# Variables as Resource for Shared-Memory Programs: Semantics and Soundness

Stephen Brookes [1,2]

*Department of Computer Science*
*Carnegie Mellon University*
*Pittsburgh, USA*

**Abstract**

Parkinson, Bornat, and Calcagno recently introduced a logic for partial correctness in which program variables are treated as resource, generalizing earlier work based on separation logic and permissions. An advantage of their approach is that it yields a logic devoid of complex side conditions: there is no need to pepper the inference rules with "modifies" clauses. They used a simple operational semantics to prove soundness of the sequential fragment of their logic, and they showed that the inference rules of concurrent separation logic can be translated directly into their framework. Their concurrency rules are strictly more powerful than those of concurrent separation logic, since the new logic allows proofs of programs that perform concurrent reads. We provide a denotational semantics and a soundness proof for the concurrent fragment of their logic, extending our earlier work on concurrent separation logic to incorporate permissions in a natural manner.

*Key words:* shared memory, concurrency, partial correctness, race condition, permission, separation logic

## 1 Introduction

Parkinson, Bornat, and Calcagno have recently introduced a logic for partial correctness in which program variables are treated as resource [12]. This amounts to a natural and significant generalization of earlier work, based on separation logic [13] and permissions [1,2,3], in which heap cells were treated as

[1] This research was sponsored by the National Science Foundation (NSF) under grant no. CCF-0429505. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.
[2] Email: brookes@cs.cmu.edu

*This paper is electronically published in*
*Electronic Notes in Theoretical Computer Science*
*URL:* `www.elsevier.nl/locate/entcs`

resource [3]. Separation logic [13] is a resource-oriented logic including formulas such as **emp** (specifying that the heap is empty), $E \mapsto E'$ (describing a singleton heap), and $P_1 \star P_2$ (separate conjunction), which holds in any heap that can be split into disjoint sub-heaps satisfying $P_1$ and $P_2$, respectively. In prior work such as concurrent separation logic [9,5], the heap was treated as resource: mutual exclusion for heap cells was enforced within the logic by means of separate conjunction; but program variables were handled in a more traditional style, the inference rules being constrained by static side conditions to ensure the absence of race conditions. In the new logic [12] program variables (as well as heap cells) are treated as resource, governed by a permission discipline designed to allow concurrent reads while preventing concurrent writes. A key ingredient is the logical manipulation of permissions in a manner that allows proper accounting for resource usage, using a form of separating conjunction to split and combine permissions, perform bookkeeping, and enforce disjointness constraints. An advantage of this approach is that it leads to a Hoare-style logic free of the traditional, rather complex, side conditions used to enforce non-interference constraints, such as those imposed in the Owicki-Gries rule for shared-memory programs [11] and its descendants [5,9].

Concurrent separation logic [5,9] extends and adapts Owicki-Gries logic to incorporate pointers and mutable state, using resource-sensitive partial correctness formulas of the form $\Gamma \vdash \{P\}C\{Q\}$, where $P$ and $Q$ are separation logic formulas and $\Gamma$ is a resource context associating a list of resource names with protection lists (of variables) and resource invariants. It was shown in [5] that this logic is sound, provided resource invariants are chosen to be *precise* separation logic formulas. A precise formula has the characteristic property that in every state there is at most one sub-heap for which the formula holds. Parkinson, Bornat, and Calcagno have shown [12] that the inference rules of concurrent separation logic can be translated into their framework. The concurrent fragment of their logic allows correctness proofs for programs that perform concurrent reads, so that their logic is more powerful than concurrent separation logic. Furthermore, their rule for parallel composition manages to avoid the need for a "modifies" clause, relying instead on blending the required constraints implicitly into the structure of formulas.

Parkinson, Bornat, and Calcagno [12] sketch a soundness proof, based on a simple operational semantics, for the sequential fragment of their logic, including procedures using call-by-reference and/or call-by-value parameters. Their paper also cites a draft manuscript dealing with the concurrency rules, based on an abstract operational model [7]. For the concurrent fragment of their logic it is not obvious that the blending in of erstwhile side conditions and well-formedness constraints on formula structure achieves the intended effect:

[3] We use the phrase "variables treated as resource" to refer to logics in which reasoning about variable usage is managed with a form of separating conjunction, in the spirit of BI [10,8].

we need a rigorous proof that the new logic manages to rule out racy programs while not letting in unsoundness through a side door. The side conditions and syntactic well-formedness constraints used in concurrent separation logic to restrict the structure of resource invariants, pre- and post-conditions, and resource contexts, were chosen very carefully to ensure soundness. One cannot, therefore, obtain a soundness proof for the concurrent fragment of the new logic simply by appealing to soundness of the original concurrent separation logic. The fact that the inference rules of concurrent separation logic can be translated faithfully into the new logic is insufficient to establish soundness of the new logic: the earlier soundness proof for concurrent separation logic does not incorporate permissions, and the new inference rules of [12] for concurrency and resources are not derivable within concurrent separation logic. Indeed, as we have said, the new logic is strictly more powerful, since it allows proofs of correctness for programs that perform concurrent reads, unlike the earlier logic.

We provide here a semantics and a soundness proof for the concurrent fragment of their logic, extending our earlier work on concurrent separation logic to incorporate permissions in a natural manner. We are able to re-use the *action traces* semantic model [5], introduced earlier in our soundness proof for the original concurrent separation logic, and we will provide a soundness proof in the same style as [5], making a series of appropriate adjustments to deal with permissions and ownership. Thus we obtain an elegant validation proof for the concurrency logic of [12], based on a straightforward trace-theoretical denotational semantics. These results are also evidence of the robustness of our action trace model. Our soundness proof also makes clear the role of a suitably permissive analogue of the notion of *precision*. Our soundness proof employs a denotational trace-based semantic model, in contrast to the operational approach of [7].

Since we build directly on the logic presented in [12], we include here a summary of the relevant background. To facilitate comparison we adopt (most of) their notational conventions, although we prefer to use semantic notation consistent with our own prior usage. As in [12] we omit pointers and heap, and we appeal to space limitations and claim that our definitions and results can be extended in a straightforward manner to incorporate pointers. We also refer to [12,1,2] for further motivation as well as justification for various design decisions, for examples of program proofs, for discussion of the way substitution interacts with ownership, and for philosophical remarks concerning the nature of logical variables.

## 2 Programs

### *2.1 Syntax*

We deal with a shared-memory language with the usual sequential program constructs and *parallel composition*, written $C_1 \| C_2$, *conditional critical regions* of the form **with** $r$ **when** $B$ **do** $C$, and *local resource blocks* of the form **resource** $r$ **in** $C$. Here $r$ ranges over the set of *resource names*.[4] We let $C$ range over commands (or processes), $B$ over boolean expressions, and $E$ over integer expressions. Such expressions may contain program variables, and we let $x$ range over program variables. We let $\mathtt{free}(C)$ be the set of free variables occurring in $C$, and we let $\mathtt{res}(C)$ be the set of free resource names in $C$. In particular,

$$\mathtt{res}(C_1 \| C_2) = \mathtt{res}(C_1) \cup \mathtt{res}(C_2)$$

$$\mathtt{res}(\textbf{with } r \textbf{ when } B \textbf{ do } C) = \{r\} \cup \mathtt{res}(C)$$

$$\mathtt{res}(\textbf{resource } r \textbf{ in } C) = \mathtt{res}(C) - \{r\}$$

We use the abbreviation **with** $r$ **do** $C$ for **with** $r$ **when true do** $C$.

Although we omit the details, the language can be extended with the usual pointer-manipulating constructs (allocation, disposal, lookup, and update) as in [13]. The forthcoming semantics and logic can also be extended similarly.

### *2.2 Semantics*

We use the action trace semantics for programs, as in [5]. We include some of the key ingredients here, but refer to [5] for details.

A command denotes a set of action traces. An action trace is a finite or infinite sequence of actions, each representing an atomic piece of state change. Actions, ranged over by $\lambda$, include $\delta$ (idle), $x = v$ (a read), $x{:=}v$ (a write), resource actions *try(r)*, *acq(r)* and *rel(r)*, and an error action *abort*. The meta-variables $v$ range over the set $V_{int}$ of integer values, $x$ over program variables, and $r$ over resource names, respectively.

A trace represents a possible interactive computation of the program in a concurrent environment. We interpret parallel composition as a form of fair interleaving that keeps track of resources (so that resource acquisition is mutually exclusive at all stages) and treats a possible race condition as a runtime error.

An integer expression $E$ denotes a set $[\![E]\!]$ of trace-value pairs, the trace indicating the read actions needed to evaluate the expression and yield the given

[4] There is a potential for confusion over our double usage of the term "resource". A resource name $r$ will behave, semantically, like a binary semaphore. We also use the term more loosely to refer for example to a collection of variables.

value. The semantics of expressions is given denotationally. For example:

$$[\![0]\!] = \{(\delta, 0)\}$$
$$[\![x]\!] = \{(x = v, v) \mid v \in V_{int}\}$$
$$[\![E_1 + E_2]\!] = \{(\rho_1\rho_2, v_1 + v_2) \mid (\rho_1, v_1) \in [\![E_1]\!], (\rho_2, v_2) \in [\![E_2]\!]\}$$

A boolean expression $B$ denotes a set $[\![B]\!]$ of trace-value pairs, and we let $[\![B]\!]_{true} = \{\rho \mid (\rho, true) \in [\![B]\!]\}$, and analogously for $[\![B]\!]_{false}$. In particular,

$$[\![\mathbf{true}]\!] = \{(\delta, true)\}$$
$$[\![E_1 = E_2]\!] = \{(\rho_1\rho_2, true) \mid \exists v.\ (\rho_1, v) \in [\![E_1]\!], (\rho_2, v) \in [\![E_2]\!]\}$$
$$\cup\ \{(\rho_1\rho_2, false) \mid \exists v_1 \neq v_2.\ (\rho_1, v_1) \in [\![E_1]\!] \wedge (\rho_2, v_2)\ \in [\![E_2]\!]\}$$

The above clauses specify that evaluation of expressions such as $E_1 + E_2$ is sequential, left-to-right, but this is not crucial in the ensuing development.

A command $C$ denotes a trace set $[\![C]\!]$, defined denotationally.

**Definition 1**
*The trace set $[\![C]\!]$ is defined by structural induction on $C$, as follows.*

$$[\![\mathbf{skip}]\!] = \{\delta\}$$
$$[\![x{:=}E]\!] = \{\rho\, x{:=}v \mid (\rho, v) \in [\![E]\!]\}$$
$$[\![C_1; C_2]\!] = \{\alpha_1\alpha_2 \mid \alpha_1 \in [\![C_1]\!],\ \alpha_2 \in [\![C_2]\!]\}$$
$$[\![\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2]\!] = [\![B]\!]_{true}[\![C_1]\!]\ \cup\ [\![B]\!]_{false}[\![C_2]\!]$$
$$[\![\mathbf{while}\ B\ \mathbf{do}\ C]\!] = ([\![B]\!]_{true}[\![C]\!])^*[\![B]\!]_{false}\ \cup\ ([\![B]\!]_{true}[\![C]\!])^\omega$$
$$[\![\mathbf{local}\ x\ \mathbf{in}\ C]\!] = \{\alpha\backslash x \mid \alpha \in [\![C]\!]_{[x:v]},\ v \in V_{int}\}$$
$$[\![C_1 \| C_2]\!] = \bigcup\{\alpha_1 \| \alpha_2 \mid \alpha_1 \in [\![C_1]\!],\ \alpha_2 \in [\![C_2]\!]\}$$
$$[\![\mathbf{with}\ r\ \mathbf{when}\ B\ \mathbf{do}\ C]\!] = wait^*\, enter\ \cup\ wait^\omega$$
$$wait = \{acq(r)\, \rho\, rel(r) \mid (\rho, false) \in [\![B]\!]\}\ \cup\ \{try(r)\}$$
$$enter = \{acq(r)\, \rho\, \alpha\, rel(r) \mid (\rho, true) \in [\![B]\!],\ \alpha \in [\![C]\!]\}$$
$$[\![\mathbf{resource}\ r\ \mathbf{in}\ C]\!] = \{\alpha\backslash r \mid \alpha \in [\![C]\!]_r\}$$

We use the obvious notation for concatenation and iteration, and we treat $\delta$ as a unit for concatenation, so that $\alpha\,\delta\,\beta = \alpha\,\beta$ for all traces $\alpha$ and $\beta$. We also treat *abort* as a left-zero, so that $\alpha\, abort\, \beta = \alpha\, abort$.

The definition of parallel composition involves book-keeping to keep track of the resources held by each process, only allowing actions that obey the mutual exclusion constraints on resource acquisition, and treats a potential race as a runtime error:

- We define a *resource enabling* relation $(A_1, A_2) \xrightarrow{\lambda} (A_1', A_2)$ for each action

$\lambda$, that specifies when a process holding resources $A_1$, in an environment that holds $A_2$, can perform this action, and the resulting effect on resources:

$$(A_1, A_2) \xrightarrow{acq(r)} (A_1 \cup \{r\}, A_2) \quad \text{if } r \notin A_1 \cup A_2$$
$$(A_1, A_2) \xrightarrow{rel(r)} (A_1 - \{r\}, A_2) \quad \text{if } r \in A_1$$
$$(A_1, A_2) \xrightarrow{\lambda} (A_1, A_2) \quad \text{for all other actions}$$

We write $(A_1, A_2) \xrightarrow{\alpha} \cdot$ to indicate that a process holding resources $A_1$ in an environment holding $A_2$ can perform the trace $\alpha$.

- We write $\lambda_1 \bowtie \lambda_2$ ($\lambda_1$ *interferes with* $\lambda_2$) when one of the actions is a write to a variable either read or written by the other, so that the two actions may produce a race condition. Notice that we do not regard two concurrent reads as a disaster.
- We define, for each disjoint pair $(A_1, A_2)$ of resource sets and each pair $(\alpha_1, \alpha_2)$ of action sequences, the set $\alpha_1 {}_{A_1}\|_{A_2} \alpha_2$ of all *mutex fairmerges* of $\alpha_1$ using $A_1$ with $\alpha_2$ using $A_2$. The definition for finite sequences is inductive:

$$\alpha_1 {}_{A_1}\|_{A_2} \epsilon = \{\alpha_1 \mid (A_1, A_2) \xrightarrow{\alpha_1} \cdot\}$$
$$\epsilon {}_{A_1}\|_{A_2} \alpha_2 = \{\alpha_2 \mid (A_2, A_1) \xrightarrow{\alpha_2} \cdot\}$$
$$(\lambda_1\alpha_1) {}_{A_1}\|_{A_2} (\lambda_2\alpha_2) =$$
$$\{\lambda_1\beta \mid (A_1, A_2) \xrightarrow{\lambda_1} (A_1', A_2) \ \&\ \beta \in \alpha_1 {}_{A_1'}\|_{A_2} (\lambda_2\alpha_2)\}$$
$$\cup \{\lambda_2\beta \mid (A_2, A_1) \xrightarrow{\lambda_2} (A_2', A_1) \ \&\ \beta \in (\lambda_1\alpha_1) {}_{A_1}\|_{A_2'} \alpha_2\}$$
$$\cup \{abort \mid \lambda_1 \bowtie \lambda_2\}$$

When $A_1 = A_2 = \{\}$ we use the notation $\alpha_1 \| \alpha_2$ instead of $\alpha_1 {}_{\{\}}\|_{\{\}} \alpha_2$. This definition lifts to handle infinite traces in the usual manner. [5]

The command **resource** $r$ **in** $C$ introduces a local resource named $r$, whose scope is $C$. Its traces are obtained from traces of $C$ in which the local resource $r$ is assumed initially available and the actions involving the local resource are executable without interference. We let $[\![C]\!]_r$ be the set of traces of $C$ which are *sequential for* $r$ in this manner. Equivalently, $\alpha$ is sequential for $r$ if $\alpha \lceil r$ is a prefix of a trace in the set $(acq(r)\, try(r)^\infty\, rel(r))^\infty$. We let $\alpha \backslash r$ be the trace obtained from $\alpha$ by replacing each action on $r$ by $\delta$.

The traces of **local** $x$ **in** $C$ are obtained in a similar manner, by hiding the actions that involve $x$ in the traces of $C$ which are sequential for $x$. We assume that the local variable is initialized to an arbitrary integer value. We write $[\![C]\!]_{[x:v]}$ for the set of traces $\alpha$ of $C$ such that the sequence of reads and writes to $x$ along $\alpha$ is sequentially consistent with the initial value $v$ for $x$.

[5] This formulation of parallel composition differs from [5] by including *all* fair interleavings even if the program is racy; the new formulation has the advantage of associativity, but this issue is not crucial for achieving soundness, and for race-free programs the two formulations coincide.

Again the structure of these traces reflects the locality of $x$: the scope of the local variable binding is $C$ alone, so no concurrent process has access to the local variable.

The iterative structure of the traces of a conditional critical region reflect its use to achieve synchronization: waiting until the resource is available and the test condition is true, followed by execution of the body command while holding the resource, and finally releasing the resource.

## 3 Stacks and permissions

To prepare for the forthcoming logic we augment the traditional notions of stack (or store) with permissions. In a "permissive stack" a program variable has a value as usual, but also a permission drawn from some given set. We make some general assumptions about the set of permissions, as in [12].

**Definition 2**
*A permissions model is a set $\mathcal{P}$, with a distinguished element $\top$ (standing for "total" permission), and with a partial "composition" function $\otimes : \mathcal{P} \times \mathcal{P} \rightharpoonup \mathcal{P}$ such that $(\mathcal{P}, \otimes, \top)$ is a partial commutative cancellative semigroup:*

$$\begin{array}{ll} \text{SYM} & \forall p_1, p_2 \in \mathcal{P}.\ p_1 \otimes p_2 \simeq p_2 \otimes p_1 \\ \text{ASSOC} & \forall p_1, p_2, p_3 \in \mathcal{P}.\ p_1 \otimes (p_2 \otimes p_3) \simeq (p_1 \otimes p_2) \otimes p_3 \\ \text{CANC} & \forall p, p_1, p_2.\ (p \otimes p_1 = p \otimes p_2 \ \Rightarrow\ p_1 = p_2), \end{array}$$

*with the following additional properties:*

$$\begin{array}{ll} \text{TOP} & \forall p \in \mathcal{P}.\ (\top \otimes p) \text{ is undefined} \\ \text{NON-ZERO} & \forall p, p' \in \mathcal{P}.\ p \otimes p' \neq p \\ \text{COMP} & \forall p \in \mathcal{P} - \{\top\}.\ \exists \bar{p} \in \mathcal{P}.\ p \otimes \bar{p} = \top. \end{array}$$

*Permissions $p_1$ and $p_2$ such that $p_1 \otimes p_2$ is defined are called compatible, and we will write $p_1 \sharp p_2$ to indicate compatibility.*

We use Kleene equality here on meta-language expressions whose value may be undefined: for example, $p_1 \otimes p_2 \simeq p_2 \otimes p_1$ means that if $p_1$ and $p_2$ are compatible, then $p_1 \otimes p_2 = p_2 \otimes p_1$; and $p_1 \otimes p_2$ is undefined if and only if $p_2 \otimes p_1$ is undefined. This should be distinguished from ordinary equality: for instance, when $p, p' \in \mathcal{P}$ we have $p \otimes p' \neq p$ if either $p \otimes p'$ is undefined or $p \otimes p'$ is defined and not equal to $p$.

The "fractional permissions" of Boyland [3,1] fit into this framework: let $\mathcal{P}$ be the set of positive rational numbers in the interval $(0, 1]$, with $\top = 1$, and $p_1 \otimes p_2 =_{\text{def}} p_1 + p_2$ when this sum belongs to the interval, undefined otherwise; thus $p_1 \sharp p_2$ iff $0 < p_1 + p_2 \leq 1$. The fractional model also satisfies the following *divisibility* property:

$$\text{DIV} \quad \forall p \in \mathcal{P}. \exists p_1, p_2 \in \mathcal{P}.\ p = p_1 \otimes p_2.$$

Similarly, "counting permissions" [1] fit into the general framework, choosing $\mathcal{P} = V_{int}$, the set of integers, $\top = 0$, and letting $p_1 \otimes p_2$ be undefined if $p_1 \geq 0 \wedge p_2 \geq 0$, or $(p_1 \geq 0 \vee p_2 \geq 0) \wedge p_1 + p_2 < 0$, and equal to $p_1 + p_2$ otherwise. The intution here is that 0 is the "total" (or source) permission, $-1$ is a read permission, and $+k$ is a source permission from which $k$ read permissions have been split off. The divisibility property fails for counting permissions.

Although specific notions of permission, such as the fractional model and the counting model, may obey additional laws such as DIV, only the properties listed above are used in our technical developments, and COMP is not needed in our soundness proof although it seems to be a reasonable assumption [1]. [6] As a result we do not need to fix in advance a particular model of permissions; the forthcoming logic relies only on the above general properties. However, for concreteness, we will provide some examples based on fractional permissions, in which the divisibility property plays a crucial role.

A *stack* is a finite partial function from program variables to integers tagged with permissions. We let $s$ range over stacks, and let $S$ be the set of stacks. Thus we have $S = \mathbf{Var} \rightharpoonup_{fin} V_{int} \times \mathcal{P}$. Note the special case of a stack in which all permissions are total: this corresponds obviously to a traditional stack comprising a finite partial function from program variables to integers.

**Definition 3**
*A stack $s$ is totally permissive if for all $x$ in $\mathtt{dom}(s)$, $s(x) = (v, \top)$ for some $v \in V_{int}$.*

**Definition 4**
*Stacks $s$ and $s'$ are compatible, written $s \sharp s'$, when they agree on values for all variables common to both of their domains, and provide compatible permissions for such variables. More formally, $s \sharp s'$ holds if and only if*

$$\forall x, v, v', p, p'.\ (s(x) = (v, p) \wedge s'(x) = (v', p') \ \Rightarrow v = v' \ \&\ p \sharp p').$$

*When $s \sharp s'$ we let $s \star s'$ be the stack consisting of all pairs $(x, (v, p))$ such that either $(s(x) = (v, p)$ and $x \notin \mathtt{dom}(s'))$, or $(s'(x) = (v, p)$ and $x \notin \mathtt{dom}(s))$, or there are $p', p''$ such that $s(x) = (v, p')$ and $s'(x) = (v, p'')$ and $p = p' \otimes p''$. Equivalently,*

$$s \star s' = s \backslash dom(s') \cup s' \backslash dom(s) \cup \{(i, (v, p \otimes p')) \mid (i, (v, p)) \in s, (i, (v, p')) \in s'\}.$$

Note that for all stacks $s$ and $s_1$ there is at most one $s_2$ such that $s = s_1 \star s_2$. (This follows from the cancellative property of permission composition.) When $s = s_1 \star s_2$ we say that $s_1$ and $s_2$ are compatible sub-stacks of $s$.

Trivially when $\mathtt{dom}(s_1) \cap \mathtt{dom}(s_2) = \{\}$ we have $s_1 \sharp s_2$, and $s_1 \star s_2 = s_1 \cup s_2$. Also note that two stacks that each provide total permission for the same variable will be incompatible, because $\top \otimes \top$ is undefined; indeed, total

[6] We are grateful to Matthew Parkinson for remarks concerning the relevance of various permission axioms.

permission for a particular variable is incompatible with *any* permission for that variable.

For a program variable $x$, and stack $s$, let $s\backslash x = \{(y,(v,p)) \in s \mid y \neq x\}$. We write $[s \mid x : (v,p)]$ for the stack $(s\backslash x) \cup \{(x,(v,p))\}$, which is also equal to $(s\backslash x) \star \{(x,(v,p))\}$.

## 4 A logic

We assume given a notion of permissions $(\mathcal{P}, \otimes, \top)$ satisfying the above properties. The syntax of assertions (as in [12]) is then given by the following abstract grammar, in which $\Phi$ ranges over assertions, $X$ over logical variables, and $p$ over permission expressions (built from permission variables and constants, using $\otimes$).

$$\Phi ::= B \mid E = E \mid \mathbf{emp_s} \mid \mathtt{Own}_p(x) \mid \Phi \star \Phi \mid$$
$$\neg\Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \Phi \Rightarrow \Phi \mid \exists X.\Phi$$

Integer and boolean expressions in the logic are built from the usual arithmetic constants and operators, and may contain logical variables as well as program variables. The three types of variables (program variables, integer logical variables, and permission logical variables) range over disjoint sets. Expressions occurring in program text, and programs themselves, only contain program variables. Logical variables are only used in the logic, typically to allow the specification of relationships between the values and permissions described in pre- and post-conditions. Quantification is only allowed for logical variables.

## 5 Semantics of the logic

An *interpretation* is a finite partial function mapping logical variables to integers and permission logical variables to permission values. We let $i$ range over interpretations, and we let $I$ be the set of interpretations. We will refer to a pair $(s,i)$ as a state, and we let $\sigma$ range over states. We let $\mathtt{dom}(s,i) = \mathtt{dom}(s) \cup \mathtt{dom}(i)$.

We assume given the evaluation semantics for logic expressions, so that whenever $(s,i)$ is a state defined on (at least) the free program variables and logical variables of $E$, $|E|(s,i)$ is the integer value of $E$. When $E$ is a program expression containing no logical variables we may omit the $i$, simply writing $|E|s$. The semantic clauses for expression evaluation are standard. For example,

$$|0|(s,i) = 0$$
$$|x|(s,i) = \textbf{if } x \in \mathtt{dom}(s) \textbf{ then } s(x) \textbf{ else } i(x)$$
$$|E_1 + E_2|(s,i) = |E_1|(s,i) + |E_2|(s,i)$$

Similarly we assume given the evaluation semantics for permission expressions, noting again that the value of an expression containing $\otimes$ may be undefined. For a permission expression $p$ and an interpretation $i$ containing values for all of the variables in $p$, $|p|i \in \mathcal{P}$ is defined in the obvious way. In particular, $|p|i = p$ when $p \in \mathcal{P}$, and $|X|i = i(X)$ when $X$ is a logical permission variable belonging to $dom(i)$.

We define (as in [12]) a *forcing semantics* for assertions. When $(s, i) \models \Phi$ we say that the state $(s, i)$ satisfies $\Phi$, or that $\Phi$ holds in $(s, i)$. We write $\models \Phi$ when $\Phi$ holds in all states. (Because of the presence of partial expressions in our logical language, we need to be careful to avoid assertions whose value is undefined.)

**Definition 5**
*For a state $(s, i)$ and formula $\Phi$ we define the truth value $(s, i) \models \Phi$ by induction on the structure of $\Phi$:*

$$
\begin{array}{ll}
(s,i) \models B & \Leftrightarrow |B|s = \mathit{true} \\
(s,i) \models E_1 = E_2 & \Leftrightarrow |E_1|(s,i) = |E_2|(s,i) \ \wedge \ \mathtt{free}(E_1, E_2) \subseteq \mathtt{dom}(s,i) \\
(s,i) \models E_1 \neq E_2 & \Leftrightarrow |E_1|(s,i) \neq |E_2|(s,i) \ \wedge \ \mathtt{free}(E_1, E_2) \subseteq \mathtt{dom}(s,i) \\
(s,i) \models \mathbf{emp_s} & \Leftrightarrow s = \{\} \\
(s,i) \models (\Phi \Rightarrow \Psi) & \Leftrightarrow (s,i) \models \Phi \text{ implies } (s,i) \models \Psi \\
(s,i) \models \Phi \wedge \Psi & \Leftrightarrow (s,i) \models \Phi \text{ and } (s,i) \models \Psi \\
(s,i) \models \Phi_1 \star \Phi_2 & \Leftrightarrow \exists s_1, s_2.\ s = s_1 \star s_2 \ \wedge \ (s_1,i) \models \Phi_1 \ \wedge \ (s_2,i) \models \Phi_2 \\
(s,i) \models \mathtt{Own}_p(x) & \Leftrightarrow \exists v, u.\ s = \{(x,(v,u))\} \ \wedge \ |p|i = u \\
(s,i) \models \exists X.\Phi & \Leftrightarrow \exists v \in V_{int} \cup \mathcal{P}.\ (s, [i \mid X : v]) \models \Phi
\end{array}
$$

*The remaining cases, dealing with the usual logical connectives, are standard.*

(For brevity of exposition we have omitted the "magic wand" connective.)

Note that we have specified that $\mathtt{Own}_p(x)$ is *false* in a state $(s, i)$ for which $|p|i$ is undefined or when $|p|i$ is defined by $x \notin \mathtt{dom}(s)$.

The characteristic properties of the set of permissions give rise to some logical equivalences, e.g. if $p_1 \sharp p_2$ then $\mathtt{Own}_{p_1}(x) \star \mathtt{Own}_{p_2}(x) \Leftrightarrow \mathtt{Own}_{p_1 \otimes p_2}(x)$ and if $\neg(p_1 \sharp p_2)$ then $\mathtt{Own}_{p_1}(x) \star \mathtt{Own}_{p_2}(x) \Leftrightarrow \mathbf{false}$. In particular, for all $p \in \mathcal{P}$ we have $\mathtt{Own}_p(x) \star \mathtt{Own}_\top(x) \Leftrightarrow \mathbf{false}$.

Let $\mathtt{free}(E)$ be the set of program variables and logical variables with free occurrences in $E$. Thus $\mathtt{free}(E) \cap \mathbf{Var}$ is the set of free program variables of $E$. We will write $\mathtt{Own}_{p_1,\ldots,p_k}(x_1, \ldots, x_k)$ for $\mathtt{Own}_{p_1}(x_1) \star \cdots \star \mathtt{Own}_{p_k}(x_k)$ when $k \geq 0$. We will refer to a formula of this kind as an *ownership claim*, and we let $O$ range over such formulas. In the case where $k = 0$ the formula is interpreted as $\mathbf{emp_s}$.

Note that $E = E$ is true in $(s, i)$ if and only if $\mathtt{free}(E) \subseteq \mathtt{dom}(s) \cup \mathtt{dom}(i)$.

Also note that when the program variables $x_1, \ldots, x_k$ are distinct the formula $\mathtt{Own}_{p_1,\ldots,p_k}(x_1, \ldots, x_k)$ is true in $(s, i)$ if and only if there are integer values $v_1, \ldots, v_k$ such that $s = \{(x_1, (v_1, [\![p_1]\!]i)), \ldots, (x_k, (v_k, [\![p_k]\!]i))\}$.[7]

# 6 A partial correctness logic

## *6.1 Formulas*

We work with resource-sensitive partial correctness formulas of the form

$$\Gamma \vdash_{vr} \{\Phi\} C \{\Phi'\},$$

where $\Gamma$ is a resource context of the form $r_1 : \Phi_1, \ldots, r_k : \Phi_k$ in which the resource names $r_i$ are distinct, and each $\Phi_j$ is a *precise* formula. The class of precise formulas is defined as follows.

**Definition 6**
*A formula $\Phi$ is precise if for all $(s, i)$ there is at most one pair $(s_1, s_2)$ such that $s = s_1 \star s_2 \wedge (s_1, i) \models \Phi$.*

For example, $\mathbf{emp_s}$ and $\mathtt{Own}_\top(x)$ are precise. More generally, even when $p$ is a permission variable, $\mathtt{Own}_p(x)$ is precise. When $\Phi$ and $\Psi$ are precise, so are $\Phi \star \Psi$ and $(B \wedge \Phi) \vee (\neg B \wedge \Psi)$.

We insist that resource invariants be precise in order to ensure proper and unambiguous accounting for permissions in the logical development that follows. Similar restrictions on the class of formulas allowed as resource invariants are necessary to achieve soundness in concurrent separation logic. The pre- and post-conditions used in formulas need not be precise.

Let $\Gamma$ be the resource context $r_1 : \Phi_1, \ldots, r_k : \Phi_k$. We write $\mathtt{inv}(\Gamma)$ for the (precise) formula $\Phi_1 \star \cdots \star \Phi_k$. When the context is empty ($k = 0$) this is interpreted as $\mathbf{emp_s}$. For a set of resource names $A$ let $\Gamma \backslash A$ be the resource context containing the entries $r_j : \Phi_j$ from $\Gamma$ for which $r_j \notin A$. We let $\mathtt{dom}(\Gamma) = \{r_1, \ldots, r_k\}$ and $\mathtt{free}(\Gamma) = \bigcup_{j=1}^{k} \mathtt{free}(\Phi_j)$. When $r \notin \mathtt{dom}(\Gamma)$ we write $\Gamma, r : \Psi$ for the context obtained by combining $\Gamma$ with $r : \Psi$. The order in which entries are listed is not important, so a context represents an unordered list.

[7] Also note that $\neg(E_1 = E_2)$ is not logically equivalent to $E_1 \neq E_2$. As remarked in [12], the former holds in the empty state, whereas the latter can hold only in states defined on the free variables of $E_1$ and $E_2$. Indeed, a formula such as $x = 0$ asserts not just that the integer value of $x$ is zero, but also that the state contains a permission for $x$. Hence $x = 0$ is equivalent to $(\exists p.\ \mathtt{Own}_p(x) \wedge x = 0) \star \mathbf{true}$.

### 6.2 *Inference rules*

Here are the inference rules [12]. In the ensuing text we will elaborate on the notation and side conditions, and comment on some structural issues.

$$\frac{}{\Gamma \vdash_{vr} \{\Phi\}\mathbf{skip}\{\Phi\}} \textsc{Skip}$$

$$\frac{}{\Gamma \vdash_{vr} \{(\mathtt{Own}_\top(x) \star O) \wedge X = E\}x{:=}E\{(\mathtt{Own}_\top(x) \star O) \wedge x = X\}} \textsc{Assn}$$

$$\frac{\Gamma \vdash_{vr} \{\Phi\}C_1\{\Psi\} \qquad \Gamma \vdash_{vr} \{\Psi\}C_2\{\Theta\}}{\Gamma \vdash_{vr} \{\Phi\}C_1; C_2\{\Theta\}} \textsc{Seq}$$

$$\frac{\Phi \Rightarrow B = B \quad \Gamma \vdash_{vr} \{\Phi \wedge B\}C_1\{\Phi'\} \quad \Gamma \vdash_{vr} \{\Phi \wedge \neg B\}C_2\{\Phi'\}}{\Gamma \vdash_{vr} \{\Phi\}\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\{\Phi'\}} \textsc{Cond}$$

$$\frac{\Phi \Rightarrow B = B \quad \Gamma \vdash_{vr} \{\Phi \wedge B\}C\{\Phi\}}{\Gamma \vdash_{vr} \{\Phi\}\mathbf{while}\ B\ \mathbf{do}\ C\{\Phi \wedge \neg B\}} \textsc{While}$$

$$\frac{\Gamma \vdash_{vr} \{\Phi_1\}C_1\{\Phi_1'\} \quad \Gamma \vdash_{vr} \{\Phi_2\}C_2\{\Phi_2'\}}{\Gamma \vdash_{vr} \{\Phi_1 \star \Phi_2\}C_1 \| C_2\{\Phi_1' \star \Phi_2'\}} \textsc{Par}$$

$$\frac{\Gamma, r : \Psi \vdash_{vr} \{\Phi\}C\{\Phi'\}}{\Gamma \vdash_{vr} \{\Phi \star \Psi\}\mathbf{resource}\ r\ \mathbf{in}\ C\{\Phi' \star \Psi\}} \textsc{Res}$$

$$\frac{(\Phi \star \Psi) \Rightarrow B = B \quad \Gamma \vdash_{vr} \{(\Phi \star \Psi) \wedge B\}C\{\Phi' \star \Psi\}}{\Gamma, r : \Psi \vdash_{vr} \{\Phi\}\mathbf{with}\ r\ \mathbf{when}\ B\ \mathbf{do}\ C\{\Phi'\}} \textsc{Reg}$$

$$\frac{\Gamma \vdash_{vr} \{\mathtt{Own}_\top(x') \star \Phi\}[x'/x]C\{\mathtt{Own}_\top(x') \star \Psi \quad x' \text{ fresh}}{\Gamma \vdash_{vr} \{\Phi\}\mathbf{local}\ x\ \mathbf{in}\ C\{\Psi\}} \textsc{Local}$$

$$\frac{\Gamma \vdash_{vr} \{\Phi\}\mathbf{resource}\ r'\ \mathbf{in}\ [r'/r]C\{\Psi\} \quad r' \notin \mathtt{res}(C)}{\Gamma \vdash_{vr} \{\Phi\}\mathbf{resource}\ r\ \mathbf{in}\ C\{\Psi\}} \textsc{Res}$$

$$\frac{\Gamma \vdash_{vr} \{\Phi\}C\{\Phi'\}}{\Gamma \vdash_{vr} \{\Phi \star \Psi\}C\{\Phi' \star \Psi\}} \textsc{Frame}$$

$$\frac{\Phi \Rightarrow \Phi' \quad \Gamma \vdash_{vr} \{\Phi'\}C\{\Psi'\} \quad \Psi' \Rightarrow \Psi}{\Gamma \vdash_{vr} \{\Phi\}C\{\Psi\}} \textsc{Conseq}$$

$$\frac{\Gamma \vdash_{vr} \{\Phi\}C\{\Psi\}}{\Gamma \vdash_{vr} \{\exists X.\Phi\}C\{\exists X.\Psi\}} \textsc{Exists}$$

$$\frac{\Gamma \vdash_{vr} \{\Phi \star \mathtt{Own}_\top(\mathcal{Y})\}C\{\Phi' \star \mathtt{Own}_\top(\mathcal{Y})\}}{\Gamma \vdash_{vr} \{\Phi\}C\backslash\mathcal{Y}\{\Phi'\}} \ (\dagger) \quad \textsc{Aux}$$

*Comments*

- The assignment rule Assn differs from the usual Hoare-style rule based on syntactic substitution, instead using a logical variable. Indeed, substitution does not always behave sensibly with respect to ownership, as shown in [1]. Nevertheless, one can derive axioms such as $\Gamma \vdash_{vr} \{\Phi\}x{:=}E\{\Phi \wedge x = E\}$ where $x$ is not free in $E$, the list $x_1, \ldots, x_k$ is an enumeration without repetition of $\mathtt{free}(E)$, and $\Phi$ is $\mathtt{Own}_\top(x) \star \mathtt{Own}_{p_1,\ldots,p_k}(x_1, \ldots, x_k)$. The derivation uses Assn together with Frame, Conseq and Exists.

- The rules Res and Reg carry the implicit side condition that $\Psi$ is precise and $r \notin \mathtt{dom}(\Gamma)$, so that the context $\Gamma, r : \Psi$ is well formed. The Ren rule for changing a bound resource name requires that the premiss must use a "fresh" resource name, i.e. one not occurring already in $C$.
- The premiss $\Phi \Rightarrow B = B$ is used in the rules Cond and While to ensure that every state satisfying $\Phi$ will contain values (and permissions) for the free variables of $B$.
- The Par rule for parallel composition does *not* require static side conditions constraining the programs and formulas, unlike traditional logics [11,5]. The usual constraint that $c_1$ must not write to any program variable occurring free in $\Phi_2$ or $\Psi_2$ (and the corresponding constraint on $c_2$) is enforced implicitly by the other inference rules. The conventional constraint that each variable written by one of the programs and read or written by the other must be "protected" by a resource is both unenforceable (resource names do not come equipped with protection lists) and rendered unnecessary by the way permissions are built into the other rules. The Frame rule avoids the usual side condition on the free variables of $\Psi$.
- The Local rule requires that the bound variable $x'$ be fresh, i.e. not free in $\Gamma, \Phi, \Psi$ and not in $\mathtt{free}(C) - \{x\}$.
- In the rules for change of bound variable, $[r'/r]C$ is obtained from $C$ by replacing each free occurrence of resource name $r$ with $r'$; similarly for $[x'/x]C$; in each case, as usual, we use a further renaming if necessary to avoid accidental capture of free names.
- In the Exists rule $X$ ranges over logical variables. We disallow quantification over program variables.
- The auxiliary variables rule Aux has a side condition (†) defined as follows: Y must be an enumeration, without repetition, of a set of program variables *auxiliary for* $C$, whose members must not occur free in $\Phi, \Phi'$ or $\Gamma$. A set of program variables is auxiliary for $C$ if every free occurrence in $C$ of a variable from in this set is in an assignment $x{:=}E$ for which $x$ also belongs to this set. The program $C\backslash\mathcal{Y}$ is obtained from $C$ by replacing each auxiliary assignment with **skip**, and eliding occurrences of **skip** by means of the obvious laws (e.g. $C; \mathbf{skip} = \mathbf{skip}; C = C$). The constraints placed on auxiliary variables ensure that these variables play no role in the resource invariants or in the pre- and post-conditions, and do not affect the control flow of the rest of the program.

## 7 Examples

The following examples serve to illustrate the style of proof supported by the permission logic rules. The reader familiar with concurrent separation logic is encouraged to try to formulate analogous proofs in the earlier logic. The advantages of the new logic include the easy accommodation of concurrent

reads, as well as the avoidance of static side conditions in many of the rules. These advantages are more compelling in the full logic dealing with pointers, but we hope that these pointer-free examples make a convincing case.

(i) One-place buffer

Let $\Psi$ be $(\mathit{full} = 1 \vee \mathit{full} = 0) \wedge (\mathtt{Own}_\top(\mathit{full}) \star \mathtt{Own}_\top(z))$

Let $\mathtt{PUT}(x)$ be **with** *buf* **when** $\mathit{full} = 0$ **do** $(z{:=}x; \mathit{full}{:=}1)$.

Let $\mathtt{GET}(y)$ be **with** *buf* **when** $\mathit{full} = 1$ **do** $(y{:=}z; \mathit{full}{:=}0)$.

The variables $z, \mathit{full}$ represent a one-place buffer protected by *buf*. The following formulas are provable.

$\mathit{buf} : \Psi \vdash_{vr} \{\mathtt{Own}_\top(x)\}\mathtt{PUT}(x)\{\mathtt{Own}_\top(x)\}$

$\mathit{buf} : \Psi \vdash_{vr} \{\mathtt{Own}_\top(y)\}\mathtt{GET}(y)\{\mathtt{Own}_\top(y)\}$

Using the parallel rule then yields:

$\mathit{buf} : \Psi \vdash_{vr} \{\mathtt{Own}_\top(x) \star \mathtt{Own}_\top(y)\}\mathtt{PUT}(x)\|\mathtt{GET}(y)\{\mathtt{Own}_\top(x) \star \mathtt{Own}_\top(y)\}$

There is an analogous derivation of a formula expressing the same correctness properties within concurrent separation logic [5,9], and it is instructive to compare the two proofs to see how the constraints imposed by side conditions in concurrent separation logic get built in automatically in the permission rules.

(ii) Concurrent reads

Let $p_1, p_2$ be permission values such that $p_1 \otimes p_2$ is defined, and let $x, y, z$ be distinct variables. The formulas

$$\vdash_{vr} \{\mathtt{Own}_\top(x) \star \mathtt{Own}_{p_1}(z)\}x{:=}z\{\mathtt{Own}_\top(x) \star \mathtt{Own}_{p_1}(z) \wedge x = z\}$$

$$\vdash_{vr} \{\mathtt{Own}_\top(y) \star \mathtt{Own}_{p_2}(z)\}y{:=}z\{\mathtt{Own}_\top(y) \star \mathtt{Own}_{p_2}(z) \wedge y = z\}$$

are easily provable. Using the parallel rule we then obtain

$$\begin{array}{l}\vdash_{vr} \{\mathtt{Own}_\top(x) \star \mathtt{Own}_\top(y) \star \mathtt{Own}_{p_1 \otimes p_2}(z)\} \\ \quad x{:=}z \| y{:=}z \\ \quad \{\mathtt{Own}_\top(x) \star \mathtt{Own}_\top(y) \star \mathtt{Own}_{p_1 \otimes p_2}(z) \wedge x = y = z\}\end{array}$$

(iii) Race conditions

It is easy to see from the assignment rule that the formula

$$\vdash_{vr} \{\mathtt{Own}_\top(x)\}x{:=}x + 1\{\mathtt{Own}_\top(x)\}$$

is provable. Hence, from the parallel rule, we can derive

$$\vdash_{vr} \{\mathtt{Own}_\top(x) \star \mathtt{Own}_\top(x)\}x{:=}x + 1 \| x{:=}x + 1\{\mathtt{Own}_\top(x) \star \mathtt{Own}_\top(x)\}$$

This formula, despite involving a racy program, asserts a triviality, since the pre-condition is not satisfiable.

In contrast, within concurrent separation logic there is no way to prove any formula of the form $\vdash \{P\}x{:=}x{+}1 \| x{:=}x{+}1\{Q\}$, because the program

falls afoul of the side conditions of the parallel rule: $x$ is concurrently written but not protected.

(iv) Auxiliary variables

Let $\top = p_1 \otimes p_2 = q_1 \otimes q_2$. (The permission values $p_1, p_2, q_1, q_2$ need not necessarily be distinct; their actual value is irrelevant in this proof.) Let $\Gamma$ be the resource context

$$r : (\mathtt{Own}_\top(x) \star \mathtt{Own}_{p_1}(i) \star \mathtt{Own}_{q_1}(j)) \wedge x = i + j$$

The following formulas are provable using the region rule:

$$\Gamma \vdash_{vr} \{\mathtt{Own}_{p_2}(i) \wedge i = 0\}\mathbf{with}\ r\ \mathbf{do}\ (x{:=}x+1; i{:=}i+1)\{\mathtt{Own}_{p_2}(i) \wedge i = 1\}$$

$$\Gamma \vdash_{vr} \{\mathtt{Own}_{q_2}(j) \wedge j = 0\}\mathbf{with}\ r\ \mathbf{do}\ (x{:=}x+1; j{:=}j+1)\{\mathtt{Own}_{q_2}(j) \wedge j = 1\}$$

By the parallel rule we deduce

$$\begin{array}{l}\Gamma \vdash_{vr} \{\mathtt{Own}_{p_2}(i) \star \mathtt{Own}_{q_2}(j) \wedge i = 0 \wedge j = 0\} \\ \quad \mathbf{with}\ r\ \mathbf{do}\ (x{:=}x+1; i{:=}i+1) \| \mathbf{with}\ r\ \mathbf{do}\ (x{:=}x+1; j{:=}j+1) \\ \quad \{\mathtt{Own}_{p_2}(i) \star \mathtt{Own}_{q_2}(j) \wedge i = 1 \wedge j = 1\}\end{array}$$

The rule for resources then gives

$$\begin{array}{l}\vdash_{vr} \{\mathtt{Own}_\top(x) \star \mathtt{Own}_\top(i) \star \mathtt{Own}_\top(j) \wedge x = i = j = 0\} \\ \quad \mathbf{resource}\ r\ \mathbf{in} \\ \quad\quad \mathbf{with}\ r\ \mathbf{do}\ (x{:=}x+1; i{:=}i+1) \| \mathbf{with}\ r\ \mathbf{do}\ (x{:=}x+1; j{:=}j+1) \\ \quad \{\mathtt{Own}_\top(x) \star \mathtt{Own}_\top(i) \star \mathtt{Own}_\top(j) \wedge x = 2 \wedge i = j = 1\}\end{array}$$

We can then derive

$$\begin{array}{l}\vdash_{vr} \{(\mathtt{Own}_\top(x) \wedge x = 0) \star (\mathtt{Own}_\top(i) \star \mathtt{Own}_\top(j))\} \\ \quad i{:=}0; j{:=}0; \\ \quad \mathbf{resource}\ r\ \mathbf{in} \\ \quad\quad \mathbf{with}\ r\ \mathbf{do}\ (x{:=}x+1; i{:=}i+1) \| \ \mathbf{with}\ r\ \mathbf{do}\ (x{:=}x+1; j{:=}j+1) \\ \quad \{(\mathtt{Own}_\top(x) \wedge x = 2) \star (\mathtt{Own}_\top(i) \star \mathtt{Own}_\top(j)\}\end{array}$$

The set $\{i, j\}$ is auxiliary for this program, and we may use the auxiliary rule to deduce

$$\begin{array}{l}\vdash_{vr} \{\mathtt{Own}_\top(x) \wedge x = 0\} \\ \quad \mathbf{resource}\ r\ \mathbf{in} \\ \quad\quad \mathbf{with}\ r\ \mathbf{do}\ x{:=}x+1 \\ \quad\quad \| \ \mathbf{with}\ r\ \mathbf{do}\ x{:=}x+1 \\ \quad \{\mathtt{Own}_\top(x) \wedge x = 2\}\end{array}$$

Again this derivation can be mimicked in concurrent separation logic. We

include it to illustrate how the permissions logic works without the need to impose side conditions.

(v) Distributed counting

Let $M, N \geq 0$ be integer constants, and let $p_1, p_2, q_1, q_2$ be permission values such that $p_1 \otimes p_2 = q_1 \otimes q_2 = \top$. Let $\Gamma$ be the resource context $r : \mathtt{Own}_\top(x) \star \mathtt{Own}_{p_1}(y) \star \mathtt{Own}_{q_1}(z) \wedge x = y + z$. Let $\mathtt{COUNT}_M$ and $\mathtt{COUNT}_N$ be the programs

$$\mathtt{COUNT}_M :: \textbf{while } y < M \textbf{ do with } r \textbf{ do } (x{:=}x+1; y{:=}y+1)$$

$$\mathtt{COUNT}_N :: \textbf{while } z < N \textbf{ do with } r \textbf{ do } (x{:=}x+1; z{:=}z+1)$$

The following formulas are provable:

$$\Gamma \vdash_{vr} \{\mathtt{Own}_{p_2}(y) \wedge y = 0\}\mathtt{COUNT}_M\{\mathtt{Own}_{p_2}(y) \wedge y = M\}$$

$$\Gamma \vdash_{vr} \{\mathtt{Own}_{q_2}(z) \wedge z = 0\}\mathtt{COUNT}_N\{\mathtt{Own}_{q_2}(z) \wedge z = N\}$$

$$\begin{array}{l}\Gamma \vdash_{vr} \{\mathtt{Own}_{p_2}(y) \star \mathtt{Own}_{q_2}(z) \wedge y = 0 \wedge z = 0\} \\ \qquad \mathtt{COUNT}_M \| \mathtt{COUNT}_N \\ \qquad \{\mathtt{Own}_{p_2}(y) \star \mathtt{Own}_{q_2}(z) \wedge y = M \wedge z = N\}\end{array}$$

$$\begin{array}{l}\vdash_{vr} \{\mathtt{Own}_\top(x) \star \mathtt{Own}_\top(y) \star \mathtt{Own}_\top(z) \wedge x = y = z = 0\} \\ \qquad \textbf{resource } r \textbf{ in } (\mathtt{COUNT}_M \| \mathtt{COUNT}_N) \\ \qquad \{\mathtt{Own}_\top(x) \star \mathtt{Own}_\top(y) \star \mathtt{Own}_\top(z) \wedge x = M + N \wedge y = M \wedge z = N\}\end{array}$$

In this example $x$ is the only "critical" variable, and is protected by $r$, whereas $y$ is only used by $\mathtt{COUNT}_M$ and $z$ is only used by $\mathtt{COUNT}_N$. Correspondingly the resource invariant has been chosen to ascribe only a read permission for $y$ and for $z$, but total permission for $x$. This leaves enough permissive latitude for $\mathtt{COUNT}_M$ to read the value of $y$ *outside* the conditional critical region (in the loop test). The loop body writes to $y$ but does so after acquiring the extra piece of permission prescribed in the resource invariant.

## 8 Towards validity

Intuitively, a formula $\Gamma \vdash \{\Phi\}C\{\Psi\}$ expresses race-free partial correctness of $C$ when executed in an environment that respects $\Gamma$. This can be expressed as a form of rely/guarantee property: when $C$ is executed from a global state that satisfies $\mathtt{inv}(\Gamma) \star \Phi$, in an environment that respects the resource invariants, $C$ is race-free, also respects the resource invariants, and if it terminates the final state will satisfy $\mathtt{inv}(\Gamma) \star \Psi$. This rely/guarantee formulation implies a more conventional form of partial correctness: when $C$ is executed *without interference* from a global states satisfying $\Phi \star \mathtt{inv}(\Gamma)$, $C$ is race-free, and if it terminates the final state will satisfy $\Psi \star \mathtt{inv}(\Gamma)$. However, the latter notion is not compositional and we need the more general formulation.

We can make these ideas concrete by appealing to action traces. The action traces of a program do not explicitly involve state, but are formed from actions that can be interpreted as having an effect on state. We are therefore well positioned to adapt our earlier soundness proof for concurrent separation logic to deal with permissions: we adapt all of the key definitions to the permissive setting and prove results analogous to the lemmas and theorems used in the original soundness proof.

First we define (global) enabling relations $\stackrel{\lambda}{\Longrightarrow}$ as in [5], adjusted to take account of permissions.

**Definition 7**
*For each action* $\lambda$ *let* $\stackrel{\lambda}{\Longrightarrow} \subseteq (S \cup \{\mathbf{abort}\}) \times (S \cup \{\mathbf{abort}\})$ *be given by:*

| | |
|---|---|
| $(s, A) \stackrel{\delta}{\Longrightarrow} (s, A)$ | always |
| $(s, A) \stackrel{x=v}{\Longrightarrow} (s, A)$ | if $\exists p.\ s(x) = (v, p)$ |
| $(s, A) \stackrel{x=v}{\Longrightarrow} \mathbf{abort}$ | if $x \notin \mathtt{dom}(s)$ |
| $(s, A) \stackrel{x:=v}{\Longrightarrow} ([s \mid x : (v, \top)], A)$ | if $\exists v_0.\ s(x) = (v_0, \top)$ |
| $(s, A) \stackrel{x:=v}{\Longrightarrow} \mathbf{abort}$ | if $x \notin \mathtt{dom}(s) \vee \exists v_0. \exists p \neq \top.\ s(x) = (v_0, p)$ |
| $(s, A) \stackrel{acq(r)}{\Longrightarrow} (s, A \cup \{r\})$ | if $r \notin A$ |
| $(s, A) \stackrel{rel(r)}{\Longrightarrow} (s, A - \{r\})$ | if $r \in A$ |
| $(s, A) \stackrel{try(r)}{\Longrightarrow} (s, A)$ | always |
| $(s, A) \stackrel{abort}{\Longrightarrow} \mathbf{abort}$ | always |
| $\mathbf{abort} \stackrel{\lambda}{\Longrightarrow} \mathbf{abort}$ | always |

The above definition allows write actions only with total permission, and read actions with any permission.

Note that when $(s, A) \stackrel{\lambda}{\Longrightarrow} (s', A')$ the two stacks are defined on the same variables and they specify equal permissions, for each program variable in their common domain: $\mathtt{dom}(s) = \mathtt{dom}(s')$ and for all $x \in \mathtt{dom}(s)$ and all $v, v', p, p'$, if $s(x) = (v, p)$ and $s(x') = (v', p')$, then $p = p'$.

We generalize from actions to traces in the obvious way, by composition. Thus we let $(s, A) \stackrel{\lambda_1 \ldots \lambda_n}{\Longrightarrow} (s', A')$ if there are stacks $s_1, \ldots, s_{n-1}$ and resource sets $A_1, \ldots, A_{n-1}$ such that $(s, A) \stackrel{\lambda_1}{\Longrightarrow} (s_1, A_1) \cdots (s_{n-1}, A_{n-1}) \stackrel{\lambda_n}{\Longrightarrow} (s', A')$.

The traces of a program $C$ obey the mutex assumptions for resources, and always acquire before release, in the following sense.

**Lemma 8**
*If* $\alpha \in [\![C]\!]$ *and* $(s, A) \stackrel{\alpha}{\Longrightarrow} (s', A')$ *then* $A = A'$ *and* $(s, \{\}) \stackrel{\alpha}{\Longrightarrow} (s', \{\})$.

Accordingly, when dealing with traces of a given program we may omit the $A$ and $A'$ without loss of generality. We write $s \stackrel{\alpha}{\Longrightarrow} s'$ when $(s, \{\}) \stackrel{\alpha}{\Longrightarrow} (s', \{\})$.

It is easy to see that global enabling for *totally permissive* states coincides

with the notion of global enabling on (permissionless) stacks from [5].

**Lemma 9**
*Let* $s : \mathbf{Var} \rightharpoonup_{fin} (V_{int} \times \mathcal{P})$ *be a totally permissive stack, and let* $\hat{s}$ *be the corresponding permissionless stack, i.e.* $\hat{s} = \{(x, v) \mid \exists p \in \mathcal{P}.\ (x, (v, p)) \in s\}$. *Then* $(s, A) \stackrel{\lambda}{\Longrightarrow} (s', A')$ *if and only if* $(\hat{s}, A) \stackrel{\lambda}{\Longrightarrow} (\hat{s'}, A')$ *using the enabling relation of [5].*

## 9 Validity and soundness

We define logical enabling relations $\stackrel{\lambda}{\Longrightarrow}_\Gamma$, as in [5], adjusted to deal with permissions.

A "local" state for a process holding resources in the set $A$ contains the variables for which the process currently "claims" a permission. For each action $\lambda$ the logical enabling relation captures the local view of a process that performs this action in a parallel environment that respects $\Gamma$.

The idea behind this notion is the following *Permission Principle*:

> At all stages, the permissions for each program variable are distributed compatibly among the concurrent processes and the available resources, and the resource invariants for all available resources hold, separately.

In terms of global and local state, we can paraphrase this as follows. Every global state of form $(s, A)$ can be viewed as a compatible combination of local states $(s_1, A_1), \ldots, (s_n, A_n)$ for the currently running processes with a piece of state $s'$ that satisfies the separate conjunction of the resource invariants, taken over all available resources. Thus $s = s_1 \star \cdots \star s_n \star s'$ and since resources are mutually exclusive, $A$ is the disjoint union of $A_1, \ldots, A_n$.

In accordance with this principle, each process only writes to variables for which it has total permission, and only reads variables for which it has a permission. When acquiring a resource, a process claims the piece of state described by the corresponding resource invariant, thus garnering additional permissions for variables mentioned in the invariant. When releasing a resource, a process must guarantee that the resource invariant holds, separately, and ceases to claim the relevant permissions.

The inference rules of the logic embody this principle, and the *logical enabling* relations will be used to formalize the notion of "respecting resource invariants" and to specify what we mean by processes claiming and ceding permissions dynamically as the program executes.

When $((s, i), A) \stackrel{\lambda}{\Longrightarrow}_\Gamma ((s', i), A')$ a process claiming $s$ and holding resource names $A$ can perform action $\lambda$ without violating the permission rules, and afterwards claims $s'$ and holds the resources in $A'$. We let $((s, i), A) \stackrel{\lambda}{\Longrightarrow}_\Gamma \mathbf{abort}$ when the action is impermissible or violates a resource invariant.

**Definition 10**
*For each action $\lambda$ and resource context $\Gamma$ we define the logical enabling relation $\stackrel{\lambda}{\Longrightarrow}_{\Gamma}$ as follows.*

$$((s,i),A) \stackrel{\delta}{\Longrightarrow}_{\Gamma} ((s,i),A) \qquad \text{always}$$

$$((s,i),A) \stackrel{x=v}{\Longrightarrow}_{\Gamma} ((s,i),A) \qquad \text{if } \exists p.\ s(x) = (v,p)$$

$$((s,i),A) \stackrel{x=v}{\Longrightarrow}_{\Gamma} \mathbf{abort} \qquad \text{if } x \notin \mathtt{dom}(s)$$

$$((s,i),A) \stackrel{x:=v}{\Longrightarrow}_{\Gamma} (([s \mid x : (v,\top)],i),A) \qquad \text{if } \exists v_0.\ s(x) = (v_0,\top)$$

$$((s,i),A) \stackrel{x:=v}{\Longrightarrow}_{\Gamma} \mathbf{abort}$$
$$\text{if } x \notin \mathtt{dom}(s) \text{ or } \exists v_0.\exists p \neq \top.\ s(x) = (v_0,p)$$

$$((s,i),A) \stackrel{acq(r)}{\Longrightarrow}_{\Gamma,r:\Psi} ((s \star s',i),A \cup \{r\})$$
$$\text{if } r \notin A, s \sharp s', (s',i) \models \Psi$$

$$((s,i),A) \stackrel{rel(r)}{\Longrightarrow}_{\Gamma,r:\Psi} ((s_2,i),A - \{r\})$$
$$\text{if } r \in A, \exists s_1.\ s = s_1 \star s_2 \wedge (s_1,i) \models \Psi$$

$$((s,i),A) \stackrel{rel(r)}{\Longrightarrow}_{\Gamma,r:\Psi} \mathbf{abort}$$
$$\text{if } r \in A, \forall s_1 \sharp s_2.\ s = s_1 \star s_2 \ \Rightarrow\ (s_1,i) \models \neg\Psi$$

$$((s,i),A) \stackrel{try(r)}{\Longrightarrow}_{\Gamma,r:\Psi} ((s,i),A) \qquad \text{always}$$

$$((s,i),A) \stackrel{abort}{\Longrightarrow}_{\Gamma} \mathbf{abort} \qquad \text{always}$$

$$\mathbf{abort} \stackrel{\lambda}{\Longrightarrow}_{\Gamma} \mathbf{abort} \qquad \text{always}$$

Note that an attempt to acquire $r$ when in a local state $(s,i)$ for which there is no $s'$ such that $s \sharp s'$ and $(s',i) \models \Psi$, so that the resource invariant for $r$ does not hold separately, is characterized as a *stuck* configuration. When the program is executing in a parallel environment that respects the resource invariants this situation does not happen, so we do not need to include an error-producing step under these circumstances. However, in sharp contrast, if a process releases a resource in a state for which there is no sub-state in which the invariant holds, this is indeed a violation of the resource rules and we treat it as an error.

Our insistence on precise resource invariants is important here, and crucial in the forthcoming soundness proof. Because of precision, the local transition rule for $rel(r)$ is unambiguous: in any state $(s,i)$ there is at most one sub-stack $s_1$ in which the invariant holds, so that when the invariant holds there is a unique $s_2$ such that $((s,i),A) \stackrel{rel(r)}{\Longrightarrow}_{\Gamma,r:\Psi} ((s_2,i),A - \{r\})$. Nevertheless, despite the precision assumption, the local enabling rules for *acquiring* allow non-determinism: for a given state $(s,i)$ there may be several compatible stacks $s'$ such that $(s',i) \models \Psi$, and the choice may depend on scheduling. It is here

that we allow for interference by the process's environment.

Since programs do not mention any logical variables, the values of logical variables are unaffected by the program's actions. It is easy to show that for all $\lambda$, all $s, i, i'$, and all $A, A'$, if $((s,i),A) \stackrel{\lambda}{\underset{\Gamma}{\Longrightarrow}} ((s',i'),A')$ then $i = i'$.

We generalize from actions to traces in the obvious way, by composition, using the obvious notation $((s,i),A) \stackrel{\alpha}{\underset{\Gamma}{\Longrightarrow}} ((s',i'),A')$.

Since program traces always acquire a resource before releasing it, by analogy with Lemma 5 we can show the following lemma.

**Lemma 11**
*If* $\alpha \in [\![C]\!]$ *and* $((s,i),A) \stackrel{\alpha}{\underset{\Gamma}{\Longrightarrow}} ((s',i),A')$
*then* $i = i'$, $A = A'$, *and* $((s,i),\{\}) \stackrel{\alpha}{\underset{\Gamma}{\Longrightarrow}} ((s',i),\{\})$.

We can now define a suitable notion of validity of resource-sensitive formulas, using the local enabling relation to formulate rigorously what it means to execute a trace in an environment that respects a resource context $\Gamma$.

**Definition 12 (Validity of formulas)**
$\Gamma \vdash_{vr} \{\Phi\}C\{\Phi'\}$ *is valid iff for all states* $(s,i)$ *and all traces* $\alpha \in [\![C]\!]$, *and all* $\sigma'$, *if* $(s,i) \models \Phi$ *and* $(s,i) \stackrel{\alpha}{\underset{\Gamma}{\Longrightarrow}} \sigma'$, *then* $\sigma' \neq$ **abort** *and* $\sigma' \models \Phi'$.

The reader should check at this point that the examples derived earlier all qualify as valid according to this definition. This is a worthwhile exercise, but is rendered unnecessary by the following result.

**Theorem 13 (Soundness of the logic)**
*Every well-formed provable formula is valid: if* $\Gamma$ *is a well-formed context and* $\Gamma \vdash_{vr} \{\Phi\}C\{\Phi'\}$ *is provable from the inference rules, then* $\Gamma \vdash_{vr} \{\Phi\}C\{\Phi'\}$ *is valid.*

**Proof**
To prove soundness we show that for each inference rule, if the premisses are valid so is the conclusion.

- For the Skip rule the result holds trivially. The proofs for Conseq and Exists are easy.
- The assignment rule Assn has no premisses, so we show directly that each instance of the rule's conclusion is valid. Consider such an instance, of the form

$$\Gamma \vdash_{vr} \{(\mathtt{Own}_\top(x) \star O) \wedge X = E\}x{:=}E\{(\mathtt{Own}_\top(x) \star O) \wedge x = X\},$$

  where $X$ is a logical variable. The pre-condition $(\mathtt{Own}_\top(x) \star O) \wedge X = E$ is true in $(s,i)$ if and only if there is an integer $v_0$ such that $s(x) = (v_0, \top)$ and $(s\backslash x, i) \models O$, $\mathtt{free}(E) \subseteq \mathtt{dom}(s)$, and $i(X) = |E|s$. This means that the state contains permissions to read the free variables of $E$ and total permission to write to $x$, so that no concurrent process can write to $x$ or write to a free variable of $E$, or read $x$, without violating the Permission

Principle. Every trace of $x{:=}E$ has the form $\rho\, x{:=}v$ for some $(\rho, v) \in [\![E]\!]$. Under the assumptions about $(s,i)$, it is clear that $\neg((s,i) \stackrel{\rho\, x:=v}{\Longrightarrow}_{\Gamma} \mathbf{abort})$. And if $(s,i) \stackrel{\rho\, x:=v}{\Longrightarrow}_{\Gamma} (s',i)$ then $v = |E|s$ and $s' = [s \mid x : (v, \top)]$, so that $(s',i) \models (\mathtt{Own}_{\top}(x) \star O) \wedge x = X$, as required for validity.

- The inference rules Cond, While, Seq are straightforward.
- Soundness of the Frame rule can be deduced from the following Lemma.

**Lemma 14 (Frame Property for Commands)**
*Let $\alpha \in [\![C]\!]$. Suppose that $s_1 \sharp s_2$ and $s = s_1 \star s_2$.*
· *If $(s,i) \stackrel{\alpha}{\Longrightarrow}_{\Gamma} \mathbf{abort}$ then $(s_1,i) \stackrel{\alpha}{\Longrightarrow}_{\Gamma} \mathbf{abort}$.*
· *If $(s,i) \stackrel{\alpha}{\Longrightarrow}_{\Gamma} (s',i)$ then either $(s_1,i) \stackrel{\alpha}{\Longrightarrow}_{\Gamma} \mathbf{abort}$ or there is a stack $s_1'$ such that $s_1' \sharp s_2$, $s' = s_1' \star s_2$, and $(s_1,i) \stackrel{\alpha}{\Longrightarrow}_{\Gamma} (s_1',i)$.*

- Soundness of the Par rule for $C_1 \| C_2$ is a consequence of the following Parallel Decomposition Lemma and its Corollary.

**Lemma 15 (Parallel Decomposition for Traces)**
*Let $\alpha_1, \alpha_2$ be traces, $A_1, A_2$ be disjoint sets of resources, and $\alpha \in \alpha_1 {}_{A_1}\|_{A_2} \alpha_2$. Let $s_1 \sharp s_2$ and $s = s_1 \star s_2$. Let $A = A_1 \cup A_2$.*
(i) *If $((s,i),A) \stackrel{\alpha}{\Longrightarrow}_{\Gamma} \mathbf{abort}$ then either $((s_1,i),A_1) \stackrel{\alpha_1}{\Longrightarrow}_{\Gamma} \mathbf{abort}$ or $((s_2,i),A_2) \stackrel{\alpha_2}{\Longrightarrow}_{\Gamma} \mathbf{abort}$.*
(ii) *If $((s,i),A) \stackrel{\alpha}{\Longrightarrow}_{\Gamma} ((s',i),A')$ then either $((s_1,i),A_1) \stackrel{\alpha_1}{\Longrightarrow}_{\Gamma} \mathbf{abort}$, or $((s_2,i),A_2) \stackrel{\alpha_2}{\Longrightarrow}_{\Gamma} \mathbf{abort}$, or there are stacks $s_1', s_2'$ and resource sets $A_1', A_2'$ such that $A_1' \cap A_2' = \{\}$, $A' = A_1 \cup A_2'$, $s' = s_1' \star s_2'$, $((s_1,i),A_1) \stackrel{\alpha_1}{\Longrightarrow}_{\Gamma} ((s_1',i),A_1')$, and $((s_2,i),A_2) \stackrel{\alpha_2}{\Longrightarrow}_{\Gamma} ((s_2',i),A_2')$.*

**Corollary 16 (Parallel Decomposition for Commands)**
*Let $\alpha_1 \in [\![C_1]\!]$, $\alpha_2 \in [\![C_2]\!]$, and $\alpha \in \alpha_1 \| \alpha_2$ be a trace of $C_1 \| C_2$. Let $s_1 \sharp s_2$ and $s = s_1 \star s_2$.*
(i) *If $(s,i) \stackrel{\alpha}{\Longrightarrow}_{\Gamma} \mathbf{abort}$ then $(s_1,i) \stackrel{\alpha_1}{\Longrightarrow}_{\Gamma} \mathbf{abort}$ or $(s_2,i) \stackrel{\alpha_2}{\Longrightarrow}_{\Gamma} \mathbf{abort}$.*
(ii) *If $(s,i) \stackrel{\alpha}{\Longrightarrow}_{\Gamma} (s',i)$, then $(s_1,i) \stackrel{\alpha_1}{\Longrightarrow}_{\Gamma} \mathbf{abort}$, or $(s_2,i) \stackrel{\alpha_2}{\Longrightarrow}_{\Gamma} \mathbf{abort}$, or there exist $s_1', s_2'$ such that $(s_1,i) \stackrel{\alpha_1}{\Longrightarrow}_{\Gamma} (s_1',i)$ and $(s_2,i) \stackrel{\alpha_2}{\Longrightarrow}_{\Gamma} (s_2',i)$, $s_1' \sharp s_2'$ and $s' = s_1' \star s_2'$.*

- Soundness of the Res rule for local resource blocks uses the following lemma. There is an analogous lemma for the Local rule.

**Lemma 17 (Local Resource)**
*Let $\Gamma$ be a resource context such that $r \notin \mathtt{dom}(\Gamma)$. Let $\beta \in [\![C]\!]_r$ and suppose $s_1 \sharp s_2, s = s_1 \star s_2$, and $(s_2,i) \models \Psi$.*
· *If $((s,i),A) \stackrel{\beta\backslash r}{\Longrightarrow}_{\Gamma} \mathbf{abort}$ then $((s_1,i),A) \stackrel{\beta}{\Longrightarrow}_{\Gamma, r:\Psi} \mathbf{abort}$.*
· *If $((s,i),A) \stackrel{\beta\backslash r}{\Longrightarrow}_{\Gamma} ((s',i),A')$ then either $((s_1,i),A) \stackrel{\beta}{\Longrightarrow}_{\Gamma, r:\Psi} \mathbf{abort}$ or there are stacks $s_1', s_2'$ such that $((s_1,i),A) \stackrel{\beta}{\Longrightarrow}_{\Gamma, r:\Psi} ((s_1',i),A')$, $s_1' \sharp s_2'$, $s' = s_1' \star s_2'$,*

*and* $(s_2', i) \models \Psi$.

- For the Reg rule our insistence that resource invariants be precise finally pays off. Let $\Gamma, r : \Psi$ be a context, where $\Psi$ is a precise formula, and suppose that $\Gamma \vdash_{vr} \{(\Phi \star \Psi) \wedge B\} C \{\Phi' \star \Psi\}$ is valid and $\Phi \star \Psi \Rightarrow B = B$ is true. We must show that

$$\Gamma, r : \Psi \vdash_{vr} \{\Phi\} \textbf{with } r \textbf{ when } B \textbf{ do } C \{\Phi'\}$$

is valid. So suppose $(s, i) \models \Phi$. Let $\alpha$ be a trace of **with** $r$ **when** $B$ **do** $C$. Without loss of generality we can assume that $\alpha$ has the form

$$acq(r)\, \beta_1\, rel(r)\, \ldots\, acq(r)\, \beta_n\, rel(r)\, acq(r)\, \beta\, \gamma\, rel(r),$$

where $\beta_1, \ldots, \beta_n \in [\![B]\!]_{false}$, $\beta \in [\![B]\!]_{true}$, and $\gamma \in [\![C]\!]$.

It is easy to show that for all $\beta_j \in [\![B]\!]_{false}$, $\neg((s, i) \xRightarrow[\Gamma, r:\Psi]{acq(r)\, \beta_j\, rel(r)} \textbf{abort})$. Indeed, the first action moves to a state $s \star s_1$ where $(s_1, i) \models \Psi$. Hence $(s \star s_1, i) \models \Phi \star \Psi$, and since $\Phi \star \Psi \Rightarrow B = B$ we have $\mathtt{free}(B) \subseteq \mathtt{dom}(s \star s_1) = \mathtt{dom}(s) \cup \mathtt{dom}(s_1)$. This means that the stack $s \star s_1$ contains permissions for the free variables of $B$, so the read actions in $\beta_j$ do not abort. Further, if $\beta_j$ is enabled from $s \star s_1$ the subsequent $rel(r)$ action will not abort, because the stack after the reads will still be $s \star s_1$ and contains a sub-stack satisfying the invariant $\Psi$. By precision, the only sub-stack with this property is $s_1$. It follows that for all $s'$, if $(s, i) \xRightarrow[\Gamma, r:\Psi]{acq(r)\, \beta_j\, rel(r)} (s', i)$ then $s = s'$.

It remains to consider the final part of the trace $\alpha$, structured as above. We must show first that $\neg((s, i) \xRightarrow[\Gamma, r:\Psi]{acq(r)\, \beta\, \gamma\, rel(r)} \textbf{abort})$. Using a similar argument to the above, the abort case can only happen if there is a stack $s_1$ such that $(s_1, i) \models \Psi$, $s \sharp s_1$, $|B|(s \star s_1) = \mathit{true}$, and $(s \star s_1, i) \xRightarrow{\gamma}_{\Gamma} \textbf{abort}$. This contradicts our previous assumption that $\Gamma \vdash_{vr} \{(\Phi \star \Psi) \wedge B\} C \{\Phi' \star \Psi\}$ is valid.

Now let $(s, i) \xRightarrow[\Gamma, r:\Psi]{acq(r)\, \beta\, \gamma\, rel(r)} (s', i)$. By definition of the logical enabling relation there must be stacks $s_{n+1}, s''$ such that $s \sharp s_{n+1}$, $(s_{n+1}, i) \models \Psi$, $(s \star s_{n+1}, i) \xRightarrow[\Gamma, r:\Psi]{\beta} (s \star s_{n+1}, i)$, $((s \star s_{n+1}, i), \{r\}) \xRightarrow[\Gamma, r:\Psi]{\gamma} ((s'', i), \{r\})$ and $((s'', i), \{r\}) \xRightarrow[\Gamma, r:\Psi]{rel(r)} (s', i)$. Thus $s \star s_1 \models B$ and $s \star s_1 \models \Phi \star \Psi$. By validity of $\Gamma \vdash_{vr} \{(\Phi \star \Psi) \wedge B\} C \{\Phi' \star \Psi\}$ it follows that $(s'', i) \models \Phi' \star \Psi$, so the final $rel(r)$ action moves to a stack $s'$ satisfying $\Phi'$, as required.

- For the Aux rule, let $\mathcal{Y}$ be an auxiliary set for $C$ disjoint from $\mathtt{free}(\Phi, \Psi, \Gamma)$, and suppose the formula $\Gamma \vdash_{vr} \{\Phi \star \mathtt{Own}_\top(\mathcal{Y})\} C \{\Psi \star \mathtt{Own}_\top(\mathcal{Y})\}$ is valid. Since $\mathcal{Y}$ is auxiliary for $C$, $[\![C \backslash \mathcal{Y}]\!] = \{\alpha \backslash \mathcal{Y} \mid \alpha \in [\![C]\!]\}$, where $\alpha \backslash \mathcal{Y}$ is the trace obtained from $\alpha$ by replacing each read or write to an auxiliary variable by $\delta$. Let $\sigma \models \Phi$ and, without loss of generality, $\mathtt{dom}(\sigma) \cap \mathcal{Y} = \{\}$. Suppose $\alpha \in [\![c]\!]$ and $\sigma \xRightarrow[\Gamma]{\alpha \backslash \mathcal{Y}} \sigma'$. We must show that $\sigma' \neq \textbf{abort}$, and $\sigma' \models \Psi$. Since $\mathtt{dom}(\sigma) \cap \mathcal{Y} = \{\}$ we can choose a totally permissive state $\sigma_1$ such that $\sigma \sharp \sigma_1$

and $\mathtt{dom}(\sigma_1) = \mathcal{Y}$. Thus $\sigma \star \sigma_1 \models \Phi \star \mathtt{Own}_\top(\mathcal{Y})$. By validity of the premiss, $\sigma \star \sigma_1 \stackrel{\alpha}{\underset{\Gamma}{\Longrightarrow}} \sigma''$ for some state $\sigma'' \neq \mathbf{abort}$ such that $\sigma'' \models \Psi \star \mathtt{Own}_\top(\Gamma)$. Since $\mathcal{Y}$ is auxiliary for $C$, $\sigma''$ must be expressible (uniquely) in the form $\sigma'' = \sigma' \star \sigma_1'$, for some totally permissive $\sigma_1'$ such that $\sigma' \sharp \sigma_1'$ and $\mathtt{dom}(\sigma_1') = \mathcal{Y}$. (Here we have $\sigma \stackrel{\alpha\backslash\mathcal{Y}}{\underset{\Gamma}{\Longrightarrow}} \sigma'$ and $\sigma_1 \stackrel{\alpha\lceil\mathcal{Y}}{\underset{\Gamma}{\Longrightarrow}} \sigma_1'$.) Hence $\sigma' \neq \mathbf{abort}$ and $\sigma' \models \Psi$, as required.

Thus we have established that concurrent permissions logic is sound. We can instantiate this result by choosing either of the two notions of permission mentioned earlier, to deduce that the logic of fractional permissions (as described in [1]) is sound, and similarly that the logic of source permissions (as in [1]) is sound. It is also obvious that augmenting the logic with additional axioms concerning a specific permissions model, such as DIV when adopting the fractional model, does not destroy soundness.

## 10 Connecting local and global

We have proven that the logic is sound with respect to a "logical enabling" relation that formalizes the sense in which a process and its environments co-operate in a rely/guarantee discipline moderated by resource invariants. It remains to connect this notion of soundness, based as it is on the logic, with a more independent notion of computation in which resource invariants play no defining rôle. We have already introduced the relevant notion: the "global" enabling relations $\stackrel{\lambda}{\Longrightarrow}$ on stacks. The following results characterize the relationship between logical (or "local") computations and "global" computations and connect with the *Permission Principle* mentioned earlier.

**Lemma 18 (Local/Global Property for Actions)**
*Let $\Gamma$ be a resource context and $A$ be a set of resources. Suppose $s_1 \sharp s_2$, $s = s_1 \star s_2$, and $(s_2, i) \models \mathtt{inv}(\Gamma\backslash A)$.*

- *If $(s, A) \stackrel{\lambda}{\Longrightarrow} \mathbf{abort}$ then $((s_1, i), A) \stackrel{\lambda}{\underset{\Gamma}{\Longrightarrow}} \mathbf{abort}$.*
- *If $(s, A) \stackrel{\lambda}{\Longrightarrow} (s', A')$ then either $((s_1, i), A) \stackrel{\lambda}{\underset{\Gamma}{\Longrightarrow}} \mathbf{abort}$, or there are $s_1', s_2'$ such that $s_1' \sharp s_2'$, $s' = s_1' \star s_2'$, $(s_2', i) \models \mathtt{inv}(\Gamma\backslash A')$, and $(s_1, A) \stackrel{\lambda}{\underset{\Gamma}{\Longrightarrow}} (s_1', A')$.*

**Corollary 19 (Local/Global Property for Commands)**
*Suppose $s_1 \sharp s_2$, $s = s_1 \star s_2$ and $(s_2, i) \models \mathtt{inv}(\Gamma)$. Let $\alpha \in [\![C]\!]$.*

*If $s \stackrel{\alpha}{\Longrightarrow} \mathbf{abort}$ then $(s_1, i) \stackrel{\alpha}{\underset{\Gamma}{\Longrightarrow}} \mathbf{abort}$.*

*If $s \stackrel{\alpha}{\Longrightarrow} s'$ then either $(s_1, i) \stackrel{\alpha}{\underset{\Gamma}{\Longrightarrow}} \mathbf{abort}$, or there are $s_1', s_2'$ such that $s_1' \sharp s_2'$, $s' = s_1' \star s_2'$, $(s_2', i) \models \mathtt{inv}(\Gamma)$, and $(s_1, i) \stackrel{\alpha}{\underset{\Gamma}{\Longrightarrow}} (s_1', i)$.*

Finally we deduce the following connection with the informal notion of validity discussed earlier.

**Theorem 20 (Provability implies no race)**
*Let* $\Gamma \vdash_{vr} \{\Phi\}C\{\Psi\}$ *be a valid formula. For all* $\sigma$ *such that* $\sigma \models \Phi \star \mathtt{inv}(\Gamma)$*, all traces* $\alpha \in [\![C]\!]$*, and all* $\sigma'$*, if* $\sigma \stackrel{\alpha}{\Longrightarrow} \sigma'$ *then* $\sigma' \neq$ **abort** *and* $\sigma' \models \Psi \star \mathtt{inv}(\Gamma)$*.*

In particular, when $C$ has no free resource names (in which case $\Gamma$ can be chosen to be the empty context, without loss of generality), validity of $\vdash_{vr} \{\Phi\}C\{\Psi\}$ implies conventional fault-free partial correctness with respect to pre-condition $\Phi$ and post-condition $\Psi$.

## 11 Conclusions

We have adapted an earlier semantics-based soundness proof for concurrent separation logic to a permissive setting, and established the soundness of a permissions logic that generalizes and extends the earlier logic to incorporate reasoning about concurrent reads. Although we omitted pointer operations from our programming language it is straightforward to deal with them in our semantic model (as shown in [5]) and the permissions logic can also deal appropriately with pointers [1,12]. Our soundness proof can also be adapted in a straightforward and systematic way to cope with pointers. Since fractional permissions [3,1] and counting permissions [1] are special cases of the general notion of permission model upon which our development is based, we can also deduce from our soundness analysis that the proof rules for fractional permissions and counting permissions given by [1] are sound.

So far our semantic model deals only with first-order concurrency, in a language without procedures. We would like to extend these ideas to cover a simply typed procedural language with shared memory concurrency. Perhaps this might be possible by combining the possible worlds approach with action traces, in the spirit with which we were able to develop a semantics for Parallel Algol [6], leading to an Algol-like (and call-by-name) shared-memory language with pointers. On the other hand, it is not clear if this approach would work for a call-by-value language. The logic of [12] also incorporates procedures with simple call-by-reference parameters and call-by-value parameters, but no procedure parameters. There may be a less elaborate way to extend action traces to this more limited setting. It would also be interesting to adapt the *footstep traces* model [4] to the permissive setting, and to incorporate procedures.

## 12 Acknowledgements

Thanks to Cristiano Calcagno for introducing me to [12] and for answering many questions. Thanks to Matthew Parkinson for several helpful suggestions and clarifications, to Richard Bornat for his unique style of encouragement, and to Ruy Ley-Wild for careful proof-reading. The anonymous referees made some helpful comments.

## References

[1] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. Parkinson. *Permission accounting in separation logic*. In POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 259-270, New York, Jan. 2005. ACM Press.

[2] R. Bornat, C. Calcagno, and H. Yang. *Variables as resource in separation logic*. Proc. MFPS XXI, Birmingham, May 2005. To appear, Electronic Notes in Theoretical Computer Science, Elsevier Science, 2006.

[3] J. Boyland. *Checking interference with fractional permissions*. Proc. $10^{th}$ Symposium on Static Analysis, R. Cousot, editor. Springer LNCS vol. 2694, pp. 55-72, 2003.

[4] S. Brookes. *A Grainless Semantics for Parallel Programs with Shared Mutable Data*. Proc. MFPS XXI. Birmingham, UK. May 2005. Elsevier ENTCS.

[5] S. Brookes. *A semantics for concurrent separation logic*. To appear, Theoretical Computer Science, 2006. Preliminary version appeared in Proc. CONCUR '04, Springer LNCS vol. 3170, pp. 16-34, August 2004.

[6] S. Brookes. *The Essence of Parallel Algol*. Proc. $11^{th}$ Annual Symposium on Logic in Computer Science, pp. 164-173. IEEE Computer Society Press, 1997.

[7] C. Calcagno, P.W. O'Hearn, and H. Yang. *Soundness of Abstract Concurrent Separation Logic*. Manuscript, 2004.

[8] S. Isthiaq and P. W. O'Hearn. *BI as an assertion language for mutable data structures*. Proc. $28^{th}$ POPL conference, pp. 36-49, January 2001.

[9] P.W. O'Hearn. *Resources, Concurrency, and Local Reasoning*. To appear in Theoretical Computer Science. Preliminary version appeared in Proc. CONCUR '04, Springer LNCS, London, August 2004.

[10] P. W. O'Hearn and D. J. Pym. *The logic of bunched implications*. Bulletin of Symbolic Logic, 5(2):215-244, June 1999.

[11] S. Owicki and D. Gries, *Verifying properties of parallel programs: An axiomatic approach*, Comm. ACM. 19(5):279-285, May 1976.

[12] M. Parkinson, R. Bornat, and C. Calcagno. *Variables as Resource in Hoare Logic*. To appear, Proc. $21^{st}$ IEEE Conference on Logic in Computer Science, LICS 2006. IEEE Computer Society Press, 2006.

[13] J.C. Reynolds, *Separation logic: a logic for shared mutable data structures*, Invited paper. Proc. $17^{th}$ IEEE Conference on Logic in Computer Science, LICS 2002, pp. 55-74. IEEE Computer Society, 2002.