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

Stephen Brookes
Carnegie Mellon University

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.

1 Introduction

In recent work, Parkinson, Bornat, and Calcagno have introduced a logic in which program variables are treated as resource [10], generalizing earlier work based on separation logic [11] and permissions [1, 2, 3]. In prior work on concurrent separation logic, for programs operating on mutable state, the heap was treated as resource: mutual exclusion for heap cells was enforced within the logic through judicious use of *separating 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 [4, 6]. In the new logic [10] program variables are treated as resource, governed by a permission discipline designed to allow concurrent reads to a variable 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 book-keeping of variable usage, 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 [9] and its descendants [4, 6].

Concurrent separation logic [4, 6] 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 Γ is a resource context associating a list of resource names with protection lists (of variables) and resource invariants. Parkinson, Bornat, and Calcagno have shown [10] 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.

Their paper [10] sketches 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. 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: 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 [10] 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. In fact we are able to re-use the *action traces* semantic model [4], introduced earlier for the original concurrent separation logic, and we will provide a soundness proof in the same style as [4], making a series of appropriate adjustments to deal with permissions and ownership. Thus we obtain an elegant validation proof for the concurrency logic of [10], based on a straightforward trace-theoretical denotational semantics.

Since we build directly on the logic presented in [10], we include here a summary of the relevant background, logic syntax and semantics, and inference rules from that paper. 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 [10] 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 [10, 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

We use a traditional shared-memory language, including the usual sequential constructs together with 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*. [1] We let C range over commands (or processes), B over boolean expressions, and E over integer expressions. Such expressions may contain program variables,

[1] 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.

and we let x range over program variables. We let $\texttt{free}(C)$ be the set of free variables occurring in C, and we let $\texttt{res}(C)$ be the set of free resource names in C. In particular,

$$\begin{aligned}
&\texttt{res}(C_1 \| C_2) = \texttt{res}(C_1) \cup \texttt{res}(C_2) \\
&\texttt{res}(\textbf{with } r \textbf{ when } B \textbf{ do } C) = \{r\} \cup \texttt{res}(C) \\
&\texttt{res}(\textbf{resource } r \textbf{ in } C) = \texttt{res}(C) - \{r\}
\end{aligned}$$

3 Program semantics

We use the action trace semantics for programs, as in [4]. We include some of the key ingredients here, but refer to [4] for details.

A program 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 λ, include δ (idle), $x = v$ (a read), $x{:=}v$ (a write), resource actions *try(r)*, *acq(r)* and *rel(r)*, and an error action *abort*. Here v ranges over the set V_{int} of integer values. 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 value. The semantics of expressions is given denotationally. Here are the semantic clauses for a constant, an identifier, and a sum expression. The remaining clauses are similar.

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

Similarly 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 $[\![B]\!]_{false}$.

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.

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

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

The definition of parallel composition performs book-keeping to keep track of the resources held by each process, only allows actions when they 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 λ, 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:

$$
\begin{array}{ll}
(A_1, A_2) \xrightarrow{\mathit{acq}(r)} (A_1 \cup \{r\}, A_2) & \text{if } r \notin A_1 \cup A_2 \\
(A_1, A_2) \xrightarrow{\mathit{rel}(r)} (A_1 - \{r\}, A_2) & \text{if } r \in A_1 \\
(A_1, A_2) \xrightarrow{\lambda} (A_1, A_2) & \text{for all other actions}
\end{array}
$$

  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 α.

- We write $\lambda_1 \bowtie \lambda_2$ (λ_1 *interferes with* λ_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 (α_1, α_2) of action sequences, the set $\alpha_1 {}_{A_1}\|_{A_2} \alpha_2$ of all *mutex fairmerges* of α_1 using A_1 with α_2 using A_2. The definition for finite sequences is inductive:

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

  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. [2]

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 r is assumed initially available and the actions involving r are executable without interference. We let $[\![C]\!]_r$ be the set of traces of C which are *sequential for* r in this manner. Equivalently, α 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 α by replacing each action on r by δ.

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 α of C such that the sequence of reads and writes to x along α is sequentially consistent with the initial value v for x. 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.

[2] This formulation of parallel composition differs from the version used in previous papers on concurrent separation logic; the new version can be used equally well to show the soundness of concurrent separation logic as in [4]. Although slightly more verbose – the trace set of $c_1 \| c_2$ includes *all* fair interleavings even if the program is racy – the new formulation has the advantage of associativity. This issue is not crucial for achieving soundness, and for race-free programs the two formulations coincide.

4 Stacks, permissions, and states

To prepare for the forthcoming logic we augment the traditional notions of stack (or store) and state with permissions. In a "permissive state" 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 [10].

A *stack* is a finite partial function from program variables to integers tagged with permissions. The set of permissions, denoted $\mathcal{P}$, is equipped with a distinguished element $\top$ (standing for "total" permission) and a partial "composition" function $\otimes$ such that $(\mathcal{P}, \otimes)$ is a partial injective commutative semigroup satisfying the following properties [3]:

1. $\forall p \in \mathcal{P}.\ (\top \otimes p)$ is undefined
2. $\forall p, p' \in \mathcal{P}.\ (p \otimes p' \neq p)$
3. $\forall p \in \mathcal{P}. \exists p_1, p_2 \in \mathcal{P}.\ (p = p_1 \otimes p_2)$

Permissions p_1 and p_2 such that $p_1 \otimes p_2$ is defined are called *compatible*. We do not need to fix in advance a particular model of permissions; the logic relies only on the above general properties. However, it is worth remarking that "fractional permissions" 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.

We let s range over stacks, and let S be the set of stacks. Thus we have $S = \mathbf{Var} \rightharpoonup_{\mathit{fin}} V_{\mathit{int}} \times \mathcal{P}$.

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

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

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

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 $x \notin \mathtt{dom}(s))$,

[3] Matthew Parkinson points out that only the first two properties are needed in the soundness proof of this paper; and that the third property is required to encode Hoare logics as in [10], and can be of considerable assistance in program proofs. We leave it in to facilitate proofs about concurrent reads.

or there are p', p'' such that $s(x) = (v, p')$ and $s'(x) = (v, p'')$ and $p = p' \otimes p''$. Note the important fact that for all stacks s and s_1 there is at most one stack s_2 such that $s_1 \sharp s_2 \ \wedge \ s = s_1 \star s_2$. When $s = s_1 \star s_2$ we say that s_1 and s_2 are sub-states of s.

Trivially when $\mathtt{dom}(s_1) \cap \mathtt{dom}(s_2) = \{\}$ we have $s_1 \sharp s_2$, and $s_1 \star s_2 = s_1 \cup s_2$.

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 $(s \backslash x) \star \{(x, (v, p))\}$.

An *interpretation* is a finite partial function mapping logical variables to integers and permission 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 σ range over states. We let $\mathtt{dom}(s, i) = \mathtt{dom}(s) \cup \mathtt{dom}(i)$.

5 A logic

The syntax of assertions (as in [10]) is given by the following abstract grammar, in which Φ ranges over assertions, X over logical variables, and p over permission expressions (built from permission variables using $\otimes$). We assume that the three types of variables range over disjoint sets.

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

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. Let **Var** be the set of program variables and **LVar** be the set of logical variables. Quantification is only allowed for logical variables.

6 Semantics of the logic

We assume given the evaluation semantics for 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 $|E|s$. Similarly we assume given the evaluation semantics for permission expressions. 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.

We define (as in [10]) a *forcing semantics* for assertions. When $(s,i) \models \Phi$ we say that the state (s,i) satisfies Φ, or that Φ holds in (s,i). We write $\models \Phi$ when Φ holds in all states.

Definition 2
For a state (s,i) and formula Φ we define the truth value $(s,i) \models \Phi$ by induction on the structure of Φ:

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

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

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

The characteristic properties of the set of permissions give rise to some logical equivalences, e.g.

$$\mathtt{Own}_p(x) \Leftrightarrow \exists p_1, p_2.(\mathtt{Own}_{p_1}(x) \star \mathtt{Own}_{p_2}(x) \ \wedge \ p = p_1 \otimes p_2).$$

and

$$\mathtt{Own}_{p_1}(x) \star \mathtt{Own}_{p_2}(x) \Leftrightarrow \exists p.\ (p = p_1 \otimes p_2 \ \wedge \mathtt{Own}_p(x)).$$

In particular, $\mathtt{Own}_{p_1}(x) \star \mathtt{Own}_{\top}(x)$ is equivalent to **false**.

Let $\mathtt{free}(E)$ be the set of program variables and logical variables with free occurrences in E. Thus $\mathtt{free}(E) \cap \mathtt{Var}$ 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 $\mathtt{free}(E) \subseteq \mathtt{dom}(s)$, since expressions E do not contain logical variables. 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 $s = \{(x_1, (v_1, [\![p_1]\!]i)), \ldots, (x_k, (v_k, [\![p_k]\!]i))\}$. [4]

7 A partial correctness logic

7.1 Formulas

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

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

where Γ 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 Φ_j is a *precise* formula. The class of precise formulas is defined as follows.

Definition 3
A formula Φ 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. Note that when Φ and Ψ are precise, so is $\Phi \star \Psi$. When Φ and Ψ are precise, so is $(B \wedge \Phi) \vee (\neg B \wedge \Psi)$.

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.

Let Γ 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 Γ for which $r_j \notin A$. We let $\mathtt{dom}(\Gamma) = \{r_1, \ldots, r_k\}$ and $\mathtt{free}(\Gamma) = \bigcup_{j=1}^{k} \mathtt{free}(\Phi_j)$.

In contrast to concurrent separation logic, a resource context does not explicitly designate a protection list of variables, in addition to a resource invariant, for each resource name. Instead the resource invariant does double duty, indicating ownership constraints directly as part of the invariant.

[4] Also note that $\neg(E_1 = E_2)$ is not logically equivalent to $E_1 \neq E_2$. As remarked in BCP, 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 $(\exists p.\ \mathtt{Own}_p(x) \wedge x = 0) \star \mathbf{true}$.

7.2 Inference rules

Here are the inference rules. In addition to the rules listed in [10] we include the natural analogue of the Owicki-Gries auxiliary variables rule. Some of the rules have side conditions expressing syntactic "freshness" assumptions. In the ensuing text we will elaborate on these and related issues.

SKIP

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

ASSIGN

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

SEQ

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

COND

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

WHILE

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

PAR

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

RESOURCE

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

Region

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

Local

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

Renaming

$$\frac{\Gamma \vdash_{vr} \{\Phi\} \textbf{resource } r' \textbf{ in } [r'/r] C \{\Psi\}}{\Gamma \vdash_{vr} \{\Phi\} \textbf{resource } r \textbf{ in } C \{\Psi\}} \qquad (\mathrm{r} \notin \mathtt{res}(\mathrm{C}))$$

Frame

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

Consequence

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

Exists

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

Aux

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

Comments

- The rules for local resource declaration and conditional critical region carry an implicit side condition: that $r \notin \mathtt{dom}(\Gamma)$ and Ψ is precise, so that the context $\Gamma, r : \Psi$ is well formed. The rule for changing a bound resource name carries the obvious constraint 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 for conditional and loops to ensure that every state satisfying Φ will contain values (and permissions) for the free variables of B.

- The rule for local variables requires that the bound variable x' be fresh, i.e. not free in Γ, Φ, Ψ 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 existential quantification rule X ranges over logical variables (including permission variables), but we disallow quantification over program variables.

- The auxiliary variables rule carries a side condition (†) defined as follows: the set Y must be auxiliary for C, and the members of Y must not occur free in Φ, Φ' or Γ. This constraint ensures 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.

  We write $\mathtt{Own}_\top(Y)$ for $\mathtt{Own}_\top(y_1) \star \cdots \star \mathtt{Own}_\top(y_n)$, where $y_1, \ldots, y_n$ is an enumeration without repetition of the set Y.

  A set Y of program variables is auxiliary for C if every free occurrence in C of a variable from Y is in an assignment $x{:=}E$ for which x also belongs to Y. The program $C\backslash Y$ 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 freshness constraint on Y means that $\mathtt{free}(\Phi, \Phi', \Gamma) \cap Y = \{\}$.

  The constraints expressed by the side condition are crucial in ensuring soundness; this can be seen even before we formulate rigorously what soundness means. For example, although $\vdash_{vr} \{y = 0\} y{:=}1 \{y = 1\}$ is provable (and surely valid!), and $\{y\}$ is clearly an auxiliary variable set for this program, the formula $\vdash_{vr} \{y = 0\}\mathbf{skip}\{y = 1\}$ is (again clearly) not valid.

  Similarly, $r : x = y \vdash_{vr} \{x = 0\}\mathbf{with}\ r\ \mathbf{do}\ (x{:=}1; y{:=}1)\{x = 1\}$ is provable and valid, again $\{y\}$ is an auxiliary set, but we should expect that $r : x = y \vdash_{vr} \{x = 0\}\mathbf{with}\ r\ \mathbf{do}\ x{:=}1\{x = 1\}$ is invalid.

8 Towards validity

Intuitively, a formula $\Gamma \vdash \{\Phi\}C\{\Psi\}$ expresses a form of race-free partial correctness property of C when executed in an environment that respects Γ. 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 $\mathtt{inv}(\Gamma) \star \Psi$. We can fomalize this notion of validity as follows, using 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 [4], adjusted to take account of permissions.

Definition 4
For each action λ we define the global enabling relation

$$\stackrel{\lambda}{\Longrightarrow} \subseteq (S \cup \{\mathbf{abort}\}) \times (S \cup \{\mathbf{abort}\})$$

as follows.

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

It is easy to see that this definition coincides with the notion of global enabling of [4] when restricted to totally permissive stacks. For a stack $s : \mathbf{Var} \rightharpoonup_{\mathit{fin}} (V_{\mathit{int}} \times \mathcal{P})$ let $\hat{s} = \{(x, v) \mid \exists p.\, (x, (v, p)) \in s\} : \mathbf{Var} \rightharpoonup_{\mathit{fin}} V_{\mathit{int}}$. For every partial function $f : \mathbf{Var} \rightharpoonup_{\mathit{fin}} V_{\mathit{int}}$ there is a unique totally permissive stack s such that $\hat{s} = f$. If every variable in $\mathtt{dom}(s)$ has total permission in s, $(s, A) \stackrel{\lambda}{\Longrightarrow} (s', A')$ if and only if $(\hat{s}, A) \stackrel{\lambda}{\Longrightarrow} (\hat{s'}, A')$ using the enabling relation of [4].

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 5
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. Of course this is not true of arbitrary traces. We write $s \stackrel{\alpha}{\Longrightarrow} s'$ when $(s, \{\}) \stackrel{\alpha}{\Longrightarrow} (s', \{\})$.

9 Validity and soundness

Next we define logical enabling relations $\stackrel{\lambda}{\Longrightarrow}_{\Gamma}$, again as in [4], adjusted to deal with permissions. The idea is that for each action λ this relation captures the local view of a process that performs this action in a parallel environment that respects the resources. A "local" state for a process contains the variables for which the process currently "claims" a permission. Since the resource invariants in Γ may mention logical variables and/or permission variables, we need to work with states of the form (s, i) here.

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

Definition 6
For each action λ and resource context Γ we define the logical enabling relation $\overset{\lambda}{\underset{\Gamma}{\Longrightarrow}}$ as follows.

$((s,i),A) \overset{\delta}{\underset{\Gamma}{\Longrightarrow}} ((s,i),A)$	always
$((s,i),A) \overset{x=v}{\underset{\Gamma}{\Longrightarrow}} ((s,i),A)$	if $\exists p.\ s(x) = (v,p)$
$((s,i),A) \overset{x:=v}{\underset{\Gamma}{\Longrightarrow}} (([s \mid x : (v,\top)],i),A)$	if $\exists v_0.\ s(x) = (v_0,\top)$
$((s,i),A) \overset{try(r)}{\underset{\Gamma}{\Longrightarrow}} ((s,i),A)$	always
$((s,i),A) \overset{acq(r)}{\underset{\Gamma}{\Longrightarrow}} ((s \star s',i),A \cup \{r\})$	if $r \notin A, r : \Psi \in \Gamma, s \sharp s'$, and $(s',i) \models \Psi$
$((s,i),A) \overset{rel(r)}{\underset{\Gamma}{\Longrightarrow}} ((s_2,i),A - \{r\})$	if $r \in A, r : \Psi \in \Gamma, s = s_1 \star s_2, (s_1,i) \models \Psi$
$((s,i),A) \overset{rel(r)}{\underset{\Gamma}{\Longrightarrow}} \mathbf{abort}$	if $r \in A, r : \Psi \in \Gamma$, and $\forall s_1, s_2.\ (s = s_1 \star s_2$ implies $(s_1,i) \models \neg\Psi)$
$((s,i),A) \overset{x=v}{\underset{\Gamma}{\Longrightarrow}} \mathbf{abort}$	if $x \notin \mathtt{dom}(s)$
$((s,i),A) \overset{x:=v}{\underset{\Gamma}{\Longrightarrow}} \mathbf{abort}$	if $x \notin \mathtt{dom}(s) \vee \exists v_0 . \exists p \neq \top.\ s(x) = (v_0,p)$
$((s,i),A) \overset{abort}{\underset{\Gamma}{\Longrightarrow}} \mathbf{abort}$	always
$\mathbf{abort} \overset{\lambda}{\underset{\Gamma}{\Longrightarrow}} \mathbf{abort}$	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-state s_1 in which the invariant holds, so that when the invariant holds there is a unique s_2 such that $((s, i), A) \xRightarrow[\Gamma]{rel(r)} ((s_2, i), A - \{r\})$. 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 states 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 λ, all s, i, i', and all A, A', if $((s, i), A) \xRightarrow[\Gamma]{\lambda} ((s', i'), A')$ then $i = i'$.

We generalize from actions to traces in the obvious way, by composition.

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 Γ.

Definition 7 (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* σ'*, if* $(s, i) \models \Phi$ *and* $(s, i) \xRightarrow[\Gamma]{\alpha} \sigma'$*, then* $\sigma' \neq$ **abort** *and* $\sigma' \models \Phi'$*.*

Theorem 8 (Soundness of the logic)
Every well-formed provable formula is valid: if Γ *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 **skip** the result holds trivially.

- The assignment rule has no premisses, so we show directly that each instance of the rule's conclusion is valid. Consider such an instance, of the form

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

  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 sufficient 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. Indeed, 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 for if-then-else, while-loops and sequential composition are straightforward cases.

- Soundness of the frame rule can be deduced easily from the following Frame Lemma and its Corollary.

  Lemma 9 (Frame Property for Actions)
  Let λ be an action, and suppose that $s_1 \sharp s_2$ and $s = s_1 \star s_2$.

  - *If $((s, i), A) \stackrel{\lambda}{\Longrightarrow}_\Gamma$ **abort** then $((s_1, i), A) \stackrel{\lambda}{\Longrightarrow}_\Gamma$ **abort**.*
  - *If $((s, i), A) \stackrel{\lambda}{\Longrightarrow}_\Gamma ((s', i), A')$ then either $((s_1, i), A) \stackrel{\lambda}{\Longrightarrow}_\Gamma$ **abort**, or there is a stack s_1' such that $((s_1, i), A) \stackrel{\lambda}{\Longrightarrow}_\Gamma ((s_1', i), A')$, $s_1' \sharp s_2$, and $s' = s_1' \star s_2$.*

  Corollary 10 (Frame Property for Commands)
  Let $\alpha \in [\![C]\!]$. Suppose that $s_1 \sharp s_2$ and $s = s_1 \star s_2$.

  - *If $(s, i) \stackrel{\alpha}{\Longrightarrow}_\Gamma$ **abort** then $(s_1, i) \stackrel{\alpha}{\Longrightarrow}_\Gamma (s_1', i)$.*

– *If* $(s,i) \stackrel{\alpha}{\Longrightarrow}_{\Gamma} (s',i)$ *then either* $(s_1,i) \stackrel{\alpha}{\Longrightarrow}_{\Gamma}$ **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 rule for $C_1 \| C_2$ follows from the following Parallel Decomposition Lemma and its Corollary.

Lemma 11 (Parallel Decomposition for Traces)
Let α_1, α_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* $A = A_1 \cup A_2$.

1. *If* $((s,i),A) \stackrel{\alpha}{\Longrightarrow}_{\Gamma}$ **abort** *then either* $((s_1,i),A_1) \stackrel{\alpha_1}{\Longrightarrow}_{\Gamma}$ **abort** *or* $((s_2,i),A_2) \stackrel{\alpha_2}{\Longrightarrow}_{\Gamma}$ **abort**.
2. *If* $((s,i),A) \stackrel{\alpha}{\Longrightarrow}_{\Gamma} ((s',i),A')$ *then either* $((s_1,i),A_1) \stackrel{\alpha_1}{\Longrightarrow}_{\Gamma}$ **abort**, *or* $((s_2,i),A_2) \stackrel{\alpha_2}{\Longrightarrow}_{\Gamma}$ **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 12 (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* $C_1 \| C_2$. *Suppose* $s = s_1 \star s_2$.

1. *If* $(s,i) \stackrel{\alpha}{\Longrightarrow}_{\Gamma}$ **abort** *then* $(s_1,i) \stackrel{\alpha_1}{\Longrightarrow}_{\Gamma}$ **abort** *or* $(s_2,i) \stackrel{\alpha_2}{\Longrightarrow}_{\Gamma}$ **abort**.
2. *If* $(s,i) \stackrel{\alpha}{\Longrightarrow}_{\Gamma} (s',i)$, *then* $(s_1,i) \stackrel{\alpha_1}{\Longrightarrow}_{\Gamma}$ **abort**, *or* $(s_1,i) \stackrel{\alpha_1}{\Longrightarrow}_{\Gamma}$ **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 rule for local resource blocks uses the following lemma.

Lemma 13 (Local Resource)
Let Γ *be a resource context such that* $r : \Psi \in \Gamma$. *Let* $\beta \in [\![C]\!]_r$ *and suppose* $s_1 \sharp s_2, s = s_1 \otimes s_2$, *and* $(s_2,i) \models \Psi$.

– *If* $((s,i),A) \stackrel{\beta \backslash r}{\Longrightarrow}_{\Gamma \backslash r}$ **abort** *then* $((s_1,i),A) \stackrel{\beta}{\Longrightarrow}_{\Gamma}$ **abort**.

– *If* $((s,i),A) \stackrel{\beta\backslash r}{\Longrightarrow}_{\Gamma\backslash r} ((s',i),A')$ *then either* $((s_1,i),A) \stackrel{\beta}{\Longrightarrow}_{\Gamma}$ **abort** *or there are stacks* s'_1, s'_2 *such that* $s'_1 \sharp s'_2$, $s' = s'_1 \star s'_2$, $(s'_2,i) \models \Psi$, *and* $((s_1,i),A) \stackrel{\beta}{\Longrightarrow}_{\Gamma} ((s'_1,i),A')$.

- There is an analogous lemma for local variable blocks.
- For the rule dealing with conditional critical regions, our insistence that resource invariants be precise finally pays off.

  Let $\Gamma, r : \Psi$ be a context, so that Ψ 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 α be a trace of **with** r **when** B **do** C. Without loss of generality we can assume that α 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 $\beta_j \in [\![B]\!]_{false}$, $\neg((s,i) \stackrel{acq(r)\,\beta_j\,rel(r)}{\Longrightarrow}_{\Gamma}$ **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 $\Phi \star \Psi \Rightarrow B = B$ this implies that $\mathtt{free}(B) \subseteq \mathtt{dom}(s \star s_1) = \mathtt{dom}(s) \cup \mathtt{dom}(s_1)$. This means that the stack $s \star s_1$ contains permissions for the free variables of B, so that the read actions in β_j do not abort. Further, if β_j is enabled from $s \star s_1$ the subsequent $rel(r)$ action will not abort, because the state after the reads will still be $s \star s_1$ and still contains a sub-state satisfying the invariant Ψ. By precision, the only sub-state with this property is s_1. It follows that for all s', if $(s,i) \stackrel{acq(r)\,\beta_j\,rel(r)}{\Longrightarrow}_{\Gamma,r:\Psi} (s',i)$ then $s = s'$.

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

Now suppose $(s,i) \stackrel{\alpha}{\Longrightarrow}_{\Gamma,r:\Psi} (s',i)$. Again as above we know that we can break this up into the following phases:

$$(s,i) \stackrel{acq(r)\,\beta_1\,rel(r)}{\Longrightarrow}_{\Gamma,r:\Psi} (s,i) \cdots \stackrel{acq(r)\,\beta_n\,rel(r)}{\Longrightarrow}_{\Gamma,r:\Psi} (s,i) \stackrel{acq(r)\,\beta\,\gamma\,rel(r)}{\Longrightarrow}_{\Gamma,r:\Psi} (s',i)$$

and there must be a stack s_{n+1} such that $s \sharp s_{n+1}$, $(s_{n+1},i) \models \Psi$, $(s \star s_{n+1},i) \stackrel{\beta}{\Longrightarrow}_{\Gamma} (s \star s_{n+1},i) \stackrel{\gamma}{\Longrightarrow}_{\Gamma} (s'',i) \stackrel{rel(r)}{\Longrightarrow}_{\Gamma,r:\Psi} (s',i)$. 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 satisfying Φ', as required.

- For the auxiliary variables rule, suppose that Y is an auxiliary set for C such that $\mathtt{free}(\Phi,\Psi,\Gamma) \cap Y = \{\}$, and suppose the formula $\Gamma \vdash_{vr} \{\Phi \star \mathtt{Own}_\top(Y)\} C \{\Psi \star \mathtt{Own}_\top(Y)\}$ is valid. Since Y is auxiliary for C, $[\![C \backslash Y]\!] = \{\alpha \backslash Y \mid \alpha \in [\![C]\!]\}$, where $\alpha \backslash Y$ is the trace obtained from α by replacing each read or write to an auxiliary variable by δ. Let $\sigma \models \Phi$ and, without loss of generality, $\mathtt{dom}(\sigma) \cap Y = \{\}$. Suppose $\alpha \in [\![c]\!]$ and $\sigma \stackrel{\alpha \backslash Y}{\Longrightarrow}_{\Gamma} \sigma'$. We must show that $\sigma' \neq \mathbf{abort}$, and $\sigma' \models \Psi$. Since $\mathtt{dom}(\sigma) \cap Y = \{\}$ we can choose a maximally permissive state σ_1 such that $\sigma \sharp \sigma_1$ and $\mathtt{dom}(\sigma_1) = Y$. Thus $\sigma \star \sigma_1 \models \Phi \star \mathtt{Own}_\top(Y)$. By validity of the premiss, $\sigma \star \sigma_1 \stackrel{\alpha}{\Longrightarrow}_{\Gamma} \sigma''$ for some state $\sigma'' \neq \mathbf{abort}$ such that $\sigma'' \models \Psi \star \mathtt{Own}_\top(\Gamma)$. Since Y is auxiliary for C, σ'' must be expressible (uniquely) in the form $\sigma'' = \sigma' \star \sigma'_1$, for some maximally permissive σ'_1 such that $\sigma' \sharp \sigma'_1$ and $\mathtt{dom}(\sigma'_1) = Y$. (Here we have $\sigma \stackrel{\alpha \backslash Y}{\Longrightarrow}_{\Gamma} \sigma'$ and $\sigma_1 \stackrel{\alpha \lceil Y}{\Longrightarrow}_{\Gamma} \sigma'_1$.) Hence $\sigma' \neq \mathbf{abort}$ and $\sigma' \models \Psi$, as required.

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 – also based on trace semantics – in which resource invariants play no defining rôle. We have already introduced the relevant notion: we refer to the "global" enabling relations $\stackrel{\lambda}{\Longrightarrow}$ on stacks.

Lemma 14 (Local/Global Property for Actions)
Let Γ 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 $(s, A) \stackrel{\lambda}{\Longrightarrow}$ **abort** then $((s_1, i), A) \stackrel{\lambda}{\Longrightarrow}_{\Gamma}$ **abort**.*
- *If $(s, A) \stackrel{\lambda}{\Longrightarrow} (s', A')$ then either $((s_1, i), A) \stackrel{\lambda}{\Longrightarrow}_{\Gamma}$ **abort**, or there exist stacks 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 $(s_1, A) \stackrel{\lambda}{\Longrightarrow}_{\Gamma} (s'_1, A')$.*

Corollary 15 (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 $s \stackrel{\alpha}{\Longrightarrow}$ **abort** then $(s_1, i) \stackrel{\alpha}{\Longrightarrow}_{\Gamma}$ **abort**.*
*If $s \stackrel{\alpha}{\Longrightarrow} s'$ then either $(s_1, i) \stackrel{\alpha}{\Longrightarrow}_{\Gamma}$ **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 $(s_1, i) \stackrel{\alpha}{\Longrightarrow}_{\Gamma} (s'_1, i)$.*

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

Theorem 16 (Provability implies no race)
*Let $\Gamma \vdash_{vr} \{\Phi\} C \{\Psi\}$ be a valid formula. For all σ such that $\sigma \models \Phi \star \mathtt{inv}(\Gamma)$, all traces $\alpha \in [\![C]\!]$, and all σ', if $\sigma \stackrel{\alpha}{\Longrightarrow} \sigma'$ then $\sigma' \neq$ **abort** and $\sigma' \models \Psi \star \mathtt{inv}(\Gamma)$.*

11 Examples

1. Let Ψ 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 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:

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

   Now let $\mathtt{GET}(y_1, y_2)$ be **with** *buf* **when** $\mathit{full} = 1$ **do** $(\mathit{full}{:=}0; (y_1{:=}z \| y_2{:=}z))$, where y_1 and y_2 are distinct program variables.

2. Concurrent reads

   Let p_1, p_2 be permission values such that $p_1 \otimes p_2$ is defined.

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

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

   $\vdash_{vr} \{\mathtt{Own}_\top(x) \star \mathtt{Own}_\top(y) \star \mathtt{Own}_{p_1 \otimes p_2}(z)\} x{:=}z \{\mathtt{Own}_\top(x) \star \mathtt{Own}_\top(y) \star \mathtt{Own}_{p_1 \otimes p_2}(z)\}$

3. 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, is valid; it asserts a triviality, since the pre-condition is not satisfiable.

4. 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 Γ 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

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

   The rule for resources then gives

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

We can then derive

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

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

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

12 Acknowledgements

Thanks to Cristiano Calcagno for introducing me to the BCP paper and answering many questions about the logic. Thanks to Matthew Parkinson for helpful suggestions and clarifications, and to Richard Bornat for his unique style of encouragement.

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 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.

[5] 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.

[6] 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.

[7] P.W. O'Hearn, H. Yang, and J.C. Reynolds. *Separation and Information Hiding*. Proc. 31^{st} POPL conference, pp 268-280, Venice. ACM Press, January 2004.

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

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

[10] M. Parkinson, R. Bornat, and C. Calcagno. *Variables as Resource in Hoare Logics*. Draft paper. Submitted, February 2006.

[11] 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.