sigma = \sigma_{i0} \bullet \sigma_{i1}\}.$$

Since v_a satisfies $\{p\}\,c\,\{q\}$ we have

(a) $(\bigsqcup_{\sigma_{i0}\in J} q_i * \{\sigma_{i0}\}) \sqsubseteq q.$

It is also easy to see that

(b) $\{\sigma\} \sqsubseteq (\bigsqcup_{\sigma_{i0}\in J} p_i * \{\sigma_{i0}\}).$

[9] This is sometimes called Adaptation Completeness, and is actually the most difficult step.

We then reason as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\{p_i\}\,c\,\{q_i\}\ \ \text{all } i \in I}{\{p_i * \{\sigma_{i0}\}\}\,c\,\{q_i * \{\sigma_{i0}\}\}\ \ \text{all } \sigma_{i0} \in J}}{\{\bigsqcup_{\sigma_{i0}\in J} p_i * \{\sigma_{i0}\}\}\,c\,\{\bigsqcup_{\sigma_{i0}\in J} q_i * \{\sigma_{i0}\}\}}}{\{\bigsqcup_{\sigma_{i0}\in J} p_i * \{\sigma_{i0}\}\}\,c\,\{q\}}}{\{\{\sigma\}\}\,c\,\{q\}}$$

The first step here uses the frame rule multiple times, the second step uses the disjunction rule, and the third and fourth use consequence with (a) and (b) above. This completes the proof of the case for primitive commands c.

The case of `skip` is straightforward.

For $C_1; C_2$, we know that v_a satisfies $\{p\}\,C_1; C_2\,\{q\}$. Therefore, it follows that $[\![C_1]\!]v_a\sigma \neq \top$ for each $\sigma \in p$. Let r be $\bigsqcup\{[\![C_1]\!]v_a\sigma \mid \sigma \in p\}$. We know that v_a satisfies $\{p\}\,C_1\,\{r\}$, and so by the induction hypothesis $\mathsf{Ax} \vdash \{p\}\,C_1\,\{r\}$. Similarly, by the definition of $[\![C_1; C_2]\!]$ it follows that v_a satisfies $\{r\}\,C_2\,\{q\}$ and by induction hypothesis $\mathsf{Ax} \vdash \{r\}\,C_2\,\{q\}$. We can then apply the rule of sequencing to derive $\{p\}\,C_1; C_2\,\{q\}$.

For $C_1 + C_2$, we know that v_a satisfies $\{p\}\,C_1 + C_2\,\{q\}$. Therefore, it follows that $[\![C_1]\!]v_a\sigma \neq \top$ and $[\![C_2]\!]v_a\sigma \neq \top$ for each $\sigma \in p$. Let r_i be $\bigsqcup\{[\![C_i]\!]v_a\sigma \mid \sigma \in p\}$ for $i = 1, 2$. We know that v_a satisfies $\{p\}\,C_i\,\{r_i\}$, so by induction hypothesis we obtain that $\mathsf{Ax} \vdash \{p\}\,C_i\,\{r_i\}$. By the rule of consequence we obtain that $\mathsf{Ax} \vdash \{p\}\,C_i\,\{r_1 \sqcup r_2\}$. We can then apply the rule for $+$ to obtain $\mathsf{Ax} \vdash \{p\}\,C_1 + C_2\,\{r_1 \sqcup r_2\}$. By the definitions of $[\![C_1 + C_2]\!]$ and $\langle\!\langle p \rangle\!\rangle - \langle\!\langle q \rangle\!\rangle$ we obtain that $r_1 \sqcup r_2 \sqsubseteq q$. One more application of the rule of consequence then gives us $\mathsf{Ax} \vdash \{p\}\,C_1 + C_2\,\{q\}$.

For $C^\star$, we know that v_a satisfies $\{p\}\,C^\star\,\{q\}$. Let $r = \bigsqcup\{[\![C^n]\!]v_a\sigma \mid \sigma \in p\}$. Unwinding the definitions, this means

(a) $r \sqsubseteq q$ (by the definition of $\langle\!\langle p \rangle\!\rangle - \langle\!\langle q \rangle\!\rangle$),

(b) $p \sqsubseteq r$ (from $n = 0$ case in r), and

(c) v_a satisfies $\{r\}\,C\,\{r\}$ (by induction on n in the definition of r).

By induction hypothesis, from (c) we obtain $\mathsf{Ax} \vdash \{r\}\,C\,\{r\}$ and using the rule for iteration, $\mathsf{Ax} \vdash \{r\}\,C^\star\,\{r\}$. From this we can derive $\mathsf{Ax} \vdash \{p\}\,C^\star\,\{q\}$ using the rule of consequence with (a) and (b). ■

6 A Logic for Concurrency

We add three new command forms for parallel composition, lock declarations $\ell.C$, and critical sections.[10]

$$C \ ::= \ \cdots \mid C \parallel C \mid \ell.C \mid \texttt{with}\,\ell\,\texttt{do}\,C$$

[10] In [22] a conditional notion of critical section was used for convenience, but this can be encoded in terms of simple sections and `assume` statements.

This language assumes that there is a fixed infinite set $Locks$, from which the ℓ's are drawn. The basic constraint on the critical sections is that different $\texttt{with}\,\ell\,\texttt{do}\,C$ for the same ℓ must be executed with mutual exclusion. In implementation terms, we can consider the critical section as being implemented by $P(\ell); C; V(\ell)$ where $P(\ell)$ and $V(\ell)$ are Dijkstra's operations on (binary) mutex semaphore ℓ.

The program logic will manipulate an *environment* mapping locks to precise predicates.

$$Env = Locks \rightharpoonup_{fin} Prec.$$

The judgments of the logic for concurrency are of the form

$$\eta \rhd \{p\}C\{q\}$$

where $\eta \in Env$ defines all the lock variables free in C. The rules for concurrency are in Figure 3.

Definition 22 (Proof-theoretic Consequence Relation, II) *We write*

$$\eta; \mathsf{Ax} \vdash \{p\}\,C\,\{q\},$$

to mean that $\eta \rhd \{p\}\,C\,\{q\}$ *is derivable from assumptions*

$$\eta' \rhd \{p\}\,c\,\{q\}, \quad \textit{where } \{p\}\,c\,\{q\} \in \mathsf{Ax} \textit{ and } \eta' \in Env,$$

by the rules in Figure 3.

7 A Concurrency Model

In broad outline, our semantics for the concurrent logic follows that of Brookes [12]. First, we define an interleaving semantics based on *action traces*. This is a denotational but completely syntactic model, that resolves all the concurrency for us. Second, we give a way to "execute" the traces in given states. Brookes did this using an additional "local enabling relation" defined for the traces. Here, trace execution just uses the denotational semantics in terms of local functions. This is what allows us to formulate our model for arbitrary separation algebras.[11]

7.1 Syntactic Trace Model

The traces will be made up of the primitive actions of our programming language, plus two additional semaphore operations to model entry and exit from critical regions.

Definition 23 *An atomic action* α *is a primitive command or* `skip` *or a race-check or an* ℓ*-action* $act(\ell)$.

$$\begin{array}{rcl} \alpha & ::= & c \mid \texttt{skip} \mid \texttt{check}(c, c) \mid act(\ell) \\ act(\ell) & ::= & P(\ell) \mid V(\ell) \end{array}$$

A trace τ *is a sequential composition of atomic actions:*

$$\tau ::= \alpha; \cdots; \alpha$$

[11] Also, in contrast to [12], there will be no notion of local enabling becoming "stuck", where stuckness is a concept distinct from faulting.

$$\frac{\eta \rhd \{p_1\}\, C_1\, \{q_1\} \quad \eta \rhd \{p_2\}\, C_2\, \{q_2\}}{\eta \rhd \{p_1 * p_2\}\, C_1 \parallel C_2\, \{q_1 * q_2\}} \qquad \frac{\eta, \ell \mapsto r \rhd \{p\}\, C\, \{q\}}{\eta \rhd \{p * r\}\, \ell.C\, \{q * r\}} \qquad \frac{\eta \rhd \{p * r\}\, C\, \{q * r\}}{\eta, \ell \mapsto r \rhd \{p\}\, \texttt{with}\, \ell\, \texttt{do}\, C\, \{q\}}$$

Plus the rules from Figure 2 with $\eta\rhd$ added uniformly

Figure 3. Rules for Concurrency

$$T(c) = \{c\} \qquad T(\texttt{skip}) = \{\texttt{skip}\} \qquad T(C_1; C_2) = \{\tau_1; \tau_2 \mid \tau_i \in T(C_i)\}$$
$$T(C_1 + C_2) = T(C_1) \cup T(C_2) \qquad T(C^\star) = (T(C))^\star \qquad T(C_1 \parallel C_2) = \{\tau_1 \mathbin{zip} \tau_2 \mid \tau_i \in T(C_i)\}$$
$$T(\ell.C) = \{(V(\ell); \tau; P(\ell)) - \ell \mid \tau \in T(C) \text{ is } \ell\text{-synchronized}\} \qquad T(\texttt{with}\, \ell\, \texttt{do}\, C) = \{P(\ell); \tau; V(\ell) \mid \tau \in T(C)\}$$

where $\tau_1 \mathbin{zip} \tau_2$ and the auxiliary $\tau_1 \mathbin{zip'} \tau_2$ are defined as follows:

$$\gamma ::= \texttt{skip} \mid act(\ell)$$
$$\epsilon \mathbin{zip} \tau = \tau \qquad \tau \mathbin{zip} \epsilon = \tau \qquad \epsilon \mathbin{zip'} \tau = \tau \qquad \tau \mathbin{zip'} \epsilon = \tau$$
$$(c_1; \tau_1) \mathbin{zip} (c_2; \tau_2) = \texttt{check}(c_1, c_2); ((c_1; \tau_1) \mathbin{zip'} (c_2; \tau_2)) \qquad (\gamma; \tau_1) \mathbin{zip} \tau_2 = (\gamma; \tau_1) \mathbin{zip'} \tau_2$$
$$\tau_1 \mathbin{zip} (\gamma; \tau_2) = \tau_1 \mathbin{zip'} (\gamma; \tau_2) \qquad (\alpha_1; \tau_1) \mathbin{zip'} (\alpha_2; \tau_2) = (\alpha_1; (\tau_1 \mathbin{zip} (\alpha_2; \tau_2))) \cup (\alpha_2; ((\alpha_1; \tau_1) \mathbin{zip} \tau_2))$$

Figure 4. Trace Semantics

We write ϵ for the empty trace, $\tau - \ell$ for the trace obtained by deleting all ℓ-actions from τ, and $\tau|_\ell$ for the trace obtained by removing all non-ℓ actions from τ.

Definition 24 *A trace τ is ℓ-synchronized if $\tau|_\ell$ is an element of the regular language $(P(\ell); V(\ell))^\star$.*

We are going, in what follows, to concentrate on ℓ-synchronized traces only. This is justified for two reasons. First, any $P(\ell)$ will have a matching $V(\ell)$ because the semaphore operations will be generated in traces by entry to and exit from critical regions $\texttt{with}\, \ell\, \texttt{do}\, C$. The second reason can be stated logically and operationally. Operationally, if one critical region for ℓ is nested within another region for the same ℓ then the inner region can never be executed. Logically, the proof rule for critical regions can never be used on the inner region, because the rule for $\texttt{with}\, \ell\, \texttt{do}\, C$ in Figure 3 deletes ℓ from the environment.

The set of traces $T(C)$ of a command C is defined in Figure 4. Most cases are straightforward. The traces of $\ell.C$ are obtained by restricting to the ℓ-synchronized traces of C and deleting ℓ-actions. The deletion of ℓ-actions is justified by Lemma 26, since ℓ-actions behave like `skip` when ℓ is mapped to emp by the environment η. The semantics of $\ell.C$ starts with a $V(\ell)$ and ends with a $P\ell$) to model the idea that the lock declaration begins by transferring state into the resource ℓ holds and terminates by releasing it. This follows the view from [22] of $P(\ell)$ and $V(\ell)$ as resource ownership transformers, a view that is formalized below using the annihilation and materialization operations discussed at the end of Section 3.2. The critical region $\texttt{with}\, \ell\, \texttt{do}\, C$ just inserts mutex operations before and after C. The traces of $C_1 \parallel C_2$ are interleavings, except that any time two primitive actions can potentially execute at the same time we insert a check for races. We remark that races are not *detected* at this stage: we merely insert check statements that will be evaluated at execution time.

7.2 Executing Traces

As an individual trace is just a sequential composition of simple commands, we can define its denotational semantics following Figure 1. In detail, the definition is

$$\begin{aligned}
[\![c]\!]v\eta &= v(c) \\
[\![\texttt{skip}]\!]v\eta\sigma &= \{\sigma\} \\
[\![C_1; C_2]\!]v\eta &= ([\![C_1]\!]v\eta); ([\![C_2]\!]v\eta) \\
[\![P(\ell)]\!]v\eta &= bla[emp, \eta(\ell)] \\
[\![V(\ell)]\!]v\eta &= bla[\eta(\ell), emp] \\
[\![\texttt{check}(c_1, c_2)]\!]v\eta &= check(v(c_1), v(c_2))
\end{aligned}$$

where $check(f,g)$ is defined as follows:

$$check(f,g)(\sigma) \;=\; \begin{cases} \{\sigma\} \text{ if } \exists \sigma_f \sigma_g.\ \sigma_f \bullet \sigma_g = \sigma \wedge \\ \qquad\qquad f(\sigma_f) \neq \top \wedge g(\sigma_g) \neq \top \\ \top \quad \text{otherwise} \end{cases}$$

In words, $check(f,g)(\sigma)$ faults if we cannot find a partition $\sigma_f \bullet \sigma_g$ of σ where the components of the partition contain sufficient resource for f and g individually. In case the entire state σ has enough resource for both f and g (meaning they don't deliver $\top$), $check(f,g)(\sigma)$ converts racing to faulting. In different models, this sense of race takes on a different import. For example, in the plain heap model, by this definition racing means that two operations touch the same location, even if they are only reading the same location, while in permission models two operations can read the same location without it being judged a race. Note, though, that this determination, whether or not we have a race, is not something that must be added to a model: it is always completely determined just by the $\bullet$ operation.

Lemma 25 *$[\![\texttt{check}(c_1,c_2)]\!]v\eta$ is local.*

Proof: It suffices to show that for all local actions f_1, f_2, function $check(f_1,f_2)$ is local. Consider σ, σ' such that $\sigma \bullet \sigma'$ is defined. We need to show that

$$check(f_1,f_2)(\sigma \bullet \sigma') \sqsubseteq check(f_1,f_2)(\sigma) * \{\sigma'\}.$$

When $check(f_1,f_2)(\sigma) = \top$, the rhs of the inequality is $\top$, so the inequality follows. Suppose $check(f_1,f_2)(\sigma) \neq \top$. Then, the rhs of the inequality is $\{\sigma \bullet \sigma'\}$, and there exists a splitting $\sigma_1 \bullet \sigma_2 = \sigma$ of σ with $f_i(\sigma_i) \neq \top$. By the locality of f_2, we have that $f_2(\sigma_2 \bullet \sigma') \sqsubseteq f_2(\sigma_2) * \{\sigma'\}$. So, $f_2(\sigma_2 \bullet \sigma') \neq \top$. Since $f_1(\sigma_1) \neq \top$, $check(f_1,f_2)(\sigma \bullet \sigma')$ is $\{\sigma_1 \bullet (\sigma_2 \bullet \sigma')\}$, the same set as $check(f_1,f_2)(\sigma) * \{\sigma'\}$. ■

A crucial property of trace execution is the following. It relies essentially on lock invariants being precise, and particularly Lemma 13 and the analysis of annihilation it provides in the context of the semantics just given for $V(\ell)$.

Lemma 26 *If τ is an ℓ-synchronized trace then*

$$[\![\, V(\ell); \tau; P(\ell)]\!]v\eta \sqsupseteq [\![\tau - \ell]\!]v\eta$$

Proof: For brevity we write $\tau[r]$ for $[\![\tau]\!]v(\eta|\ell{:=}r)$, and $P(r)$ (resp. $V(r)$) for $(P(\ell))[r]$ (resp. $(V(\ell))[r]$). Since $P(emp) = V(emp) = \texttt{skip}$, we obtain the result by proving

$$V(r); \tau[r]; P(r) \sqsupseteq \tau[emp].$$

The proof is by induction on the length of τ. If τ does not contain ℓ-actions, then $\tau = \tau[emp]$, and the result follows immediately from the following property

$$V(r); f; P(r) \sqsupseteq f. \qquad (1)$$

Otherwise, because of the ℓ-synchronized assumption, $\tau[r]$ must be of the form $\tau_1; P(r); \tau_2; V(r); \tau'[r]$ where τ_1, τ_2 do not contain ℓ-actions and τ' is ℓ-synchronized. We then have

$$\begin{array}{ll} & V(r); \tau_1; P(r); \tau_2; V(r); \tau'[r]; P(r) \\ \sqsupseteq & \tau_1; \tau_2; V(r); \tau'[r]; P(r) \\ \sqsupseteq & \tau_1; \tau_2; \tau'[emp] = \tau[emp]. \end{array}$$

The first inclusion uses the same result as the base case, and the last inclusion uses the induction hypothesis.

We are left with proving property (1). We prove the inclusion for all σ. If $V(r)\sigma = \top$, the conclusion is immediate. Otherwise, $\sigma = \sigma_1 \bullet \sigma_2$ for $\sigma_2 \in r$. We then have

$$\begin{array}{lll} (V(r); f; P(r))\sigma & \sqsupseteq & (f; P(r))\sigma_1 \\ & = & (f\sigma_1) * r \\ & \sqsupseteq & (f\sigma_1) * \{\sigma_2\} \\ & \sqsupseteq & f\sigma. \end{array}$$

In the first two steps we use the semantics of $V(r)$ and $P(r)$. In the last two steps we use monotonicity of $*$ and locality of f. ■

7.3 Interpreting the Logic

Definition 27 (Semantic Consequence Relation, II) *Given a set of traces S, we define the semantics $[\![S]\!]v\eta = \bigsqcup_{\tau \in S} [\![\tau]\!]v\eta$. We write*

$$\eta; \mathsf{Ax} \models_{\mathcal{I}} \{p\}\, C\, \{q\}$$

to mean that for all valuations v, if v satisfies Ax then $\langle\!\langle p \rangle\!\rangle\ [\![T(C)]\!]v\eta\ \langle\!\langle q \rangle\!\rangle$ is true according to Definition 6.

The following lemma is the essential part of the proof of soundness for parallel composition. It says that if a trace τ is obtained as an interleaving of two traces τ_1 and τ_2, and executing τ_1 and τ_2 sequentially on separate substates satisfies two postconditions, then executing τ on the combination of the separate states satisfies the separating conjunction of the postconditions[12].

Lemma 28 *If $\sigma = \sigma_1 \bullet \sigma_2$ and $[\![\tau_i]\!]v\eta\sigma_i \sqsubseteq q_i$ for $i = 1, 2$, and $\tau \in (\tau_1\ zip\ \tau_2)$ then $[\![\tau]\!]v\eta\sigma \sqsubseteq q_1 * q_2$.*

Proof: The proof is by induction on the definition of *zip* and *zip′*.

The first interesting case involves race checking. Consider $\tau_1 = c_1; \tau_1'$ and $\tau_2 = c_2; \tau_2'$. Then $\tau = \texttt{check}(c_1,c_2); \tau'$ for some $\tau' \in (\tau_1\ zip'\ \tau_2)$. Since $[\![c_i; \tau_i]\!]v\eta\sigma_i \sqsubseteq q_i$, we have $[\![c_i]\!]v\eta\sigma_i \neq \top$, hence

$$check([\![c_1]\!]v\eta, [\![c_2]\!]v\eta)(\sigma_1 \bullet \sigma_2) = \sigma_1 \bullet \sigma_2 \neq \top.$$

[12] This is a cousin of Brookes's Parallel Decomposition Lemma

That is, $[\![\texttt{check}(c_1, c_2)]\!]v\eta\sigma = \sigma$. The conclusion $[\![\texttt{check}(c_1, c_2); \tau']\!]v\eta\sigma \sqsubseteq q_1 * q_2$ follows directly by induction hypothesis on τ'.

The other interesting case is the interleaving case of zip' (the bottom right equality in Figure 4). Consider $\tau_1 = \alpha_1; \tau_1'$ and $\tau_2 = \alpha_2; \tau_2'$. and suppose that $\tau \in (\alpha_1; (\tau_1' \; zip \; (\alpha_2; \tau_2')))$ (the other case being symmetrical). Then there is $\tau' \in (\tau_1' \; zip \; (\alpha_2; \tau_2'))$ with $\tau = \alpha_1; \tau'$.

Since $[\![\tau_i]\!]v\eta\sigma_i \sqsubseteq q_i$ by assumption, we know that $[\![\tau_1']\!]v\eta\sigma_1' \sqsubseteq q_1$ for each $\sigma_1' \in v(\alpha_1)\sigma_1$, where $v(\alpha_i)\sigma_i \neq \top$ by the denotational semantics of sequential composition in Figure 1 and the fact that $q_1 \neq \top$. By induction hypothesis, for any such σ_1' where $\sigma_1' \# \sigma_2$, we have $[\![\tau']\!]v\eta(\sigma_1' \bullet \sigma_2) \sqsubseteq q_1 * q_2$. This says exactly that $[\![\tau']\!]v\eta(\sigma') \sqsubseteq q_1 * q_2$, for all $\sigma' \in v(\alpha_1)\sigma_1 * \{\sigma_2\}$. Since α_1 satisfies the locality condition we have $v(\alpha_1)(\sigma_1 \bullet \sigma_2) \sqsubseteq v(\alpha_1)\sigma_1 * \{\sigma_2\}$, and so by the denotational semantics of sequential composition we obtain $[\![\tau]\!]v\eta\sigma \sqsubseteq q_1 * q_2$. ■

Note the use of the locality property for the basic actions α (including the semaphore operations) near the end of this proof.

We now have all the information we need to prove our main soundness result.

Theorem 29 (Soundness, II) *All of the proof rules preserve validity. As a result,*

$$\eta; \mathsf{Ax} \vdash \{p\}\, C\, \{q\} \qquad \textit{implies} \qquad \eta; \mathsf{Ax} \models_{\mathcal{I}} \{p\}\, C\, \{q\}$$

Proof: The proof is by induction on the derivation of $\eta; \mathsf{Ax} \vdash \{p\}\, C\, \{q\}$. For the rules in Figure 2 the proof is straightforward. We consider the concurrency rules in Figure 3.

For the parallel rule, assume $\eta; \mathsf{Ax} \models_{\mathcal{I}} \{p_i\}\, C_i\, \{q_i\}$ for $i = 1, 2$. We need to show $\eta; \mathsf{Ax} \models_{\mathcal{I}} \{p_1 * p_2\}\, C_1 \parallel C_2\, \{q_1 * q_2\}$. Consider a valuation v that satisfies Ax and a trace $\tau \in T(C_1 \parallel C_2)$. We need to show that $\langle\!\langle p_1 * p_2 \rangle\!\rangle\, [\![\tau]\!]v\eta\, \langle\!\langle q_1 * q_2 \rangle\!\rangle$ is true. Take $\sigma = \sigma_1 \bullet \sigma_2$ such that $\sigma_i \in p_i$ for $i = 1, 2$. We need to show that $[\![\tau]\!]v\eta\sigma \sqsubseteq q_1 * q_2$. Since $\tau \in T(C_1 \parallel C_2)$, we have $\tau = \tau_1 \; zip \; \tau_2$ for $\tau_i \in T(C_i)$. By assumption, $[\![\tau_i]\!]v\eta\sigma_i \sqsubseteq q_i$ for $i = 1, 2$. Lemma 28 gives $[\![\tau]\!]v\eta\sigma \sqsubseteq q_1 * q_2$, as required.

For the lock declaration rule, assume $(\eta|\ell{:=}r); \mathsf{Ax} \models_{\mathcal{I}} \{p\}\, C\, \{q\}$. We need to show $\eta; \mathsf{Ax} \models_{\mathcal{I}} \{p * r\}\, \ell.C\, \{q * r\}$. Consider a valuation v that satisfies Ax and a trace $\tau \in T(\ell.C)$. We need to show that $\langle\!\langle p * r \rangle\!\rangle\, [\![\tau]\!]v\eta\, \langle\!\langle q * r \rangle\!\rangle$ holds. Take $\sigma \in p * r$. We need to prove that $[\![\tau]\!]v\eta\sigma \sqsubseteq q * r$. Since $\tau \in T(\ell.C)$, we have $(V(\ell); \tau'; P(\ell)) - \ell$ for ℓ-synchronized $\tau' \in T(C)$. By assumption and the semantics of $P(\ell)$ and $V(\ell)$ we have $[\![V(\ell); \tau'; P(\ell)]\!]v(\eta|\ell{:=}r)\sigma \sqsubseteq q * r$. Lemma 26 gives $[\![V(\ell); \tau'; P(\ell)]\!]v(\eta|\ell{:=}r) \sqsupseteq [\![\tau' - \ell]\!]v\eta$. Since $\tau = (\tau' - \ell)$, we have shown $[\![\tau]\!]v\eta\sigma \sqsubseteq q * r$, as required.

The proof for the $\texttt{with}\, \ell\, \texttt{do}\, C$ rule is straightforward.

For the critical region rule, the traces of $\texttt{with}\, \ell\, \texttt{do}\, C$ are of the form $P(\ell); \tau'; V(\ell)$. If we know that $\{p * r\} C \{q * r\}$ holds then we can reason in sequential separation logic about this trace as in this proof outline which shows intermediate assertions for use with the sequencing rule

$$\{p\}\; P(\ell)\; \{p * r\}\; \tau'\; \{q * r\}\; V(\ell) \{q\}$$

This overall pre and post is what we need to establish for any trace of $\texttt{with}\, \ell\, \texttt{do}\, C$. The given preconditions and postconditions for $P(\ell)$ and $V(\ell)$ follow from their semantic definitions as best local actions: you use p as a frame axiom in the P case, and q as a frame axiom in the V case. (This syntactic reasoning about the traces can be easily recast in more semantic terms.)

■

Because failure of a race check results in value $\top$, and because a Hoare triple is falsified by $\top$, the theorem also implies that any proven program is race-free. Of course, this notion of race-freedom is relative to the given separation algebra. In a plain heap model $[L \rightharpoonup_{fin} R]$ any access to a common location is regarded as a race, while in permission models concurrent reads are not judged as racy. The point is that the combining operation $\bullet$ of the separation algebra contains all the information that is needed to define "race" for the model.

In this paper we have not addressed further issues in the semantics of concurrency such as independence [15] or granularity [34, 10]. We hope that these issues can be approached in a general setting like that of the present paper.

Remarks on other rules. We did not include the auxiliary variable elimination rule, which comes to separation logic from Owicki and Gries. This rule requires variables to be present in Σ, while the general notion of separation algebra does not require variables to be present. It is easy to validate the rule in Σ's that contain variables-as-resource.

Part of the issue dealt with by auxiliary variables can be seen, though, on the general level by appeal to a cousin of Milner's expansion law. For example,

$$\{p\}\; (\texttt{with}\, \ell\, \texttt{do}\, C_1 + \texttt{with}\, \ell\, \texttt{do}\, C_2); (C_1' \parallel C_2')\; \{q\}$$

it follows that

$$\{p\}\; ((\texttt{with}\, \ell\, \texttt{do}\, C_1); C_1') \parallel ((\texttt{with}\, \ell\, \texttt{do}\, C_2); C_2')\; \{q\}$$

and this inference holds in the general models.[13]

Neither did we explicitly include the Hoare logic rule for introducing existentials, which is crucial in [21] for deriving

[13] Note that, in contrast to Milner, expansion requires protection by a critical region (or else racing could ensue); see [15] for further discussion and analysis.

completeness results. The existential rule, semantically, just boils down to the disjunction rule, as in the inference

$$\frac{\{p(d))\}\, C\, \{q(d)\}, \qquad \text{all } d \in D}{\{\bigvee_{d\in D} p(d))\}\, C\, \{\bigvee_{d\in D} q(d)\}}$$

where $p, q : D \to P(\Sigma)$ for some set D.

Finally, we did not include a version of the substitution rule from [21]. This would require formulation of a notion of parameterized local action.

8 Operational Semantics

The interleaving semantics is still removed from the way that programs work in two respects (even under timeslicing on a single-CPU machine). The first is that the semantics of lock declarations $\ell.C$ simply drops all ℓ-actions from traces. The second is that we presume that traces have structure inspired by the intuition of mutual exclusion, but we do not explicitly represent blocking or busy-waiting in the semaphores used in their interpretation. To complete our story we should justify the structure in the trace model further, and we do so in this section by connecting to an operational semantics in the style of Plotkin.

The operational semantics, defined in Figure 8, acts on intermediate configurations (ρ, C, σ), where $\rho : \mathit{Locks} \rightharpoonup_{\mathit{fin}} \{\mathit{free}, \mathit{busy}\}$ is the environment for locks, C the current command, and σ the current state. The semantics corresponds to a blocking interpretation of the semaphores, in that the rule for P will not fire when its lock is busy.

A single reduction step is described by the relation $(\rho, C, \sigma) \overset{\alpha}{\rightsquigarrow}_v d$, where the result d can be a new intermediate configuration (ρ', C', σ'), a terminal configuration (ρ', σ'), or the error value fault. The annotation α on $\overset{\alpha}{\rightsquigarrow}_v$, used to ease the comparison with the trace semantics, indicates the current action, which ranges over the following set α^+ (of "extended" actions)

$$\alpha^+ \quad ::= \quad \cdot \mid c \mid \mathtt{skip} \mid \mathtt{check}(c, c) \mid \mathit{act}(\ell) \mid \mathit{decl}(\ell)$$

where $\cdot$ is the unit of action composition, and $\mathit{decl}(\ell)$ indicates the declaration of lock ℓ; the ℓ actions are not deleted from the traces in the operational semantics. Most of the rules in Figure 8 are straightforward. The rule for $C_1 \parallel C_2$ produces the auxiliary command $C_1 * C_2$. The former performs race checking, and the latter pure interleaving.

A trace τ^+ (an "extended" trace) is a sequence of actions α^+, where $\mathit{decl}(\ell)$ acts as a binder for the reminder of the sequence. We write $(\rho, C, \sigma) \overset{\tau^+}{\rightsquigarrow}_v d$ for the many-step reduction. We write $|\tau^+|$ for the trace obtained from τ by deleting all the bound ℓ-actions. We write $\tau_1^+ \le \tau_2^+$ if $\exists \tau_3^+ . \tau_1^+; \tau_3^+ = \tau_2^+$.

We now relate the τ^+ actions from the operational semantics with τ actions from the denotational semantics, and prove soundness as a corollary.

Theorem 30 (Operational Adequacy Theorem) *Let C be a command with no free lock variables, and with no nested $\mathtt{with}\,\ell\,\mathtt{do}\,C'$ commands over the same ℓ*[14]*. Let η_0 and ρ_0 be the empty environments. Then the following hold:*

- *If $(\rho_0, C, \sigma) \overset{\tau^+}{\rightsquigarrow}_v \mathsf{fault}$ then $\exists \tau \in T(C). |\tau^+| \le \tau$ and $[\![\tau]\!] v\eta(\sigma) = \top$.*
- *If $(\rho_0, C, \sigma) \overset{\tau^+}{\rightsquigarrow}_v (\rho', \sigma')$ then $\exists \tau \in T(C). |\tau^+| = \tau$ and $\sigma' \in [\![\tau]\!] v\eta(\sigma)$.*

The proof of this theorem is given in the appendix.

Definition 31 (Semantic Consequence Relation, III) *Let C be a command with no free lock variables, and let η_0 be the empty environment. We write*

$$\eta_0; \mathsf{Ax} \models_{op} \{p\}\, C\, \{q\}$$

to mean that for all valuations v and states $\sigma \in p$, if v satisfies Ax then

- *$(\eta_0, C, \sigma) \overset{\tau^+}{\not\rightsquigarrow}_v \mathsf{fault}$*
- *$(\eta_0, C, \sigma) \overset{\tau^+}{\rightsquigarrow}_v (\eta', \sigma')$ implies $\sigma' \in q$.*

Corollary 32 (Soundness, III) *Let C be a command with no free lock variables, and let η_0 be the empty environment.*

$$\eta_0; \mathsf{Ax} \vdash \{p\}\, C\, \{q\} \qquad \textit{implies} \qquad \eta_0; \mathsf{Ax} \models_{op} \{p\}\, C\, \{q\}$$

Proof: The proof is an immediate consequence of Theorems 29 and 30. The no-nested condition of Theorem 30 is a consequence of the assumption that $\{p\}\, C\, \{q\}$ is derivable. ■

Note that this corollary does *not* show that the proof rules themselves are all sound in the operational semantics. It just shows that the overall conclusion of a Hoare triple corresponds to what we expect from the operational semantics.

8.1 Proof of Operational Adequacy

As usual, to obtain a strong enough induction hypothesis in the proof of Theorem 30 we need to show a stronger result. In preparation for this we now define well-formed commands, which generalize the notion of nesting-free command to intermediate configurations of the operational semantics, and constrain the occurrences of $V(\ell)$ to be in continuation position.

[14] The no-nesting condition is necessary to ensure that the operational semantics produces (initial segments of) ℓ-synchronized traces.

$$\frac{\sigma' \in v(c)(\sigma)}{(\rho, c, \sigma) \overset{c}{\rightsquigarrow}_v (\rho, \sigma')} \qquad \frac{v(c)(\sigma) = \top}{(\rho, c, \sigma) \overset{c}{\rightsquigarrow}_v (\rho, \mathsf{fault})}$$

$$\frac{\rho(\ell) = \mathit{free}}{(\rho, \mathtt{with}\,\ell\,\mathtt{do}\,C, \sigma) \overset{P(\ell)}{\rightsquigarrow}_v ((\rho|\ell{:=}\mathit{busy}), (C;\, V(\ell)), \sigma)}$$

$$\frac{}{(\rho,\, V(\ell), \sigma) \overset{V(\ell)}{\rightsquigarrow}_v ((\rho|\ell{:=}\mathit{free}), \sigma)}$$

$$\frac{\ell \notin \mathit{dom}(\rho)}{(\rho, \ell.C, \sigma) \overset{\mathit{decl}(\ell)}{\rightsquigarrow}_v ((\rho|\ell{:=}\mathit{free}), C, \sigma)}$$

$$\frac{}{(\rho, C^\star, \sigma) \overset{\cdot}{\rightsquigarrow}_v (\rho, \mathtt{skip} + (C; C^\star), \sigma)}$$

$$\frac{}{(\rho, C_1 + C_2, \sigma) \overset{\cdot}{\rightsquigarrow}_v (\rho, C_1, \sigma)} \qquad \frac{}{(\rho, C_1 + C_2, \sigma) \overset{\cdot}{\rightsquigarrow}_v (\rho, C_2, \sigma)}$$

$$\frac{\mathit{check}(v(c_1), v(c_2))(\sigma) \neq \top}{(\rho, (c_1; C_1) \parallel (c_2; C_2), \sigma) \overset{\mathtt{check}(c_1,c_2)}{\rightsquigarrow}_v (\rho, (c_1; C_1) * (c_2; C_2), \sigma)} \qquad \frac{C_1 \not\equiv c_1; C_1' \text{ or } C_2 \not\equiv c_2; C_2'}{(\rho, C_1 \parallel C_2, \sigma) \overset{\cdot}{\rightsquigarrow}_v (\rho, C_1 * C_2, \sigma)}$$

$$\frac{\mathit{check}(v(c_1), v(c_2))(\sigma) = \top}{(\rho, (c_1; C_1) \parallel (c_2; C_2), \sigma) \overset{\mathtt{check}(c_1,c_2)}{\rightsquigarrow}_v \mathsf{fault}}$$

$$\frac{(\rho, C_1, \sigma) \overset{\alpha}{\rightsquigarrow}_v (\rho', C_1', \sigma')}{(\rho, C_1 * C_2, \sigma) \overset{\alpha}{\rightsquigarrow}_v (\rho', C_1' \parallel C_2, \sigma')} \qquad \frac{(\rho, C_1, \sigma) \overset{\alpha}{\rightsquigarrow}_v (\rho', \sigma')}{(\rho, C_1 * C_2, \sigma) \overset{\alpha}{\rightsquigarrow}_v (\rho', C_2, \sigma')} \qquad \frac{(\rho, C_1, \sigma) \overset{\alpha}{\rightsquigarrow}_v \mathsf{fault}}{(\rho, C_1 * C_2, \sigma) \overset{\alpha}{\rightsquigarrow}_v \mathsf{fault}}$$

$$\frac{(\rho, C_2, \sigma) \overset{\alpha}{\rightsquigarrow}_v (\rho', C_2', \sigma')}{(\rho, C_1 * C_2, \sigma) \overset{\alpha}{\rightsquigarrow}_v (\rho', C_1 \parallel C_2', \sigma')} \qquad \frac{(\rho, C_2, \sigma) \overset{\alpha}{\rightsquigarrow}_v (\rho', \sigma')}{(\rho, C_1 * C_2, \sigma) \overset{\alpha}{\rightsquigarrow}_v (\rho', C_1, \sigma')} \qquad \frac{(\rho, C_2, \sigma) \overset{\alpha}{\rightsquigarrow}_v \mathsf{fault}}{(\rho, C_1 * C_2, \sigma) \overset{\alpha}{\rightsquigarrow}_v \mathsf{fault}}$$

$$\frac{(\rho, C_1, \sigma) \overset{\alpha}{\rightsquigarrow}_v (\rho', C_1', \sigma')}{(\rho, C_1; C_2, \sigma) \overset{\alpha}{\rightsquigarrow}_v (\rho', C_1'; C_2, \sigma')} \qquad \frac{(\rho, C_1, \sigma) \overset{\alpha}{\rightsquigarrow}_v (\rho', \sigma')}{(\rho, C_1; C_2, \sigma) \overset{\alpha}{\rightsquigarrow}_v (\rho', C_2, \sigma')} \qquad \frac{(\rho, C_1, \sigma) \overset{\alpha}{\rightsquigarrow}_v \mathsf{fault}}{(\rho, C_1; C_2, \sigma) \overset{\alpha}{\rightsquigarrow}_v \mathsf{fault}}$$

Figure 5. Small-step Operational Semantics

Definition 33 *A command C' is defined to be in continuation position in C if*

- $C \equiv C'$, *or*
- $C \equiv C_1; C_2; C_3$ *and C' is in continuation position in C_2, or*
- $C \equiv C_1 * C_2$ *and C' is in continuation position in C_1 or in C_2.*

Definition 34 (Well-Formed Commands) *A command C is well-formed w.r.t. an environment ρ if*

- *C does not contain nested $(\texttt{with}\,\ell\,\texttt{do}-)$ commands over the same lock ℓ;*
- *the free lock variables of C are contained in $dom(\rho)$;*
- *if $\rho(\ell) = free$ then C does not contain $V(\ell)$;*
- *if $\rho(\ell) = busy$ then following hold:*
  - *C does not contain $\texttt{with}\,\ell\,\texttt{do}-$;*
  - *C contains exactly one $V(\ell)$ in continuation position.*

The set of traces $T(C)$ is defined for the extended commands used in the operational semantics as:

$$T(V(\ell)) = \{V(\ell)\} \quad T(C_1 * C_2) = \{\tau_1 \mathit{zip}' \tau_2 \mid \tau_i \in T(C_i)\}$$

We now define ρ-synchronized traces, to capture the possible synchronization patterns starting from an intermediate configuration of the operational semantics.

Definition 35 *A trace τ of ℓ-actions for $\ell \in dom(\rho)$ is said to be ρ-synchronized if*

- *$\tau = \cdot$ (the empy trace) and $\rho(\ell) = free$ for all $\ell \in dom(\rho)$; or*
- *$\tau = P(\ell); \tau'$ and $\rho(\ell) = free$ and τ' is ρ'-synchronized for $\rho' = (\rho|\ell{:=}busy)$; or*
- *$\tau = V(\ell); \tau'$ and $\rho(\ell) = busy$ and τ' is ρ'-synchronized for $\rho' = (\rho|\ell{:=}free)$.*

A general trace τ is said to be ρ-synchronized if $\tau|_{dom(\rho)}$ is ρ-synchronized.

This is the stronger version of the Operational Adequacy Theorem.

Proposition 36 *Let C be well-formed w.r.t. ρ and let η be an environment such that $dom(\eta) = dom(\rho)$ and $\forall \ell \in dom(\eta).\, \eta(\ell) = emp$. Then the following hold:*

1. $\forall \sigma, \forall \tau^+$
   - $(\rho, C, \sigma) \overset{\tau^+}{\rightsquigarrow_v}$ fault *implies* $\exists \tau \in T(C).\, |\tau^+| \leq \tau$ *and τ is ρ-synchronized and* $[\![\tau]\!] v\eta(\sigma) = \top$*;*
   - $(\rho, C, \sigma) \overset{\tau^+}{\rightsquigarrow_v} (\rho', \sigma')$ *implies* $\exists \tau \in T(C).\, |\tau^+| = \tau$ *and τ is ρ-synchronized and $\sigma' \in [\![\tau]\!] v\eta(\sigma)$.*
2. $\forall \sigma, \forall \tau \in T(C)$
   - *if τ is ρ-synchronized and* $[\![\tau]\!] v\eta(\sigma) = \top$ *then* $\exists \tau^+.\, |\tau^+| \leq \tau$ *and* $(\rho, C, \sigma) \overset{\tau^+}{\rightsquigarrow_v}$ fault*;*
   - *if τ is ρ-synchronized and $\sigma' \in [\![\tau]\!] v\eta(\sigma)$ then* $\exists \rho'.\, \exists \tau^+.\, |\tau^+| = \tau$ *and* $(\rho, C, \sigma) \overset{\tau^+}{\rightsquigarrow_v} (\rho', \sigma')$.

Proof: The proof of part 1 is by induction on the length of the derivation of $(\rho, C, \sigma) \overset{\tau^+}{\rightsquigarrow_v} d$.

The most interesting case is when $C \equiv \ell.C_1$ and $(\rho, C, \sigma) \overset{\tau^+}{\rightsquigarrow_v}$ fault. Then $(\rho, \ell.C_1, \sigma) \overset{decl(\ell)}{\rightsquigarrow_v} (\rho_1, C_1, \sigma)$ and $(\rho_1, C_1, \sigma) \overset{\tau_1^+}{\rightsquigarrow_v}$ fault with $\rho_1 = (\rho|\ell{:=}free)$ and $\tau^+ = decl; \tau_1^+$. By induction hypothesis $\exists \tau_1 \in T(C_1).\, |\tau_1^+| \leq \tau_1$ and τ_1 is ρ_1-synchronized and $[\![\tau_1]\!] v\eta_1(\sigma) = \top$ where $\eta_1 = (\eta|\ell{:=}emp)$. Since τ_1 is ρ_1-synchronized and $\rho_1(\ell) = free$, we have that τ_1 is ℓ-synchronized. Let $\tau = \tau_1 - \ell$. By definition of trace set we have that $\tau \in T(\ell.C_1)$. Observe also that $|\tau^+| \leq \tau$. Finally, since $\eta_1(\ell) = emp$, we have $[\![\tau]\!] v\eta(\sigma) = [\![\tau_1]\!] v\eta_1(\sigma) = \top$, which concludes the case. The case when $(\rho, C, \sigma) \overset{\tau^+}{\rightsquigarrow_v} (\rho', \sigma')$ is analogous.

Case $C \equiv \texttt{with}\,\ell\,\texttt{do}\,C_1$ and $(\rho, C, \sigma) \overset{\tau^+}{\rightsquigarrow_v}$ fault. Then $(\rho, \texttt{with}\,\ell\,\texttt{do}\,C_1, \sigma) \overset{P(\ell)}{\rightsquigarrow_v} (\rho_1, (C_1; V(\ell)), \sigma)$ and $(\rho_1, (C_1; V(\ell)), \sigma) \overset{\tau_1^+}{\rightsquigarrow_v}$ fault with $\rho_1 = (\rho|\ell{:=}busy)$ and $\tau^+ = P(\ell); \tau_1^+$. Since C is well-formed w.r.t. ρ then $\rho(\ell) = free$. Furthermore, $(C_1; V(\ell))$ is well-formed w.r.t. ρ_1. We can then apply the induction hypothesis and obtain $\exists \tau_1 \in T(C_1; V(\ell)).\, |\tau_1^+| \leq \tau_1$ and τ_1 is ρ_1-synchronized and $[\![\tau_1]\!] v\eta(\sigma) = \top$. Notice that $\tau_1 = \tau_1'; V(\ell)$ for $\tau_1' \in T(C_1)$. Let $\tau = P(\ell); \tau_1'; V(\ell)$. Then $\tau \in T(C)$ by construction. Also, $|\tau^+| \leq \tau$. Finally, τ is ρ-synchronized since τ_1 is ρ_1-synchronized, and $[\![\tau]\!] v\eta(\sigma) = \top$ since $\eta(\ell) = emp$. This concludes the case. The case when $(\rho, C, \sigma) \overset{\tau^+}{\rightsquigarrow_v} (\rho', \sigma')$ is analogous.

[More cases to be added]

The proof of part 2 is by induction on the lexicographic order:

i) size of the current command, where the size relation $>$ is the well-founded relation defined by
   - $C^\star > (C; \ldots; C)$
   - $\ell.C > C$
   - $C_1 + C_2 > C_i \quad (\text{for } i \in 1..2)$

- $C_1 \parallel C_2 > C_1 * C_2$

ii) length of τ.

Notice that the lexicographic order is necessary because case $\ell.C$ requires the application of the induction hypothesis to a longer trace.

To illustrate the use of the size relation, first consider the case when $C \equiv C_1^\star$ and $[\![\tau]\!]v\eta(\sigma) = \top$. Then $\tau = \tau_1; \cdots; \tau_n$ with $\tau_i \in T(C_1)$ for $i \in 1..n$. Let $C' = C_1; \ldots; C_1$ with n occurrences of C_1. By construction $C^\star > C'$ and $\tau \in T(C')$ and C' is well-formed w.r.t. ρ. We can then apply the induction hypothesis and obtain the conclusion immediately. The case when $[\![\tau]\!]v\eta(\sigma) \neq \top$ is analogous.

Case $C \equiv \ell.C_1$ when $[\![\tau]\!]v\eta(\sigma) = \top$. Then $\tau = (V(\ell); \tau_1; P(\ell)) - \ell$ for ℓ-synchronized $\tau_1 \in T(C_1)$. Let $\rho_1 = (\rho|\ell{:=}\mathit{free})$ and $\eta_1 = (\eta|\ell{:=}\mathit{emp})$. Then we have that C_1 is well-formed w.r.t. ρ_1 and τ_1 is ρ_1-synchronized and $[\![\tau_1]\!]v\eta_1(\sigma) = \top$. Since $\ell.C_1 > C_1$ we can apply the induction hypothesis and obtain $\exists \tau_1^+. |\tau_1^+| \leq \tau_1$ and $(\rho_1, C_1, \sigma) \overset{\tau_1^+}{\rightsquigarrow}_v$ fault. Let $\tau^+ = \mathit{decl}(\ell); \tau$. Then $|\tau^+| \leq \tau$ and $(\rho, C, \sigma) \overset{\tau^+}{\rightsquigarrow}_v$ fault, which concludes the case. The case when $[\![\tau]\!]v\eta(\sigma) \neq \top$ is analogous. ■

9 Conclusion and Related Work

There are three main precursors to this work. The first is the model theory of BI [23, 30], which Pym emphasized can be understood as providing a general model of resource. At first, models of BI were given in terms of total commutative monoids and then, prodded by the development in [16], in terms of partial monoids. Separation algebras are a special case of the models in [29], corresponding to (certain) Boolean BI algebras.

The second precursor is [37], which identified Safety Monotonicity and the Frame Property, conditions on an operational semantics corresponding to the frame rule. In comparison to [37] the main step forward – apart from concurrency and consideration of a class of models rather than a single one – is the use of functions into the poset $P(\Sigma)^\top$ instead of relations satisfying Safety Monotonicity and the Frame Property. This shift has led to dramatic simplifications. For example, the formulation of the best state transformers (Definition 10) is much simpler and easier to understand than its relational cousin in [24].

The relational version of local actions for separation algebras was used in [17]. Again, that notion of local action was based on Safety Monotonicity and the Frame Property, prior to our move to the topped powerset. Also, the focus there was on program refinement, rather than abstract separation logic.

The third precursor is Brookes's proof of the soundness of concurrent separation logic, for the RAM model [12]. One of the key insights of Brookes's work shows up again here; namely,, where the semantics is factored into two parts: i) a (stateless) trace model, where interleaving is done on the (syntactic) actions; ii) a semantics that interprets the actions' effects on states. With this factoring concurrency is handled in the action-trace model, in a way that is largely independent of the meanings of the primitive commands, and this means that the imperative (state transforming) meaning of commands needs only to be given in a sequential setting, for the traces, after concurrency has been resolved by interleaving. We attempted to prove soundness for an interleaving operational semantics of concurrency in the style of Plotkin, but doing so directly turned out to be prohibitively difficult, particularly in the case of lock (resource) declarations: use of a deletion operation on traces to filter out locally-declared locks makes declarations much easier to handle.

There are, though, several differences in our semantics and Brookes's. Most importantly, Brookes's traces are made up of items that are tightly tied to the RAM model, and are not themselves primitive commands in the language under consideration. Because we use the primitive commands themselves (as well as semaphore operations) as the elements of the interleaving, we are able to see that soundness depends *only* on the locality properties of the primitive commands: this gives, we feel, a sharper explanation of the conditions needed for soundness, and it transfers immediately to the more general class of models. Also, because of this choice our proof of soundness for an infinite class of models is (arguably) simpler than Brookes's proof for a single model; for example, our Lemma 28 is considerably simpler in its statement than the Parallel Decomposition Lemma, which plays an analogous role in Brookes's work. There are also other detailed differencessuch as: that we detect races while executing traces, after interleaving, while Brookes detects races at an earlier stage (during interleaving); and that we connect to a blocking interpretation of critical regions, where Brookes's are like uses busy waiting.

This being said, we fully acknowledge the leading influence of Brookes's remarkable semantic analysis.

In formulating our results we have not aimed for the maximum possible generality. Our results on the sequential subset of ASL could almost certainly be redone using context logic [13], which replaces the primitive of separation by the more general primitive of pulling a state apart into a state-with-a-hole (a context) and its filler; however, more work on the basics of context logic would be required to generalize our more significant results on concurrency. Abstract predicates and higher-order separation logic have been used to approach modules, while the treatment here

avoids higher-order predicates [25, 6]. Finally, it would be desirable to go beyond algebra and formulate the essence of local action at a categorical level, perhaps on the level of the general theory of effects [18, 28].

Rather than shooting for maximum generality, we have chosen a tradeoff between complexity and generality, that demonstrates the existence of at least one abstract account of a basis for local reasoning about programs. It is, though, but one possible path through the subject. Recent work on the logic for low-level code often choose to allow commands that violate Safety Monotonicity and the Frame Property, opting instead to have locality concentrated in a novel interpretation of Hoare triples [7], or to express locality explicitly by polymorphism [19, 3]. The work on Boogie [2] achieves modularity using ideas that have at least some hints of the primitives in separation logic [20], and a study of the abstract principles underlying Boogie could be valuable. And, it does not appear that the locality condition in our model can be used to explain the "procedure local semantics" of [35]. Generally, we believe that there is more to be learnt about local reasoning about programs, particularly concurrent programs, and about semantics expressing local program behaviour.

Finally, we have shown that one can get some way using just the notion of local action on a separation algebra, but it appears that one can go still further. For example, we can define the parallel composition of local actions by

$$(f_1 \parallel f_2)\sigma \;=\; \bigsqcap\{(f_1\sigma_1) * (f_2\sigma_2) \mid \sigma = \sigma_1 \bullet \sigma_2\}$$

We would not advocate this as a parallel semantics of composite commands (it is an overapproximation of the expected meaning), but it works well for primitive commands that are considered to be atomic. Using this notion, one could conceivably obtain a truly concurrent semantics, where primitive actions could be run at the same time rather than be interleaved, in arbitrary separation algebras.

Acknowledgments. We are grateful to Philippa Gardner and Martin Hyland for trenchant criticisms at decisive points in this work.

We acknowledge the financial support of the EPSRC.

References

[1] A. Ahmed, L. Jia, and D. Walker. Reasoning about hierarchical storage. In *18th LICS*, pp33-44, 2003.

[2] M. Barnett, R. DeLine, M. Fahndrich, K.R.M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[3] N. Benton. Abstracting allocation. In *CSL 2006*, pages 182–196, 2006.

[4] J. Berdine, C. Calcagno, and P.W. O'Hearn. Smallfoot: Automatic modular assertion checking with separation logic. In *4th FMCO*, pp115-137, 2006.

[5] J. Berdine, B. Cook, D. Distefano, and P. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *18th CAV*, pp386-400, 2006.

[6] L. Birkedal and N. Torp-Smith. Higher-order separation logic and abstraction. submitted, 2005.

[7] L. Birkedal and H. Yang. Relational parametricity and separation logic. In *10th FOSSACS*, to appear, 2007.

[8] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *32nd POPL*, pp59–70, 2005.

[9] R. Bornat, C. Calcagno, and H. Yang. Variables as resource in separation logic. In *21st MFPS*, pp247–276, 2005.

[10] S. Brookes. A grainless semantics for parallel programs with shared mutable data. In *21st MFPS*, pp277-307, 2005.

[11] S. Brookes. Variables as resource for shared-memory programs: Semantics and soundness. In *22nd MFPS*, pp123–150, 2006.

[12] S. D. Brookes. A semantics for concurrent separation logic. *Proceedings of the 15th CONCUR*, London. August, 2004.

[13] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic and tree update. In *32nd POPL*, pp271-282, 2005.

[14] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *16th ESOP*, to appear, 2007.

[15] J. Hayman and G. Winskel. Independence and concurrent separation logic. In *21st LICS*, pp147-156, 2006.

[16] S. Isthiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *28th POPL*, pages 36–49, 2001.

[17] I. Mijajlovic, N. Torp-Smith, and P. O'Hearn. Refinement and separation contexts. Proceedings of FSTTCS, LNCS 3328, Chennai, December, 2004.

[18] E. Moggi. Notions of computation and monads. *Information and Computation* 93(1), pp55-92, 1991.

[19] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in hoare type theory. In *ICFP 2006*, pages 62–73, 2006.

[20] D. A. Naumann and M. Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state. In *19th LICS*, 2004.

[21] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *15th CSL*, pp1–19, 2001.

[22] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, May 2007. Preliminary version appeared in CONCUR'04, LNCS 3170, 49–67.

[23] P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, June 99.

[24] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *31st POPL*, pages 268–280, 2004.

[25] M. Parkinson and G. Bierman. Separation logic and abstraction. In *32nd POPL*, pp59–70, 2005.

[26] M. Parkinson, R. Bornat, and C. Calcagno. Variables as resource in Hoare logics. In *21st LICS*, 2006.

[27] M. Parkinson, R. Bornat, and P. O'Hearn. Modular verification of a non-blocking stack. In *34th POPL*, 2007.

[28] A.J. Power and E.P. Robinson. Premonoidal categories and notions of computation. *Math. Struct. Comp. Sci.* 7(5), pp453-468, 1997.

[29] D. Pym, P. O'Hearn, and H. Yang. Possible worlds and resources: the semantics of BI. *Theoretical Computer Science*, 315(1):257–305, 2004.

[30] D.J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*. Applied Logic Series. Kluwer Academic Publishers, 2002.

[31] W. Reisig. Petri nets. EATCS Monographs, vol. 4, 1995.

[32] J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In Jim Davies, Bill Roscoe, and Jim Woodcock, editors, *Millennial Perspectives in Computer Science*, pages 303–321, Houndsmill, Hampshire, 2000. Palgrave.

[33] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th LICS*, pp 55-74, 2002.

[34] J. C. Reynolds. Towards a grainless semantics for shared variable concurrency. In *24th FSTTCS*, pp35-48, 2004.

[35] N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. *32nd POPL*, pp296–309, 2005.

[36] N. Torp-Smith, L. Birkedal, and J. Reynolds. Local reasoning about a copying garbage collector. In *31st POPL*, pp220–231, 2004.

[37] H. Yang and P. O'Hearn. A semantic basis for local reasoning. In *5th FOSSACS*, LNCS 2303, 2002.