Syntactic Control of Interference for Separation Logic

Uday S. Reddy
University of Birmingham
u.s.reddy@cs.bham.ac.uk

John C. Reynolds
Carnegie-Mellon University
jcr@cs.cmu.edu

Abstract

Separation Logic has witnessed tremendous success in recent years in reasoning about programs that deal with heap storage. Its success owes to the fundamental principle that one should keep separate areas of the heap storage separate in program reasoning. However, the way Separation Logic deals with program variables continues to be based on traditional Hoare Logic without taking any benefit of the separation principle. This has led to unwieldy proof rules suffering from lack of clarity as well as questions surrounding their soundness. In this paper, we extend the separation idea to the treatment of variables in Separation Logic, especially Concurrent Separation Logic, using the system of Syntactic Control of Interference proposed by Reynolds in 1978. We extend the original system with permission algebras, making it more powerful and able to deal with the issues of concurrent programs. The result is a streamined presentation of Concurrent Separation Logic, whose rules are memorable and soundness obvious. We also include a discussion of how the new rules impact the semantics and devise static analysis techniques to infer the required permissions automatically.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Syntax; F.1.2 [*Theory of Computation*]: Computation by Abstract Devices—Models of Computation – Parallelism and concurrency; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Logics of programs; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Denotational semantics

General Terms Program Logic, Concurrency, Denotational Semantics, Type Systems

Keywords Separation Logic, Syntactic Control of Interference, Conditional Critical Regions, Fractional Permissions, Static Analysis

1. Introduction

In reasoning about programs that alter the state, one often encounters stylized side conditions that have to do with how variable symbols are used. For example, the "invariance" rule of Hoare Logic [3] (or the "constancy" rule in Specification Logic [37]), written as

$$\frac{\{P\}\, C\, \{Q\}}{\{P \wedge R\}\, C\, \{Q \wedge R\}}$$

has a side condition that states that "C should not modify any variables occurring free in R." This rule becomes the all-important "frame rule" in Separation Logic [30] but the same side condition is retained. Similar conditions also occur in the rules for procedures [3, 37]. In fact, such conditions are not only employed in the proof rules for procedures, but it has also been argued that intelligible programming requires adherence to them even if no efforts are made at formal reasoning. A procedure call $P(A)$ is regarded as intelligible only if the procedure P does not modify any of the variables occurring free in the argument A and, likewise, the argument A does not cause state changes via the variables occurring free in P. (This is more commonly called "aliasing" control. Consider call-by-name or call-by-reference parameter passing methods or higher-order arguments to see the full effect of this observation.)

These observations are also made with additional force in concurrent programming. Hoare [17, 18] and Brinch Hansen [10] have argued convincingly that parallel processes should not interfere with each other. A process should not modify variables that are concurrently used by other processes unless the variables are under the control of resources enforcing mutual exclusion. Program logic proof rules similarly employ these conditions in their proof rules [17, 25, 31].

Noticing that essentially the same side condition arises in all such contexts, Reynolds [36] formulated it as a uniform principle of *non-interference*. Two program phrases P_1 and P_2 are considered non-interfering if the variables used in one of them for state-modification do not occur free in the other phrase. This work presents a system of rules called "syntactic control of interference" (SCI) which bring structure to the conditions employed in intelligible programming as well as the formal rules of programming logics. These rules incorporate, at syntactic level, what we now regard as the "separation principle," the same principle that is responsible for the success of Separation Logic in reasoning about heap storage. The SCI system has been studied quite extensively since this early work. O'Hearn [26] reformulated Reynolds's rules in the notation of type systems (or proof theory) and noted its overriding similarity to Girard's Linear Logic [15]. Reddy [33] formulated a novel semantic model for programs in the SCI framework, exploiting the non-interference property of the programs in a crucial way, which turns out to be the first fully abstract model ever discovered for a higher-order imperative programming language [22]. (The games models of Abramsky et al. [1, 2] generalize Reddy's model to deal with interference and represent fully abstract models as well.) Generalizing the SCI framework, O'Hearn and Pym formulated bunched type systems and the logic of Bunched Implications [27], the latter of which forms the foundation for Separation Logic [30, 34]. In retrospect, it is fair to say that SCI has proved to be a deep foundational principle of imperative programs leading to numerous developments in our understanding of their structure.

Curiously, despite all the historical background, SCI has not been used in formulating Separation Logic itself. We believe that this has led to unwieldy proof rules fraught with side conditions. The problems become critical in the formulation of Concurrent Separation Logic [25]. Brookes's attempt to formalize such rules [11] turned out to be faulty, with known counterexamples to their soundness [4].

In this paper, we reformulate the rules of sequential as well as concurrent Separation Logic using the principles of SCI to bring structure to their side conditions. It turns out that the traditional SCI is not quite adequate to the task. It incorporates a limited treatment of "passive" or "read-only" uses of variables which is unable to deal with the more advanced usage of variables in concurrent programs. We enrich the traditional SCI with the idea of fractional permissions, borrowed from Boyland [8] and Bornat et al [6], to devise a more powerful variant. This system is then used to create a streamlined formulation of Separation Logic.

While fractional permissions for variables allow a streamlined presentation of the proof rules and clarify their semantics, they would be an overkill to use in practice. A programmer should not have to explicitly annotate all the variable uses in processes and shared resources with fractional annotations. To address the issue, we devise a *permission inference system*, which can take a Separation Logic proof outline without any permissions specified with shared resources and fills them in if at all possible according to the rules of the logic. The effect is similar to that of Hindley-Milner type inference in programming languages like ML [14, 23].

Related Work

Hoare [17] and Brinch Hansen [9] have formulated conventions for controlling variable aliasing in concurrent programs. Hoare also proposed proof rules for reasoning with conditional critical regions. Owicki and Gries [31] generalized Hoare's conventions as well as the proof rules. O'Hearn [25] extended the Owicki-Gries system to deal with heap storage, formulating a *Concurrent Separation Logic*, which is currently a subject of active study [11, 13, 16, 20, 32, 38]. We refer to this logic as the "Owicki-Gries-O'Hearn system."

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. (For instance, the Smallfoot verification tool implements such global analysis [19].) 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 [11] 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. In fact, subsequently, Ian Wehrman has found counterexamples to their soundness in one particular case [4]. The second attempt was that of Bornat *et al.* [7, 32], where they treat variables as "resources" similar to heap locations, whose access is controlled via programming logic proof rules. Their rules do address the non-compositionality issue mentioned above and the soundness of their rules is more immediate. However the rules are clumsy to use in practice because the normal pun of variable symbols as mutable variable and logical variables is not retained. For instance, a formula such as $x = x$ is true if x is a logical variable, but not necessarily so when x is a program variable.

Although we arrived at our formulation via a different route, using the ideas of syntactic control of interference to formalize the original Owicki-Gries-O'Hearn system, our system can also be seen as a syntactic variant of the Parkinson-Bornat-Calcagno logic described in [32]. The benefits of using the syntactic approach are:

- The normal conventions of variable usage in Hoare-style logics are respected. So pitfalls in reasoning from improper treatment of variable symbols can be avoided.
- We are able to devise a compositional (or "modular") static analysis system to automatically infer permissions required for variable usage.
- The system should be extensible to higher-order languages with procedures and objects. For instance, the methods of objects can be categorized as active or passive. (Even though we do not discuss the higher-order aspects in this paper, the traditional principles of syntactic control of interference for higher-order procedures apply. These principles however do not generalize to storable procedures as in ML. So, further work is needed to address such issues.)

In recent work, Brookes [12] has independently devised a revised set of proof rules for Concurrent Separation Logic using ideas resembling ours. His rules do not employ fractional permissions as ours do and the relationship to SCI is less clear cut. We do not know at present the precise relationship between his formulation and ours, but we anticipate that the two are very close.

Inference of fractional permissions has been studied by Yasuoka and Terauchi [40] and Bierhoff [5]. Both these pieces of work address the permissions needed for heap cells (which is a harder problem than that for variables). However, they do not deal with concurrency, which is our main concern. Yasuoka et al. use a region-based analysis to make the heap permission problem tractable, which may be seen as a reduction to the corresponding problem for variable names. The techniques employed in their work rely on sophisticated constraint-solving methods. In contrast, our permission inference algorithm a two-pass algorithm on the syntax tree, similar to regular compiler analysis methods, with only rudimentary constraint solving issues.

The rest of this paper is organized as follows. In Section 2, we informally motivate the issues addressed by our formulation of Separation Logic. The logical system itself is described in Sections 3 and 4. We also include a detailed comparison with the proof systems of O'Hearn [25] Parkinson et al. [32]. A comparison with Brookes's original system [11] is include in Appendix B. In Section 5, we describe the denotational semantic framework of the proof system and indicate how the soundness is proved. Finally, in Section 6, we describe an algorithm to automatically infer variable permissions needed in the proof system.

2. Motivation

As mentioned in the Introduction, Hoare and Brinch Hansen advocated the avoidance of interference between concurrent processes as a good practice of intelligible programming. That requires that, in forming a parallel composition of commands $C_1 \parallel C_2$, one must ensure that C_1 does not modify any variable that occurs free in C_2 and *vice versa*. We use the terminology "*actively used*" for variables that are used for state-modification. Any variables that are used *only* for reading the state are said to be "*passively used*."

We first consider how to treat active free variables of phrases using Syntactic Control of Interference using the notations of [26, 29]. If a command C is formed using a set of active free variables Σ, it is described using a judgement of the form $\Sigma \vdash C\ \mathbf{Comm}$.

```
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

Now, the well-formedness of an intelligible parallel composition in the sense of Hoare and Brinch Hansen can be described by the rule:

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

Notice that the active free variables of C_1 and C_2 are combined "multiplicatively," requiring them to be separate or disjoint. Thus, the non-interference conditions of Hoare and Brinch Hansen can be described in a pleasingly symmetric fashion without employing side conditions.

The judgements used above describe the *well-formedness* of commands. Rules of program logic can be stated in essentially the same way. The Separation Logic proof rule for parallel composition becomes:

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

Each judgement in this rule asserts the well-formedness of a Hoare triple specification *as well as* the truth of the specification. Once again, *no side conditions are required* to describe a sound inference.

While Reynolds [36] only considered independent parallel composition, it is possible to add shared resources, e.g., Hoare-style resources and conditional critical regions, in the same way. A resource declaration command

$$\mathbf{resource}\ r(\Sigma_0)\ \mathbf{in}\ C$$

should split the available active variable context into two separate parts, Σ_0 for the variables encapsulated in the resource and the remainder of the context for the body C. A critical region command:

$$\mathbf{with}\ r\ \mathbf{when}\ B\ \mathbf{do}\ C\ \mathbf{od}$$

should add the encapsulated context of the resource r to the current context for the scope C. All this seems essentially straightforward. However, it turns out to be *inadequate* in practice.

To see the problem, consider the example proof outline shown in Table 1, discussed by Owicki and Gries [31]. Even though we use separating conjunction $\star$ in assertions, $\star$ has the same force as the ordinary conjunction $\wedge$ here because the formulas involved are pure. The purpose of the proof outline is to argue that running $x := x + 1$ in parallel with itself increments x by 2. The variable x is placed in a resource r, allowing it to be safely shared across the parallel branches. Notice that placing it in the resource precludes it from being mentioned in the parallel branches outside any critical regions. So, it is not possible to write assertions that show that each critical region increments x.

To solve the problem, Owicki and Gries recommend adding auxiliary variables a and b and using them to record control information about the increment actions performed in the two processes. The auxiliary variables are also included in the shared resource. So, a and b can only be modified inside critical regions.[1] The resource invariant $x = a + b$ captures the control information recorded by a and b. However, notice that a and b need to be mentioned in assertions *outside* the critical regions. Owicki and Gries tailor their proof rules to allow such usage. Evidently, we are entering tricky territory here. The variable x cannot be used outside critical regions whereas the variables a and b are allowed to be used. The difference is that x is modified in *both* the processes. Making assertions about it in one of the processes would not be sound because the other process can invalidate the assertions. On the other hand, the variable a is only modified in the left process. So, assertions mentioning a remain true independent of the progress of the other process. Thus, Owicki-Gries as well as O'Hearn's proof systems use a critical region proof rule *which allows the variables owned by a resource to appear in local assertions of a process, as long as they are not modified in "other processes."*

Note that the notion of a variable being "modified in other processes" is quite subtle. One might expect that neither x nor a should be regarded as being modified in the "other process" because the other process does not have direct access to them. Any modification happens only inside critical regions. So, the modification actions cannot be attributed to the process. Rather they should be charged to the resource, with the understanding that entering critical sections adds the access rights of the resource to the process. The putative Syntactic Control of Interference framework we alluded to above would treat the variables in that way.

To handle these issues, we generalize the active versus passive free variable distinction inherited from [29, 36] to *total* versus *partial* ownership of the free variables [6, 8]. (It has become conventional to call such ownership constraints "permissions." We continue to use that terminology even though we regard it as misleading.) A total permission for a variable allows writing to the variable (in other words, an *active* use) and a partial permission allows only reading (a *passive* use). In the algebra of *fractional permissions*, a total permission is denoted by 1 and a partial permission by some non-zero fraction. The use of permissions gives us more powerful control over variable usage because fractional permissions can be combined, possibly leading to a total permission, which then allows writing.

Returning to our example in Table 1, we can define the the resource r to contain 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 $b^{\frac{1}{2}}$ are 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 use a and b in their local assertions because such usage is passive. When the left process enters its critical region, its local permissions are combined with those owned by the resource, leading to the set of permissions x^1, a^1, $b^{\frac{1}{2}}$. This allows the critical region to modify x and a, while only reading is permitted for b. The right process is similar. This provides a *compositional description* of the variable usage in the example, eliding the references to "other processes."

In the following sections, we formalize the system of Syntactic Control of Interference with permissions and use it to formulate the rules of sequential as well as concurrent Separation Logic.

[1] In Brookes's variant of the Owicki-Gries-O'Hearn system [11], a and b need not be included among the owned variables of the resource. Thus, Brookes's logic is subtly more general than the original Concurrent Separation Logic.

3. Sequential Separation Logic

Our form of syntactic control is a modified version of Reynolds SCI, using the ideas of permissions for read-only access [6, 8].

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 the following axioms [32]:

(non-zero)	$\forall p, p' \in \mathcal{P}.\, p \oplus p' \neq p$
(top)	$\forall p \in \mathcal{P}.\, \top \oplus p$ is undefined
(divisibility)	$\forall p \in \mathcal{P}.\, \exists p_1, p_2 \in \mathcal{P}.\, p = p_1 \oplus p_2$

A significant case of permission algebras 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.

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 $norm(\Sigma)$. Whenever two variable contexts are combined, as in "Σ_1, Σ_2", one needs to ensure that the combination is well-defined. We say that Σ_1 and Σ_2 are compatible, and denote this fact by $\Sigma_1 \sharp \Sigma_2$.

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 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.

[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.

The following rules will be *admissible rules* in our proof systems (if the premises are derivable then so is the conclusion):

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

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

The substitution rules allow a variable with a *full* permission to be substituted by an expression.

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

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

More generally, for all expressions and assertions, the requirement is that all their free variables must have some permission in Σ. We omit the formal rules for brevity.

We can write down well-formedness rules for commands as well, but we will save a bit of work by combining the well-formedness of commands with program logic, which we look at next. (For completeness, we include the well-formedness rules in Appendix A.)

A judgement of sequential Separation Logic is of the form

$$\Sigma \vdash \{P\}\, C\, \{Q\} \tag{1}$$

which means that:

1. P, C and Q are well-formed phrases in the context Σ, and
2. 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 shown in Table 2. 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:

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

(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 $x^\top$. But 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

$$\frac{\Sigma \vdash P', Q' \ \mathbf{Assert} \quad \Sigma \vdash \{P\}\, C\, \{Q\}}{\Sigma \vdash \{P'\}\, C\, \{Q'\}} \quad \text{if } P' \supset P \text{ and } Q \supset Q'$$

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

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

$$\frac{\Sigma \vdash \{P \wedge B\}\, C_1\, \{Q\} \quad \Sigma \vdash \{P \wedge \neg B\}\, C_2\, \{Q\}}{\Sigma \vdash \{P\}\, \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\, \{Q\}} \qquad \frac{\Sigma \vdash P, Q \ \mathbf{Assert} \quad \Sigma, x^{\top} \vdash \{P\}\, C\, \{Q\}}{\Sigma \vdash \{P\}\, \mathbf{local}\ x\ \mathbf{in}\ C\, \{Q\}}$$

Table 2. Proof rules of sequential Separation Logic

$$\textit{ASSIGN} \ \frac{\Sigma \vdash x \ \mathbf{Var} \quad \Sigma \vdash E \ \mathbf{Exp} \quad \Sigma \vdash P \ \mathbf{Assert}}{\Sigma \mid \Gamma \vdash \{P[E/x]\}\, x := E\, \{P\}} \qquad \textit{COND} \ \frac{\Sigma \mid \Gamma \vdash \{P \wedge B\}\, C_1\, \{Q\} \quad \Sigma \mid \Gamma \vdash \{P \wedge \neg B\}\, C_2\, \{Q\}}{\Sigma \mid \Gamma \vdash \{P\}\, \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\, \{Q\}}$$

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

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

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

$$\textit{AUXILIARY} \ \frac{\Sigma \vdash P, Q \ \mathbf{Assert} \quad \Sigma, X^{\top} \mid \Gamma \vdash \{P\}\, C\, \{Q\}}{\Sigma \mid \Gamma \vdash \{P\}\, C \setminus X\, \{Q\}} \quad \text{if } X \text{ is auxiliary for } C$$

Table 3. Proof rules of Concurrent Separation Logic

same force as the usual side condition "C does not modify any free variables of R" in the conventional formulation of Separation Logic.

As an example, using the fractional permission algebra, we can derive the inference using *FRAME*:

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

4. Concurrent Separation Logic

In this section, we formalize the rules of O'Hearn's Concurrent Separation Logic, treating Hoare-style resources and conditional critical regions. The context-free syntax of the commands is:

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

Note that the resource declarations include permission contexts Σ for the variables associated with them. The notation enhances that of Owicki and Gries [31] and O'Hearn [25], who list only variable names with resource declarations. In Section 6, we present an inference algorithm that allows the resource declarations to be written simply in the form $\mathbf{resource}\ r\ \mathbf{in}\ C$ and finds the appropriate permission contexts Σ to be used with them, avoiding the annotation burden for the programmer.

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

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

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.

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 a "resource invariant" formula R_i which is a *precise* assertion [25] and satisfies $\Sigma_i \vdash R_i$ **Assert**. This means 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, see the rules for assignment and conditional commands in Table 3. The resource contexts do not play any rule in the sequential fragment of the programming language.

The proof rule for parallel composition is the rule *PAR*. As one would expect, the variable context of the composite command, Σ_1, Σ_2, needs to be split into separate portions Σ_1 and Σ_2, 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 (the *CRIT* rule). The body of the critical region, C, can use the combined variable contexts of the process and the resource, Σ and Σ_0 respectively. 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".

The rule for the resource declaration is *RESOURCE*. 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 any critical regions.

Finally, the rule *AUXILIARY*, which is similar to the rule for local variable declaration in its form, allows a set of variables $X = \{x_1, \ldots, x_n\}$ to be deleted from a command C along with all assignments to them, provided they are "auxiliary", i.e., each free occurrence in C of a variable from X is in an assignment whose left hand side also belongs to X. The notation $C \setminus X$ denotes the command obtained by deleting all the assignments to variables in X. Note that all the variables in X are assigned the $\top$ permission in the second premise. This guarantees that the variables do not occur in Σ or the permission contexts in Γ.

4.1 Comparison with Owicki-Gries-O'Hearn system

O'Hearn's version of Concurrent Separation Logic [25] is based on the Owicki-Gries system [31] 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.

```
{$57 \mapsto -$}
**resource** $r_1(p^{\frac{1}{2}})$ **in**
  **resource** $r_2(p^{\frac{1}{2}})$ **in begin**
      **with** $r_1$ **do** (**with** $r_2$ **do** p := 0 **od**); [57] := 3 **od**
    || **with** $r_2$ **do** (**with** $r_1$ **do** p := 1 **od**); [57] := 4 **od**
  **end**
{$57 \mapsto -$}
```

Table 4. "Problematic program" due to Berdine and Reynolds

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 variable 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 $\frac{1}{2}$.

However, our version of the Concurrent Separation Logic is more expressive. By associating permission contexts with resources, we make it possible for the permission to be combined in nested critical regions. For example, consider the program fragment shown in Table 4 due to Berdine and Reynolds [35]. The purpose of the two resources r_1 and r_2 is to achieve mutual exclusion to a shared data structure, in this case just the location 57. If the specification has a proof in Concurrent Separation Logic, the race-freedom property of the logic guaruantees that only one process can potentially access the memory location 57 at any given time. A proof can be given in our version of the logic using the following resource invariants:

$$\begin{array}{lcl} R_1 & = & (p = 0 \wedge 57 \mapsto -) \vee (p \neq 0 \wedge \mathbf{emp}) \\ R_2 & = & (p = 0 \wedge \mathbf{emp}) \vee (p \neq 0 \wedge 57 \mapsto -) \end{array}$$

Note that $R_1 \star R_2$ is equivalent to $57 \mapsto -$. So, both the pre-condition and the post-condition can be rewritten to $R_1 \star R_2$.

What makes the proof work is the idea that the permissions for the variable p are split across the two resources. So, a process can modify it only by entering critical regions for both the resources. This form of split-permissions for variables is not available in the Owicki-Gries-O'Hearn system.

Brookes [11], in his effort to prove the soundness of Concurrent Separation Logic, defined a variant of the original system which is subtly more general. Unfortunately, the generalization proved to be unsound. However, all the valid proofs that can be carried out in Brookes's system can be represented in our system. A detailed comparison with Brookes's system, along with soundness issues, appears in Appendix B.

4.2 Comparison with "Variables as Resource" systems

Parkinson et al. [32] and Brookes [13] 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 $\Sigma = (x_1^{p_1}, \dots, x_n^{p_n})$, 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 [32] 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 a 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.

5. Semantics and soundness

The standard proof of soundness for sequential Separation Logic is due to Yang and O'Hearn [39]. Bornat et al. [6] have extended it to deal with permissions. The soundness proof of the original Concurrent Separation Logic was provided by Brookes [11] using novel denotational methods. Brookes [13] has also used these methods to prove the soundness of the "variables as resource" system. Since then, other proofs of soundness have appeared. See [38] for an overview. We regard Brookes's semantics as the canonical one since it is denotationally based and allows easy extensions and adaptations.

In this section, we discuss how the presentation of Separation Logic using the SCI principles impacts the semantics. We regard the SCI judgements for phrases and specifications as a form of type system, and use the approach of "Church typing" to define the semantics, i.e., we regard well-formedness judgement $\Sigma \vdash C\ \mathbf{Comm}$ and $\Sigma \mid \Gamma \vdash C\ \mathbf{Comm}$ as a form of typing for C and interpret C using denotations that are *appropriate* for the specified context $\Sigma \mid \Gamma$. It is also possible to conceive of a "Curry typing" semantics where the commands are interpreted without regard to their contexts of well-formedness, and the well-formed judgements are given a logical meaning as properties of the untyped denotations. However, we follow the Church typing approach here because it seems more natural.

5.1 Sequential Separation Logic

A *state* is modelled as a pair (s, h) of a "store" and a "heap," which are finite partial functions from, respectively, variables and addresses. To keep track of permissions, we define them to map their arguments to *pairs of values and permissions*:

$$\begin{array}{rcl} \mathrm{Store} & = & \mathrm{Vars} \rightharpoonup \mathrm{Val} \times \mathcal{P} \\ \mathrm{Heap} & = & \mathrm{Addr} \rightharpoonup \mathrm{Val} \times \mathcal{P} \end{array}$$

We refer to such maps as *permissive store* and *permissive heap* respectively, and both kinds of maps generically as *permissive maps*. Two permissive maps ϕ_1 and ϕ_2 are said to be *compatible*, denoted $\phi_1 \sharp \phi_2$, if, for all arguments common to both of their domains, they agree on values and provide compatible permissions. More formally, $\phi_1 \sharp \phi_2$ iff:

$$\phi_1(x) = (v, p) \wedge \phi_2(x) = (v', p') \implies v = v' \wedge p \oplus p' \text{ is defined}$$

If ϕ_1 and ϕ_2 are compatible, their joining operation is denoted $\phi_1 \cdot \phi_2$ (which combines permissions whenever both ϕ_1 and ϕ_2 are defined). It is extended to states by defining $(s_1, h_1) \cdot (s_2, h_2) = (s_1 \cdot s_2,\ h_1 \cdot h_2)$.

Given a variable context Σ with $norm(\Sigma) = (x_1^{p_1}, \dots, x_n^{p_n})$, a store s is said to be *of type* Σ if $\operatorname{dom} s = \{x_1, \dots, x_n\}$ and the permission component of $s(x_i)$ is p_i for every i. It is easy to see that, whenever $\Sigma_1 \sharp \Sigma_2$, any stores s_1 of type Σ_1 and s_2 of type Σ_2 are compatible, and $s_1 \cdot s_2$ is of type Σ_1, Σ_2.

A state $\sigma = (s, h)$ is said to be *of type* Σ just if s is of type Σ. The heap component of the state is unconstrained. If (s_1, h_1) and (s_2, h_2) are states of of type Σ_1 and Σ_2 respectively, $\Sigma_1 \sharp \Sigma_2$ and $h_1 \sharp h_2$, then $(s_1, h_1) \cdot (s_2, h_2)$ is of type Σ_1, Σ_2.

The meaning of a command in the sequential programming language is defined in [39] as a *local state transformer*, i.e., a binary relation $[\![C]\!] \subseteq \mathrm{State} \times \mathrm{State} \uplus \{\mathsf{fault}\}$ satisfying safety monotonicity, termination monotonicity and the frame property. It was extended to permissive states in [6]. While it is not stated there, it is also easy to see that $[\![C]\!]$ always preserves the domain and permission structure of the store. This allows us to define a typed semantics for commands. If $\Sigma \vdash C\ \mathbf{Comm}$ is a well-formedness judgement then its meaning is a relation $[\![C]\!]_\Sigma$ consisting of just the pairs (σ, σ') where both σ and σ' are of type Σ.

DEFINITION 1. *A judgement of the sequential Separation Logic $\Sigma \vdash \{P\}\, C\, \{Q\}$ is valid iff, for all states σ of type Σ satisfying P:*

- $(\sigma, \mathsf{fault}) \notin [\![C]\!]_\Sigma$, *and*
- *if $(\sigma, \sigma') \in [\![C]\!]_\Sigma$ then σ' is of type Σ and satisfies Q.*

THEOREM 2 (Soundness). *Every derivable judgement of sequential Separation Logic is valid.*

The proof is by induction on the derivation of the judgement. Consider the FRAME rule as a significant example. Let σ be a state of type Σ, Σ' satisfying $P \star R$. Then σ can be written as $\sigma_1 \cdot \sigma_0$ where σ_1 is of type Σ and satisfies P and σ_0 is of type Σ' and satisfies R. Then by inductive hypothesis, $\Sigma \vdash \{P\}\, C\, \{Q\}$ is valid. Hence $[\![C]\!]_\Sigma$ is safe for σ_1 and, whenever $(\sigma_1, \sigma_1') \in [\![C]\!]_\Sigma$, σ_1' is of type Σ and satisfies Q. So, by the safety monotonicity and frame properties, $[\![C]\!]_{\Sigma,\Sigma'}$ is safe for σ, and $(\sigma, \sigma') \in [\![C]\!]_{\Sigma,\Sigma'}$ implies σ' is of type Σ, Σ' and σ' satisfies $Q \star R$.

5.2 Concurrent Separation Logic

The denotational semantics of commands in the concurrent programming language is given in two stages. First, commands are interpreted as *traces*, i.e., stylized sequences of actions. Second, these traces are described by their effect on states as state transitions. It is not possible to interpret the commands directly as state transitions, because such transitions only relate initial and final states whereas parallel composition makes intermediate states visible.

Trace semantics

A *pre-action* (or an untyped action) is a syntactic token given by the syntax:

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

As in [11], δ is a do-nothing or idle action, $x = v$ denotes the action of reading the variable x, $x := v$ denotes the action of writing to 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 respectively. The token abort denotes the action of aborting a computation in case of an error.

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

We model the actions and action traces "appropriate" for a syntactic context $\Sigma \mid \Gamma$ as a form of typing. First of all, the contexts enable certain actions and prohibit others. A variable action $x = v$ or $x := v$ would only be possible in a context that contains x with requisite permissions. The resource actions $\mathit{try}(r)$ and $\mathit{acq}(r)$ would only be possible in a 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 resource $r(\Sigma_0)$ from the resource context and adding its variables Σ_0 to the variable context. A $\mathit{rel}(r)$ action has the opposite effect. We represent these effects by a transition relation $\xrightarrow{\lambda}$ on contexts. Finally, when a resource is acquired by a process, it is not available for another acquisition until it is released. At the same time, the type information of the resource should continue to be retained in the context. Therefore, we work with a form of extended contexts where the resources acquired by a trace are marked "busy," by enclosing them in square brackets as $[r(\Sigma)]$.

An *extended context* is a context of the form

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

such that

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

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 of the resources are marked busy. The notation $(\widetilde{\Gamma})^\circ$ denotes the underlying resource context of $\widetilde{\Gamma}$ where all the busy markers are erased.

An *action* is a triple $\langle \Sigma|\widetilde{\Gamma},\ \Sigma'|\widetilde{\Gamma}',\ \lambda\rangle$ consisting of the initial and final contexts and a pre-action that leads from the former to the latter. We write it using the notation $\Sigma|\widetilde{\Gamma} \xrightarrow{\lambda} \Sigma'|\widetilde{\Gamma}'$. The list of actions used in the semantics of the programming language are shown in Table 6. There are no constraints on the actions for reading and writing heap locations because the access to heap locations is controlled in the programming logic rather than the syntax.

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

Table 6. Actions used in traces

A trace is a finite or infinite sequence of the form

$$\Sigma_0|\widetilde{\Gamma}_0 \xrightarrow{\lambda_1} \Sigma_1|\widetilde{\Gamma}_1 \xrightarrow{\lambda_2} \Sigma_2|\widetilde{\Gamma}_2 \xrightarrow{\lambda_3} \cdots$$

If the sequence $\alpha = \lambda_1\lambda_2\cdots$ is finite, we use the notation $\Sigma_0|\widetilde{\Gamma}_0 \xrightarrow{\alpha} \Sigma_n|\widetilde{\Gamma}_n$ to denote the corresponding trace. If it is infinite, we use the notation $\Sigma_0|\widetilde{\Gamma}_0 \xrightarrow{\alpha} \infty$. We also use the notation $\Sigma_0|\widetilde{\Gamma}_0 \xrightarrow{\alpha} \cdot$ for both finite and infinite traces, and say that the pretrace α is *enabled* in the context $\Sigma_0|\widetilde{\Gamma}_0$.

For defining the meaning of parallel composition, we define an operation of *interleaving* two traces. Suppose α_1 and α_2 are two traces with α_1 enabled in a context $\Sigma_1|\widetilde{\Gamma}_1$ and α_2 enabled in a context $\Sigma_2|\widetilde{\Gamma}_2$. Then $\widetilde{\Gamma}_1$ and $\widetilde{\Gamma}_2$ should have the same underlying resource contexts, i.e., $(\widetilde{\Gamma}_1)^\circ = (\widetilde{\Gamma}_2)^\circ$, and they should mark disjoint sets of resources as busy. Then the resource context obtained by marking the busy resources of both $\widetilde{\Gamma}_1$ and $\widetilde{\Gamma}_2$ is denoted $\widetilde{\Gamma}_1 \wedge \widetilde{\Gamma}_2$. Interleaving is only possible for traces α_1 and α_2 such that $\widetilde{\Gamma}_1$ and $\widetilde{\Gamma}_2$ are in this form and $\Sigma_1, \Sigma_2 \mid \widetilde{\Gamma}_1 \wedge \widetilde{\Gamma}_2$ is well-defined.

Two actions λ_1 and λ_2 are said to *interfere*, written $\lambda_1 \not\parallel \lambda_2$, 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 fairmerges* of $\Sigma_1|\widetilde{\Gamma}_1 \xrightarrow{\alpha_1} \cdot$ and $\Sigma_2|\widetilde{\Gamma}_2 \xrightarrow{\alpha_2} \cdot$ is a set of traces of type $\Sigma_1, \Sigma_2|\widetilde{\Gamma}_1 \wedge \widetilde{\Gamma}_2 \longrightarrow \cdot$ given by induction on the lengths of α_1 and α_2:

$$\begin{array}{l}
\alpha_1 \parallel \epsilon = \{\alpha_1\} \\
\epsilon \parallel \alpha_2 = \{\alpha_2\} \\
(\lambda_1\alpha_1) \parallel (\lambda_2\alpha_2) = \\
\quad \{\, \mathit{abort} \mid \lambda_1 \not\parallel \lambda_2 \} \cup \\
\quad \{\, \Sigma_1, \Sigma_2 \mid \widetilde{\Gamma}_1 \wedge \widetilde{\Gamma}_2 \xrightarrow{\lambda_1} \Sigma'_1, \Sigma_2 \mid \widetilde{\Gamma}'_1 \wedge \widetilde{\Gamma}_2 \xrightarrow{\beta} \cdot \\
\qquad \text{such that } \beta \in \alpha_1 \parallel (\lambda_2\alpha_2) \,\} \cup \\
\quad \{\, \Sigma_1, \Sigma_2 \mid \widetilde{\Gamma}_1 \wedge \widetilde{\Gamma}_2 \xrightarrow{\lambda_2} \Sigma_1, \Sigma'_2 \mid \widetilde{\Gamma}_1 \wedge \widetilde{\Gamma}'_2 \xrightarrow{\beta} \cdot \\
\qquad \text{such that } \beta \in (\lambda_1\alpha_1) \parallel \alpha_2 \,\}
\end{array}$$

This definition is a typed version of the notion of mutex fairmerges in Brookes [11]. Note that the typing information of traces obviates the need to consider possible interference via variable usage.

The definition is extended to *sets* of traces in the natural way. If T_1 and T_2 are trace sets enabled in contexts $\Sigma_1|\widetilde{\Gamma}_1$ and $\Sigma_2|\widetilde{\Gamma}_2$ then the trace set $T_1 \parallel T_2$, obtained as the union of all $\alpha_1 \parallel \alpha_2$ for all $\alpha_1 \in T_1$ and $\alpha_2 \in T_2$ is enabled in the context $\Sigma_1, \Sigma_2 \mid \widetilde{\Gamma}_1 \wedge \widetilde{\Gamma}_2$.

A *(well-bracketed) trace for an extended context* $\Sigma \mid \widetilde{\Gamma}$ is either abort, a finite trace α such that $\Sigma \mid \widetilde{\Gamma} \xrightarrow{\alpha} \Sigma \mid \widetilde{\Gamma}$, or an infinite trace whose every finite prefix can be extended to a well-bracketed finite trace. The terminology is motivated by thinking of the $\mathit{acq}(r)$ and $\mathit{rel}(r)$ actions as brackets. A trace set T is a *(well-bracketed)*

$$\llbracket x \rrbracket_\Sigma = \{\, (x = v, v) \mid v \text{ Value} \,\}$$
$$\llbracket E_1 + E_2 \rrbracket_\Sigma = \{\, (\rho_1\rho_2, v_1 + v_2) \mid (\rho_1, v_1) \in \llbracket E_1 \rrbracket_\Sigma \wedge (\rho_2, v_2) \in \llbracket E_2 \rrbracket_\Sigma \,\}$$

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

Table 5. Trace semantics of phrases

trace set for context $\Sigma \mid \widetilde{\Gamma}$ if each trace in T is a well-bracketed trace for the context.

LEMMA 3 (Weakening of contexts). *If α is a trace for an extended context $\Sigma|\widetilde{\Gamma}$, and $\Sigma, \Sigma'|\widetilde{\Gamma}, \widetilde{\Gamma}'$ is a longer well-defined extended context, then α is a trace for $\Sigma, \Sigma'|\widetilde{\Gamma}, \widetilde{\Gamma}'$.*

LEMMA 4 (Parallel composition preserves contexts). *If α_1 and α_2 are traces for extended contexts $\Sigma_1|\widetilde{\Gamma}$ and $\Sigma_2|\widetilde{\Gamma}$ respectively, and $\Sigma_1, \Sigma_2|\widetilde{\Gamma}$ is a well-defined extended context then $\alpha_1 \parallel \alpha_2$ is a trace set for the context $\Sigma_1, \Sigma_2|\widetilde{\Gamma}$.*

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

- The meaning of an expression $\Sigma \vdash E\ \mathbf{Exp}$ is a set of pairs (ρ, v) where ρ is an action trace of type $\Sigma| \xrightarrow{\rho} \Sigma|$ (i.e., a 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 $\llbracket E \rrbracket_\Sigma$.
- The meaning of a command $\Sigma \mid \Gamma \vdash C\ \mathbf{Comm}$ is a set of traces ρ for the context $\Sigma|\Gamma$. We denote it by $\llbracket C \rrbracket_{\Sigma|\Gamma}$.

The semantics is defined in the standard fashion [11]. However, it is defined by induction on the *derivations* of well-formedness judgements $\Sigma \vdash E\ \mathbf{Exp}$ and $\Sigma \mid \Gamma \vdash C\ \mathbf{Comm}$, instead of induction on the structure of terms. We show the meanings of sample phrases in Table 5. The notation $\llbracket E \rrbracket_\Sigma \restriction v$ denotes the set of traces $\{\, \rho \mid (\rho, v) \in \llbracket E \rrbracket_\Sigma \,\}$. The notations $\rho \setminus x$ and $\rho \setminus r$ remove the actions mentioning x and r respectively from ρ.

THEOREM 5 (Type soundness of trace semantics). *The meaning of command $\Sigma \mid \Gamma \vdash C\ \mathbf{Comm}$ is a (well-bracketed) trace set for the context $\Sigma \mid \Gamma$. Likewise, for every (ρ, v) in the meaning of an expression $\Sigma \vdash E\ \mathbf{Exp}$, ρ is a (well-bracketed) trace for the context $\Sigma \mid$.*

Local state semantics

A *state* for a concurrent program is a triple (s, h, A) where s is a permissive store, h is a permissive heap and A is a set of resource names (deemed to have been acquired by the process). We also use an error state $\mathbf{abort}$. The types for states will be annotated extended contexts of the form $\Sigma \mid \widetilde{\Gamma}$ where the resources are *annotated* with resource invariants as in $r(\Sigma_0) : R$. It is a characteristic of Brookes's semantics for Concurrent Separation Logic that the resource invariants play a central role in the state transition semantics.

A *state of type* $\Sigma \mid \widetilde{\Gamma}$ is either $\mathbf{abort}$ or a normal state (s, h, A) where s is a store of type Σ, h is a heap, and A is a subset of the resources marked busy in $\widetilde{\Gamma}$.

We can interpret actions (and action traces) of type $\Sigma \mid \widetilde{\Gamma} \longrightarrow \Sigma' \mid \widetilde{\Gamma}'$ as state transformations that transform states of type $\Sigma \mid \widetilde{\Gamma}$ to states of type $\Sigma' \mid \widetilde{\Gamma}'$. For actions of type $\Sigma \mid \widetilde{\Gamma} \longrightarrow \Sigma \mid \widetilde{\Gamma}$ (where the state type is unchanged), the state transformations are as follows:

$$\begin{array}{l}
(s,h,A) \xrightarrow{\delta} (s,h,A) \\
(s,h,A) \xrightarrow{\mathit{abort}} \mathbf{abort} \\
(s,h,A) \xrightarrow{x=v} (s,h,A) \quad \text{iff} \quad \exists p.\, s(x) = (v,p) \\
(s,h,A) \xrightarrow{x:=v} (s[x \mapsto (v,\top)], h, A) \quad \text{iff} \quad \exists v_0.\, s(x) = (v_0, \top) \\
(s,h,A) \xrightarrow{[l]=v} (s,h,A) \quad \text{iff} \quad \exists p.\, h(l) = (v,p) \\
(s,h,A) \xrightarrow{[l]=v} \mathbf{abort} \quad \text{iff} \quad l \notin \mathrm{dom}(h) \\
(s,h,A) \xrightarrow{[l]:=v} (s, h[l \mapsto (v,\top)], A) \quad \text{iff} \quad \exists v_0.\, h(l) = (v_0, \top) \\
(s,h,A) \xrightarrow{[l]:=v} \mathbf{abort} \quad \text{iff} \quad l \notin \mathrm{dom}(h) \\
(s,h,A) \xrightarrow{\mathit{try}(r)} (s,h,A)
\end{array}$$

For an acq action of type

$$\Sigma \mid \widetilde{\Gamma}, r(\Sigma_0) \colon R \xrightarrow{\mathit{acq}(r)} \Sigma, \Sigma_0 \mid \widetilde{\Gamma}, [r(\Sigma_0) \colon R]$$

the transformations are given by:

$$\begin{array}{c}
(s,h,A) \xrightarrow{\mathit{acq}(r)} (s \cdot s_0,\ h \cdot h_0,\ A \cup \{r\}) \\
\text{iff} \quad (s_0, h_0) \models R,\ s \sharp s_0,\ h \sharp h_0
\end{array}$$

For a rel action of type

$$\Sigma, \Sigma_0 \mid \widetilde{\Gamma}, [r(\Sigma_0) \colon R] \xrightarrow{\mathit{rel}(r)} \Sigma \mid \widetilde{\Gamma}, r(\Sigma_0) \colon R$$

the transformations are:

$$\begin{array}{l}
(s \cdot s_0,\ h \cdot h_0,\ A \uplus \{r\}) \xrightarrow{\mathit{rel}(r)} (s,h,A) \quad \text{iff} \quad (s_0,h_0) \models R \\
(s,h,A) \xrightarrow{\mathit{rel}(r)} \mathbf{abort} \quad \text{iff} \quad \forall h_0 \subseteq h.\, \neg(s,h_0) \models R
\end{array}$$

The key property of these transformations, inherited from Brookes [11], is that the transitions for $\mathit{acq}(r)$ *extend* the current state with an *arbitrary* state of the resource satisfying the resource invariant R. The condition $s \sharp s_0$ ensures that the values of any common variables agree. The transitions for $\mathit{rel}(r)$ do the opposite: they remove the state of the resource from the current state. If and when the resource is reacquired in a future action, the state of the resource obtained may bear no relationship to the state previously released.

In fact, since other processes can intervene in the interim, nothing more can be assumed about the reacquired state of the resource.

LEMMA 6 (Type soundness of traces). *Given a trace α of type $\Sigma \mid \widetilde{\Gamma} \xrightarrow{\alpha} \Sigma' \mid \widetilde{\Gamma}'$ and a state (s, h, A) of type $\Sigma \mid \widetilde{\Gamma}$, if $(s, h, A) \xrightarrow{\alpha} (s', h', A')$ then (s', h', A') is of type $\Sigma' \mid \widetilde{\Gamma}'$.*

Soundness

DEFINITION 7 (Validity). *A judgement $\Sigma \mid \Gamma \vdash \{P\}\ C\ \{Q\}$ is valid iff, for all well-bracketed traces α for the context $\Sigma|\Gamma$ in $[\![C]\!]_{\Sigma|\Gamma}$, all local states $(s, h, \emptyset)$ and σ' of type $\Sigma|\Gamma$,*

$$\begin{aligned}(s, h) \models P \ \wedge\ (s, h, \emptyset) \xrightarrow{\alpha} \sigma' \implies \\ \exists s', h'.\, \sigma' = (s', h', \emptyset) \wedge \sigma' \models Q\end{aligned}$$

THEOREM 8 (Soundness). *Every provable judgement of concurrent SCI Separation Logic is valid.*

Standard semantics

In addition to the semantics defined above, which is with respect to a program proof, traces can be interpreted as actions on global states. The relation is denoted $(s, h, A) \overset{\alpha}{\Longrightarrow} (s', h', A')$ and is similar to an untyped version of the local state transition semantics, except that the rules for *acq* and *rel* actions are modified as follows:

$$\begin{array}{lll}(s, h, A) \overset{acq(r)}{\Longrightarrow} (s,\ h,\ A \cup \{r\}) & & \text{if } r \notin A \\ (s, h, A \uplus \{r\}) \overset{rel(r)}{\Longrightarrow} (s, h, A \setminus \{r\}) & & \text{if } r \in A\end{array}$$

This relation corresponds to running a process on the global state without any interference from any other processes.

The following result says that the standard semantics obtained by executing traces on the global state corresponds to the local state semantics defined above. The notation $\mathsf{inv}(\Gamma)$ stands for the conjunction of all the resource invariants in Γ.

THEOREM 9. *Let $(s, h, \emptyset)$ be a global state and $\Sigma \mid \Gamma$ a context. Suppose the state (s, h) can be split as $(s_1, h_1) \cdot (s_2, h_2)$ where $(s_1, h_1, \emptyset)$ is of type $\Sigma \mid \Gamma$ and $(s_2, h_2) \models \mathsf{inv}(\Gamma)$.*

- *If $(s, h, \emptyset) \overset{\alpha}{\Longrightarrow}$ **abort** then $(s_1, h_1, \emptyset) \xrightarrow{\alpha}$ **abort**.*
- *If $(s, h, \emptyset) \overset{\lambda}{\Longrightarrow} (s', h', \emptyset)$ then either*
  - *$(s_1, h_1, \emptyset) \xrightarrow{\lambda}$ **abort**, or*
  - *(s', h') can be split as $(s'_1, h'_1) \cdot (s'_2, h'_2)$ such that $(s'_1, h'_1, \emptyset)$ is of some type $\Sigma' \mid \Gamma'$, $(s_1, h_1, \emptyset) \xrightarrow{\lambda} (s'_1, h'_1, \emptyset)$ and $(s'_2, h'_2) \models \mathsf{inv}(\Gamma')$.*

6. Permission inference

In this section, we investigate the problem of permission inference. We construct an algorithm which, given a program and a proof outline with no variable contexts listed with resources, fills them in if at all possible in accordance with the rules of SCI Separation Logic.

We restrict our attention to the permission algebra of fractional permissions, the real interval $(0, 1]$ with addition as the partial binary operation. For theoretical simplicity, we extend the algebra to include 0 as an abnormal permission value, indicating that the resource or the process possesses no permission for the variable, and extend addition to 0 in the standard way. An element of $[0, 1]$ is referred to as an "extended permission."

A normal form context with n variables and m resources is of the form

$$\begin{array}{l}x_1^{p_{01}}, \ldots, x_n^{p_{0n}} \mid \\ \quad r_1(x_1^{p_{11}}, \ldots, x_n^{p_{1n}}) : R_1,\ \ldots,\ r_m(x_1^{p_{m1}}, \ldots, x_n^{p_{mn}}) : R_m \vdash\end{array}$$

where each p_{ij} is an extended permission, with the index i corresponding to the owner of the permission (0 for the process or "**self**," $1, \ldots, n$ for the shared resources), and the index j corresponding to the variable. We represent all the data in the context by two finite functions:

$$\Delta : \mathrm{Vars} \to \mathrm{Owners} \to [0, 1] \qquad \Upsilon : \mathrm{Resources} \to \mathrm{Invariant}$$

where $\mathrm{Owners} = \{\mathbf{self}\} \uplus \mathrm{Resources}$, and Δ satisfies

$$\Sigma_{o \in \mathrm{Owners}}\, (\Delta\, v\, o) \le 1$$

The sets Vars and Resources include all the variable and resource names appearing in the program fragment being analyzed.

Using these notations, the proof system of SCI Separation Logic can be rewritten using judgements of the form:

$$\begin{array}{lll}\Delta|\Upsilon \vdash E\ \mathbf{Exp} & \Delta|\Upsilon \vdash P\ \mathbf{Assert} & \text{(Passive)} \\ \Delta|\Upsilon \vdash x\ \mathbf{Var} & \Delta|\Upsilon \vdash \{P\}\ C\ \{Q\} & \text{(Active)}\end{array}$$

(where the first three forms have the resource context Υ added for uniformity in discussion). For example, the parallel composition rule is rewritten as:

$$\frac{\Delta_1 \mid \Upsilon \vdash \{P_1\}\ C_1\ \{Q_1\} \quad \Delta_2 \mid \Upsilon \vdash \{P_2\}\ C_2\ \{Q_2\}}{\Delta \mid \Upsilon \vdash \{P_1 \star P_2\}\ C_1 \parallel C_2\ \{Q_1 \star Q_2\}}$$

$$\begin{array}{ll}\text{if} & \Delta_1\, v\, o = \Delta_2\, v\, o = \Delta\, v\, o \text{ for all } o \ne \mathbf{self} \\ & \Delta\, v\, \mathbf{self} = \Delta_1\, v\, \mathbf{self} + \Delta_2\, v\, \mathbf{self} \le 1\end{array}$$

We also use abbreviated rules for the passive judgements:

$$\frac{}{\Delta \mid \Upsilon \vdash E\ \mathbf{Exp}} \quad \text{if } \forall v \in FV(E), \Delta\, v\, \mathbf{self} > 0$$

$$\frac{}{\Delta \mid \Upsilon \vdash P\ \mathbf{Assert}} \quad \text{if } \forall v \in FV(P), \Delta\, v\, \mathbf{self} > 0$$

Define a *write-proof* as a proof where the side conditions of passive judgements are ignored. Since the passive judgements involve variable reading, this means that the permissions needed for variable reading are not checked. However, the permissions needed for variable writing are still checked, hence the name.

Define a "pre-judgement" as a judgement with variable contexts Δ erased, i.e., a judgement of one of the forms:

$$\begin{array}{lll}\Upsilon \vdash E\ \mathbf{Exp} & \Upsilon \vdash P\ \mathbf{Assert} & \text{(Passive)} \\ \Upsilon \vdash x\ \mathbf{Var} & \Upsilon \vdash \{P\}\ C\ \{Q\} & \text{(Active)}\end{array}$$

A "pre-rule" is an SCI Separation Logic rule with variable contexts erased. A "pre-inference" is an instance of a pre-rule and a "pre-proof" is a derivation made up of pre-inferences. The *erasure* of a judgement, rule, inference or proof X is a pre-judgement, pre-rule, pre-inference or pre-proof (respectively) denoted X^0, obtained by erasing all the variable contexts. In that case, we say that X "erases" to X^0 or X "extends" X^0.

The problem of permission inference is now stated formally as follows:

Given a pre-proof P^0, is there a proof P whose erasure is P^0?

The algorithm described below answers the question. Moreover, if the answer is yes, it produces a maximally permissive proof $P^{\max}$ that extends P^0.

We regard proof trees as *formal trees*, i.e., graphs satisfying the tree conditions, labelled by judgements. P^0 and P are different labellings of the *same* formal tree. We use the notation $(P^0)_N$ and $(P)_N$, respectively, to refer to the judgements labeling a node N of the formal tree.

We use a few auxiliary concepts:

- A *permission restriction* Φ is an assignment

  $$[v_1 : O_1,\ \ldots,\ v_k : O_k]$$

  where $v_i \in \mathrm{Vars}$ and $O_i \subseteq \mathrm{Owners}$. We also feel free to treat Φ as a partial function of type $\mathrm{Vars} \rightharpoonup \mathcal{P}(\mathrm{Owners})$. Such a

```
$\{57 \mapsto -\}$
resource $r_1$ in
  resource $r_2$ in begin
      with $r_1$ do (with $r_2$ do p := 0 od); [57] := 3 od
    || with $r_2$ do (with $r_1$ do p := 1 od); [57] := 4 od
  end
$\{57 \mapsto -\}$
```

Table 7. "Problematic program" due to Berdine and Reynolds

Φ represents the condition that, for each of the variables v_i, the owners in O_i share the full permission for v_i. A variable v_i will occur in a permission restriction exactly when the program phrase being described contains an assignment to v_i. The corresponding O_i lists all the owners that can contribute permissions required for that assignment to be legal. Formally, the satisfaction of a permission restriction by a variable context is defined as:

$$\Delta \models \Phi \iff \forall (v_i : O_i) \in \Phi.\ \Sigma_{o \in O_i}\,(\Delta\ v_i\ o) = 1$$

Note that $\Delta\ v_i\ o$ must be 0 for all owners outside O_i. There are *no* constraints on Δ for the other variables not mentioned in Φ.

- We define a *permission ordering* on variable contexts $\Delta \preceq \Delta'$ by the rule:

$$\Delta\ v\ o > 0 \Longrightarrow \Delta'\ v\ o > 0$$

We say that Δ' is "more permissive" than Δ. The intuition is that Δ' has non-zero permissions for at least as many combinations as Δ.

A permission restriction $\Phi = [v_1 : O_1, \ldots, v_k : O_k]$ is satisfiable only if every O_i is nonempty. In that case, a maximally permissive variable context satisfying Φ can be defined as follows:

$$\Delta^{\max}\ v\ o = \begin{cases} 1/\#O, & \text{if } (v : O) \in \Phi \wedge o \in O \\ 0, & \text{if } (v : O) \in \Phi \wedge o \notin O \\ 1/(\#\text{Owners} + 1), & \text{if } v \notin \text{dom}\,\Phi \end{cases}$$

where $\#S$ denotes the size of the set S. In other words, a full permission is apportioned among all the owners permitted by Φ or, if Φ imposes no restriction, then a partial permission is apportioned among all owners.

Our algorithm for permission inference is a two-phase algorithm. The first phase traverses a pre-proof leaves to root ("bottom-up" in the syntax tree), and computes, at each inference step, the permission restriction that must be satisfied by any write-proof. If any permission restriction computed in this phase is unsatisfiable then there is no proof corresponding to the pre-proof. The second phase traverses the pre-proof from the root to leaves ("top-down" in the syntax tree), computing variable contexts that extend the pre-proof to a *maximally permissive write-proof* in the sense of the pre-order $\preceq$. The maximally permissive write-proof is then checked to verify that it contains non-zero permissions for all the passive uses of variables.

We illustrate the algorithm using the "problematic program" of Berdine and Reynolds [35], reproduced in Table 7 for ease of reference: Let Υ stand for the resource context $r_1 : R_1,\ r_2 : R_2$.

The first phase of the algorithm traverses the pre-proof leaves to root and computes, at each inference step, the permission restrictions needed to extend the pre-proof to a write proof. Since the inference steps correspond to program terms, we just show the terms involved in each case.

1. For the variable p, i.e., the inference step concluding $\Upsilon \vdash p\ \mathbf{Var}$, the permission restriction is $\Phi_1 = [p : \{\mathbf{self}\}]$. The total permission must be owned by $\mathbf{self}$ at this point in order to allow assignments to p.
2. For the command $p := 0$, the permission restriction is the same, $\Phi_2 = [p : \{\mathbf{self}\}]$.
3. For the critical section $\mathbf{with}\ r_2\ \mathbf{do}\ p := 0\ \mathbf{od}$, the permission restriction is $\Phi_3 = [p : \{\mathbf{self}, r_2\}]$. This means that both the process and the resource r_2 could have non-zero permission for p. Since the critical section combines the permissions from $\mathbf{self}$ and r_2 to execute the body, this is well-justified.
4. The command $[57] := 3$ does not write to any variables. So, its permission restriction is empty: $\Phi_4 = [\,]$.
5. For the outer critical section
   $$P_1 \equiv \mathbf{with}\ r_1\ \mathbf{do}\ (\mathbf{with}\ r_2\ \mathbf{do}\ p := 0\ \mathbf{od}); [57] := 3\ \mathbf{od}$$
   r_1 is added to the restriction: $\Phi_5 = [p : \{\mathbf{self}, r_1, r_2\}]$.
6. The second process P_2 similarly has the permission restriction $\Phi_6 = [p : \{\mathbf{self}, r_1, r_2\}]$.
7. For the parallel composition $P_1 \parallel P_2$, the permission restriction is $\Phi_7 = [p : \{r_1, r_2\}]$, i.e., $\mathbf{self}$ is *removed* from the permission restrictions obtained from the component processes.

   Why? In this phase of the algorithm, we are only considering what permissions are needed for writing variables. Since both the processes have permission restrictions for p, that means that they are both writing to p, which is only possible if each of them has 0 as the $\mathbf{self}$ permission for p. (If the first process has non-zero permission for p then, since the second process has the sum of all its permissions for p summing to 1, the total sum of the permissions for p in the parallel composition would exceed 1, which is forbidden.) All the permissions for writing to p in both the processes must be obtained by entering critical regions for the resources.
8. $\mathbf{resource}\ r_2\ \mathbf{in}\ P_1 \parallel P_2$ has the permission context $\Phi_8 = [p : \{r_1, \mathbf{self}\}]$, which is obtained by replacing r_2 in Φ_7 by $\mathbf{self}$. This is justified by noting that the resource declaration allows the process to shift some portion of the permission for p from $\mathbf{self}$ to r_2. Since Φ_7 potentially requires a non-zero permission for p in r_2, Φ_8 must require it in $\mathbf{self}$.
9. $\mathbf{resource}\ r_1\ \mathbf{in}\ \mathbf{resource}\ r_2\ \mathbf{in}\ P_1 \parallel P_2$ has the permission restriction $\Phi_9 = [p : \{\mathbf{self}\}]$, using the same reasoning as in the previous step.

The key observation is the fact that permission restriction Φ_7 for $P_1 \parallel P_2$ does not contain $\mathbf{self}$. This requires us to divide the full permission for p among only the two resources r_1 and r_2.

Since all the permission restrictions computed in phase 1 are satisfiable, we proceed to phase 2 of the algorithm. This phase moves *top-down*, from the root to the leaves, using the permission restrictions computed in the previous phase.

1. For the overall program, the permission restriction is $\Phi_9 = [p : \{\mathbf{self}\}]$. The maximally permissive variable context satisfying Φ_9 is given by $\Delta\ p = [\mathbf{self} : 1]$.
2. The last inference step is of the form:

$$\frac{\begin{array}{lr} \Delta_1 \mid\ \vdash R_1\ \mathbf{Assert} & [\,] \\ \Delta_2 \mid r_1 : R_1 \vdash \{R_2\}\ \mathbf{resource}\ r_2\ \mathbf{in}\ P_1 \parallel P_2\ \{R_2\} & [\Phi_8] \end{array}}{\Delta \mid\ \vdash \{R_1 \star R_2\} \begin{pmatrix} \mathbf{resource}\ r_1\ \mathbf{in} \\ \quad \mathbf{resource}\ r_2\ \mathbf{in} \\ \quad\quad P_1 \parallel P_2 \end{pmatrix} \{R_1 \star R_2\}\ [\Phi_9]}$$

(where the Δ's need to satisfy various side conditions detailed in the formal rules given below). Note that the permission restriction for the first premise is empty because it is a passive judgement. We calculate maximally permissive variable contexts Δ_1 and Δ_2 using Δ (obtained in the previous step) and the permission restrictions for the premises $[\,]$ and Φ_8 calculated in the first phase. Recall that $\Phi_8 = [p : \{r_1, \mathbf{self}\}]$. This implies that $\Delta_2\ p$ should be of the form $[r_1 : \pi_1,\ \mathbf{self} : \pi_s]$ for some non-zero fractions π_1 and π_s such that $\pi_1 + \pi_s = 1$. The precise fractions do not matter, just that they should be non-zero. For instance, we can pick $\pi_1 = \pi_s = \frac{1}{2}$. Δ_1 should be of the form $[p : [\mathbf{self} : \pi_1]]$ because the permission allocated to **self** in the resource invariant should be the permission allocated to r_1 in Δ_2.

3. Moving top-down in the pre-proof, we need to construct the inference:

$$\frac{\begin{array}{lr}\Delta_1' \mid\ \vdash R_2\ \mathbf{Assert} & [\,] \\ \Delta_2' \mid r_1 : R_1,\ r_2 : R_2 \vdash \{\mathbf{emp}\}\ P_1 \parallel P_2\ \{\mathbf{emp}\} & [\Phi_7]\end{array}}{\Delta_2 \mid r_1 : R_1 \vdash \{R_2\}\ \mathbf{resource}\ r_2\ \mathbf{in}\ P_1 \parallel P_2\ \{R_2\}\ [\Phi_8]}$$

where $\Delta_2 = [p : [r_1 : \pi_1,\ \mathbf{self} : \pi_s]]$ is the variable context from the previous step. Proceeding similarly to the previous step, we can calculate that the variable context Δ_2' in the judgement should be of the form $[r_1 : \pi_1',\ r_2 : \pi_2',\ \mathbf{self} : \pi_s']$ such that $\pi_1' = \pi_1$ and $\pi_2' + \pi_s' = \pi_s$. However, the permission restriction Φ_7 only lists r_1 and r_2 for p. Hence, π_s should be 0, and $\pi_2' = \pi_s$. If $\pi_1 = \pi_s = \frac{1}{2}$ was chosen in the previous step, then we obtain $\pi_1' = \pi_2' = \frac{1}{2}$.

We omit the remaining steps, which are straightforward. Note that the main task of the algorithm is now accomplished. The permissions for p in the two resources r_1 and r_2 have been inferred. They are $\frac{1}{2}$ each.

The algorithm for permission inference takes as input a pre-proof P^0, regarded as a labeling function $(P^0)_N$ of a formal tree of nodes. In phase 1, it traverses the tree leaf-to-root and constructs a permission restriction Φ_N for each node N. If the pre-inference for $(P^0)_N$ is of the form

$$X^0 : \frac{\Upsilon_{N_1} \vdash S_{N_1} \quad \cdots \quad \Upsilon_{N_k} \vdash S_{N_k}}{\Upsilon_N \vdash S_N} \tag{2}$$

then the algorithm computes the permission restriction Φ_N for node N as a partial function $\mathcal{F}_R(\Phi_{N_1}, \ldots, \Phi_{N_k})$ of the permission restrictions of its children (antecedents of the pre-inference), satisfying:

Property L0: If each $\mathrm{dom}\,\Phi_{N_i}$ contains exactly the modified free variables of S_{N_i} (i.e., varaibles that occur on the left hand sides of assignments) then $\mathrm{dom}\,\Phi_N$ likewise contains exactly the modified free variables of S_N.

Property L1: For every inference X that extends X^0:

$$X : \frac{\Delta_1 \mid \Upsilon_{N_1} \vdash S_{N_1} \quad \cdots \quad \Delta_k \mid \Upsilon_{N_k} \vdash S_{N_k}}{\Delta \mid \Upsilon_N \vdash S_N}$$

we have $(\bigwedge_{i=1}^k \Delta_i \models \Phi_{N_i}) \implies \Delta \models \Phi_N$ where $\Phi_N = \mathcal{F}_R(\Phi_{N_1}, \ldots, \Phi_{N_k})$.

If, on the other hand, $\mathcal{F}_R(\Phi_{N_1}, \ldots, \Phi_{N_k})$ is undefined then there exists no inference X extending X^0. This case arises only for the variable declaration rule.

LEMMA 10. *Given a pre-proof P^0, if Φ_N is a family of of permission restrictions for the nodes of P^0 produced in phase 1, then, for every write-proof P^w that extends P^0, the variable context Δ_N of $(P^w)_N$ satisfies Φ_N.*

The proof is by induction on the structure of the underlying tree of P^0. Thus, the result holds for all sub-proofs of P^0 as well.

In phase 2, we construct a maximally permissive write-proof that extends P^0 by calculating $\Delta_N^{\max}$ for every node N. For the root node, we choose a maximally permissive Δ satisfying Φ_{root}. Then phase 2 proceeds from the root to leaves, constructing $\Delta_N^{\max}$ for each node using the $\Delta^{\max}$ of the the parent (consequent of the pre-inference) and the permission restrictions computed in phase 1. Specifically, given a pre-inference X^0

$$X^0 : \frac{\Upsilon_{N_1} \vdash S_{N_1} \quad \cdots \quad \Upsilon_{N_k} \vdash S_{N_k}}{\Upsilon_N \vdash S_N}$$

and $\Delta_N^{\max}$ satisfying Φ_N, it computes $\Delta_{N_1}^{\max}, \ldots, \Delta_{N_k}^{\max}$ for the child nodes of N (antecedents of the pre-inference) as a function $\mathcal{G}_R(\Phi_{N_1}, \ldots, \Phi_{N_k}, \Delta_N^{\max})$ of the given $\Delta_N^{\max}$ and the permission restrictions $\Phi_{N_1}, \ldots, \Phi_{N_k}$ of the child nodes, satisfying:

Property L2: $\Delta_{N_1}^{\max} \models \Phi_{N_1}, \ldots, \Delta_{N_k}^{\max} \models \Phi_{N_k}$, and the following is a legal inference that extends X^0:

$$X^{\max} : \frac{\Delta_{N_1}^{\max} \mid \Upsilon_{N_1} \vdash S_{N_1} \quad \cdots \quad \Delta_{N_k}^{\max} \mid \Upsilon_{N_k} \vdash S_{N_k}}{\Delta_N^{\max} \mid \Upsilon_N \vdash S_N}$$

Moreover, for any other legal inference X that extends X^0:

$$X : \frac{\Delta_1 \mid \Upsilon_{N_1} \vdash S_{N_1} \quad \cdots \quad \Delta_k \mid \Upsilon_{N_k} \vdash S_{N_k}}{\Delta \mid \Upsilon_N \vdash S_N}$$

such that $\forall i.\ \Delta_i \models \Phi_{N_i}$ and $\Delta \models \Phi_N$, we have $\Delta \preceq \Delta_N^{\max} \implies \forall i. \Delta_i \preceq \Delta_{N_i}^{\max}$.

LEMMA 11. *Given a pre-proof P^0, a family of permission restrictions Φ_N produced in phase 1, and a variable context $\Delta_{\mathrm{root}}^{\max}$ satisfying Φ_{root}, let $P^{\max}$ be the write-proof on the same underlying tree of nodes obtained by using the given $\Delta_{\mathrm{root}}^{\max}$ and contexts $\Delta_N^{\max}$ satisfying the Property L2. Then*

1. *$P^{\max}$ is a legal write-proof extending P^0.*
2. *if P is any other write-proof extending P^0 using variable contexts Δ_N, and $\Delta_{\mathrm{root}} \preceq \Delta_{\mathrm{root}}^{\max}$, then $\Delta_N \preceq \Delta_N^{\max}$ for all nodes N.*

The proof is by induction on the depth of the nodes in the underlying tree of P^0.

We describe all these aspects compactly by writing down the rules of SCI Separation Logic using the notations of this section, and displaying the computations of both the phases of the algorithm. We decorate the judgements with schematic permission restrictions Φ:

$$\Delta \mid \Upsilon \vdash S\ [\Phi]$$

in order to refer to the permission restrictions computed in phase 1 and used in phase 2. The side conditions of passive rules are ignored in phase 1, but used in phase 2.

Some of rules are as follows (the others are similar):

Expressions:

$$\frac{}{\Delta \mid \Upsilon \vdash E\ \mathbf{Exp}\ [\,]} \quad \text{where } \forall v \in FV(E).\ \Delta\ v\ \mathbf{self} > 0$$

Phase 1 is trivial: $\Phi = [\,]$.
Phase 2 checks to verify that $\Delta^{\max}$ satisfies the side condition. If and only if the side condition is satisfied, the write-proof that extends P^0 with $\Delta^{\max}$ for this node will be a proof.

The rule for Assertions is similar.

Assignable Variable:

$$\frac{}{\Delta \mid \Upsilon \vdash x\ \mathbf{Var}\ [\Phi]} \quad \text{where } \Delta\ x\ o = \begin{cases} 1, & \text{if } o = \mathbf{self} \\ 0, & \text{otherwise} \end{cases}$$

Phase 1: $\Phi = [x : \{\mathbf{self}\}]$
Phase 2 computation is trivial because there are no premises.

Sequencing:

$$\frac{\Delta_1 \mid \Upsilon \vdash \{P\}\, C_1\, \{Q\}\ [\Phi_1] \quad \Delta_2 \mid \Upsilon \vdash \{Q\}\, C_2\, \{R\}\ [\Phi_2]}{\Delta \mid \Upsilon \vdash \{P\}\, C_1; C_2\, \{R\}\ [\Phi]}$$

where $\Delta_1 = \Delta_2 = \Delta$

Phase 1:

$$\begin{aligned} &\mathrm{dom}\ \Phi = \mathrm{dom}\ \Phi_1 \cup \mathrm{dom}\ \Phi_2 \\ &\Phi\ v = \begin{cases} \Phi_1\, v & \text{if } v \in \mathrm{dom}\ \Phi_1 \setminus \mathrm{dom}\ \Phi_2 \\ \Phi_2\, v & \text{if } v \in \mathrm{dom}\ \Phi_2 \setminus \mathrm{dom}\ \Phi_1 \\ (\Phi_1 v \cap \Phi_2 v) & \text{if } v \in \mathrm{dom}\ \Phi_1 \cap \mathrm{dom}\ \Phi_2 \end{cases} \end{aligned}$$

Phase 2: $\Delta_1^{\max} = \Delta_2^{\max} = \Delta^{\max}$.

All other rules such as conditionals, assignment, lookup and mutation are similar to Sequencing in that Δ remains unchanged in the premises. Their Phase 1 and Phase 2 computations are exactly the same as for Sequencing.

Parallel composition:

$$\frac{\Delta_1 \mid \Upsilon \vdash \{P_1\}\, C_1\, \{Q_1\}\ [\Phi_1] \quad \Delta_2 \mid \Upsilon \vdash \{P_2\}\, C_2\, \{Q_2\}\ [\Phi_2]}{\Delta \mid \Upsilon \vdash \{P_1 \star P_2\}\, C_1 \parallel C_2\, \{Q_1 \star Q_2\}\ [\Phi]}$$

where $\Delta_1\ v\ o = \Delta_2\ v\ o = \Delta\ v\ o$ for all $o \neq \mathbf{self}$
$\Delta\ v\ \mathbf{self} = \Delta_1\ v\ \mathbf{self} + \Delta_2\ v\ \mathbf{self} \leq 1$

Phase 1:

$$\begin{aligned} &\mathrm{dom}\ \Phi = \mathrm{dom}\ \Phi_1 \cup \mathrm{dom}\ \Phi_2 \\ &\Phi\ v = \begin{cases} \Phi_1\, v & \text{if } v \in \mathrm{dom}\ \Phi_1 \setminus \mathrm{dom}\ \Phi_2 \\ \Phi_2\, v & \text{if } v \in \mathrm{dom}\ \Phi_2 \setminus \mathrm{dom}\ \Phi_1 \\ (\Phi_1 v \cap \Phi_2 v) \setminus \{\mathbf{self}\} & \text{if } v \in \mathrm{dom}\ \Phi_1 \cap \mathrm{dom}\ \Phi_2 \end{cases} \end{aligned}$$

Phase 2: The pair $(\Delta_1^{\max}\, v\, o,\ \Delta_2^{\max}\, v\, o)$ is as follows:

- If $o \neq \mathbf{self}$, it is $(\Delta^{\max}\, v\, o,\ \Delta^{\max}\, v\, o)$.
- If $o = \mathbf{self}$ and $v \in \mathrm{dom}\ \Phi_1 \setminus \mathrm{dom}\ \Phi_2$, it is $(\Delta^{\max}\, v\, \mathbf{self},\ 0)$
- If $o = \mathbf{self}$ and $v \in \mathrm{dom}\, \Phi_2 \setminus \mathrm{dom}\, \Phi_1$, it is $(0,\ \Delta^{\max}\, v\, \mathbf{self})$.
- If $o = \mathbf{self}$ and $v \in \mathrm{dom}\ \Phi_1 \cap \mathrm{dom}\ \Phi_2$, it is $(\frac{1}{2}\,\Delta^{\max}\, v\, \mathbf{self},\ \frac{1}{2}\,\Delta^{\max}\, v\, \mathbf{self})$.
- If $o = \mathbf{self}$ and $v \notin \mathrm{dom}\ \Phi_1 \cup \mathrm{dom}\ \Phi_2$, it is $(\frac{1}{2}\,\Delta^{\max}\, v\, \mathbf{self},\ \frac{1}{2}\,\Delta^{\max}\, v\, \mathbf{self})$.

The Frame rule is similar to parallel composition.

Critical regions:

$$\frac{\begin{array}{ll} \Delta_1 \mid \Upsilon,\, r\colon R \vdash P, Q\ \mathbf{Assert} & [\,] \\ \Delta_2 \mid \Upsilon \vdash \{(P \star R) \wedge B\}\, C\, \{Q \star R\} & [\Phi_2] \end{array}}{\Delta \mid \Upsilon,\, r\colon R \vdash \{P\}\ \mathbf{with}\ r\ \mathbf{when}\ B\ \mathbf{do}\ C\ \mathbf{od}\ \{Q\}\ \ [\Phi]}$$

where $\Delta_2\ v\ o = \Delta\ v\ o$ for all $o \notin \{\mathbf{self}, r\}$
$\Delta_2\ v\ \mathbf{self} = \Delta\ v\ \mathbf{self} + \Delta\ v\ r \leq 1$
$\Delta_2\ v\ r = 0$
$\Delta_1 = \Delta$

Phase 1: $\mathrm{dom}\ \Phi = \mathrm{dom}\ \Phi_2$
$\Phi\ v = \{\, o \in \Phi_2\, v \mid o \neq \mathbf{self},\ o \neq r\,\} \cup$
$\{\,\mathbf{self} \mid \mathbf{self} \in \Phi_2\, v\,\} \cup \{\, r \mid \mathbf{self} \in \Phi_2\, v\,\}$
Phase 2:

- For $o \notin \{\mathbf{self}, r\}$, $\Delta_2^{\max}\ v\ o = \Delta^{\max}\ v\ o$.
- $\Delta_2^{\max}\ v\ \mathbf{self} = \Delta^{\max}\ v\ \mathbf{self} + \Delta^{\max}\ v\ r$.
- $\Delta_1^{\max} = \Delta^{\max}$ and $\Delta_2^{\max}\ v\ r = 0$.

Resource declaration:

$$\frac{\Delta_1 \mid \Upsilon \vdash R\ \mathbf{Assert}\ \ [\,] \quad \Delta_2 \mid \Upsilon, r : R \vdash \{P\}\, C\, \{Q\}\ \ [\Phi_2]}{\Delta \mid \Upsilon \vdash \{P \star R\}\ \mathbf{resource}\ r\ \mathbf{in}\ C\ \{Q \star R\}\ \ [\Phi]}$$

where R is precise
$\Delta_2\ v\ o = \Delta\ v\ o$ for all $o \notin \{\mathbf{self}, r\}$
$\Delta\ v\ \mathbf{self} = \Delta_2\ v\ \mathbf{self} + \Delta_2\ v\ r \leq 1$
$\Delta_1\ v\ o = \begin{cases} \Delta_2\ v\ r, & \text{if } o = \mathbf{self} \\ 0, & \text{otherwise} \end{cases}$

Phase 1: $\mathrm{dom}\ \Phi = \mathrm{dom}\ \Phi_2$
$\Phi\ v = \{\, o \in \Phi_2\, v \mid o \neq \mathbf{self}, o \neq r\,\} \cup$
$\{\,\mathbf{self} \mid \mathbf{self} \in \Phi_2\, v \vee r \in \Phi_2\, v\,\}$
Phase 2: The context $\Delta_2^{\max}$ is defined as follows:

- For $o \notin \{\mathbf{self}, r\}$, $\Delta_2^{\max}\, v\, o$ is $\Delta^{\max}\ v\ o$.
- The pair $(\Delta_2^{\max}\, v\mathbf{self},\ \Delta_2^{\max}\, v r)$ is as follows: If $v \notin \mathrm{dom}\Phi_2$ then it is $(\frac{1}{2}\,\Delta^{\max}\, v\, \mathbf{self},\ \frac{1}{2}\,\Delta^{\max}\, v\, \mathbf{self})$. If $v \in \mathrm{dom}\ \Phi_2$:
  - If $\mathbf{self} \in \Phi_2\ v$ and $r \in \Phi_2\ v$, it is $(\frac{1}{2}\,\Delta^{\max}\, v\ \mathbf{self},\ \frac{1}{2}\,\Delta^{\max}\, v\, \mathbf{self})$.
  - If $\mathbf{self} \in \Phi_2\ v$ and $r \notin \Phi_2\ v$, it is $(\Delta^{\max}\, v\, \mathbf{self},\ 0)$.
  - If $\mathbf{self} \notin \Phi_2\ v$ and $r \in \Phi_2\ v$, it is $(0,\ \Delta^{\max}\, v\, \mathbf{self})$.
  - If $\mathbf{self} \notin \Phi_2\ v$ and $r \notin \Phi_2\ v$, it is $(0, 0)$.

$\Delta_1^{\max}\ v\ \mathbf{self}$ is the same as $\Delta_2^{\max}\ v\ r$. For all $o \neq \mathbf{self}$, $\Delta_1^{\max}\ v\ o$ is 0.

Variable declaration:

$$\frac{\Delta_1 \mid \Upsilon \vdash P, Q\ \mathbf{Assert}\ \ [\,] \qquad \Delta_2 \mid \Upsilon \vdash \{P\}\, C\, \{Q\}\ [\Phi_2]}{\Delta \vdash \Upsilon \vdash \{P\}\ \mathbf{local}\ x\ \mathbf{in}\ C\ \{Q\}\ \ [\Phi]}$$

where $\Delta_2\ v\ o = \Delta\ v\ o$ for all $v \neq x$
$\Delta_2\ x\ o = \begin{cases} 1, & \text{if } o = \mathbf{self} \\ 0, & \text{otherwise} \end{cases}$
$\Delta_1\ v\ o = \Delta\ v\ o$ for all $v \neq x$
$\Delta_1\ x\ o = 0$

Phase 1: If $\mathbf{self} \in \Phi_2\ x$ or $x \notin \mathrm{dom}\ \Phi_2$ then the computation is:

$\mathrm{dom}\ \Phi = (\mathrm{dom}\ \Phi_2) \setminus \{x\}$
$\Phi\ v = \Phi_2\ v$ for all $v \neq x$

If $x \in \mathrm{dom}\ \Phi_2$ and $\mathbf{self} \notin\ \Phi_2\ x$ then Phase 1 fails, i.e., there is no write-proof extending P^0.
Phase 2:

- $\Delta_2^{\max}\ x\ o$ is 1 when o is $\mathbf{self}$, 0 otherwise. For all other v, $\Delta_2^{\max}\ v\ o = \Delta^{\max}\ v\ o$.
- $\Delta_1^{\max} = \Delta^{\max}$.

It may be verified that all the phase 1 and phase 2 computations listed above satisfy the properties L1 and L2 respectively, completing the proof of correctness.

7. Conclusion

We have provided a streamlined formulation of Sequential and Concurrent Separation Logic rules without awkward side conditions for variable usage. The rules are more expressive than the original Owicki-Gries-O'Hearn system. Yet, they retain the "syntactic" character of the variable conditions without adding proof burden in the programming logic itself. This syntactic character is exploited in devising an algorithm to automatically infer the annotations required in resource declarations. This should prove useful for Separation Logic-based verification tools like Smallfoot.

Our work is also a modest contribution to the theory of Syntactic Control of Interference, which dates back to 1978. While the system has been studied from a semantics point of view, it has not been previously applied to the formulation of programming logics, which is somewhat paradoxical given its natural fit with reasoning principles. We have extended the traditional framework with permission algebras, which should prove useful for further development.

Further work along this line would include the extension of Concurrent Separation Logic with higher-order features such as procedures and objects, for which Syntactic Control of Interference is well-suited.

Acknowledgments

We are grateful to Philippa Gardner and her group for the "yak" session at Imperial College where this work was first presented. Peter O'Hearn has provided much appreciated encouragement and inspiration throughout the development of this work. Special thanks go to Josh Berdine and Ian Wehrman for making us aware of the technical problems with the earlier proof systems. John Reynolds's work was partially supported by National Science Foundation Grant CCF-0916808.

References

[1] S. Abramsky and G. McCusker. Linearity, sharing and state. In *Algol-like Languages* O'Hearn and Tennent [28], chapter 20.

[2] S. Abramsky, K. Honda, and G. McCusker. A fully abstract game semantics for general references. In *LICS 1998*, pages 334–344, 1998.

[3] K. R. Apt. Ten years of Hoare's logic: A survey. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483, Oct. 1981.

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

[5] K. Bierhoff. API protocol compliance in object-oriented software. Technical Report CMU-ISR-09-108, Carnegie-Mellon University, Apr 2009.

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

[7] R. Bornat, C. Calcagno, and H. Yang. Variables as resource in Separation Logic. In *Proc. 22nd Ann. Conf. on Math. Found. of Program. Semantics (MFPS XXII)* Main et al. [21], pages 247–276.

[8] 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.

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

[10] P. Brinch Hansen. Structured multiprogramming. *Comm. ACM*, 15: 574–577, July 1972.

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

[12] S. D. Brookes. A revisionist history of Concurrent Separation Logic. In Mislove and Ouaknine [24], pages 5–28.

[13] 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)* Main et al. [21], pages 123–150. doi: DOI: 10.1016/j.entcs.2006.04.008.

[14] L. Damas and R. Milner. Principal type-schemes for functional programs. In *ACM Symp. on Princ. of Program. Lang.*, pages 207–212, 1982.

[15] J.-Y. Girard. Linear logic. *Theoretical Comput. Sci.*, 50:1–102, 1987.

[16] A. Gotsman, J. Berdine, and B. Cook. Precision and the conjunction rule in Concurrent Separation Logic. In Mislove and Ouaknine [24].

[17] 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.

[18] C. A. R. Hoare. Monitors: An operating system structuring concept. *Comm. ACM*, 17(10):549–558, Oct. 1974.

[19] B. J., C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with Separation Logic. In F. S. de Boer, editor, *Formal Methods for Components and Objects, 4th Intern. Symp.*, volume 4111 of *LNCS*, pages 115–137. Springer-Verlag, 2005.

[20] K. Kapoor, K. Lodaya, and U. S. Reddy. Fine grained concurrency with Separation Logic. *J. Philosophical Logic*, 40(5):583–632, Oct 2011. doi: 10.1007/s10992-011-9195-1.

[21] M. Main, A. Melton, and M. Mislove. *Proc. 22nd Ann. Conf. on Math. Found. of Program. Semantics (MFPS XXII)*, volume 158 of *Elect. Notes in Theor. Comput. Sci.* Elsevier, 2006.

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

[23] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17:348–375, 1978.

[24] M. Mislove and J. Ouaknine, editors. *Proc. 27nd Ann. Conf. on Math. Found. of Program. Semantics (MFPS XXVII)*, volume 276 of *Elect. Notes in Theor. Comput. Sci.* Elsevier, 2011.

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

[26] 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.

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

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

[29] 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 [28]).

[30] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *CSL 2001*, volume 2142 of *LNCS*, pages 1–19, Berlin, 2001. Springer-Verlag.

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

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

[33] 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 [28]).

[34] J. Reynolds. Separation Logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[35] J. C. Reynolds. A problematic program (joint work with Josh Berdine). Presentation at the Dagstuhl workshop on Types, Logics and Semantics for State, 2008.

[36] 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 [28]).

[37] J. C. Reynolds. Idealized Algol and its specification logic. In D. Neel, editor, *Tools and Notions for Program Construction*, pages 121–161. Cambridge Univ. Press, 1982. (Reprinted as Chapter 6 of [28]).

[38] V. Vafeiadis. Concurrent Separation Logic and operational semantics. In Mislove and Ouaknine [24].

[39] H. Yang and P. W. O'Hearn. A semantics basis for local reasoning. In *FOSSACS*, pages 402–416, Berlin, 2002. Springer-Verlag.

[40] H. Yasuoka and T. Terauchi. Polymorphic fractional capabilities. In *Static Analysis Symposium/Workshop on Static Analysis*, pages 36–51, 2009. doi: 10.1007/978-3-642-03237-0_5.

Technical Appendix

A. Well-formedness of commands

For completeness, we include the well-formedness rules for commands. They can be obtained from the programming logic rules by simply omitting all the judgements corresponding to assertions.

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

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

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

$$\frac{\Sigma, x^{\top} \vdash C\ \textbf{Comm}}{\Sigma \vdash (\textbf{local}\ x\ \textbf{in}\ C)\ \textbf{Comm}}$$

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 concurrent commands are shown in below. 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.

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

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

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

The rule for parallel composition follows the general pattern of original syntactic control of interference in [29, 36]. 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 an 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.

B. Comparison with Brookes's system

Brookes [11], 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: $\textbf{owned}(\Gamma)$ is the set of variables included in the resource declarations and $\textbf{free}(\Gamma)$ 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 $\textbf{owned}(\Gamma)$ 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 $\textbf{free}(\Gamma) \setminus \textbf{owned}(\Gamma)$ 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 $\textbf{free}(R)$ 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 permission) 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 8, due to Ian Wehrman [4]. In this example, the variable x is in $\textbf{owned}(\Gamma)$ and a is in $\textbf{free}(\Gamma)$. 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. 4.1. However, Owicki-Gries place the additional requirement (the "rider" mentioned in Sec. 4.1) 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

```
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 8. Example proof outline in Brookes's system

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 $\mathbf{free}(\Gamma)$ are variables that have partial permissions in the resource. So, the distinction between $\mathbf{owned}(\Gamma)$ and $\mathbf{free}(\Gamma)$ is one of permission levels, and Brookes's system fits in between the Owicki-Gries system and our system of syntactic control with permissions.

Recently, Brookes proposed a revised system [12], which avoids the problem mentioned above. It uses sets of free variables called "rely sets" in its judgements, similar to our use of variable contexts. However, there are no permissions associated with the variables in rely sets. There are also other technical differences in the way the variable conditions are treated. We do not at present have a precise comparison with the SCI system and Brookes's revised system.

C. Selected proofs of results

Proof of Theorem 5 The proof is by induction on the derivation of well-formed terms:

- If the command is $\Sigma \mid \Gamma \vdash (x := E)$ $\mathbf{Comm}$ then we know that $x^{\top} \in norm(\Sigma)$ and $\Sigma \vdash E$ $\mathbf{Exp}$. So, for any $(\rho, v) \in [\![E]\!]_{\Sigma}$, ρ is a well-bracketed trace for $\Sigma \mid$ and, hence, for $\Sigma \mid \Gamma$. Since $x^{\top} \in norm(\Sigma)$, $(x := v)$ is also a trace for $\Sigma \mid \Gamma$.

- If the command is
$$\Sigma \mid \Gamma, r(\Sigma_0) \vdash (\mathbf{with}\ r\ \mathbf{when}\ B\ \mathbf{do}\ C\ \mathbf{od})\ \mathbf{Comm}$$
then we know that $\Sigma, \Sigma_0 \vdash B$ $\mathbf{Exp}$ and $\Sigma, \Sigma_0 \vdash C$ $\mathbf{Comm}$ are well-formed. 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 set $wait$ is a trace set for $\Sigma \mid \Gamma, r(\Sigma_0)$. The trace set $enter$ is a trace set for $\Gamma \mid \Gamma, r(\Sigma_0)$. 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:
$$\begin{array}{lll} \Sigma \mid \Gamma, r(\Sigma_0) & & \\ \quad \xrightarrow{acq(r)} \Sigma, \Sigma_0 \mid \Gamma & & \text{by definition} \\ \quad \xrightarrow{\rho} \Sigma, \Sigma_0 \mid \Gamma, [r(\Sigma_0)] & & \text{by ind. hyp. for } B \text{ and Lemma 3} \\ \quad \xrightarrow{\gamma} \Sigma, \Sigma_0 \mid \Gamma, [r(\Sigma_0)] & & \text{by ind. hyp. for } C \text{ and Lemma 3} \\ \quad \xrightarrow{rel(r)} \Sigma \mid \Gamma, r(\Sigma_0) & & \text{by definition} \end{array}$$

- If the command is
$$\Sigma_1, \Sigma_2 \mid \Gamma \vdash (C_1 \parallel C_2)\ \mathbf{Comm}$$
then we have well-formed commands $\Sigma_i \mid \Gamma \vdash C_i$ $\mathbf{Comm}$ for $i = 1, 2$. By inductive hypothesis, each $[\![C_i]\!]_{\Sigma_i|\Gamma}$ is a trace set for the context $\Sigma_i \mid \Gamma$. Then, by Lemma 2, $[\![C_1]\!]_{\Sigma_1|\Gamma} \parallel [\![C_2]\!]_{\Sigma_2|\Gamma}$ is a trace set for the context $\Sigma_1, \Sigma_2 \mid \Gamma$.

Proof of Theorem 8

Proof: By induction on the derivation of $\Sigma \mid \Gamma \vdash \{P\}\ C\ \{Q\}$. We show selected cases.

- If the last rule is the assignment rule for $\Sigma \mid \Gamma \vdash \{P\,[E/x]\} x := E\ \{P\}$, then we have $\Sigma \vdash x$ $\mathbf{Var}$, $\Sigma \vdash E$ $\mathbf{Exp}$ and $\Sigma \vdash P$ $\mathbf{Assert}$. The trace set $[\![x := E]\!]_{\Sigma|\Gamma}$ consists of traces of the form $\Sigma|\Gamma \xrightarrow{\rho(x:=v)} \Sigma|\Gamma$ where $\rho \in [\![E]\!]_{\Sigma}$. Let (s, h) be a state satisfying $P\,[E/x]$. Then
every state transition for $\rho(x := v)$ is of the form $(s, h) \xrightarrow{\rho(x:=v)} (s\,[x \mapsto v]\,, h)$. We have $(s, v) \in |e|$ and by Substitution Lemma $(s\,[x \mapsto v]\,, h) \models P$.
- If the last rule is the parallel composition rule for $\Sigma_1, \Sigma_2 \mid \Gamma \vdash \{P_1 \star P_2\}\ C_1 \parallel C_2\ \{Q_1 \star Q_2\}$ derived from the premises $\Sigma_1 \mid \Gamma \vdash \{P_1\}\ C_1\ \{Q_1\}$ and $\Sigma_2 \mid \Gamma \vdash \{P_2\}\ C_2\ \{Q_2\}$, let $(s_1 \cdot s_2, h_1 \cdot h_2)$ be an initial state satisfying $P_1 \star P_2$ such that $h_1 \perp h_2$ and each (s_i, h_i) is of type $\Sigma_i \mid \Gamma_i$ and satisfies P_i.
The trace set $[\![C_1 \parallel C_2]\!]_{\Sigma_1,\Sigma_2|\Gamma}$ is the set of mutex fairmerges $[\![C_1]\!]_{\Sigma_1|\Gamma} \parallel [\![C_2]\!]_{\Sigma_2|\Gamma}$. Let α be a well-bracketed trace in this set and σ' a state such that $(s, h) \xrightarrow{\alpha} \sigma'$. Then we know that α is a finite trace $\Sigma_1, \Sigma_2 \mid \Gamma \xrightarrow{\alpha} \Sigma_1, \Sigma_2 \mid \Gamma$ and it is in the mutex fairmerges of some $\alpha_1 \in [\![C_1]\!]_{\Sigma_1|\Gamma}$ and $\alpha_2 \in [\![C_2]\!]_{\Sigma_2|\Gamma}$.
If $\sigma' = \mathbf{abort}$ then it must be the case that $(s_i, h_i) \xrightarrow{\alpha_i} \mathbf{abort}$ for some $i = 1, 2$. If so, the corresponding premise $\Sigma_i \mid \Gamma \vdash \{P_i\}\ C_i\ \{Q_i\}$ would be invalid, contradicting the inductive hypothesis.
If σ' is of the form (s', h') then by the Parallel Decomposition Lemma, there are states (s'_i, h'_i) of type $\Sigma_i|\Gamma$, for $i = 1, 2$, such that $s' = s'_1 \cdot s'_2$, $h' = h'_1 \cdot h'_2$, and $(s_i, h_i) \xrightarrow{\alpha_i} (s'_i, h'_i)$. By the inductive hypothesis $(s'_i, h'_i) \models Q_i$, for $i = 1, 2$. This gives the result $(s'_1 \cdot s'_2,\ h'_1 \cdot h'_2) \models Q_1 \star Q_2$.
- If the last rule used is the critical region rule for
$$\Sigma \mid \Gamma, r(\Sigma_0) : R \vdash \{P\}\ \mathbf{with}\ r\ \mathbf{when}\ B\ \mathbf{do}\ C\ \{Q\}$$
derived from $\Sigma \vdash P, Q$ $\mathbf{Assert}$, $\Sigma, \Sigma_0 \vdash B$ $\mathbf{Exp}$ and $\Sigma, \Sigma_0 \mid \Gamma \vdash \{(P \star R) \wedge B\}\ C\ \{Q \star R\}$, let (s, h) be an initial state of type $\Sigma \mid \Gamma, r(\Sigma_0)$ satisfying P, α a well-bracketed trace for context $\Sigma \mid \Gamma, r(\Sigma_0)$ in $[\![\mathbf{with}\ r\ \mathbf{when}\ B\ \mathbf{do}\ C]\!]_{\Sigma|\Gamma,r(\Sigma_0)}$, and σ' a state such that $(s, h) \xrightarrow{\alpha} \sigma'$.
Ignoring try actions and repeated tests, α is of the form
$$\begin{array}{rcl} \Sigma \mid \Gamma, r(\Sigma_0) \colon R & \xrightarrow{acq(r)} & \Sigma, \Sigma_0 \mid \Gamma, [r(\Sigma_0) \colon R] \\ & \xrightarrow{\rho} & \Sigma, \Sigma_0 \mid \Gamma, [r(\Sigma_0) \colon R] \\ & \xrightarrow{\beta} & \Sigma, \Sigma_0 \mid \Gamma, [r(\Sigma_0) \colon R] \\ & \xrightarrow{rel(r)} & \Sigma \mid \Gamma, r(\Sigma_0) \colon R \end{array}$$
where $\rho \in [\![B]\!]_{\Sigma,\Sigma_0} \restriction true$ and $\beta \in [\![C]\!]_{\Sigma,\Sigma_0|\Gamma}$.
We argue that there are states (s_0, h_0) and (s'_0, h'_0) of type $\Sigma_0|$ and (s', h') of type $\Sigma \mid \Gamma$ such that
$$\begin{array}{rcl} (s, h, \emptyset) & \xrightarrow{acq(r)} & (s \cdot s_0,\ h \cdot h_0,\ \{r\}) \\ & \xrightarrow{\rho} & (s \cdot s_0,\ h \cdot h_0,\ \{r\}) \\ & \xrightarrow{\beta} & (s' \cdot s'_0,\ h' \cdot h'_0,\ \{r\}) \\ & \xrightarrow{rel(r)} & (s', h', \emptyset) \end{array}$$

Here, (s_0, h_0) is an arbitrary state satisfying R. Since (s, h) satisfies P, $(s \cdot s_0,\, h \cdot h_0$ satisfies $P \star R$. By virtue of the fact that $\rho \in [\![B]\!]_{\Sigma,\Sigma_0}$, it also satisfies B, and hence $(P \star R) \wedge B$. The premise for the validity of $\Sigma, \Sigma_0 \vdash \{(P \star R) \wedge B\}\, C\, \{Q \star R\}$ implies that the target state of β satisfies $Q \star R$. In particular, it cannot be $\mathbf{abort}$. Since this state is of type $\Sigma, \Sigma_0 \mid \Gamma, [r(\Sigma_0)\colon R]$, the store is decomposable into s' and s'_0 of types Σ and Σ_0 respectively. Moreover, the heap of this state must have a subheap of satisfying R. The fact that R is precise means that this subheap is unique. So, the heap is decomposable into h' and h'_0. The remaining part of the target state (s', h') satisfies Q. Finally, the $rel(r)$ action removes the resource part of the state, giving (s', h'), which satisfies Q.