Syntactic Control of Interference for Separation Logic (Preliminary Report)

Uday S. Reddy
University of Birmingham

April 12, 2011

In an important paper in 1978 [26], Reynolds formulated a system of rules for "syntactic control of interference" formalizing the extant conventions for good programming practice in controlling variable aliasing as well as the conventions used in the programming logics formulated by Hoare [11, 12]. The focus of the rules at that time was the use of procedures. However, concurrency poses very much the same issues of controlling variable aliasing. Despite considerable further work on Reynolds's system [15, 16, 17, 18, 21, 20, 25], its use for concurrency has not be explored in detail. See, however, the work of Ghica and his colleagues [8, 9] for some applications to concurrency.

Hoare [13] and Brinch Hansen [4] have formulated conventions for controlling variable aliasing in concurrent programs. Owicki and Gries [23] formulated a programming logic formalizing the reasoning principles enabled by these conventions. O'Hearn [19] extended the Owicki-Gries system to deal with heap storage, formulating a *Concurrent Separation Logic*, which is currently a subject of active study [6, 5, 10, 14, 24, 27]. In this paper, we bring to bear the principles of syntactic control of interference on the issues of Concurrent Separation Logic

The main issue of our concern is how the variable usage is controlled across parallel processes and the interplay between such control and the proof rules of the programming logic. The original logic formulated by Owicki and Gries employed informal statements of the form "variable not modified by any other process". Such a statement is ambiguous (e.g., does it include modification inside critical regions). It is also non-compositional. Checking if a proof is correctly constructed involves examining the entire program. It is also problematic in defining semantics of the programming logic and verifying its soundness.

Two previous attempts have been made to formalise the variable usage rules of Concurrent Separation Logic. Brookes [6] formulated a compositional set of rules in his effort to prove the soundness of the logic. However, the rules generalize the original Owicki-Gries rules in new ways and their soundness is not obvious. The second attempt was that of Parkinson *et al.* [24], where they treat variables as "resources" similar to heap locations whose access is controlled via programming logic proof rules. The soundness of these rules is more immediate. However the rules are clumsy to use in practice because normal conventions of variable usage are not respected. For instance, a formula such as $x = x$ is not universally true.

We provide a system of rules based on the principles of syntactic control of interference, which provides a clear formalization of the Owicki-Gries rules whose soundness is obvious. It also respects the traditional conventions of variable usage so that pitfalls in reasoning can be avoided. In addition, we believe that it also throws a clear light on the semantic structure of concurrent programs along with their reasoning principles.

1 Syntactic control framework

Our form of syntactic control is a modified version of Reynolds SCI, using the ideas of permissions for read-only access [2, 3]. In the original syntactic control system, as formulated in [20], term-formation rules are expressed using judgements of the form

$$\Sigma \mid \Sigma' \vdash M \ \mathbf{Comm}$$

where the variable context of the term is split into two zones Σ and Σ', called the *active zone* and *passive zone* respectively. The variables in Σ are "actively used" in the term, i.e., they are used for both reading and writing the state, whereas the variables in Σ' are "passively used", i.e., they are used for only reading. There are rules in the system to temporarily regard an active free variable as a passive free variable within localized contexts. However, a passive free variable cannot be turned into an active one. (That would violate the promise to use it passively.) By using a permission algebra instead of zones, we regain the ability to combine multiple passive copies of a variable back into an active free variable. Thus, our system will be strictly more powerful than the traditional syntactic control of interference.

We assume a permission algebra $(\mathcal{P}, \oplus, \top)$, i.e., a partial commutative semigroup that is cancellative, has a distinguished element $\top$ denoting full permission and satisfies certain additional axioms [2]. A convenient example of a permission algebra is that of *fractional permissions*: the real interval $(0, 1]$ with $\oplus$ being the partial operation of addition and $\top = 1$. The idea is that a full permission (1 in the fractional permission algebra) allows an "active" usage, i.e., both reading and writing, whereas a partial permission (represented by fractional values in the fractional algebra) allows a read-only or "passive" usage.[1]

A *variable context* Σ is an unordered list of the form $x_1^{p_1}, \ldots, x_n^{p_n}$ where $x_1, \ldots, x_n$ are variable symbols and $p_1, \ldots, p_n$ are permissions, subject to the following condition:[2]

- *if the same variable symbol x occurs in Σ multiple times with permissions $p_{i_1}, \ldots, p_{i_k}$ respectively then $p_{i_1} \oplus \cdots \oplus p_{i_k}$ is defined.*

We call a putative variable context *well-defined* when it satisfies this condition. If the variables $x_1, \ldots, x_n$ are pairwise distinct, then we say that the variable context is in *normal form*. A non-normal form variable context can be *normalized* by replacing the multiple copies of each variable by a single copy and associating with it the permission $p_{i_1} \oplus \cdots \oplus p_{i_k}$ as above. We denote the normalized version of variable context Σ by $\mathit{norm}(\Sigma)$.

We assume that all the variable contexts appearing in legal inferences are well-defined, i.e., any inference that leads to an ill-defined variable context is illegal. (Formally, our system of rules is a *natural deduction system*, where the variable contexts are used as assumptions of the deductions. Even though we use the notation of sequents for presenting the deduction rules, it is *not* a sequent calculus.)

The syntactic well-formedness of program phrases is expressed using a variety of judgements:

$$\Sigma \vdash x \ \mathbf{Var} \quad \Sigma \vdash E \ \mathbf{Exp} \quad \Sigma \vdash P \ \mathbf{Assert} \quad \Sigma \vdash C \ \mathbf{Comm}$$

These say, respectively, that the displayed phrase is a well-formed variable, expression, assertion or command in the variable context Σ. All these forms of judgements have a bidirectional

[1] Even though the term "permission" has become conventional for the elements of this algebra, it is really best to think of the elements as representing *levels of ownership* by program and logic phrases. "Half permission" may not make much intuitive sense, but "half ownership" is a perfectly reasonable notion.

[2] It is more conventional to require that all the variable symbols listed in a context are distinct. It would be possible to formulate variants of our rules using such a convention. But we feel that our approach is more intuitive.

structural rule:

$$\textit{Contraction}\quad \frac{\Sigma, x^p, x^q \vdash \mathcal{S}}{\Sigma, x^{p \oplus q} \vdash \mathcal{S}}$$

This allows two copies of a variable x to be combined into a single copy or to split a single copy into two, while keeping account of the permissions. The following rules will be *admissible* in our proof systems:

$$\textit{Weakening}\quad \frac{\Sigma \vdash \mathcal{S}}{\Sigma, \Sigma' \vdash \mathcal{S}}$$

$$\textit{Subst}_E\quad \frac{\Sigma \vdash E\ \mathbf{Exp} \quad \Sigma, x^\top \vdash E'\ \mathbf{Exp}}{\Sigma \vdash E'[E/x]\ \mathbf{Exp}}$$

$$\textit{Subst}_A\quad \frac{\Sigma \vdash E\ \mathbf{Exp} \quad \Sigma, x^\top \vdash P\ \mathbf{Assert}}{\Sigma \vdash P[E/x]\ \mathbf{Assert}}$$

(In this preliminary version of this document, we only mention the case of a variable with a full permission being substituted. But, clearly, substitution also makes sense when the variable has a partial permission. Treating the general case would require a multiplication operation on permissions, i.e., we need to move to partial commutative semirings instead of semigroups.)

To use a variable symbol x as a variable phrase in a program (thereby allowing assignments to it), one needs the full permission for the variable:

$$\frac{}{\Sigma, x^\top \vdash x\ \mathbf{Var}}$$

On the other hand, to use a variable as an expression, any permission will do.

$$\frac{}{\Sigma, x^p \vdash x\ \mathbf{Exp}} \qquad \frac{\Sigma \vdash E_1\ \mathbf{Exp} \quad \Sigma \vdash E_2\ \mathbf{Exp}}{\Sigma \vdash E_1 + E_2\ \mathbf{Exp}}$$

The second rule shows an example of an expression construct. Note that the same variable context is used for both the components of the expression. Rules for the other expression constructs can be formulated similarly.

Assertions are similar to expressions. Some sample rules are:

$$\frac{\Sigma \vdash E_1\ \mathbf{Exp} \quad \Sigma \vdash E_2\ \mathbf{Exp}}{\Sigma \vdash E_1 \overset{p}{\longmapsto} E_2\ \mathbf{Assert}} \qquad \frac{\Sigma \vdash P_1\ \mathbf{Assert} \quad \Sigma \vdash P_2\ \mathbf{Assert}}{\Sigma \vdash P_1 \star P_2\ \mathbf{Assert}}$$

$$\frac{\Sigma, x^\top \vdash P\ \mathbf{Assert}}{\Sigma \vdash \exists x.\, P\ \mathbf{Assert}}$$

It is worth noting that there is no requirement for the two conjuncts of a separating conjunction $P_1 \star P_2$ to use separate variable contexts. In the rule for existential quantifier, we are assuming a variable hygiene convention that requires that the bound variable x is distinct from the free variables listed in Σ. The variable x is assumed to carry the full permission $\top$ in the body of the quantifier. (It is interesting to consider a variant of the rule where some non-$\top$ permission is assumed for the variable x. Nothing would go wrong, of course, since x is a logical variable and cannot be modified inside a program.)

Well-formed rules for commands are straightforward as well:

$$\frac{}{\Sigma \vdash \textbf{skip Comm}} \qquad \frac{\Sigma \vdash x \textbf{ Var} \quad \Sigma \vdash E \textbf{ Exp}}{\Sigma \vdash (x := E) \textbf{ Comm}}$$

$$\frac{\Sigma \vdash x \textbf{ Var} \quad \Sigma \vdash E \textbf{ Exp}}{\Sigma \vdash (x := [E]) \textbf{ Comm}} \qquad \frac{\Sigma \vdash E_1 \textbf{ Exp} \quad \Sigma \vdash E_2 \textbf{ Exp}}{\Sigma \vdash ([E_1] := E_2) \textbf{ Comm}}$$

$$\frac{\Sigma \vdash B \textbf{ Exp} \quad \Sigma \vdash C_1 \textbf{ Comm} \quad \Sigma \vdash C_2 \textbf{ Comm}}{\Sigma \vdash (\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2) \textbf{ Comm}}$$

$$\frac{\Sigma \vdash E \textbf{ Exp} \quad \Sigma, x^{\top} \vdash C \textbf{ Comm}}{\Sigma \vdash (\textbf{local } x := E \textbf{ in } C) \textbf{ Comm}}$$

An example of a well-formed command judgement is

$$x^1, y^{\frac{1}{2}} \vdash (x := y) \textbf{ Comm}$$

The variable context needs to contain a full permission for x because it is used on the left hand side of an assignment, but a half permission will do for y because it is only used for reading.

The well-formedness rules for commands are somewhat redundant because our programming logic will be formulated in such a way that a specification $\Sigma \vdash \{P\}\, C\, \{Q\}$ is derivable only if $\Sigma \vdash C$ **Comm** is a valid well-formedness judgement. So, no special attention need be paid to the well-formedness of commands.

This observation is more important than it might appear at first sight. We are trying to formulate programming logics where the specifications that can be derived are well-formed and valid. So, it is not simply enough for commands to be well-formed. They must be well-formed in the context of their correctness proofs, which is a stronger requirement.

2 Sequential Separation Logic

A judgement of sequential Separation Logic is of the form $\Sigma \vdash \{P\}\, C\, \{Q\}$, which means that the failure-avoiding specification $\{P\}\, C\, \{Q\}$ holds assuming a variable context Σ. The variables that are modified in the command C would be required to have $\top$ permission in Σ. Other variables, which might be employed in C in a read-only fashion or employed only in assertions, can have non-$\top$ permissions.

The rules for commands are as follows:

$$\frac{\Sigma \vdash P \textbf{ Assert}}{\Sigma \vdash \{P\}\, \textbf{skip}\, \{P\}} \qquad \frac{\Sigma \vdash x \textbf{ Var} \quad \Sigma \vdash E \textbf{ Exp} \quad \Sigma \vdash P \textbf{ Assert}}{\Sigma \vdash \{P[E/x]\}\, x := E\, \{P\}}$$

$$\frac{\Sigma \vdash x \textbf{ Var} \quad \Sigma \vdash E \textbf{ Exp} \quad \Sigma \vdash E' \textbf{ Exp}}{\Sigma \vdash \{P[E'/x] \wedge E \mapsto E'\}\, x := [E]\, \{P \wedge E \mapsto E'\}} \text{ if } x \notin FV(E, E')$$

$$\frac{\Sigma \vdash E \textbf{ Exp} \quad \Sigma \vdash E' \textbf{ Exp}}{\Sigma \vdash \{E \mapsto \_\}\, [E] := E'\, \{E \mapsto E'\}}$$

$$\frac{\Sigma \vdash B \textbf{ Exp} \quad \Sigma \vdash \{P \wedge B\}\, C_1\, \{Q\} \quad \Sigma \vdash \{P \wedge \neg B\}\, C_2\, \{Q\}}{\Sigma \vdash \{P\}\, \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2\, \{Q\}}$$

$$\frac{\Sigma \vdash E \textbf{ Exp} \quad \Sigma \vdash P \textbf{ Assert} \quad \Sigma \vdash Q \textbf{ Assert} \quad \Sigma, x^{\top} \vdash \{P \wedge x = E\}\, C\, \{Q\}}{\Sigma \vdash \{P\}\, \textbf{local } x := E \textbf{ in } C\, \{Q\}}$$

Since we incorporate the well-formedness of assertions and commands in specifications, most rules have premises to do with well-formedness of assertions, commands or components of commands. In the rule for assignment, we depend on the admissible rule $Subst_A$ which allows us to substitute for a variable symbol with the $\top$ permission. The rule for heap cell lookup illustrates the use of side conditions for specifying genuine logical conditions about the occurrence of free variables (as opposed to the conditions that are purely to do with well-formedness issues). Contrast this with the rule for local variable declaration, where we require that E, P and Q should be well-formed in the *outer* variable context. So, they cannot have x occurring free. This seems to be a reasonable choice, because most programmers understand the scope of x to be command C. So, its free occurrence in other places would be considered odd.

The frame rule of Separation Logic gives us the first application of the syntactic control of interference:

$$FRAME \quad \frac{\Sigma \vdash \{P\}\, C\, \{Q\} \quad \Sigma' \vdash R \ \mathbf{Assert}}{\Sigma, \Sigma' \vdash \{P \star R\}\, C\, \{Q \star R\}}$$

(Note that there is an implicit side condition for the rule that says that Σ, Σ' is a well-formed variable context.) Since the variable contexts of $\{P\}\, C\, \{Q\}$ and R are required to be separate, it is not possible for C to modify any free variables of R. If C modifies a variable x then Σ needs to include $x^{\top}$. Then x^p cannot occur in Σ', for any permission p, because $\top \oplus p$ is undefined. Thus the splitting of the variable context into Σ and Σ' has exactly the same force as the usual side condition "C does not modify any free variables of R" in the conventional formulation of Separation Logic.

For example, using the fractional permission algebra, we can derive the judgement:

$$x^1, y^{\frac{1}{2}} \vdash \{y = 0\}\, x := y\, \{x = 0\}$$

We can use the frame rule with a separate assertion $y^{\frac{1}{2}}, z^{\frac{1}{2}} \vdash y = z$ to derive:

$$x^1, y^1, z^{\frac{1}{2}} \vdash \{y = 0 * y = z\}\, x := y\, \{x = 0 * y = z\}$$

However, it is not possible to derive:

$$x^1, y^{\frac{1}{2}}, \Sigma' \vdash \{y = 0 * x = z\}\, x := y\, \{x = 0 * x = z\}$$

for any Σ' that provides the content for $(x = z)$ **Assert**. To do so, Σ' must include some permission p for x, but $1 \oplus p$ is undefined.

3 Concurrent Separation Logic

To motivate the design of the rules for Concurrent Separation Logic, we look at an example due to Owicki and Gries [23], shown in Table 1. Even though we use separating conjunction $\star$ in assertions, $\star$ has the same force as the ordinary conjunction $\wedge$ because the formulas involved are pure.

The remarkable fact about the proof is that the variables a and b appear in the local assertions of the processes, despite being part of the resource r. This is justified in the Owicki-Gries and O'Hearn's proof systems using a critical region proof rule *which allows the variables owned by a resource to appear in local assertions as long as they are not modified in "other processes."* This is similar to the side condition employed in the frame rule, and it can be formalized using permissions in the same way.[3]

[3]Note that Brookes [6] defines a variant of the O'Hearn's system, where a and b need not be included among the owned variables of the resource. However, since they are among the free variables of the resource invariant, separate restrictions apply to their usage.

x := 0; a := 0; b := 0;
$\{x = a + b \star a = 0 \star b = 0\}$
resource r(x, a, b) $\{x = a + b\}$ **in**
begin

$\{a = 0\}$		$\{b = 0\}$
with r **do**		**with** r **do**
$\{a = 0 \star x = a + b\}$		$\{b = 0 \star x = a + b\}$
x := x+1;	||	x := x+1;
a := 1		b := 1
$\{a = 1 \star x = a + b\}$		$\{b = 1 \star x = a + b\}$
od		**od**
$\{a = 1\}$		$\{b = 1\}$

end
$\{x = a + b \star a = 1 \star b = 1\}$
$\{x = 2\}$

Table 1: Example proof outline in Concurrent Separation Logic

Suppose we allow resources to own, not merely collections of variables, but specific permissions for them. Then the resource r can be defined to own the permissions x^1, $a^{\frac{1}{2}}$, $b^{\frac{1}{2}}$. The entire program is specified in the context x^1, a^1, b^1. The remaining permissions $a^{\frac{1}{2}}$ and $b^{\frac{1}{2}}$ can be distributed to the two processes: $a^{\frac{1}{2}}$ to the left process and $b^{\frac{1}{2}}$ to the right process. This allows the two processes to employ a and b in their local assertions. When the left process enters its critical region, its local permissions are combined with those owned by the resource, leading to total permissions x^1, a^1, $b^{\frac{1}{2}}$. This allows the critical region to modify x and a. The right process is similar.

This discussion motivates the use of variables with permissions, i.e., *variable contexts*, to be declared along with resources. So, the syntax of commands is as follows:

$$\begin{array}{rcl} C & ::= & x := E \mid x := [E] \mid [E] := E' \mid \textbf{skip} \mid C_1; C_2 \mid \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \\ & & \mid C_1 \parallel C_2 \mid \textbf{with } r \textbf{ when } B \textbf{ do } C \textbf{ od} \mid \textbf{resource } r(\Sigma) \textbf{ in } C \end{array}$$

The well-formedness of commands is defined using judgements of the form

$$\Sigma \mid \Gamma \vdash C \textbf{ Comm}$$

Here, Σ is a variable context and Γ is a *resource context* of the form $r_1(\Sigma_1), \ldots, r_n(\Sigma_n)$ where r_i are resource names, Σ_i are variable contexts owned by the resources, subject to the following conditions:

- *The resource names r_i are distinct from each other.*
- *The variable context $\Sigma, \Sigma_1, \ldots, \Sigma_n$ is well-defined.*

A putative syntactic context satisfying these conditions is said to be *well-defined*. Note that only commands require resource contexts (which get used in checking the well-formedness of critical regions). Variables, expressions, and assertions only need variable contexts.

The well-formedness rules of commands are shown in Table. 2. All the rules of the sequential programming language can be lifted to the concurrent language by adding " $\mid \Gamma$" to the syntactic

contexts of all the commands. We show two rules for assignment and conditional as examples. The rule for parallel composition follows the general pattern of original syntactic control of interference in [20, 26]. The resource context is shared between the parallel branches but the variable contexts are required to be separate. The critical region rule shows that the variable context of the resource becomes part of the normal variable context of the critical region. This is where the use of a permission algebra adds value to the traditional syntactic control of interference. It is possible for the critical region to combine the variable permissions in Σ and Σ_0 to convert a passive free variable into active one. If Σ and Σ_0 each contain $x^{\frac{1}{2}}$ then, by combining them, the critical region obtains the permission x^1, which allows it to modify the variable x. The rule for resource declaration requires a part of the current variable context (Σ_0) to be sliced off and handed to the resource, which is then available only by entering critical regions.

$$\frac{\Sigma \vdash x\ \mathbf{Var} \quad \Sigma \vdash E\ \mathbf{Exp}}{\Sigma \mid \Gamma \vdash (x := E)\ \mathbf{Comm}} \qquad \frac{\Sigma \vdash B\ \mathbf{Exp} \quad \Sigma \mid \Gamma \vdash C_1\ \mathbf{Comm} \quad \Sigma \mid \Gamma \vdash C_2\ \mathbf{Comm}}{\Sigma \mid \Gamma \vdash (\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2)\ \mathbf{Comm}}$$

$$\frac{\Sigma_1 \mid \Gamma \vdash C_1\ \mathbf{Comm} \quad \Sigma_2 \mid \Gamma \vdash C_2\ \mathbf{Comm}}{\Sigma_1, \Sigma_2 \mid \Gamma \vdash (C_1 \parallel C_2)\ \mathbf{Comm}}$$

$$\frac{\Sigma, \Sigma_0 \vdash B\ \mathbf{Exp} \quad \Sigma, \Sigma_0 \mid \Gamma \vdash C\ \mathbf{Comm}}{\Sigma \mid \Gamma, r(\Sigma_0) \vdash (\mathbf{with}\ r\ \mathbf{when}\ B\ \mathbf{do}\ C\ \mathbf{od})\ \mathbf{Comm}}$$

$$\frac{\Sigma \mid \Gamma, r(\Sigma_0) \vdash C\ \mathbf{Comm}}{\Sigma, \Sigma_0 \mid \Gamma \vdash (\mathbf{resource}\ r(\Sigma_0)\ \mathbf{in}\ C)\ \mathbf{Comm}}$$

Table 2: Well-formedness of concurrent commands

Just as in the sequential case, our rules of the programming logic incorporate the well-formedness of commands. So, no special attention needs to be paid to their well-formedness.

The programming logic is formulated using judgements of the form

$$\Sigma \mid \Gamma \vdash \{P\}\ C\ \{Q\}$$

Here, Σ is a variable context and Γ is an *annotated resource context* where each resource $r_i(\Sigma_i)$ is annotated with "resource invariant" formula R_i which is a *precise* assertion and satisfies $\Sigma_i \vdash R_i\ \mathbf{Assert}$. Note that a resource invariant for a resource can only employ the variables available in its variable context.

All the rules of the sequential Separation Logic can be lifted to Concurrent Separation Logic by simply adding " $\mid \Gamma$" to all the specification judgements. For example,

$$\frac{\Sigma \vdash x\ \mathbf{Var} \quad \Sigma \vdash E\ \mathbf{Exp} \quad \Sigma \vdash P\ \mathbf{Assert}}{\Sigma \mid \Gamma \vdash \{P[E/x]\}\ x := E\ \{P\}}$$

$$\frac{\Sigma \vdash B\ \mathbf{Exp} \quad \Sigma \mid \Gamma \vdash \{P \wedge B\}\ C_1\ \{Q\} \quad \Sigma \mid \Gamma \vdash \{P \wedge \neg B\}\ C_2\ \{Q\}}{\Sigma \mid \Gamma \vdash \{P\}\ \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \{Q\}}$$

The resource contexts do not play any rule in the sequential fragment of the programming language.

The proof rule for parallel composition is as follows:

$$PAR \quad \frac{\Sigma_1 \mid \Gamma \vdash \{P_1\}\ C_1\ \{Q_1\} \quad \Sigma_2 \mid \Gamma \vdash \{P_2\}\ C_2\ \{Q_2\}}{\Sigma_1, \Sigma_2 \mid \Gamma \vdash \{P_1 \star P_2\}\ C_1 \parallel C_2\ \{Q_1 \star Q_2\}}$$

As one would expect, the variable context of the composite command needs to be split into separate portions for the two processes. The resource context, on the other hand, is shared. The rule allows C_1 and C_2 to share read-only variables, via separate copies with partial permissions. However, it is not possible for one process to modify a variable employed in the other process or *its proof*.

A resource's variables can be imported when a critical region is entered:

$$CRIT \quad \frac{\Sigma \vdash P, Q \textbf{ Assert} \quad \Sigma, \Sigma_0 \vdash B \textbf{ Exp} \quad \Sigma, \Sigma_0 \mid \Gamma \vdash \{P \star R \wedge B\}\, C\, \{Q \star R\}}{\Sigma \mid \Gamma,\, r(\Sigma_0){:}\, R \vdash \{P\} \textbf{ with } r \textbf{ when } B \textbf{ do } C \textbf{ od } \{Q\}}$$

The body of the critical region, C, can use the combined variable contexts of the process and the resource. However, the pre-condition and the post-condition can only employ the variables available in the process's context. This captures the Owicki-Gries requirement that they should only employ variables not modified by "other processes".

Finally, the rule for the resource declaration is as follows:

$$RESOURCE \quad \frac{\Sigma_0 \vdash R \textbf{ Assert} \quad \Sigma \mid \Gamma,\, r(\Sigma_0){:}\, R \vdash \{P\}\, C\, \{Q\}}{\Sigma, \Sigma_0 \mid \Gamma \vdash \{P \star R\} \textbf{ resource } r(\Sigma_0) \textbf{ in } C\, \{Q \star R\}} \quad (R \text{ precise})$$

The variable context Σ_0 is sliced out of the current context, and transferred to the resource r. The resource invariant is based on these variables. The body of the resource declaration, C, can only use the remaining context Σ outside the critical regions.

For example, the body of the critical region in the left process in Table 1 (abbreviated to C_1) has the following specification:

$$x^1, a^1, b^{\frac{1}{2}} \mid \vdash \{a = 0 \star x = a + b\}\, C_1\, \{a = 1 \star x = a + b\}$$

By the *CRIT* rule, we obtain

$$a^{\frac{1}{2}} \mid r(x^1, a^{\frac{1}{2}}, b^{\frac{1}{2}}) : x = a + b \vdash \{a = 0\} \textbf{ with } r \textbf{ do } C_1 \textbf{ od } \{a = 1\}$$

Similarly, the right process has the specification

$$b^{\frac{1}{2}} \mid r(x^1, a^{\frac{1}{2}}, b^{\frac{1}{2}}) : x = a + b \vdash \{b = 0\} \textbf{ with } r \textbf{ do } C_2 \textbf{ od } \{b = 1\}$$

and the parallel rule gives:

$$a^{\frac{1}{2}}, b^{\frac{1}{2}} \mid r(x^1, a^{\frac{1}{2}}, b^{\frac{1}{2}}) : x = a + b \vdash \{a = 0 \star b = 0\}\ \ldots \parallel \ldots\ \{a = 1 \star b = 1\}$$

Finally, the *RESOURCE* rule allows us to derive:

$$x^1, a^1, b^1 \mid \vdash \{x = a{+}b \star a = 0 \star b = 0\} \textbf{resource } r(x^1, a^{\frac{1}{2}}, b^{\frac{1}{2}}) \textbf{ in } \ldots \parallel \ldots \{x = a{+}b \star a = 1 \star b = 1\}$$

3.1 Comparison with Owicki-Gries-O'Hearn system

O'Hearn's version of Concurrent Separation Logic [19] is based on the Owicki-Gries system [23] as its underlying framework for variable usage. In this system, the free variables of the resource invariant must be listed in the resource, similar to our *RESOURCE* rule. The rules governing the variables of a resources are as follows:

1. If a variable x belongs to a resource r, it cannot appear in a parallel process except in a critical region for r.

2. If a variable x is changed in process S_i, it cannot appear in S_j $(i \neq j)$ unless it belongs to a resource.

The rule 1 is relaxed in our proof rules. Recall that our resources encapsulate not merely variables but variables with permissions. So, if x belongs to a resource with permission $\top$ then the restrictions on its usage in our rules are exactly the same as in the Owicki-Gries system. However, if x belongs to the resource with a partial permission, then one or more processes can possibly use x in a read-only fashion using the remaining partial permission.

The rule 2 is represented exactly the same way in our proof rules.

The rule 1 is somewhat misleading. While it requires that a variable x belonging to a resource cannot appear in the *code* of a parallel process except in a critical region, it nevertheless permits it to appear in the *assertions* of the process outside critical regions. Thus, the proof outline of Table 1 is legal in the Owicki-Gries-O'Hearn system. However, there is a rider to this allowance in the Owicki-Gries proof rule for critical regions. A variable occurring free in the assertions surrounding a critical region should not be changed in "another process". The allowance as well as its rider are already covered in our relaxation of the rule 1 above. We treat the free occurrences of variables in assertions as well as read-only occurrences in code in exactly the same way. A variable that is not modified in "another process," is available to the current process with a partial permission. So, it can use it in a read-only fashion in both code and assertions. Our relaxation of the Owicki-Gries rule 1 leads to a simpler formulation.

Thus all valid proof outlines of the Owicki-Gries-O'Hearn system remain valid proof outlines in our logic with syntactic control of interference. It is quite straightforward to come up with an assignment of permissions to the variables listed in a resource.

- If a variable appears in multiple processes, either in code or assertions, and modified in at least one of them, then the resource should contain the $\top$ permission for the variable.
- If a variable has read-only occurrences in one or more processes, then then resource may contain any permission p for the variable and the complement of p should be distributed to all processes that use it outside critical sections.
- If a variable is used in only one process (but possibly in assertions outside critical regions), then the resource may contain any permission p for the variable and the complement of p is given to the process.

For the example in Table 1, the variable x appears in multiple processes. So, it gets the permission 1 in the resource. The variables a (respectively, b) is used only in the left process (respectively, the right process). So, the resource is given $\frac{1}{2}$ permission and the process is given the remaining $\frac{1}{2}$.

3.2 Comparison with Brookes's system

Brookes [6], in his effort to prove the soundness of the Concurrent Separation Logic, defined a variant of the original logic defined by O'Hearn. In his formulation, the resource invariant of a resource can have additional variables that are not declared in the resource. He defines two sets of variables for a resource context: **owned**(Γ) is the set of variables included in the resource declarations and **free**(Γ) is the set of variables that occur free in the resource invariants in Γ. The two sets of variables are governed by different rules.

1. Variables in **owned**(Γ) can be used only inside critical regions for the resources. They cannot occur free in either assertions or expressions outside the critical regions.

2. Variables in **free**(Γ) that are not in **owned**(Γ) can be modified only in the critical regions for the resources. However, they *can* occur free in assertions and expressions outside the critical regions.

So, the proof outline of Table 1 is not valid in the Brookes's version of Concurrent Separation Logic. The variables a and b are owned by the resource, but they occur free outside critical regions. However, the proof outline can be transformed to a legal Brookes outline by removing the variables a and b from owned list of the resource. Since each of these variables is modified in at most one process, Brookes does not require it to be owned by the resource. It can simply remain a free variable of the resource invariant. However, the rule 2 restricts each of these variables to be modified only in critical regions.

Valid proof outlines in the Brookes's system can be transformed to our system. If $r(x_1, \ldots, x_n)$ is a Brookes resource declaration used with an invariant R, and **free**(R) includes additional variables $y_1, \ldots, y_m$, then the resource declaration should be transformed to $r(x_1^\top, \ldots, x_n^\top, y_1^{p_1}, \ldots, y_m^{p_m})$ in our system, where the permissions $p_1, \ldots, p_m$ are chosen to satisfy the constraints on their use:

1. If a variable y_i is modified in the critical regions of a process A then it cannot occur in the other processes. (Brookes's parallel composition rule requires that any variable modified in one process and occurring free in another process — called a "critical" variable — has to be owned by a resource. But y is not owned by r by assumption, and well-formedness of resource contexts prohibits it from being owned by another resource.) In this case, p_i can be some partial permission, and the complement of p_i is allocated to the process A for the variable y_i.

2. If a variable y_i is not modified in any of the processes, then it is a read-only variable in the **resource** declaration command. So, the available permission of y_i in the variable context (which might be a partial peermssion) should be split into the permission for the resource (p_i) and the various processes.

However, there is a third, more troublesome, case. Brookes's rules, like the Owicki-Gries rules, make a distinction between read-only uses of variables in code and their use in assertions. While the first case above prohibits the read-only uses of y_i in the *code* of processes other than A, it does not prohibit its uses in their *assertions*. This turns out to be unsound, as shown by the example in Table 3, due to Ian Wehrman [1]. In this example, the variable x is in **owned**(Γ) and a is in **free**(Γ). Since a does not occur free in the *code* of the left process, this is permitted by Brookes's rules. However, a occurs in the *assertions* of the left process, immediately after the first critical region. This represents invalid reasoning. The right process can intervene between the two critical regions of the left process and modify a. So, the assertion $t = a$ may not continue to hold when the second critical region is entered.

The distinction between read-only uses in code and uses in assertions was also made by Owicki-Gries, as noted in Sec. 3.1. However, Owicki-Gries place the additional requirement (the "rider" mentioned in Sec. 3.1) that the assertions surrounding critical regions can only use variables that are not modified by other processes. The assertion $t = a$ used after the first critical region of the left process is thus prohibited by Owicki-Gries.

Brookes's system can be repaired using a similar rider. This would have the unfortunate consequence that the rules are not compositional any more. However, it would bring it closer to the Owicki-Gries system as well as our syntactic control system. In effect, the variables listed in the resources are the variables with full permissions, and the remaining variables in **free**(Γ) are variables that have partial permissions in the resource. So, the distinction between **owned**(Γ) and **free**(Γ) is one of permission levels, and Brookes's system fits in between the Owicki-Gries system and our system of syntactic control with permissions.

x := a;
resource r(x) $\{x = a\}$ **in**
begin

$\{true\}$		$\{true\}$
with r **do**		**with** r **do**
$\{x = a\}$		$\{x = a\}$
t := x	||	x := x+1;
$\{x = a = t\}$		a := a+1
od		$\{x = a\}$
$\{t = a\}$		**od**
with r **do**		$\{true\}$
$\{x = a = t\}$		
x := t		
$\{x = a\}$		
od		
$\{true\}$		

end
$\{x = a\}$

Table 3: Example proof outline in Brookes's system

3.3 Comparison with "Variables as Resource" systems

Parkinson et al. [24] and Brookes [5] define a general scheme of treating variables as resources with permissions. In contrast to our approach of syntactic control, the variable resources are included in program assertions, through ownership formulas of the form $\mathbf{own}_p(x)$ and used with all the normal logical connectives. So, this approach can be termed "logical control of interference" for variables.

It is easy to see that the syntactic control system can be translated to the logical control system. For every variable context $\Sigma = (x_1^{p_1}, \ldots, x_n^{p_n})$, there is an ownership formula $O_\Sigma \equiv \mathbf{own}_{p_1}(x_1) \star \cdots \star \mathbf{own}_{p_n}(x_n)$. A judgement $\Sigma \mid \Gamma \vdash \{P\}\, C\, \{Q\}$ of our system can be translated to a judgement $\Gamma \vdash \{O_\Sigma \wedge P\}\, C\, \{O_\Sigma \wedge Q\}$ in the "Variables as resource" system. In fact, Parkinson et al [24] give translations of this form for Hoare logics.

It is not possible to go in the reverse direction. The "Variables as resource" system uses logical formulas to express ownership of variables. So, it can express much richer set of ownership constraints than possible in the syntactic control system. For example, the formula

$$(x = 0 \wedge \mathbf{own}_\top(y)) \vee (x \neq 0 \wedge \mathbf{own}_\top(z))$$

does not correspond to any syntactic variable context.

Thus, the "Variables as Resource" logic is more expressive than the syntactic control system. However, we argue that the syntactic control system offers considerable simplicity and convenience. In particular,

- There are no issues of undefinedness in expressions and formulas. So, one does not need to write formulas of the form $E = E$ just to ensure that E is defined in the current context.
- Substitution is a valid operation in expressions and assertions.

- The system has no logical anomalies, e.g., the equivalence $\neg(E_1 = E_2) \iff E_1 \neq E_2$ holds in our system, whereas the two formulas have different interpretations in the Variables as Resource logic.
- We need no special treatment of logical variables. The "pun" of program variables as logical variables, characteristic of Hoare logics, continues to work in our system.

4 Semantics and soundness

This section represents work in progress.

The syntactic control system for Concurrent Separation Logic has a semantics in terms of Brookes's action traces [6] and the soundness of the logic follows from it. We indicate some of this structure to characterize the role played by the variable contexts in the semantics.

An *action* is a syntactic token given by the syntax:

$$\lambda ::= \delta \mid x = v \mid x := v \mid [l] = v \mid [l] := v \mid \mathit{try}(r) \mid \mathit{acq}(r) \mid \mathit{rel}(r) \mid \mathit{abort}$$

As in [6], δ is a do-nothing or idle action, $x = v$ denotes the action of reading the variable x to obtain a value v, $x := v$ denotes the action of writing a value v into the variable x. The actions $[l] = v$ and $[l] := v$ denote similar actions for heap locations. The tokens $\mathit{try}(r)$, $\mathit{acq}(r)$ and $\mathit{rel}(r)$ denote the actions of attempting to acquire a resource, acquiring a resource and releasing a resource. The token *abort* denotes the action of aborting a computation in case of an error.

A *trace* is a possibly infinite sequence of actions subject to the identifications $\alpha \cdot \delta \cdot \beta = \alpha \cdot \beta$, and $\alpha \cdot \mathit{abort} \cdot \beta = \alpha \cdot \mathit{abort}$.

The syntactic control system imposes some structure on the actions and traces (just as a type system would impose structure on computations in a typed programming language). First of all, the syntactic contexts $\Sigma \mid \Gamma$ enable certain actions and prohibit others. For instance, a variable action $x = v$ or $x := v$ would only be possible in a syntactic context that contains the variable x and the resource actions $\mathit{try}(r)$ and $\mathit{acq}(r)$ would only be possible in a syntactic context that contains a resource named r. Secondly, as a result of an action, the context available for the rest of a trace might change. For instance, $\mathit{acq}(r)$ has the effect of removing the a resource $r(\Sigma_0)$ from the resource context and adding its variables Σ_0 to the variable context. A $\mathit{rel}(r)$ action would have the opposite effect. We represent these effects by a transition relation $\xrightarrow{\lambda}$ on syntactic contexts.

Unfortunately, we cannot work with syntactic contexts directly because the traces represent a low-level language of computations which lacks the block structure of the high-level languages. The action of $\mathit{acq}(r)$ cannot be to simply remove a resource from the resource context because we need to reconstruct it again when a $\mathit{rel}(r)$ action occurs. So, we continue to retain the acquired resource in the context but mark it as "busy". An *extended context* is a context of the form

$$\Sigma, \Sigma'_1, \ldots, \Sigma'_m \mid r_1(\Sigma_1), \ldots, r_n(\Sigma_n), [r'_1(\Sigma'_1)], \ldots, [r'_m(\Sigma'_m)]$$

such that

- the resource names $r_1, \ldots, r_n, r'_1, \ldots, r'_m$ are all distinct, and
- the variable context $\Sigma, \Sigma'_1, \ldots, \Sigma'_m, \Sigma_1, \ldots, \Sigma_n$ is well-defined.

The idea is that the resources $r'_1, \ldots, r'_m$ have been acquired by the trace in previous actions and their variables have been incorporated into the current variable context. If and when these resources are released, the variables would be removed from the variable context and returned

to the resource context. Once again, a putative extended context satisfying these conditions is said to be *well-defined*. We use the letter $\widetilde{\Gamma}$ to range over extended resource contexts where some resources might be marked busy.

The transition relation on extended contexts is as follows:

$$\begin{array}{l}
\Sigma \mid \widetilde{\Gamma} \xrightarrow{\delta} \Sigma \mid \widetilde{\Gamma} \\
\Sigma \mid \widetilde{\Gamma} \xrightarrow{x=v} \Sigma \mid \widetilde{\Gamma} \quad \text{iff} \quad x^p \in \Sigma \text{ for some } p \\
\Sigma \mid \widetilde{\Gamma} \xrightarrow{x:=v} \Sigma \mid \widetilde{\Gamma} \quad \text{iff} \quad x^\top \in \mathit{norm}(\Sigma) \\
\Sigma \mid \widetilde{\Gamma} \xrightarrow{[l]=v} \Sigma \mid \widetilde{\Gamma} \\
\Sigma \mid \widetilde{\Gamma} \xrightarrow{[l]:=v} \Sigma \mid \widetilde{\Gamma} \\
\Sigma \mid \widetilde{\Gamma}, r(\Sigma_0) \xrightarrow{try(r)} \Sigma \mid \widetilde{\Gamma}, r(\Sigma_0) \\
\Sigma \mid \widetilde{\Gamma}, r(\Sigma_0) \xrightarrow{acq(r)} \Sigma, \Sigma_0 \mid \widetilde{\Gamma}, [r(\Sigma_0)] \\
\Sigma, \Sigma_0 \mid \widetilde{\Gamma}, [r(\Sigma_0)] \xrightarrow{rel(r)} \Sigma \mid \widetilde{\Gamma}, r(\Sigma_0)
\end{array}$$

Note that the transition relation is single-valued, i.e., a partial function. It extends to traces $\xrightarrow{\alpha}$ in the obvious way via relational composition. Note that there are no constraints on the actions for reading and writing heap locations. This is because because the access to heap locations is controlled in the programming logic rather than the syntax.

For any finite or infinite trace $\alpha = \lambda_1\lambda_2\ldots$, we use the notation:

$$\Sigma \mid \widetilde{\Gamma} \xrightarrow{\alpha} \cdot \quad \text{iff} \quad \exists \Sigma_i, \widetilde{\Gamma}_i.\ \Sigma \mid \widetilde{\Gamma} \xrightarrow{\lambda_1} \Sigma_1 \mid \widetilde{\Gamma}_1 \xrightarrow{\lambda_2} \Sigma_2 \mid \widetilde{\Gamma}_2 \xrightarrow{\lambda_3} \cdots$$

If $\Sigma \mid \widetilde{\Gamma} \xrightarrow{\alpha} \cdot$, we say that the trace α is *enabled* in the context $\Sigma \mid \widetilde{\Gamma}$.

If α_1 is enabled in a context $\Sigma_1 \mid \widetilde{\Gamma}_1$ and α_2 is enabled in a context $\Sigma_2 \mid \widetilde{\Gamma}_2$ such that Σ_1, Σ_2 is well-defined and $\widetilde{\Gamma}_1$ and $\widetilde{\Gamma}_2$ are identical except for marking disjoint sets of resources as being busy, then we consider the notion of *interleaving* α_1 and α_2. Two actions λ_1 and λ_2 are said to *interfere*, written $\lambda_1 \sharp \lambda_2$, if λ_1 writes to a heap location l and λ_2 reads or writes the same location l, or *vice versa*. The set of *mutex fair merges* of α_1 and α_2, denoted $\alpha_1 \,{}_{\widetilde{\Gamma}_1}\|_{\widetilde{\Gamma}_2}\, \alpha_2$ is defined by induction on the lengths of α_1 and α_2:

$$\begin{array}{rcl}
\alpha_1 \,{}_{\widetilde{\Gamma}_1}\|_{\widetilde{\Gamma}_2}\, \epsilon & = & \{\alpha_1\} \\
\epsilon \,{}_{\widetilde{\Gamma}_1}\|_{\widetilde{\Gamma}_2}\, \alpha_2 & = & \{\alpha_2\} \\
(\lambda_1\alpha_1) \,{}_{\widetilde{\Gamma}_1}\|_{\widetilde{\Gamma}_2} (\lambda_2\alpha_2) & = & \{\, \mathit{abort} \mid \lambda_1 \sharp \lambda_2 \,\} \cup \\
 & & \{\, \lambda_1\beta \mid (\Sigma_1 \mid \widetilde{\Gamma}_1) \xrightarrow{\lambda_1} (\Sigma_1' \mid \widetilde{\Gamma}_1') \wedge \beta \in \alpha_1 \,{}_{\widetilde{\Gamma}_1'}\|_{\widetilde{\Gamma}_2} (\lambda_2\alpha_2) \,\} \cup \\
 & & \{\, \lambda_2\beta \mid (\Sigma_2 \mid \widetilde{\Gamma}_2) \xrightarrow{\lambda_2} (\Sigma_2' \mid \widetilde{\Gamma}_2') \wedge \beta \in (\lambda_1\alpha_1) \,{}_{\widetilde{\Gamma}_1}\|_{\widetilde{\Gamma}_2'}\, \alpha_2 \,\}
\end{array}$$

This definition is similar to a corresponding definition of Brookes [6]. It extends to trace sets in the expected manner: $T_1 \,{}_{\widetilde{\Gamma}_1}\|_{\widetilde{\Gamma}_2}\, T_2 = \bigcup\{\, \alpha_1 \,{}_{\widetilde{\Gamma}_1}\|_{\widetilde{\Gamma}_2}\, \alpha_2 \mid \alpha_1 \in T_1 \wedge \alpha_2 \in T_2 \,\}$.

A *(well-bracketed) trace for an extended context* $\Sigma \mid \widetilde{\Gamma}$ is either *abort* or a trace α such that $\Sigma \mid \widetilde{\Gamma} \xrightarrow{\alpha} \Sigma \mid \widetilde{\Gamma}$.[4] The terminology is motivated by thinking of the $acq(r)$ and $rel(r)$ actions as brackets. A trace set T is a *(well-bracketed) trace set for context* $\Sigma \mid \widetilde{\Gamma}$ if each trace in T is a well-bracketed trace for the context.

Lemma 1 *If α is a trace for an extended context $\Sigma \mid \widetilde{\Gamma}$, and $\Sigma, \Sigma' \mid \widetilde{\Gamma}, \widetilde{\Gamma}'$ is a longer well-defined extended context, then α is a trace for $\Sigma, \Sigma' \mid \widetilde{\Gamma}, \widetilde{\Gamma}'$.*

[4] This definition is specific to finite traces. We omit the case of infinite traces in this preliminary version of the paper.

Lemma 2 *If α_1 and α_2 are traces for extended contexts $\Sigma_1 \mid \widetilde{\Gamma}$ and $\Sigma_2 \mid \widetilde{\Gamma}$ respectively, and $\Sigma_1, \Sigma_2 \mid \widetilde{\Gamma}$ is a well-defined extended context then $\alpha_1 \,{}_{\widetilde{\Gamma}}\|_{\widetilde{\Gamma}}\, \alpha_2$ is a trace set for the context $\Sigma_1, \Sigma_2 \mid \widetilde{\Gamma}$.*

All expressions and commands can be given a compositional semantics in terms of trace sets.

- The meaning of an expression $\Sigma \vdash E$ **Exp** is a set of pairs (ρ, v) where ρ is an action trace enabled in the syntactic context $\Sigma \mid$ (i.e., a syntactic context with no resources, because expressions do not access resources), and v is a value (obtained as the result of evaluating E). We denote it by $[\![E]\!]_\Sigma$.
- The meaning of a command $\Sigma \mid \Gamma \vdash C$ **Comm** is a set of traces ρ for the context $\Sigma \mid \Gamma$. We denote it by $[\![C]\!]_{\Sigma|\Gamma}$.

The semantics defined in the standard fashion [6]. However, it is defined by induction on the derivations of well-formedness judgements $\Sigma \vdash E$ **Exp** and $\Sigma \mid \Gamma \vdash C$ **Comm**, instead of induction on the structure of terms.

$$[\![x]\!]_\Sigma = \{\, (x = v, v) \mid v \text{ Value} \,\}$$
$$[\![E_1 + E_2]\!]_\Sigma = \{\, (\rho_1\rho_2, v_1 + v_2) \mid (\rho_1, v_1) \in [\![E_1]\!]_\Sigma \wedge (\rho_2, v_2) \in [\![E_2]\!]_\Sigma \,\}$$

We use the notation $[\![E]\!]_\Sigma \restriction v$ to denote the set of traces $\{\, \rho \mid (\rho, v) \in [\![E]\!]_\Sigma \,\}$.

$$[\![\mathbf{skip}]\!]_{\Sigma|\Gamma} = \{\delta\}$$
$$[\![x := E]\!]_{\Sigma|\Gamma} = \{\, \rho(x := v) \mid (\rho, v) \in [\![E]\!]_\Sigma \,\}$$
$$[\![x := [E]]\!]_{\Sigma|\Gamma} = \{\, \rho([v] := v')(x := v') \mid (\rho, v) \in [\![E]\!]_\Sigma \,\}$$
$$[\![[E] := E']\!]_{\Sigma|\Gamma} = \{\, \rho\rho'([l] := v') \mid (\rho, v) \in [\![E]\!]_\Sigma \wedge (\rho', v') \in [\![E']\!]_\Sigma \,\}$$
$$[\![\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2]\!]_{\Sigma|\Gamma} = ([\![B]\!]_\Sigma \restriction \mathbf{true})\, [\![C_1]\!]_{\Sigma|\Gamma} \cup ([\![B]\!]_\Sigma \restriction \mathbf{false})\, [\![C_2]\!]_{\Sigma|\Gamma}$$
$$[\![\mathbf{local}\ x := E\ \mathbf{in}\ C]\!]_{\Sigma|\Gamma} = \{\, \rho(\alpha \setminus x) \mid (\rho, v) \in [\![E]\!]_\Sigma \wedge \alpha \in ([\![C]\!]_{\Sigma, x^\top|\Gamma})_{[x:v]} \,\}$$
$$[\![C_1 \parallel C_2]\!]_{\Sigma_1,\Sigma_2|\Gamma} = [\![C_1]\!]_{\Sigma_1|\Gamma} \parallel [\![C_2]\!]_{\Sigma_2|\Gamma}$$
$$[\![\mathbf{with}\ r\ \mathbf{when}\ B\ \mathbf{do}\ C\ \mathbf{od}]\!]_{\Sigma|\Gamma, r(\Sigma_0)} = \mathit{wait}^*\, \mathit{enter} \cup \mathit{wait}^\omega$$
$$\text{where}\quad \mathit{wait} = \mathit{acq}(r)\, ([\![B]\!]_{\Sigma,\Sigma_0} \restriction \mathbf{false})\, \mathit{rel}(r) \cup \{\mathit{try}(r)\}$$
$$\mathit{enter} = \mathit{acq}(r)\, ([\![B]\!]_{\Sigma,\Sigma_0} \restriction \mathbf{true})\, [\![C]\!]_{\Sigma,\Sigma_0|\Gamma}\, \mathit{rel}(r)$$
$$[\![\mathbf{resource}\ r\ \mathbf{in}\ C]\!]_{\Sigma,\Sigma_0|\Gamma} = \{\, \rho \setminus r \mid \rho \in [\![C]\!]_{\Sigma|\Gamma, r(\Sigma_0)} \,\}$$

The notation $T_{x:v}$, for a trace set T, stands for the subset of T containing all the traces that follow from states in which $x = v$.

Theorem 3 *The meaning of command $\Sigma \mid \Gamma \vdash C$ **Comm** is a (well-bracketed) trace set for the context $\Sigma \mid \Gamma$. Likewise, the meaning of an expression $\Sigma \vdash E$ **Exp** consists of (well-bracketed) trace sets for every $[\![E]\!]_\Sigma \restriction v$.*

The proof is by induction on the derivation of well-formed terms:

- If the command is $\Sigma \mid \Gamma(x := E)$ **Comm** then we know that $x^\top \in \mathit{norm}(\Sigma)$ and $\Sigma \vdash E$ **Exp**. So, for any $(\rho, v) \in [\![E]\!]_\Sigma$, ρ is a well-bracketed trace for $\Sigma \mid$ and, hence, for $\Sigma \mid \Gamma$. Since $x^\top \in \mathit{norm}(\Sigma)$, $(x := v)$ is also a trace for $\Sigma \mid \Gamma$.
- If the command is $\Sigma \mid \Gamma, r(\Sigma_0) \vdash$ (**with** r **when** B **do** C **od**) **Comm** then we know that $\Sigma, \Sigma_0 \vdash B$ **Exp** and $\Sigma, \Sigma_0 \vdash C$ **Comm**. By inductive hypothesis, $[\![B]\!]_{\Sigma,\Sigma_0} \restriction b$ and $[\![C]\!]_{\Sigma,\Sigma_0|\Gamma}$ are trace sets for $\Sigma, \Sigma_0 \mid \Gamma$. It then follows that the trace sets *wait* and *enter* are trace

sets for $\Sigma \mid \Gamma, r(\Sigma_0)$. For example, considering arbitrary elements $\rho \in [\![B]\!]_{\Sigma,\Sigma_0} \restriction \mathbf{true}$ and $\gamma \in [\![C]\!]_{\Sigma,\Sigma_0|\Gamma}$, we have the transition sequence:

$$\begin{aligned}
&\Sigma \mid \Gamma, r(\Sigma_0) \\
&\quad \xrightarrow{acq(r)} \Sigma, \Sigma_0 \mid \Gamma, [r(\Sigma_0)] \\
&\quad \xrightarrow{\rho} \Sigma, \Sigma_0 \mid \Gamma, [r(\Sigma_0)] \\
&\quad \xrightarrow{\gamma} \Sigma, \Sigma_0 \mid \Gamma, [r(\Sigma_0)] \\
&\quad \xrightarrow{rel(r)} \Sigma \mid \Gamma, r(\Sigma_0)
\end{aligned}$$

where the second and third steps are obtained by lemma 1.

References

[1] J. Berdine and I. Wehrman. Variable conditions and CSL. Private communication, 4th April, 2011.

[2] R. Bornat, C. Calcagno, P.W. O'Hearn, and M. Parkinson. Permission accounting in Separation Logic. In *ACM Symp. on Princ. of Program. Lang.*, pages 59–70. ACM Press, 2005.

[3] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: 10th Intern. Symp.*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.

[4] P. Brinch Hansen. *Operating System Principles*. Prentice-Hall, Englewood Cliffs, 1973.

[5] S. D. Brookes. Variables as resource for shared-memory programs: Semantics and soundness. In *Proc. 22nd Ann. Conf. on Math. Found. of Program. Semantics (MFPS XXII)*, volume 158 of *Elect. Notes in Theor. Comput. Sci.*, pages 123–150. Elsevier, 2006.

[6] S. D. Brookes. A semantics for Concurrent Separation Logic. *Theoretical Comput. Sci.*, 375(1-3):227–270, Apr 2007.

[7] S. D. Brookes, M. Main, A. Melton, and M. Mislove. *Proc. 27nd Ann. Conf. on Math. Found. of Program. Semantics (MFPS XXVII)*, volume (to appear) of *Elect. Notes in Theor. Comput. Sci.* Elsevier, 2011.

[8] D. R. Ghica. Geometry of synthesis: A structured approach to VLSI design. In *POPL2007*. ACM, 2007.

[9] D. R. Ghica, A. S. Murawski, and C.-H. L. Ong. Syntactic control of concurrency. *Theoretical Comput. Sci.*, 350(2-3):234–251, Feb 2006.

[10] A. Gotsman, J Berdine, and B Cook. Precision and the conjunction rule in Concurrent Separation Logic. In *Proc. 27nd Ann. Conf. on Math. Found. of Program. Semantics (MFPS XXVII)* [7].

[11] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12:576–583, 1969.

[12] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In E. Engeler, editor, *Symp. Semantics of Algorithmic Languages*, volume 188 of *Lect. Notes Math.*, pages 102–116. Springer-Verlag, 1971.

[13] C. A. R. Hoare. Towards a theory of parallel programming. In C. A. R. Hoare and R. H. Perrott, editors, *Operating Systems Techniques*, pages 61–71. Academic Press, 1972.

[14] K. Kapoor, K. Lodaya, and U. S. Reddy. Fine grained concurrency using Separation Logic. *J. Philosophical Logic*, (to appear), 2012.

[15] G. McCusker. Categorical models of syntactic control of interference revisited, revisited. *LMS J. of Comp. and Math.*, 10:176–206, 2007.

[16] G. McCusker. A graph model for imperative computation. *Logical Methods in Comp. Sci.*, 6(1-2), Jan 2010.

[17] P. W. O'Hearn. Linear logic and interference control. In *Category Theory and Computer Science*, volume 350 of *LNCS*, pages 74–93. Springer-Verlag, 1991.

[18] P. W. O'Hearn. A model for syntactic control of interference. *Math. Struct. Comput. Sci.*, 3:435–465, 1993.

[19] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Comput. Sci.*, 375(1-3):271–307, May 2007.

[20] P. W. O'Hearn, A. J. Power, M. Takeyama, and R. D. Tennent. Syntactic control of interference revisited. In S. D. Brookes, M. Main, A. Melton, and M. Mislove, editors, *Math. Found. of Program. Semantics: Eleventh Ann. Conference*, volume 1 of *Elect. Notes in Theor. Comput. Sci.* Elsevier, 1995. (Reprinted as Chapter 18 of [22]).

[21] P. W. O'Hearn and J. C. Reynolds. From Algol to polymorphic linear lambda-calculus. *J. ACM*, 47(1):167–223, Jan 2000.

[22] P. W. O'Hearn and R. D. Tennent. *Algol-like Languages (Two volumes).* Birkhäuser, Boston, 1997.

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

[24] M. Parkinson, R. Bornat, and Calcagno. Variables as resource in Hoare Logics. In *Symp. on Logic in Comput. Sci.*, pages 137–146. IEEE, 2006.

[25] U. S. Reddy. Global state considered unnecessary: An introduction to object-based semantics. *J. Lisp and Symbolic Computation*, 9:7–76, 1996. (Reprinted as Chapter 19 of [22]).

[26] J. C. Reynolds. Syntactic control of interference. In *ACM Symp. on Princ. of Program. Lang.*, pages 39–46. ACM, 1978. (Reprinted as Chapter 10 of [22]).

[27] V. Vafeiadis. Concurrent Separation Logic and operational semantics. In *Proc. 27nd Ann. Conf. on Math. Found. of Program. Semantics (MFPS XXVII)* [7].