Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 276 (2011) 5–28

www.elsevier.com/locate/entcs

# A Revisionist History of Concurrent Separation Logic

Stephen Brookes

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

**Abstract**

Concurrent Separation Logic is a resource-sensitive logic for fault-free partial correctness of concurrent programs with shared mutable state, combining separation logic with Owicki-Gries inference rules, in a manner proposed by Peter O'Hearn. The Owicki-Gries rules and O'Hearn's original logic lacked compositionality, being limited to programs with a rigid parallel structure, because of a crucial constraint that "no other process modifies" certain variables, imposed as a side condition in the inference rule for conditional critical regions. In prior work we proposed a more general formulation of a concurrent separation logic using resource contexts, and we offered a soundness proof based on a trace semantics. Recently Ian Wehrman and Josh Berdine discovered an example showing that this soundness proof relies on a hidden assumption, tantamount to "no concurrent modification", so that the proposed logic also suffices only for rigid programs. Here we show that, with a natural and simple adjustment we can avoid this problem. The key idea is to augment each assertion with a "rely set" of variables, assumed to be unmodified by other processes, and adjust the inference rules to validate and take advantage of these assumptions. This revised concurrent separation logic is compositional, allowing rigid and non-rigid programs, and the extra constraints imposed by rely set requirements ensure soundness. At the same time, we relax the Owicki-Gries constraints on the use of critical variables, allowing variables to be protected by multiple resources and building into the logic a simpler, yet more general, protection discipline. In the revised logic, a process wanting to write to a shared variable must acquire all resources that protect it, while a process wishing to read a shared variable need only acquire one such resource. This generalization brings concurrent separation logic closer in spirit to permission-based logics, in which processes may be allowed to perform concurrent reads.

*Keywords:* concurrency, shared memory, denotational semantics, resources, separation logic

## 1 Introduction

Concurrent Separation Logic (CSL) is a resource-sensitive logic for reasoning about fault-free partial correctness of shared-memory concurrent programs. CSL combines separation logic, originally introduced in [10] by John Reynolds for reasoning about sequential pointer programs, with Owicki-Gries rules for pointer-free shared-memory programs [7], in a manner proposed by Peter O'Hearn [6]. The Owicki-Gries and O'Hearn logics lack compositionality, being limited to programs with rigid parallel structure, because of a static constraint that "no other process modifies" certain variables, imposed as a side condition in the rule for conditional critical regions.


doi:10.1016/j.entcs.2011.09.013

In prior work we formulated a more general concurrent separation logic [3] using resource contexts in an attempt to avoid these limitations, and we gave a soundness proof, using on a trace-based denotational semantics. A major feature in this development was a semantic formalization of O'Hearn's notion of "ownership transfer" based on resource invariants, and O'Hearn's principle that processes "mind their own business" [6]. Recently Ian Wehrman and Josh Berdine found a counterexample [12] showing that this soundness proof makes a hidden assumption, tantamount to "no other process modifies", leading to the realization that the soundness analysis of [3] only suffices for rigid programs.

We show here that, with a systematic natural adjustment to the prior formulation, we can develop a fully compositional concurrent separation logic that avoids this problem. The key idea is to augment the assertions of CSL with a "rely set", representing a set of variables assumed to be left unmodified by the "environment". By making this set an integral part of assertions, we avoid the need for a non-compositional side condition; we are able to properly account for the assumptions and guarantees that a process makes about modifications to shared variables, in a purely syntax-directed manner.

At the same time, we relax the Owicki-Gries constraints on the use of critical variables, allowing variables to be protected by multiple resources and building into the logic a simpler, more general, protection discipline. This brings concurrent separation logic closer in spirit to permission-based logics, in which processes may be allowed to perform concurrent reads [1,2].

Again using action trace semantics, we sketch a soundness proof for the revised logic, this time without the hidden assumption and without requiring rigid program structure. We offer a series of examples, addressing Wehrman's problem, and showing that the augmented logic can deal with a wider variety of programs than the original, because of our relaxation of the Owicki-Gries constraints. We intend the revised and augmented logic presented here to replace the original.

We assume familiarity with separation logic, as defined by Reynolds [10].

## 2 Syntax

The syntax of our programming language (as in [3]) is given by the following abstract grammar, in which $c$ ranges over the set **Com** of commands.

$$\begin{aligned} c ::= {} & \mathbf{skip} \mid i{:=}e \mid i{:=}[e] \mid [e]{:=}e' \mid i{:=}\mathbf{cons}\, E \mid \mathbf{dispose}\, e \\ \mid {} & c_1; c_2 \mid \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \mid \mathbf{while}\ b\ \mathbf{do}\ c \\ \mid {} & \mathbf{with}\ r\ \mathbf{when}\ b\ \mathbf{do}\ c \mid c_1 \| c_2 \end{aligned}$$

Let $e, b$ range over integer expressions and boolean expressions, respectively, and $E$ range over list expressions of form $[e_1, \ldots, e_n]$. Expressions are *pure*, i.e. independent of the heap.

We distinguish syntactically between *identifiers* ($i \in \mathbf{Ide}$) denoting integer vari-

ables, and *resource names* ($r \in \mathbf{Res}$), which behave like binary semaphores, to be represented semantically as integer variables whose value is constrained to be 0 or 1.

Let $\mathit{free}(c) \subseteq \mathbf{Ide}$ be the set of identifiers with a free occurrence in $c$, $\mathit{mod}(c)$ be the set of identifiers with a free write occurrence, and $\mathit{res}(c) \subseteq \mathbf{Res}$ be the set of resource names with a free occurrence in $c$. These are defined as usual, by structural induction. For example,

$$\mathit{free}(i{:=}e) = \mathit{free}(e) \cup \{i\}$$

$$\mathit{free}(c_1; c_2) = \mathit{free}(c_1 \| c_2) = \mathit{free}(c_1) \cup \mathit{free}(c_2)$$

$$\mathit{free}(\mathbf{with}\ r\ \mathbf{when}\ b\ \mathbf{do}\ c) = \mathit{free}(b) \cup \mathit{free}(c)$$

$$\mathit{free}(\mathbf{resource}\ r\ \mathbf{in}\ c) = \mathit{free}(c)$$

$$\mathit{res}(i{:=}e) = \{\}$$

$$\mathit{res}(c_1; c_2) = \mathit{res}(c_1 \| c_2) = \mathit{res}(c_1) \cup \mathit{res}(c_2)$$

$$\mathit{res}(\mathbf{with}\ r\ \mathbf{when}\ b\ \mathbf{do}\ c) = \mathit{res}(c) \cup \{r\}$$

$$\mathit{res}(\mathbf{resource}\ r\ \mathbf{in}\ c) = \mathit{res}(c) - \{r\}$$

$$\mathit{mod}(i{:=}e) = \mathit{mod}(i{:=}\mathbf{cons}\,E) = \mathit{mod}(i{:=}[e]) = \{i\}$$

$$\mathit{mod}([e]{:=}e') = \mathit{mod}(\mathbf{dispose}\ e) = \{\}$$

$$\mathit{mod}(c_1; c_2) = \mathit{mod}(c_1 \| c_2) = \mathit{mod}(c_1) \cup \mathit{mod}(c_2)$$

$$\mathit{mod}(\mathbf{with}\ r\ \mathbf{when}\ b\ \mathbf{do}\ c) = \mathit{mod}(c)$$

$$\mathit{mod}(\mathbf{resource}\ r\ \mathbf{in}\ c) = \mathit{mod}(c)$$

## 3 Assertions

As in Owicki-Gries [7], we associate to each resource name $r$ a set $X \subseteq \mathbf{Ide}$ of "protected variables" and a "resource invariant" $R$ [5]. As in O'Hearn's CSL [6], pre- and post-conditions, and resource invariants, are separation logic formulas. As in [3], instead of assuming a fixed choice of resource invariants and protection sets, we extend the syntax of partial correctness assertions to include a *resource context* $\Gamma$. We relax the permission rules from Owicki-Gries, O'Hearn, and the original concurrent separation logic [3], by not insisting that protection sets be pairwise disjoint. As in [3] we require resource invariants to be *precise* [10]. A separation logic formula $R$ is precise iff, for all stores $s$ and heaps $h$, there is at most one sub-heap $h' \subseteq h$ such that $(s, h') \models R$.

**Definition 3.1** A *well-formed* resource context $\Gamma$ has the form

$$r_1(X_1) : R_1, \ldots, r_n(X_n) : R_n$$

where $r_1, \ldots, r_n$ are distinct resource names, $X_1, \ldots, X_n$ are sets of identifiers, each $R_i$ is precise, and $\mathit{free}(R_i) \subseteq X_i$ for each $i$.

We say that $r$ *protects* $x$ in $\Gamma$ when $r(X) : R$ is in $\Gamma$ and $x \in X$. Let $\mathit{owned}(\Gamma) = \bigcup_{i=1}^{n} X_i$, and $\mathit{inv}(\Gamma) = R_1 * \cdots * R_n$. Let $\mathit{dom}(\Gamma)$ be $\{r_1, \ldots, r_n\}$. We let $(\Gamma, r(X) : R)$ be the context formed by augmenting $\Gamma$ with $r(X) : R$, provided this is well-formed.

**Definition 3.2** An *assertion* has form

$$\Gamma \vdash_A \{p\}c\{q\}$$

where $A$ is a set of identifiers, which we will call a *rely set*. The pre- and post-conditions $p$ and $q$, and the resource invariants in $\Gamma$, do not mention resource names. Such an assertion is *well-formed* iff $\Gamma$ is a well-formed context, $\mathit{free}(p, q) \subseteq A$, and $\mathit{free}(c) \subseteq \mathit{owned}(\Gamma) \cup A$.

In a well-formed assertion $\Gamma \vdash_A \{p\}c\{q\}$ the pre- and post-conditions may mention identifiers owned by resources in $\Gamma$, but only if they also belong to the rely set $A$; the command $c$ may use variables owned by resources or belonging to the rely set. The inference rules (to be introduced shortly) will constrain how and where $c$ is allowed to read and write these variables: in particular, $c$ can only write to a variable protected by $r$ inside a critical region that names $r$; and reads of a protected variable must be inside a critical section, unless the variable belongs to the rely set. The rules keep track (in the rely sets) of the variables used (outside of critical regions) in a proof: these must not be modified by any other process, and this constraint is enforced as a side condition of the parallel rule. Our revised logic actually enforces the following protection regime: every write in $c$ to a protected variable must be inside (nested) critical regions naming *all* resources that protect it; and every read occurrence in $c$ of a protected variable must be inside a critical region naming *some* resource that protects it. In the special case where the protection sets are pairwise disjoint, this coincides with the usual Owicki-Gries discipline: reads or writes to a protected variable must be inside a critical region naming the (unique) protecting resource.

# 4 Validity

Even before we introduce a semantic model for the programming language we can provide an intuitive characterization of what an assertion is intended to say about program behavior. An assertion expresses a "guarantee" on program behavior in suitably constrained environments, i.e. when executed concurrently with other processes whose behavior is assumed to behave as specified.

**Definition 4.1** The assertion $\Gamma \vdash_A \{p\}c\{q\}$ is valid iff every finite interactive computation of $c$, from a state (with values for all variables in $\Gamma, A$) satisfying $p * \mathit{inv}(\Gamma)$, in an environment that respects $\Gamma$ and does not modify the variables in $A$, is fault-free, respects $\Gamma$, and ends in a state satisfying $q * \mathit{inv}(\Gamma)$.

Respect for $\Gamma$ means obeying the protection regime implied by $\Gamma$ and preservation of each resource invariant (separately). Fault-freedom means no runtime errors such as dangling pointers, and no race conditions involving concurrent writes to shared variables or heap.

This notion of validity will be made formal later, by means of action trace semantics and "local enabling" relations $\xrightarrow[\Gamma,A]{\lambda}$ defined as before [3] except that we only allow "environment" moves that do not modify the variables in the rely set $A$.

Validity of $\Gamma \vdash_A \{p\}c\{q\}$ implies that when $c$ is executed *in isolation* from a state satisfying $p * inv(\Gamma)$, the execution is fault-free, and if it terminates the final state satisfies $q * inv(\Gamma)$. This is because the empty environment vacuously respects $\Gamma$ and does not modify any variable. Hence the usual slogan that provable programs are safe.

## 5 Inference rules

The inference rules for our revised logic are obtained from the original CSL [3] by adding rely sets, and relaxing the constraints on use of protected variables. We only allow well-formed instances of the rules, so every provable formula will be well-formed, as specified above.

- Skip

$$\frac{}{\Gamma \vdash_A \{p\}\mathbf{skip}\{p\}} \qquad \text{if } free(p) \subseteq A$$

- Assignment

$$\frac{}{\Gamma \vdash_A \{[e/i]p\}i{:=}e\{p\}} \qquad \text{if } i \notin owned(\Gamma),\ free(e) \subseteq A$$

- Sequence

$$\frac{\Gamma \vdash_{A_1} \{p_1\}c_1\{p_2\} \quad \Gamma \vdash_{A_2} \{p_2\}c_2\{p_3\}}{\Gamma \vdash_{A_1 \cup A_2} \{p_1\}c_1; c_2\{p_3\}}$$

- Conditional

$$\frac{\Gamma \vdash_{A_1} \{p \wedge b\}c_1\{q\} \quad \Gamma \vdash_{A_2} \{p \wedge \neg b\}c_2\{q\}}{\Gamma \vdash_{A_1 \cup A_2} \{p\}\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\{q\}}$$

- Loop

$$\frac{\Gamma \vdash_A \{p \wedge b\}c\{p\}}{\Gamma \vdash_A \{p\}\mathbf{while}\ b\ \mathbf{do}\ c\{p \wedge \neg b\}}$$

- Parallel

$$\frac{\Gamma \vdash_{A_1} \{p_1\}c_1\{q_1\} \quad \Gamma \vdash_{A_2} \{p_2\}c_2\{q_2\}}{\Gamma \vdash_{A_1 \cup A_2} \{p_1 * p_2\}c_1 \| c_2\{q_1 * q_2\}} \qquad \text{if } mod(c_1) \cap A_2 = mod(c_2) \cap A_1 = \{\}$$

- Critical region

$$\frac{\Gamma \vdash_{A \cup X} \{(p \wedge b) * R\}c\{q * R\}}{\Gamma, r(X) : R \vdash_A \{p\}\mathbf{with}\ r\ \mathbf{when}\ b\ \mathbf{do}\ c\{q\}}$$

- Local resource

$$\frac{\Gamma, r(X) : R \vdash_A \{p\}c\{q\}}{\Gamma \vdash_{A \cup X} \{p * R\}\mathbf{resource}\ r\ \mathbf{in}\ c\{q * R\}}$$

- Renaming

$$\frac{\Gamma \vdash_A \{p\}\mathbf{resource}\ r'\ \mathbf{in}\ [r'/r]c\{q\}}{\Gamma \vdash_A \{p\}\mathbf{resource}\ r\ \mathbf{in}\ c\{q\}} \qquad \text{if } r' \notin res(c)$$

- Lookup

$$\frac{}{\Gamma \vdash_A \{[e'/i]p \wedge e \mapsto e'\}i{:=}[e]\{p \wedge e \mapsto e'\}} \qquad \text{if } i \notin free(e, e'),\ i \notin owned(\Gamma)$$

- Update

$$\frac{}{\Gamma \vdash_A \{e \mapsto -\}[e]{:=}e'\{e \mapsto e'\}}$$

- Allocation

$$\frac{}{\Gamma \vdash_A \{\mathbf{emp}\}i{:=}\mathbf{cons}(E)\{i \mapsto E\}} \qquad \text{if } i \notin free(E),\ i \notin owned(\Gamma)$$

- Disposal

$$\frac{}{\Gamma \vdash_A \{e \mapsto -\}\mathbf{dispose}\, e\{\mathbf{emp}\}}$$

- Frame

$$\frac{\Gamma \vdash_A \{p\}c\{q\}}{\Gamma \vdash_{A \cup free(R)} \{p * R\}c\{q * R\}} \qquad \text{if } mod(c) \cap free(R) = \{\}$$

- Consequence

$$\frac{\Gamma \vdash_A \{p\}c\{q\}}{\Gamma \vdash_{A'} \{p'\}c\{q'\}} \qquad \text{if } A \subseteq A',\ p' \Rightarrow p,\ q \Rightarrow q'$$

- Auxiliary

$$\frac{\Gamma \vdash_{A \cup X} \{p\}c\{q\}}{\Gamma \vdash_A \{p\}c \backslash X\{q\}}$$

if $X$ is auxiliary for $c$, $X \cap free(p, q) = \{\}$, $X \cap owned(\Gamma) = \{\}$

- Conjunction

$$\frac{\Gamma \vdash_{A_1} \{p_1\}c_1\{q_1\} \quad \Gamma \vdash_{A_2} \{p_2\}c_2\{q_2\}}{\Gamma_{A_1 \cup A_2}\{p_1 \wedge p_2\}c\{q_1 \wedge q_2\}}$$

## Commentary

We offer some intuition and explanation for the side conditions, and the (mostly implicit) rôle played by the well-formedness requirements.

- Skip: the side condition that $free(p) \subseteq A$ ensures well-formedness.

- Assignment: allowed to read identifiers belonging to the rely set $A$; only allowed to write to $i$ if $i \notin \mathit{owned}(\Gamma)$. Well-formedness implies $i \in A$.
- Sequence: rely set $A_1 \cup A_2$ includes the free variables of the intermediate condition $p_2$, not just those of the pre- and post-condition.
- Conditional, Loop: well-formedness implies that $\mathit{free}(b) \subseteq A$.
- Parallel: $c_1$ relies on its environment to modify variables in $A_1$, hence the side condition $\mathit{mod}(c_2) \cap A_1 = \{\}$; similarly for $c_2$ and $\mathit{mod}(c_1) \cap A_2 = \{\}$. Suppose $\Gamma \vdash_{A_1} \{p_1\}c_1\{q_1\}$ and $\Gamma \vdash_{A_2} \{p_2\}c_2\{q_2\}$ are well-formed, and the side conditions hold. Then $\mathit{free}(p_1, q_1) \subseteq A_1$ and $\mathit{free}(p_2, q_2) \subseteq A_2$, so $\mathit{mod}(c_1) \cap \mathit{free}(p_2, q_2) = \{\}$ and $\mathit{mod}(c_2) \cap \mathit{free}(p_1, q_1) = \{\}$, as in Owicki-Gries. We also have $\mathit{free}(c_1) \subseteq \mathit{owned}(\Gamma) \cup A_1$ and $\mathit{free}(c_2) \subseteq \mathit{owned}(\Gamma) \cup A_2$, so "critical variables are protected", i.e. $\mathit{mod}(c_1) \cap \mathit{free}(c_2) \subseteq \mathit{owned}(\Gamma)$ and $\mathit{mod}(c_2) \cap \mathit{free}(c_1) \subseteq \mathit{owned}(\Gamma)$, as in Owicki-Gries. The rely set $A_1$ includes the variables used (outside of critical regions) in the *proof* for $c_1$, and similarly for $A_2$ and $c_2$, so we enforce *here* the "no concurrent modification" requirement on the relevant variables. It is natural to do so here, since after all this *is* the inference rule for concurrent processes.
- Critical region: The premiss relies on $A \cup X$ because mutual exclusion for $r$ implies that no concurrent process can touch the variables in $X$. In the conclusion there is no need to include the protected variables in the rely set, although this is allowed. If the premiss is well-formed, so is the conclusion.
- Local resource: The conclusion relies on $A \cup X$, which ensures well-formedness because $\mathit{free}(R) \subseteq X$ by well-formedness of the premiss.
- Update, Lookup, Allocation, Disposal: axioms as before, with side conditions to ensure well-formedness.
- Frame: as before, $c$ must not write to any variable occurring free in $R$. There is no need to insist that $\mathit{free}(R) \cap \mathit{owned}(\Gamma) = \{\}$ (as in the original CSL formulation), because pre- and post-conditions are allowed to mention protected variables; instead we add the variables occurring free in $R$ to the rely set, reflecting the assumption that no concurrent processes modify these variables.
- Consequence: as usual, except that we also allow strengthening of the rely set. If $\Gamma \vdash_A \{p\}c\{q\}$ is valid and $A \subseteq A'$, then $\Gamma \vdash_{A'} \{p\}c\{q\}$ is also valid, since it expresses a less general semantic property of $c$, quantifying over a more constrained set of possible environments.
- Auxiliary: a set $X$ of variables is auxiliary for $c$ iff each free occurrence in $c$ of a variable from $X$ is in an assignment whose target identifier also belongs to $X$. The requirement that auxiliary variables do not occur free in the pre- or post-condition is standard; that they do not appear in protection lists is also crucial.
- Conjunction: as noted by John Reynolds, precision of resource invariants is crucial in showing the soundness of this rule. When $R$ is precise, for all $p_1$ and $p_2$ we have $(p_1 \wedge p_2) * R \Leftrightarrow (p_1 * R) \wedge (p_2 * R)$.

# 6 Examples

We now discuss some example programs and assertions, to illustrate the way the inference rules work and to contrast the new CSL with the original one. Examples (i) and (ii) address the issues raised by Wehrman and Berdine.

We adopt the convention that when $free(R) = X$, we may omit $X$ from $r(X) : R$ and just write $r : R$, since this leads to more succinct assertions.

(i) The assertions

$$(a) \quad r : x = a \wedge \mathbf{emp} \vdash_{\{a,t\}} \{\mathbf{emp}\}\mathbf{with}\ r\ \mathbf{do}\ t{:=}x\{t = a \wedge \mathbf{emp}\}$$

$$(b) \quad r : x = a \wedge \mathbf{emp} \vdash_{\{a,t\}} \{t = a \wedge \mathbf{emp}\}\mathbf{with}\ r\ \mathbf{do}\ x{:=}t\{\mathbf{emp}\}$$

are valid and well-formed. Each is also provable from Region, having first proven

$$\vdash_{\{a,t,x\}} \{x = a \wedge \mathbf{emp}\}t{:=}x\{t = a \wedge \mathbf{emp}\},$$

$$\vdash_{\{a,t,x\}} \{t = a \wedge x = a \wedge \mathbf{emp}\}x{:=}t\{x = a \wedge \mathbf{emp}\}$$

by Assignment and Consequence. However, the assertions

$$(a') \quad r : x = a \wedge \mathbf{emp} \vdash_{\{t\}} \{\mathbf{emp}\}\mathbf{with}\ r\ \mathbf{do}\ t{:=}x\{t = a \wedge \mathbf{emp}\}$$

$$(b') \quad r : x = a \wedge \mathbf{emp} \vdash_{\{t\}} \{t = a \wedge \mathbf{emp}\}\mathbf{with}\ r\ \mathbf{do}\ x{:=}t\{\mathbf{emp}\}$$

are invalid (and not well-formed), and not provable.

Let $c_1$ be $\mathbf{with}\ r\ \mathbf{do}\ t{:=}x; \mathbf{with}\ r\ \mathbf{do}\ x{:=}t$. The assertion

$$(c) \quad r : x = a \wedge \mathbf{emp} \vdash_{\{a,t\}} \{\mathbf{emp}\}c_1\{\mathbf{emp}\}$$

is valid, well-formed, and provable from $(a)$ and $(b)$ using Sequence. But the assertion

$$(c') \quad r : x = a \wedge \mathbf{emp} \vdash_{\{t\}} \{\mathbf{emp}\}c_1\{\mathbf{emp}\}$$

is invalid (even though well-formed), and unprovable.

(ii) Let $c_1$ be as above and let $c_2$ be $\mathbf{with}\ r\ \mathbf{do}\ (x{:=}x + 1; a{:=}a + 1)$. There is no set $A$ of identifiers for which the assertion

$$r : x = a \wedge \mathbf{emp} \vdash_A \{\mathbf{emp}\}c_1 \| c_2\{\mathbf{emp}\}$$

is valid. Indeed, even for the most restrictive rely set $A = \{x, a, t\}$ the assertion is invalid: executing $c_1 \| c_2$ without interference does not necessarily preserve equality of $x$ and $a$.

Moreover, there is also no set $A$ for which this assertion is provable, because $A$ would need to be expressible as $A_1 \cup A_2$ with both

$$r : x = a \wedge \mathbf{emp} \vdash_{A_1} \{\mathbf{emp}\}c_1\{\mathbf{emp}\}$$

and

$$r : x = a \wedge \mathbf{emp} \vdash_{A_2} \{\mathbf{emp}\} c_2 \{\mathbf{emp}\}$$

being provable. The first of these would need an intermediate condition that mentions ($t$ and) either $x$ or $a$, so $A_1$ would have to contain $x$ or $a$. But $c_2$ modifies both of these variables, so the side condition on the parallel rule would fail.

This example, without the rely set, was found by Wehrman and Berdine, who showed that the assertion

$$r : x = a \wedge \mathbf{emp} \vdash \{\mathbf{emp}\} c_1 \| c_2 \{\mathbf{emp}\}$$

is provable in the original concurrent separation logic but not valid (with respect to the notion of validity used in CSL). The analysis above shows that the use of rely sets avoids this problem.

(iii) The assertion

$$\begin{aligned} r : x = a + b \wedge \mathbf{emp} \vdash_{\{a\}} \; & \{a = 0 \wedge \mathbf{emp}\} \\ & \mathbf{with}\ r\ \mathbf{do}\ (x{:=}x + 1; a{:=}a + 1) \\ & \{a = 1 \wedge \mathbf{emp}\} \end{aligned}$$

is valid, and provable from Region and Consequence, because

$$\begin{aligned} \vdash_{\{x,a,b\}} \; & \{x = a + b \wedge a = 0 \wedge \mathbf{emp}\} \\ & x{:=}x + 1; a{:=}a + 1 \\ & \{x = a + b \wedge a = 1 \wedge \mathbf{emp}\} \end{aligned}$$

is provable from Sequence, Assignment, and Consequence.

Similarly we can prove

$$\begin{aligned} r : x = a + b \wedge \mathbf{emp} \vdash_{\{b\}} \; & \{b = 0 \wedge \mathbf{emp}\} \\ & \mathbf{with}\ r\ \mathbf{do}\ (x{:=}x + 1; b{:=}b + 1) \\ & \{b = 1 \wedge \mathbf{emp}\}. \end{aligned}$$

Using Parallel and Consequence we can then derive

$$\begin{aligned} r : x = a + b \wedge \mathbf{emp} \vdash_{\{a,b\}} \; & \{a = 0 \wedge b = 0 \wedge \mathbf{emp}\} \\ & \quad \mathbf{with}\ r\ \mathbf{do}\ (x{:=}x + 1; a{:=}a + 1) \\ & \| \ \mathbf{with}\ r\ \mathbf{do}\ (x{:=}x + 1; b{:=}b + 1) \\ & \{a = 1 \wedge b = 1 \wedge \mathbf{emp}\}. \end{aligned}$$

This assertion is also valid.

Using the RESOURCE rule and CONSEQUENCE we then obtain

$$\begin{array}{ll}\vdash_{\{a,b,x\}} & \{a = 0 \wedge b = 0 \wedge x = a + b \wedge \mathbf{emp}\} \\ & \mathbf{resource}\ r\ \mathbf{in} \\ & \quad \mathbf{with}\ r\ \mathbf{do}\ (x{:=}x + 1; a{:=}a + 1) \| \mathbf{with}\ r\ \mathbf{do}\ (x{:=}x + 1; b{:=}b + 1) \\ & \{a = 1 \wedge b = 1 \wedge x = a + b \wedge \mathbf{emp}\}.\end{array}$$

By SEQUENCE, ASSIGNMENT, and CONSEQUENCE we then have

$$\begin{array}{ll}\vdash_{\{a,b,x\}} & \{x = 0 \wedge \mathbf{emp}\} \\ & a{:=}0; b{:=}0; \\ & \mathbf{resource}\ r\ \mathbf{in} \\ & \quad \mathbf{with}\ r\ \mathbf{do}\ (x{:=}x + 1; a{:=}a + 1) \| \mathbf{with}\ r\ \mathbf{do}\ (x{:=}x + 1; b{:=}b + 1) \\ & \{x = 2 \wedge \mathbf{emp}\}.\end{array}$$

Finally, since $\{a, b\}$ is an auxiliary variable set for this program, and $a, b$ do not occur in the pre- or post-condition, we can use the AUXILIARY rule to obtain

$$\begin{array}{ll}\vdash_{\{x\}} & \{x = 0 \wedge \mathbf{emp}\} \\ & \mathbf{resource}\ r\ \mathbf{in} \\ & \quad \mathbf{with}\ r\ \mathbf{do}\ x{:=}x + 1 \| \mathbf{with}\ r\ \mathbf{do}\ x{:=}x + 1 \\ & \{x = 2 \wedge \mathbf{emp}\}.\end{array}$$

(iv) We revisit O'Hearn's one-place buffer program [6,3]. The example goes through almost unchanged, except for the insertion of rely sets. Let $R$ be $(\mathit{full} = 1 \wedge z \mapsto -) \vee (\mathit{full} = 0 \wedge \mathbf{emp})$. Let $PUT$ and $GET$ be the commands

$$\begin{array}{lcl} PUT & :: & \mathbf{with}\ \mathit{buf}\ \mathbf{when}\ \mathit{full} = 0\ \mathbf{do}\ (z{:=}x; \mathit{full}{:=}1) \\ GET & :: & \mathbf{with}\ \mathit{buf}\ \mathbf{when}\ \mathit{full} = 1\ \mathbf{do}\ (y{:=}z; \mathit{full}{:=}0). \end{array}$$

The assertions

$$\begin{array}{l} \mathit{buf}(z, \mathit{full}) : R \vdash_{\{x\}} \{x \mapsto -\} PUT \{\mathbf{emp}\} \\ \mathit{buf}(z, \mathit{full}) : R \vdash_{\{y\}} \{\mathbf{emp}\} GET \{y \mapsto -\} \end{array}$$

are valid and provable. Similarly,

$$\mathit{buf}(z, \mathit{full}) : R \vdash_{\{y\}} \{\mathbf{emp}\}(GET; \mathbf{dispose}\, y)\{\mathbf{emp}\}$$

is provable, and so is

$$buf(z, full) : R \vdash_{\{x,y\}} \{x \mapsto -\}\, PUT \| (GET;\, \mathbf{dispose}\, y)\{\mathbf{emp}\}.$$

Now let $R'$ be $(full = 1 \wedge z \mapsto -) \vee (full = 0 \wedge \mathbf{emp})$. The assertions

$$buf(z, full) : R' \vdash_{\{x\}} \{x \mapsto -\} PUT \{x \mapsto -\}$$
$$buf(z, full) : R' \vdash_{\{y\}} \{\mathbf{emp}\} GET \{\mathbf{emp}\}$$

are valid and provable. Similarly

$$buf(z, full) : R' \vdash_{\{x\}} \{x \mapsto -\}(PUT;\, \mathbf{dispose}\, x)\{\mathbf{emp}\}$$

is provable, and so is

$$buf(z, full) : R' \vdash_{\{x,y\}} \{x \mapsto -\}\, (PUT; \mathbf{dispose}\, x) \| GET \{\mathbf{emp}\}.$$

(v) The assertion

$$r : x = y \wedge \mathbf{emp} \vdash_{\{\}} \{\mathbf{emp}\}\mathbf{with}\ r\ \mathbf{do}\ (x{:=}x + 1; y{:=}y + 1)\{\mathbf{emp}\}$$

is valid. Clearly $\{y\}$ is auxiliary for the command here. The assertion

$$r : x = y \wedge \mathbf{emp} \vdash_{\{\}} \{\mathbf{emp}\}\mathbf{with}\ r\ \mathbf{do}\ (x{:=}x + 1)\{\mathbf{emp}\}$$

is obviously invalid. This shows that the side condition requiring that auxiliary variables must not appear in resource invariants is crucial, as noted previously.

(vi) Let $c_1$ and $c_2$ be:

$$c_1 :: \mathbf{with}\ r_1\ \mathbf{do}\ ((\mathbf{with}\ r_2\ \mathbf{do}\ a{:=}1); [42]{:=}1)$$
$$c_2 :: \mathbf{with}\ r_2\ \mathbf{do}\ ((\mathbf{with}\ r_1\ \mathbf{do}\ a{:=}2); [42]{:=}2).$$

Let $R_1$ and $R_2$ be the assertions

$$R_1 :: (a = 1 \wedge 42 \mapsto 1) \vee (a = 2 \wedge \mathbf{emp})$$
$$R_2 :: (a = 1 \wedge \mathbf{emp}) \vee (a = 2 \wedge 42 \mapsto 2).$$

Note that $R_1 * R_2$ is equivalent to $(a = 1 \wedge 42 \mapsto 1) \vee (a = 2 \wedge 42 \mapsto 2)$. The following assertions are provable:

| | |
|---|---|
| $(a)\ \vdash_{\{a\}} \{R_1 * R_2\}a{:=}1\{(42 \mapsto \_ \wedge a = 1) * R_2\}$ | Assignment, Consequence |
| $(b)\ r_2(a) : R_2 \vdash_{\{a\}} \{R_1\}\mathbf{with}\ r_2\ \mathbf{do}\ a{:=}1\{42 \mapsto \_ \wedge a = 1\}$ | Region, $(a)$ |
| $(c)\ r_2(a) : R_2 \vdash_{\{a\}} \{42 \mapsto \_ \wedge a = 1\}[42]{:=}1\{R_1\}$ | Update, Consequence, $(b)$ |
| $(d)\ r_2(a) : R_2 \vdash_{\{a\}} \{R_1\}(\mathbf{with}\ r_2\ \mathbf{do}\ a{:=}1); [42]{:=}1\{R_1\}$ | Sequence, $(b)$, $(c)$ |
| $(e)\ r_1(a) : R_1, r_2(a) : R_2 \vdash_{\{\}} \{\mathbf{emp}\}c_1\{\mathbf{emp}\}$ | Region, Consequence, $(d)$ |

Similarly we can derive

$$(f) \quad r_1(a) : R_1, r_2(a) : R_2 \vdash_{\{\}} \{\mathbf{emp}\} c_2 \{\mathbf{emp}\}$$

Then, by PARALLEL from $(e), (f)$ we obtain

$$r_1(a) : R_1, r_2(a) : R_2 \vdash_{\{\}} \{\mathbf{emp}\} c_1 \| c_2 \{\mathbf{emp}\}$$

and finally, by RESOURCE (used twice),

$$\vdash_{\{a\}} \{R_1 * R_2\} \mathbf{resource}\ r_1\ \mathbf{in}\ \mathbf{resource}\ r_2\ \mathbf{in}\ (c_1 \| c_2) \{R_1 * R_2\}.$$

Note that this program cannot be proven correct in the original CSL, because it violates the more stringent Owicki-Gries constraints on use of critical variables. Indeed, we made use in the proof of a resource context $r_1(a) : R_1.r_2(a) : R_2$ in which the protection lists are not disjoint. The same example has been discussed as a "problematic program" by John Reynolds [11], and can also be handled cleanly in Uday Reddy's logic [9] by using fractional permissions. We include this example to indicate an advantage of employing weaker restrictions on variable use.

## 7 States, actions and traces

As in [10] a state $\sigma$ is a pair $(s, h)$ consisting of a store $s$ and a heap $h$. The store maps (a finite set of) identifiers to (integer) values; the heap maps (a finite set of) locations to values; locations are also integers. In a given state $dom(s)$ is the set of identifiers currently in scope, and $dom(h)$ is the set of active locations. We regard resource names as identifiers whose value is restricted to be 0 (in use) or 1 (available). Let **St** be the set of all states.

As in [3] the set $\Lambda$ of *actions* (ranged over by $\lambda, \mu$) is specified as follows, where $v, v', v_0 \ldots, v_n$ range over integer values, $i$ ranges over identifiers, and $r$ ranges over resource names:

$$\begin{aligned} \lambda ::= {} & \delta \mid i{=}v \mid i{:=}v \\ \mid {} & [v]{=}v' \mid [v]{:=}v' \mid alloc(v, [v_0, \ldots, v_n]) \mid disp(v) \\ \mid {} & acq(r) \mid rel(r) \mid try(r) \\ \mid {} & abort \end{aligned}$$

We let $mod(\lambda)$ be the set of identifiers whose value is modified by $\lambda$: $mod(i{:=}v) = \{i\}$ and $mod(\lambda) = \{\}$ otherwise. We let $writes(\lambda) \supseteq mod(\lambda)$ be the set of identifiers or heap cells modified by $\lambda$: $writes(i{:=}v) = \{i\}$, $writes([v]{:=}v') = \{v\}$, $writes(alloc(v, [v_0, \ldots, v_n])) = \{v, v+1, \ldots, v+n\}$, $writes(disp(v)) = \{v\}$, $writes(\lambda) = \{\}$ otherwise. Similarly we let $reads(\lambda)$ be the set of identifiers or heap cells whose value is read by $\lambda$: $reads(i{=}v) = \{i\}$, $reads([v]{=}v') = \{v\}$, $reads(\lambda) = \{\}$

$$(s,h) \stackrel{\delta}{\Longrightarrow} (s,h)$$

$$(s,(h) \stackrel{i=v}{\Longrightarrow} (s,h) \qquad \text{if } (i,v) \in s$$
$$(s,h) \stackrel{i=v}{\Longrightarrow} abort \qquad \text{if } i \notin dom(s)$$

$$(s,h) \stackrel{i:=v}{\Longrightarrow} ([s \mid i:v],h) \qquad \text{if } i \in dom(s)$$
$$(s,h) \stackrel{i:=v}{\Longrightarrow} abort \qquad \text{if } i \notin dom(s)$$

$$(s,h) \stackrel{[v]=v'}{\Longrightarrow} (s,h) \qquad \text{if } (v,v') \in h$$
$$(s,h) \stackrel{[v]=v'}{\Longrightarrow} abort \qquad \text{if } v \notin dom(h)$$

$$(s,h) \stackrel{[v]:=v'}{\Longrightarrow} (s,([h \mid v:v']) \qquad \text{if } v \in dom(h)$$
$$(s,h) \stackrel{[v]:=v'}{\Longrightarrow} abort \qquad \text{if } v \notin dom(h)$$

$$(s,h) \stackrel{disp(v)}{\Longrightarrow} (s,h\backslash v) \qquad \text{if } v \in dom(h)$$
$$(s,h) \stackrel{disp(v)}{\Longrightarrow} abort \qquad \text{if } v \notin dom(h)$$

$$(s,h) \stackrel{alloc(v,[v_0,\ldots,v_n])}{\Longrightarrow} (s,h') \qquad \text{if } \{v,\ldots,v+n\} \cap dom(h) = \{\},$$
$$\text{where } h' = [h \mid v:v_0,\ldots,v+n:v_n]$$

$$(s,h) \stackrel{acq\, r}{\Longrightarrow} ([s \mid r:0],h) \qquad \text{if } (r,1) \in s$$

$$(s,h) \stackrel{try\, r}{\Longrightarrow} (s,h) \qquad \text{if } (r,0) \in s$$

$$(s,h) \stackrel{rel\, r}{\Longrightarrow} ([s \mid r:1],h) \qquad \text{if } (r,0) \in s$$

$$(s,h) \stackrel{abort}{\Longrightarrow} abort$$

Fig. 1. Enabling relations

otherwise. And we let $free(\lambda)$ be the set of identifiers or heap cells used in $\lambda$: $free(\lambda) = reads(\lambda) \cup writes(\lambda)$. And we let $res(\lambda)$ be the set of resource names used

in $\lambda$.

Actions have an effect on the state, as specified by the *enabling* relations $\stackrel{\lambda}{\Longrightarrow} \subseteq \mathbf{St} \times (\mathbf{St} \cup \{abort\})$. These relations are given in Figure 1.

Note that read and write actions ($i = v$ and $i{:=}v$) depend only on, and only affect, the store; lookup, update, allocation and disposal actions depend only on, and only affect, the heap; and resource actions depend only on, and only affect the values of resource names (again, in the store). It is also obvious that the enabledness of a store action, and its effect, only depends on the value of the identifier or resource name attached to the action.

## 8 Semantics

We use the same denotational semantic model as before [3]. We summarize the key concepts and technical details. Expressions denote sets of evaluation traces: an evaluation trace has form $(\rho, v)$, where $\rho$ is a finite sequence of read actions (or $\delta$) and $v$ is a value. Expression evaluation always terminates, so $\rho$ ranges over finite traces. Since expressions are pure their traces only involve read actions. We assume given the semantics of expressions: for integer expressions $e$, $[\![e]\!] \subseteq \Lambda^* \times V_{int}$; for boolean expressions $b$, $[\![b]\!] \subseteq \Lambda^* \times \{true, false\}$; and for list expressions $E$, $[\![E]\!] \subseteq \Lambda^* \times V_{int}^*$. For boolean expressions we let $[\![b]\!]_{true} = \{\rho \mid (\rho, true) \in [\![b]\!]\}$, and $[\![b]\!]_{false} = \{\rho \mid (\rho, false) \in [\![b]\!]\}$. We write $V_{int}$ for the set of integers, $V_{bool}$ for the set of truth values, and $V_{int}^*$ for the set of finite sequences of integers.

Commands denote sets of action traces, which may be finite or infinite, and whose structure reflects the mutual exclusion assumption, that resources behave like binary semaphores. Command semantics is defined by structural induction, as follows.

$$[\![\mathbf{skip}]\!] = \{\delta\}$$

$$[\![i{:=}e]\!] = \{\rho\, i{:=}v \mid (\rho, v) \in [\![e]\!]\}$$

$$[\![c_1; c_2]\!] = [\![c_1]\!][\![c_2]\!]$$

$$[\![\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2]\!] = [\![b]\!]_{true}[\![c_1]\!] \cup [\![b]\!]_{false}[\![c_2]\!]$$

$$[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] = ([\![b]\!]_{true}[\![c]\!])^*[\![b]\!]_{false} \cup ([\![b]\!]_{true}[\![c]\!])^\omega$$

$$[\![\mathbf{local}\ i = e\ \mathbf{in}\ c]\!] = \{\rho(\alpha \backslash i) \mid (\rho, v) \in [\![e]\!]\ \&\ \alpha \in [\![c]\!]_{[i:v]}\}$$

$$[\![i{:=}[e]]\!] = \{\rho\,[v]{=}v'\, i{:=}v' \mid (\rho, v) \in [\![e]\!]\ \&\ v' \in V_{int}\}$$

$$[\![[e]{:=}e']\!] = \{\rho\rho'\,[v]{:=}v' \mid (\rho, v) \in [\![e]\!]\ \&\ (\rho', v') \in [\![e']\!]\}$$

$$[\![i{:=}\mathbf{cons}(E)]\!] = \{\rho\, alloc(v, L) \mid (\rho, L) \in [\![E]\!]\ \&\ v \in V_{int}\}$$

$$[\![\mathbf{dispose}(e)]\!] = \{\rho\, disp(v) \mid (\rho, v) \in [\![e]\!]\}$$

$$[\![\textbf{with } r \textbf{ when } b \textbf{ do } c]\!] = wait^* enter \cup wait^\omega$$
$$\text{where } wait = \{try(r)\} \cup \{acq(r)\,\rho\,rel(r) \mid \rho \in [\![b]\!]_{false}\}$$
$$\text{and } \quad enter = \{acq(r)\,\rho\,\alpha\,rel(r) \mid \rho \in [\![b]\!]_{true} \;\&\; \alpha \in [\![c]\!]\}$$
$$[\![\textbf{resource } r \textbf{ in } c]\!] = \{\alpha\backslash r \mid \alpha \in [\![c]\!]_{[r:1]}\}$$
$$[\![c_1 \| c_2]\!] = \bigcup\{\alpha_1 {}_{\{\}}\|_{\{\}} \alpha_2 \mid \alpha_1 \in [\![c_1]\!] \;\&\; \alpha_2 \in [\![c_2]\!]\}$$

The semantic clause for local resources uses enabling relations to characterize traces that are *sequentially executable* under an assumption that a resource name is "local", i.e. assuming that no other process changes its value. We say that $\alpha$ is executable for $r$ iff $\alpha \restriction r$ is enabled from the store $[r : 1]$ (i.e. assuming that $r$ is initially available). We write $[\![c]\!]_{[r:1]}$ for the set of traces of $c$ that are executable from $[r : 1]$. The clause for local variables uses a similar construction.

The semantic clause for parallel composition uses *fair interleaving* [8] with *race detection* and attention to resources, treating a potential race as a catastrophe. If $\alpha_1$ and $\alpha_2$ are traces, and $O_1, O_2$ are disjoint sets of resource names, we define $\alpha_1 {}_{O_1}\|_{O_2} \alpha_2$ to be the set of traces obtainable by interleaving $\alpha_1$ with $\alpha_2$, paying attention to resources, assuming that the process executing $\alpha_1$ starts with resources $O_1$ and the process executing $\alpha_2$ starts with resources $O_2$. When $O_1 \cap O_2 = \{\}$ we define the relation $O_1 \overset{\lambda}{\underset{O_2}{\Longrightarrow}} O_1'$, which holds when a process with resources $O_1$ can execute $\lambda$ in an environment with resources $O_2$, after which the process will have resources $O_1'$. These relations are given by:

$$\begin{aligned}
&O_1 \overset{acq(r)}{\underset{O_2}{\Longrightarrow}} O_1 \cup \{r\} && \text{if } r \notin O_1 \cup O_2\\
&O_1 \overset{rel(r)}{\underset{O_2}{\Longrightarrow}} O_1 - \{r\} && \text{if } r \in O_1\\
&O_1 \overset{try(r)}{\underset{O_2}{\Longrightarrow}} O_1 && \text{if } r \in O_1 \cup O_2\\
&O_1 \overset{\lambda}{\underset{O_2}{\Longrightarrow}} O_1 && \text{if } \lambda \notin \{acq(r), rel(r), try(r) \mid r \in \textbf{Res}\}
\end{aligned}$$

We extend to finite traces in the obvious way, so that $O_1 \overset{\alpha}{\underset{O_2}{\Longrightarrow}} O_2'$ iff a process starting with resources $O_1$ can, in an environment with resources $O_2$, execute the sequence $\alpha$ and afterwards possess resources $O_1'$. Then for finite traces we define the fair merge operators ${}_{O_1}\|_{O_2}$ inductively as usual:

$$\begin{aligned}
\alpha \,{}_{O_1}\|_{O_2}\, \epsilon &= \{\alpha \mid \exists O_1'.\ O_1 \overset{\alpha}{\underset{O_2}{\Longrightarrow}} O_1'\}\\
\epsilon \,{}_{O_1}\|_{O_2}\, \beta &= \{\beta \mid \exists O_2'.\ O_2 \overset{\beta}{\underset{O_1}{\Longrightarrow}} O_2'\}\\
(\lambda\alpha) \,{}_{O_1}\|_{O_2}\, (\mu\beta) &= \{\lambda\gamma \mid O_1 \overset{\lambda}{\underset{O_2}{\Longrightarrow}} O_1' \;\&\; \gamma \in \alpha \,{}_{O_1'}\|_{O_2}\, (\mu\beta)\}\\
&\cup\ \{\mu\gamma \mid O_2 \overset{\mu}{\underset{O_1}{\Longrightarrow}} O_2' \;\&\; \gamma \in (\lambda\alpha) \,{}_{O_1}\|_{O_2'}\, \beta\}\\
&\cup\ \{abort \mid \lambda \bowtie \mu\}
\end{aligned}$$

We define the *conflict* relation $\lambda \bowtie \mu$ to hold iff $writes(\lambda) \cap free(\mu) \neq \{\}$ or $writes(\mu) \cap free(\lambda) \neq \{\}$. Note that conflict thus corresponds to a write of an *identifier* or a *heap cell* that is being used concurrently. This definition extends to infinite traces in the standard way, but since we only need to deal here with finite traces we will omit the details.

The following result shows that commands respect resources.

**Lemma 8.1** *If $\alpha \in [\![c]\!]$ then for each resource name $r$, $\alpha \upharpoonright \{acq\, r, rel\, r\}$ is a prefix of $((acq\, r)(rel\, r))^\omega$.*

The following property of finite (non-aborting) traces, is also useful.

**Lemma 8.2** *If $\alpha \in [\![c]\!]$ and $(s,h) \stackrel{\alpha}{\Longrightarrow} (s',h')$ then $dom(s') = dom(s)$, $s'(r) = s(r)$ for all $r$, and $s'(i) = s(i)$ for all $i \notin writes(c)$.*

# 9 Local state

A global state $(s,h)$, seen from the perspective of a process interacting with its environment, can be represented as $(s,(h_1,O_1),(h_2,O_2),H)$, where $(h_1,O_1)$ represents the heap portion and resource set owned by the process, $(h_2,O_2)$ is owned by the environment, and $H$ is the rest of the heap, containing a portion in which the resource invariants hold, separately, for the currently available resources. The global heap is the disjoint union of these portions, i.e. $h = h_1 \cdot h_2 \cdot H$. As in [10], heaps $h_1$ and $h_2$ are *separate*, written $h_1 \perp h_2$, iff $dom(h_1) \cap dom(h_2) = \{\}$, and when this holds we write $h_1 \cdot h_2$ for the (disjoint) union of $h_1$ and $h_2$.

For a set of resource names $O$ let $\Gamma \upharpoonright O = \{r(X) : R \in \Gamma \mid r \in O\}$ and $\Gamma \backslash O = \{r(X) : R \in \Gamma \mid r \notin O\}$. When $O$ is a singleton we write $\Gamma \upharpoonright r$ as an abbreviation for $\Gamma \upharpoonright \{r\}$, and similarly $\Gamma \backslash r$ for $\Gamma \backslash \{r\}$. Recall that $inv(\Gamma)$ is the separate conjunction of the invariants in $\Gamma$; when $\Gamma$ is empty this is **emp**.

**Definition 9.1** A local state for $\Gamma$ is a combination $(s,(h_1,O_1),(h_2,O_2),H)$ satisfying the following *separation properties*:

- $h_1, h_2, H$ are separate
- $O_1 \cap O_2 = \{\}$, $s(r) = 0$ for all $r \in O_1 \cup O_2$, $s(r) = 1$ otherwise
- $(s,H) \models inv(\Gamma \backslash (O_1 \cup O_2)) * \mathbf{true}$.

In other words, the process, its environment, and the available resources, own *separate* heap portions; the process and its environment own *disjoint* resources; and in the rest of the heap, the invariants for available resources hold, separately. Let $\Sigma_\Gamma$ be the set of local states for $\Gamma$.

A local state $(s,(h_1,O_1),(h_2,O_2),H)$ corresponds to a global state, namely $(s, h_1 \cdot h_2 \cdot H)$, obtained by combining the heap portions and ignoring the resource ownership partition. In general, a global state may be representable in such manner by many different local states, or by no such local state, for example if no subheap of the global heap satisfies the resource invariants.

$$(s,(h_1,O_1),(h_2,O_2),H) \xrightarrow[\Gamma,A]{\delta} (s,(h_1,O_1),(h_2,O_2),H)$$

$$(s,(h_1,O_1),(h_2,O_2),H) \xrightarrow[\Gamma,A]{i=v} (s,(h_1,O_1),(h_2,O_2),H)$$
if $(i,v) \in s$ and $i \in \mathit{owned}(\Gamma \upharpoonright O_1) \cup A$

$$(s,(h_1,O_1),(h_2,O_2),H) \xrightarrow[\Gamma,A]{i:=v} ([s \mid i:v],(h_1,O_1),(h_2,O_2),H)$$
if $i \notin \mathit{owned}(\Gamma \backslash O_1)$

$$(s,(h_1,O_1),(h_2,O_2),H) \xrightarrow[\Gamma,A]{[v]=v'} (s,(h_1,O_1),(h_2,O_2),H) \qquad \text{if } (v,v') \in h_1$$

$$(s,(h_1,O_1),(h_2,O_2),H) \xrightarrow[\Gamma,A]{[v]:=v'} (s,([h_1 \mid v:v'],O_1),(h_2,O_2),H) \qquad \text{if } v \in \mathit{dom}(h_1)$$

$$(s,(h_1,O_1),(h_2,O_2),H) \xrightarrow[\Gamma,A]{\mathit{disp}(v)} (s,(h_1 \backslash v,O_1),(h_2,O_2),H) \qquad \text{if } v \in \mathit{dom}(h_1)$$

$$(s,(h_1,O_1),(h_2,O_2),H) \xrightarrow[\Gamma,A]{\mathit{alloc}(v,[v_0,\ldots,v_n])} (s,(h_1',O_1),(h_2,O_2),H)$$
if $\{v,\ldots,v+n\} \cap \mathit{dom}(h_1 \cdot h_2 \cdot H) = \{\}$, where $h_1' = [h_1 \mid v:v_0,\ldots,v+n:v_n]$

$$(s,(h_1,O_1),(h_2,O_2),H) \xrightarrow[\Gamma,A]{\mathit{acq}\, r} ([s \mid r:0],(h_1 \cup h_r,O_1 \cup \{r\}),(h_2,O_2),H - h_r)$$
if $r \notin O_1 \cup O_2, h_r \subseteq H, (s,h_r) \models \mathit{inv}(\Gamma \upharpoonright r)$

$$(s,(h_1,O_1),(h_2,O_2),H) \xrightarrow[\Gamma,A]{\mathit{try}\, r} (s,(h_1,O_1),(h_2,O_2),H) \qquad \text{if } r \in O_1 \cup O_2$$

$$(s,(h_1,O_1),(h_2,O_2),H) \xrightarrow[\Gamma,A]{\mathit{rel}\, r} ([s \mid r:1],(h_1 - h_r,O_1 - \{r\}),(h_2,O_2),H \cup h_r)$$
if $r \in O_1, h_r \subseteq h_1, (s,h_r) \models \mathit{inv}(\Gamma \upharpoonright r)$

Fig. 2. Local enabling relations

For each action $\lambda$ we define a partial relation

$$\xrightarrow[\Gamma,A]{\lambda} \subseteq \Sigma_\Gamma \times (\Sigma_\Gamma \cup \{\mathit{abort}\})$$

$$(s,(h_1,O_1),(h_2,O_2),H) \xrightarrow[\Gamma,A]{i=v} abort \qquad \text{if } i \notin owned(\Gamma \upharpoonright O_1) \cup A$$

$$(s,(h_1,O_1),(h_2,O_2),H) \xrightarrow[\Gamma,A]{i:=v} abort \qquad \text{if } i \in owned(\Gamma \backslash O_1)$$

$$(s,(h_1,O_1),(h_2,O_2),H) \xrightarrow[\Gamma,A]{[v]=v'} abort \qquad \text{if } v \notin dom(h_1)$$

$$(s,(h_1,O_1),(h_2,O_2),H) \xrightarrow[\Gamma,A]{[v]:=v'} abort \qquad \text{if } v \notin dom(h_1)$$

$$(s,(h_1,O_1),(h_2,O_2),H) \xrightarrow[\Gamma,A]{disp(v)} abort \qquad \text{if } v \notin dom(h_1)$$

$$(s,(h_1,O_1),(h_2,O_2),H) \xrightarrow[\Gamma,A]{rel\, r} abort \qquad \text{if } \forall h' \subseteq h_1.\, (s,h') \models \neg inv(\Gamma \upharpoonright r)$$

$$(s,(h_1,O_1),(h_2,O_2),H) \xrightarrow[\Gamma,A]{abort} abort$$

Fig. 3. Local enabling relations

reflecting execution of action $\lambda$ by the process, describing when this action is enabled, and its effect on ownership of heap and resources. Such an action is only legal if it respects the ownership discipline and maintains the separation properties. An *abort* result indicates an action that breaks the rules. Note that we assume throughout that $dom(s)$ contains all variables read or written by the process or its environment. The definitions for these *local enabling* relations are given in Figures 2 and 3.

It is easy to check that the local enabling relations are well defined, in that whenever $\sigma \in \Sigma_\Gamma$ and $\sigma \xrightarrow[\Gamma,A]{\lambda} \sigma' \neq abort$, it follows that $\sigma' \in \Sigma_\Gamma$.

The following lemma summarizes some obvious, but useful, properties.

**Lemma 9.2**

- *If* $(s,(h_1,O_1),(h_2,O_2),H) \xrightarrow[\Gamma,A]{\lambda} (s',(h_1',O_1'),(h_2',O_2'),H')$, *then* $h_2' = h_2$, $O_2' = O_2$, $O_1 \xRightarrow[O_2]{\lambda} O_1'$, *and* $s'$ *agrees with* $s$ *except on* $writes(\lambda) \cup res(\lambda)$.
- *If* $s_1$ *and* $s_2$ *agree on* $free(\lambda)$ *then* $(s_1,(h_1,O_1),(h_2,O_2),H) \xrightarrow[\Gamma,A]{\lambda} abort$ *if and only if* $(s_2,(h_1,O_1),(h_2,O_2),H) \xrightarrow[\Gamma,A]{\lambda} abort$
- *If* $s_1$ *and* $s_2$ *agree on* $free(\lambda)$ *and*

$$(s_1,(h_1,O_1),(h_2,O_2),H) \xrightarrow[\Gamma,A]{\lambda} (s_1',(h_1',O_1'),(h_2,O_2),H'),$$

*there is a store $s_2'$ such that $s_1'$ agrees with $s_2'$ on $writes(\lambda) \cup res(\lambda)$ and*

$$(s_2, (h_1, O_1), (h_2, O_2), H) \xrightarrow[\Gamma,A]{\lambda} (s_2', (h_1', O_1'), (h_2, O_2), H').$$

By interchanging the rôle of process and environment we obtain the dual notion of an *environment move*, and we formalize the notion of an environment that respects $\Gamma$ and does not modify the variables in $A$.

**Definition 9.3** We define $\rightsquigarrow_{\Gamma,A} \subseteq \Sigma_\Gamma \times \Sigma_\Gamma$ by

$$(s, (h_1, O_1), (h_2, O_2), H) \rightsquigarrow_{\Gamma,A} (s', (h_1, O_1), (h_2', O_2'), H')$$

iff there is an action $\mu$ such that $writes(\mu) \cap A = \{\}$, $O_2 \xRightarrow[O_1]{\mu} O_2'$, and

$$(s, (h_2, O_2), (h_1, O_1), H) \xrightarrow[\Gamma,A]{\mu} (s', (h_2', O_2'), (h_1, O_1), H').$$

We then define $\xRightarrow[\Gamma,A]{\lambda} = (\rightsquigarrow_{\Gamma,A})^* \circ (\xrightarrow[\Gamma,A]{\lambda}) \circ (\rightsquigarrow_{\Gamma,A})^*$.

These relations extend to traces in the obvious way, by composition. In particular, when $\alpha$ is $\lambda_1 \ldots \lambda_n$,

$$\xRightarrow[\Gamma,A]{\alpha} = \xRightarrow[\Gamma,A]{\lambda_1} \circ \cdots \circ \xRightarrow[\Gamma,A]{\lambda_n}$$

where we interpret relational composition from left to right, i.e.

$$f \circ g = \{(\sigma, \sigma'') \mid \exists \sigma'. (\sigma, \sigma') \in f \ \&\ (\sigma', \sigma'') \in g\}.$$

Since command traces respect resources, and environment moves do not modify variables in the rely set, we obtain the following result, expressing these properties in semantic terms.

**Lemma 9.4**
*If $\alpha \in [\![c]\!]$ and $(s, (h_1, O_1), (h_2, O_2), H) \xRightarrow[\Gamma,A]{\alpha} (s', (h_1', O_1'), (h_2', O_2'), H')$, then $O_1' = O_1$, and $s'$ agrees with $s$ on $writes(\alpha) \cap A$.*

The following result connects the effects of an action on a local state with its effect on the corresponding global state, as in the Local/Global Connection Theorem of [3].

**Lemma 9.5**

- *If $(s, h) \xRightarrow{\lambda} abort$ and $(s, (h_1, O_1), (h_2, O_2), H) \in \Sigma_\Gamma$ with $h = h_1 \cdot h_2 \cdot H$, then $(s, (h_1, O_1), (h_2, O_2), H) \xrightarrow[\Gamma,A]{\lambda} abort$.*
- *If $(s, (h_1, O_1), (h_2, O_2), H) \xrightarrow[\Gamma,A]{\lambda} (s', (h_1', O_1'), (h_2, O_2), H')$, then $(s, h_1 \cdot h_2 \cdot H) \xRightarrow{\lambda} (s', h_1' \cdot h_2 \cdot H')$.*

Note that the first result holds for *any* ownership partition of resources and heap, for any global state $(s, h)$. The second result specifies a particular ownership partition of the global state.

## 10 Validity, made formal

We can now offer a formal definition of validity for an assertion $\Gamma \vdash_A \{p\}c\{q\}$, expressed in terms of the semantics and local enabling relations. This makes precise the informal characterization given earlier; we now have a semantic formalization of "interactive execution in an environment that respects $\Gamma$ and does not modify $A$". The definition enforces implicitly the requirement that $c$ preserves the invariants (separately), in any environment that preserves the invariants (separately), because of the separation property built into local states in $\Sigma_\Gamma$.

**Definition 10.1** The assertion $\Gamma \vdash_A \{p\}c\{q\}$ is valid iff, for all traces $\alpha \in [\![c]\!]$, and all local states $\sigma = (s, (h_1, O_1), (h_2, O_2), H) \in \Sigma_\Gamma$ with $dom(s) \supseteq owned(\Gamma) \cup A$, if $(s, h_1) \models p$ then

(i) $\neg(\sigma \stackrel{\alpha}{\Longrightarrow}_{\Gamma,A} abort)$

(ii) if $\sigma \stackrel{\alpha}{\Longrightarrow}_{\Gamma,A} (s', (h_1', O_1), (h_2', O_2), H')$, then $(s', h_1') \models q$.

## 11 Soundness

As usual, soundness of the inference rules follows from the property that the rules preserve validity: For all well-formed instances of each inference rule, if the premisses are valid and the side conditions hold, then the conclusion is valid. The proof is a rule-by-rule case analysis, mostly as in [3] but augmented with the use and manipulation of rely sets. We summarize the main steps in showing soundness of the Parallel rule, and the rule dealing with local resources.

The following Parallel Decomposition Lemma is the key to the Parallel rule. Again, it is obtained by augmenting the analogous lemma from [3], and the rely sets play a crucial rôle here in the technical justification.

**Theorem 11.1** *Let* $mod(\alpha_1) \cap A_2 = mod(\alpha_2) \cap A_1 = \{\}$, $free(\alpha_1) \subseteq owned(\Gamma) \cup A_1$, *and* $free(\alpha_2) \subseteq owned(\Gamma) \cup A_2$. *Let* $O_1 \cap O_2 = \{\}$ *and* $\alpha \in \alpha_1 \,{}_{O_1}\|_{O_2} \alpha_2$. *Let* $\sigma = (s, (h, O), (h_3, O_3), H) \in \Sigma_\Gamma$ *and* $h = h_1 \cdot h_2, O = O_1 \cdot O_2, A = A_1 \cup A_2$. *Let* $\sigma_1 = (s, (h_1, O_1), (h_2 \cdot h_3, O_2 \cdot O_3), H)$ *and* $\sigma_2 = (s, (h_2, O_2), (h_1 \cdot h_3, O_1 \cdot O_3), H)$.

(i) *If* $\sigma \stackrel{\alpha}{\Longrightarrow}_{\Gamma,A} abort$, *then* $\sigma_1 \stackrel{\alpha_1}{\Longrightarrow}_{\Gamma,A_1} abort$ *or* $\sigma_2 \stackrel{\alpha_2}{\Longrightarrow}_{\Gamma,A_2} abort$.

(ii) *If* $\sigma \stackrel{\alpha}{\Longrightarrow}_{\Gamma,A} (s', (h', O'), (h_3', O_3'), H')$, *then* $\sigma_1 \stackrel{\alpha_1}{\Longrightarrow}_{\Gamma,A_1} abort$, $\sigma_2 \stackrel{\alpha_2}{\Longrightarrow}_{\Gamma,A_2} abort$, *or there are* $h_1', h_2', O_1', O_2'$ *such that* $O_1' \cap O_2' = \{\}$, $O' = O_1' \cdot O_2'$, $h' = h_1' \cdot h_2'$, *and*

$$\sigma_1 \stackrel{\alpha_1}{\Longrightarrow}_{\Gamma,A_1} (s', (h_1', O_1'), (h_2' \cdot h_3', O_2' \cdot O_3'), H')$$
$$\sigma_2 \stackrel{\alpha_2}{\Longrightarrow}_{\Gamma,A_2} (s', (h_2', O_2'), (h_1' \cdot h_3', O_1' \cdot O_3'), H').$$

Proof: by induction on the length of $\alpha$. (*End of Proof*)

**Corollary 11.2** *If* $\Gamma \vdash_{A_1} \{p_1\}c_1\{q_1\}$ *and* $\Gamma \vdash_{A_2} \{p_2\}c_2\{q_2\}$ *are well-formed and valid, and* $mod(c_1) \cap A_2 = mod(c_2) \cap A_1 = \{\}$, *then*

$$\Gamma \vdash_{A_1 \cup A_2} \{p_1 * p_2\}c_1 \| c_2\{q_1 * q_2\}$$

*is valid.*

This shows soundness of the parallel rule.

For dealing with local resource blocks of form **resource** $r$ **in** $c$ we need the following lemma.

**Theorem 11.3**

(i) *If* $(s, (h_1 \cdot h, O_1), (h_2, O_2), H) \in \Sigma_\Gamma$, $(s, h) \models R$, $free(R) \subseteq X$, *and* $r \notin dom(\Gamma)$, *then* $([s \mid r : 1], (h_1, O_1), (h_2, O_2), H \cdot h) \in \Sigma_{\Gamma, r(X):R}$.

(ii) *If* $r \notin dom(\Gamma)$, $(s, (h_1 \cdot h, O_1), (h_2, O_2), H) \xRightarrow[\Gamma, A \cup X]{\beta \backslash r}$ *abort, and* $\beta \in [\![c]\!]_{[r:1]}$, *then* $([s \mid r : 1], (h_1, O_1), (h_2, O_2), H) \xRightarrow[(\Gamma, r(X):R), A]{\beta}$ *abort.*

(iii) *If* $([s \mid r : 1], (h_1, O_1), (h_2, O_2), H \cdot h) \xRightarrow[(\Gamma, r(X):R), A]{\beta} ([s' \mid r : 1], (h_1', O_1'), (h_2', O_2'), H')$, $(s, h) \models R$, *and* $\beta \in [\![c]\!]_{[r:1]}$, *there is a heap* $h' \subseteq H'$ *such that* $(s', h') \models R$ *and* $(s, (h_1 \cdot h, O_1), (h_2, O_2), H) \xRightarrow[\Gamma, A \cup X]{\beta \backslash r} (s', (h_1' \cdot h', O_1'), (h_2', O_2'), H - h')$.

We then obtain soundness for the Local Resource rule.

**Lemma 11.4** *Suppose* $\Gamma, r(X) : R \vdash_A \{p\}c\{q\}$ *is valid and well-formed, so* $r \notin dom(\Gamma)$ *and* $free(R) \subseteq X$. *Then* $\Gamma \vdash_{A \cup X} \{p * R\}$**resource** $r$ **in** $c\{q * R\}$ *is valid.*

Proof: Let $\sigma = (s, (h_1 \cdot h, O_1), (h_2, O_2), H)$ be a local state for $\Gamma$ with $(s, h_1) \models p$ and $(s, h) \models R$, so that $(s, h_1 \cdot h) \models p * R$. Let $\alpha \in [\![c]\!]_{[r:1]}$.

If $\sigma \xRightarrow[\Gamma, A \cup X]{\alpha \backslash r}$ *abort* then there is a local computation of form

$$(s, (h_1, O_1), (h_2, O_2), H \cdot h) \xRightarrow[(\Gamma, r(X):R), A]{\alpha} \textit{abort}.$$

But this would contradict validity of the premiss, since $(s, h_1) \models p$.

Now suppose

$$\sigma \xRightarrow[\Gamma, A \cup X]{\alpha \backslash r} (s', (h_1'', O_1), (h_2', O_2'), H').$$

Then there must be heaps $h_1', h'$ such that $h_1'' = h_1' \cdot h'$, $(s', h') \models R$, $h' \perp H'$, and

$$(s, (h_1, O_1), (h_2, O_2), H \cdot h) \xRightarrow[(\Gamma, r(X):R), A]{\alpha} (s', (h_1', O_1), (h_2', O_2'), H' \cdot h').$$

By validity of the premiss and the assumption that $(s, h_1) \models p$, it follows that $(s', h_1') \models q$. Hence $(s', h_1'') \models q * R$, as required. (*End of Proof*)

## 12 Provable programs are fault-free

**Theorem 12.1** *If $\Gamma \vdash_A \{p\}c\{q\}$ is valid, then every finite execution of $c$ from a global state satisfying $p * inv(\Gamma)$ is race-free, and ends in a state satisfying $q * inv(\Gamma)$.*

Proof: Suppose $\Gamma \vdash_A \{p\}c\{q\}$ is valid, and let $(s,h)$ be a global state satisfying $p * inv(\Gamma)$. Let $s = h_1 \cdot H$ with $(s,h_1) \models p$ and $(s,H) \models inv(\Gamma)$. Let $\alpha \in [\![c]\!]$. If $(s,h) \stackrel{\alpha}{\Longrightarrow} abort$, then $(s,(h_1,\{\}),(\{\},\{\}),H) \stackrel{\alpha}{\underset{\Gamma,A}{\Longrightarrow}} abort$. But this would contradict validity, so there must be $s'$, $h'$ such that $(s,h) \stackrel{\alpha}{\Longrightarrow} (s',h')$.
By assumption, $(s,(h_1,\{\}),(\{\},\{\}),H)$ is a local state for $\Gamma$, in which all resources are available. A global computation of $c$, without interference, from $(s,h)$ corresponds to a local computation of $c$ from $(s,(h_1,\{\}),(\{\},\{\}),H)$ in which the environment steps are trivial. By validity, there must be $h_1', H'$ such that $h' = h_1' \cdot H'$, $(s,(h_1,\{\}),(\{\},\{\}),H) \stackrel{\alpha}{\underset{\Gamma,A}{\Longrightarrow}} (s',(h_1',\{\}),(\{\},\{\}),H')$, and $(s',h_1') \models q$. (A trivial environment obeys the rely constraint vacuously.) Moreover, since this local computation produces a legal local state for $\Gamma$, we must also have $(s',H') \models inv(\Gamma)$. Thus $(s,h') \models q * inv(\Gamma)$, as required. (*End of Proof*)
Special case:
When $c$ has no free resource names, and $\vdash_A \{p\}c\{q\}$ is valid, the conventional partial correctness assertion $\{p\}c\{q\}$ is valid, and $c$ is race-free from all states satisfying $p$.

## 13 Further extensions

Our inference rules include a FRAME rule [6] as in the original CSL. This rule allows the "framing on" (by separate conjunction) of an additional formula in the pre- and post-condition of an assertion, provided the formula does not mention any variable written by the program. It is natural to ask whether there is an analogous rule that permits framing inside resource invariants. Our semantic investigation into the revised logic suggests the following new rule, which we call Invariant Framing:

$$\frac{\Gamma, r(X) : R \vdash_A \{p\}c\{q\}}{\Gamma, r(X \cup X') : R * R' \vdash_A \{p\}c\{q\}} \qquad \text{if } mod(c) \cap X' = \{\} \text{ and } free(R') \subseteq X'$$

Of course this rule can be used repeatedly to deal with multiple resources at a time. Together with the PARALLEL rule, this would allow the following Parallel Framing rule to be derived:

$$\frac{\Gamma, r(X_1) : R_1 \vdash_{A_1} \{p_1\}c_1\{q_1\} \quad \Gamma, r(X_2) : R_2 \vdash_{A_2} \{p_2\}c_2\{q_2\}}{\Gamma, r(X_1 \cup X_2) : R_1 * R_2 \vdash_{A_1 \cup A_2} \{p_1 * p_2\}c_1 \| c_2\{q_1 * q_2\}}$$

provided $mod(c_1) \cap (A_2 \cup X_2) = \{\}$ and $mod(c_2) \cap (A_1 \cup X_1) = \{\}$. This rule has a pleasing symmetry.

# 14 Conclusions

We have presented a revised formulation of Concurrent Separation Logic, using assertions annotated with rely sets, and sketched the details behind a proof of soundness for this logic, based on an action trace semantics. This repairs a defect hidden inside our earlier version of CSL, in response to a counterexample found by Wehrman and Berdine [12]. At the same time we have been able to weaken the syntactic constraints imposed on shared variable use in Owicki-Gries and adopted without change in the original CSL. As a benefit of this generalization, the new CSL can handle programs in which shared variables are protected by multiple resources, and the new rules require that a program must acquire *all* protecting resources before writing a shared variable and must acquire *some* protecting resource before reading a shared variable.

It should be straightforward to adapt the semantic framework and technical development to incorporate permissions as in [4], using a permission algebra $(\mathcal{P}, \oplus, \top)$ as in [1], and working with permissive stores $S = \mathbf{Ide} \rightharpoonup_{fin} V_{int} \times \mathcal{P}$ and heaps $H = \mathbf{Loc} \rightharpoonup_{fin} V_{int} \times \mathcal{P}$. A global state would then be representable "locally" as a tuple $(\sigma_1, \sigma_2, \tau)$ of *compatible* state portions, where $\sigma_1$ represents the local state of the process, $\sigma_2$ the local state of the environment, and $\tau$ the state belonging to available resources. We believe that this would offer a way to show soundness of Uday Reddy's logic [9] based on syntactic control of interference, in which permissions are used statically. We plan to explore the connections between Reddy's logic and the revised CSL, in future work; it seems likely that a provable assertion in our logic corresponds to a multitude of alternative assertions in Reddy's logic, differing only in the choice of permission attached to identifiers used by the program.

# Acknowledgement

This work was supported by the NATIONAL Science Foundation (NSF) No. 1017011.

# References

[1] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. Parkinson. *Permission accounting in separation logic.* POPL 2005, SIGPLAN Notices, 40(1), 259-270.

[2] 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.

[3] S. Brookes. *A semantics for concurrent separation logic.* Invited paper, CONCUR 2004, London. Springer LNCS 3170, August 2004. Journal version in: *Theoretical Computer Science*, 375(1-3), May 2007.

[4] S. Brookes. *Variables as Resource for Shared-Memory Programs: Semantics and Soundness.* MFPS 2006, Genova, Italy. ENTCS, Volume 158, 123-150. May 2006

[5] C.A.R. Hoare, *Towards a Theory of Parallel Programming.* In **Operating Systems Techniques**, C. A. R. Hoare and R. H. Perrott, editors, pp. 61-71, Academic Press, 1972.

[6] P. W. O'Hearn. *Resources, Concurrency, and Local Reasoning*. Invited paper, CONCUR 2004, London. Springer LNCS 3170, pp. 49-67, August 2004. Journal version in: *Theoretical Computer Science*, 375(1-3), 271-307, May 2007.

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

[8] D. Park. *On the semantics of fair parallelism*. In: **Abstract Software Specifications**, Springer-Verlag LNCS vol. 86, 504–526, 1979.

[9] U. Reddy. *Syntactic Control of Interference for Separation Logic*. Talk presented at MFPS 2011, Pittsburgh, May 2011. A preliminary version is available at `http://www.cs.bham.ac.uk/ udr/papers/sci-seplogic.pdf`.

[10] 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.

[11] J, C. Reynolds. Invited Talk, MFPS 2011. Pittsburgh, May 2011.

[12] I. Wehrman and J. Berdine. Private communication. January 2011.