Local Action and Abstract Separation Logic

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 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 [33, 17, 23, 34]. 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 [17, 23] 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. 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 and further papers on concurrent separation logics [10, 11, 16, 15], 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 class of models.

This paper will not contain any examples of using the logic; see, e.g., [36, 28, 5, 4] and their references.

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, 27] for further discussion. This is for theoretical simplicity; our re-

sults 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 [23, 34]. 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 [33, 17]. 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 cancellativity 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, 27] 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 [32].

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 [30]. Because they are left adjoints they preserve all joins, so we automatically get two distributive laws

$$\begin{array}{rcl} \bigsqcup X * p & = & \bigsqcup \{x * p \mid x \in X\} \\ \bigsqcup X \sqcap p & = & \bigsqcup \{x \sqcap p \mid x \in X\}. \end{array}$$

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 of Predicates.

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 nonempty $X \subseteq P(\Sigma)$,

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

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

$$\bigsqcap X * \{\sigma\} = \bigsqcap \{x * \{\sigma\} \mid x \in X\} \text{ for all nonempty } X.$$

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 [25, 12, 22]. This property is used in the characterization of the lattice structure of local functions, in the definition of the "best" or largest local action below, and again later when we turn to concurrency.

3 Local Actions

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

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

$(\sigma, \sigma') \in R$ signifies that the program can deliver final state σ' when started in σ, and $(\sigma, \texttt{fault}) \in R$ 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: $(\sigma, \texttt{fault}) \notin R$ and $\sigma \preceq \sigma'$ implies $(\sigma', \texttt{fault}) \notin R$.

2. Frame Property: If $(\sigma_0, \texttt{fault}) \notin R, \sigma = \sigma_0 \bullet \sigma_1$ 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 [17, 37, 34]. 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 represents divergence. To account 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.

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

Technical Development. We now make the main definitions following on from these ideas.

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***:*

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

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 $[\Sigma \to P(\Sigma)^\top]$).*

The proof is straightforward, but it is instructive to give part of it to show a use of precision in action.

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

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

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

Given any precondition p_1 and postcondition p_2, we can define the best or largest local action satisfying the triple $\langle\!\langle p_1 \rangle\!\rangle\ -\ \langle\!\langle p_2 \rangle\!\rangle$[2].

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

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

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

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

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

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 $g^\dagger : P(\Sigma)^\top \to P(\Sigma)^\top$.[3] 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 $\bullet$.

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, in the sense that it simply removes part of the state satisfying p.

4 Programming Language

The commands of our language are as follows:

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

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

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

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

Figure 1. Denotational Semantics

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 $x, y \in Var$ and $l \in L$, we can define

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

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 if it were a ghost variable.

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.[4]

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 [17], 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 $\texttt{assume}(B)$ for some intuitionistic predicate $B \in P(\Sigma)$. Our valuation v has to map $\texttt{assume}(B)$ to a local action $v(\texttt{assume}(B)) \in LocAct$ 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

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

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.[5]

5 Sequential Abstract Separation Logic

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).

[4] This 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.

[5] 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.

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 12 (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.[6]

Definition 13 (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.*

The semantics of judgements is based on a notion of valuation, that maps primitive commands to local actions.

Definition 14 (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 15 (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 16 (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 of the frame rule follows Definition 8, and the other rules are straightforward. We have also proven the converse of Theorem 16, included in the longer version of this paper [13].

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

6 A Logic for Concurrency

We add three new command forms for parallel composition, lock declarations $\ell.C$, and critical sections.[7]

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

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 $\mathtt{with}\,\ell\,\mathtt{do}\,C$ 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.

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

The need for precision of these predicates (the lock invariants) can be seen from the Reynolds counterexample for concurrent separation logic [22, 12].

The judgments of the concurrency logic are of the form

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

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

Definition 17 (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*

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

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 allows us to formulate our model for arbitrary separation algebras.

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

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

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:

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

Figure 4. Trace Semantics

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 18 *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:

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

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 19 *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 21, 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 lock ℓ and terminates by releasing it. This follows the view of $P(\ell)$ and $V(\ell)$ as resource ownership transformers [22], formalized below using the annihilation and materialization operations discussed at the end of Section 3. 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.

Executing Traces As an individual trace is just a sequential composition of simple commands, we can define its de-

notational semantics following Figure 1.

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

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 $\top$), $check(f, g)(\sigma)$ skips. 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 20 $[\![\texttt{check}(c_1, c_2)]\!]v\eta$ *is local.*

A crucial property of trace execution is the following, which it relies essentially on lock invariants being precise.

Lemma 21 *If τ is an ℓ-synchronized trace then*

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

Interpreting the Logic. We can now define a semantics of Hoare triples that takes the lock invariants into account.

Definition 22 (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.

Lemma 23 (Parallel Decomposition Lemma) *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

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

That is, $[\![\texttt{check}(c_1, c_2)]\!]v\eta\sigma = \sigma$. Induction hypothesis on τ' gives $[\![\texttt{check}(c_1, c_2); \tau']\!]v\eta\sigma \sqsubseteq q_1 * q_2$.

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 execution semantics of sequential composition 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 $[\![\tau]\!]v\eta\sigma \sqsubseteq q_1 * q_2$ by the semantics of sequential composition. ■

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

Theorem 24 (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 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 require $\langle\!\langle p_1 * p_2 \rangle\!\rangle\, [\![\tau]\!]v\eta\, \langle\!\langle q_1 * q_2 \rangle\!\rangle$. 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 23 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 21 gives $[\![V(\ell); \tau'; P(\ell)]\!]v(\eta|\ell{:=}r) \sqsupseteq [\![\tau' - \ell]\!]v\eta$. Since $\tau = (\tau' - \ell)$, we have shown $[\![\tau]\!]v\eta\sigma \sqsubseteq q * r$.

For the critical region rule, the traces of $\texttt{with}\ \ell\ \texttt{do}\ C$ are of the form $P(\ell); \tau'; V(\ell)$. Given $\{p * r\}C\{q * r\}$ and a trace τ' of C, we can reason sequentially as follows, where intermediate assertions indicate use of 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 post-conditions 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. ■

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. 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 racy.

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. It is possible, though, on the general level, to validate a version of Milner's expansion law, which captures part of the import of the use of auxiliary variables.

Neither did we explicitly include the Hoare logic rule for introducing existentials [23]; it, semantically, just boils down to the disjunction rule. Finally, we did not include a version of the substitution rule from [23]. This would require formulation of a notion of parameterized local action.

8 Conclusion and Related Work

There are three main precursors to this work. The first is the model theory of BI [24, 31], 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 [17], in terms of partial monoids. Separation algebras are a special case of the models in [30], 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 [25].

The relational version of local actions for separation algebras was used in [18]. This 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, 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 difficult, particularly in the case of lock (resource) declarations: these are handled easily via filtering in the trace semantics.

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 (plus 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 a sharper explanation of the conditions needed for soundness, and it transfers immediately to the more general class of models. 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 23 is considerably simpler in its statement than Brookes's Parallel Decomposition Lemma. There are other detailed differences, such as that we detect races while executing traces, after interleaving, while Brookes detects races at an earlier stage (during interleaving). This being said, we fully acknowledge the influence of Brookes's original analysis.

The concurrency model given in this paper 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. In the long version of the paper [13] we justify these aspects of the model by connecting it to an operational semantics in the style of Plotkin.[8]

[8] Also, because of race-freedom it should be possible to connect to a

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 [14], which replaces the primitive of separation by the primitive of pulling a state apart into a state-with-a-hole (a context) and its filler; more work on 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 [26, 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 [19, 29].

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 but one possible path through the subject. Recent work on the logic for low-level code sometimes chooses to reflect locality in a novel interpretation of Hoare triples [7] rather than in the semantics of commands, or to express locality explicitly by polymorphism [20, 3]. The work on Boogie [2] achieves modularity using ideas that have hints of the primitives in separation logic [21], 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.

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*, pp182–196, 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.

[7] L. Birkedal and H. Yang. Relational parametricity and separation logic. In *10th FOSSACS*, LNCS 4423, pp93–107, 2007.

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

weak memory model, but we have not formulated such a connection.

[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. O'Hearn, and H. Yang. Local Action and Abstract Separation Logic (Longer version). www.dcs.qmul.ac.uk/∼ohearn/papers/asl-long.pdf, 2007.

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

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

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

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

[18] I. Mijajlovic, N. Torp-Smith, and P. O'Hearn. Refinement and separation contexts. Proceedings of FSTTCS, 2004.

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

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

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

[22] P. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science 375(1-3)*, pp271-307, May 2007. (Preliminary version appeared in CONCUR'04, LNCS 3170, 49–67.)

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

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

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

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

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

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

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

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

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

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

[33] J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millennial Perspectives in Computer Science*, pp 303–321, 2000. Palgrave.

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

[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.