A semantics for concurrent separation logic

Stephen Brookes

Carnegie Mellon University
Department of Computer Science

Abstract. We present a denotational semantics based on action traces, for parallel programs which share mutable data and synchronize using resources and conditional critical regions. We introduce a resource-sensitive logic for partial correctness, adapting separation logic to the concurrent setting, as proposed by O'Hearn. The logic allows program proofs in which "ownership" of a piece of state is deemed to transfer dynamically between processes and resources. We prove soundness of this logic, using a novel "local" interpretation of traces, and we show that every provable program is race-free.

1 Introduction

Parallel programs involve the concurrent execution of processes which share state and are intended to cooperate interactively. It is notoriously difficult to ensure absence of runtime errors such as *races*, in which one process changes a piece of state being used by another process, and *dangling pointers*, which may occur if two processes attempt simultaneously to deallocate the same storage. Such phenomena can cause unpredictable or irreproducible behavior.

Rather than relying on assumptions about the granularity of hardware primitives, it is preferable to use program design rules and proof techniques that guarantee error-freedom. The classic example is the syntax-directed logic for partial correctness properties of (pointer-free) parallel programs introduced by Owicki and Gries [15], building on prior work of Hoare [7]. This approach focusses on *critical variables*, the identifiers concurrently written by one process and read or written by another. The programmer must partition the critical variables among named *resources*, and each occurrence of a critical variable must be inside a region naming the relevant resource. Assuming that resource management is implemented by a suitable synchronization primitive, such as semaphores [6, 1], the design rules guarantee mutually exclusive access to critical variables and therefore freedom from races. Each process *relies* on its environment to ensure that when a resource is available the corresponding *resource invariant* holds, and *guarantees* that when the process releases the resource the invariant will hold again (*cf.* rely/guarantee methodology as in [9]). This use of resource invariants abstracts away from what happens "inside" a critical region and focusses on the places where synchronization occurs.

This method works well for simple (pointer-free) parallel programs, but the task of reasoning about parallel pointer-programs is made more difficult by the potential for aliasing, when distinct expressions may denote the same pointer: static design rules no longer suffice to prevent races involving pointer values. For example, the program $[x]{:=}0\|[y]{:=}1$ has a race if x and y are aliases, and this aliasing cannot be determined from the syntax of the program. O'Hearn [11, 12] has proposed an adaptation of the Owicki-Gries inference rules to handle parallel pointer-programs, incorporating ideas from separation logic [17, 14, 8]. The main technical novelty in this adaptation involves the use of *separating conjunction* in the rules dealing with resource invariants and parallel composition. Although this may appear superficially to produce "obvious" variants of the traditional rules, the original rules (using the standard form of conjunction) are unsound for pointer-programs, and soundness of the new rules is far from obvious. Indeed, Reynolds has shown that O'Hearn's rules are unsound without restrictions on resource invariants [18, 13].

O'Hearn provides a series of compelling examples with informal correctness proofs, but (as he remarks) the logic cannot properly be assessed without a suitable semantic model [11]. Such a model is not readily available in the literature: traditional models for concurrency do not include pointers or race-detection, and models for pointer-programs do not typically handle concurrency. In this paper we give a denotational semantics, using *action traces*, that solves these problems, using a form of parallel composition that detects races and treats them as catastrophic[1]. Our semantic model embodies a classic principle of concurrent program design, originally articulated by Dijkstra [6] and echoed in the design of the classic inference rules for shared-memory programs [7, 15]:

> ... processes should be loosely connected; by this we mean that apart from the (rare) moments of explicit intercommunication, the individual processes are to be regarded as completely independent of each other.

In other words, concurrent processes do not interfere (or cooperate) except through explicit synchronization. Our semantics makes this idea concrete through the interplay between traces, which describe interleaved behaviors of processes, and an enabling relation on "local states" that models "no interference from outside except at synchronization". This interplay permits a formalization of O'Hearn's "processes that mind their own business" [12], and leads to a Parallel Decomposition Lemma that reflects the intuition behind Dijkstra's principle in a semantically precise manner.

The Owicki-Gries logic, and O'Hearn's adaptation, assume a fixed collection of resources and a fixed set of parallel processes. We reformulate O'Hearn's inference rules in a more semantically natural manner, allowing statically scoped resource declarations and nested parallel compositions. We assume that each resource invariant is a *precise* separation logic formula, so that every time a program acquires or releases a resource there is a uniquely determined portion of the heap whose ownership can be deemed to transfer. We give a suitably general

[1] This idea was suggested by John Reynolds [18].

(and compositional) notion of validity, and we prove that the proof rules, using precise invariants, are sound. Our soundness proof demonstrates that a verified program has no race conditions.

We omit proofs, and we do not include examples to illustrate the logic; the reader should see O'Hearn's paper [12] for such examples, which may be replicated quite straightforwardly in our more formal setting. O'Hearn's paper also discusses the limitations of the logic and identifies opportunities for further research. We assume familiarity with the syntax and semantics of separation logic [17]. Apart from this we have tried to include enough technical detail to make the paper self-contained.

2 Syntax

Our programming language combines shared-memory parallelism with pointer operations. The syntax for *commands* (ranged over by c) is given by the following abstract grammar, in which r ranges over *resource names*, i over *identifiers*, e over *integer expressions*, and b over *boolean expressions*:

$$\begin{aligned} c ::= {} & \mathbf{skip} \mid i{:=}e \mid i{:=}[e] \mid [e]{:=}e' \mid i{:=}\mathbf{cons}\,(e_0,\ldots,e_n) \mid \mathbf{dispose}\, e \mid \\ & c_1; c_2 \mid c_1 \| c_2 \mid \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \mid \mathbf{while}\ b\ \mathbf{do}\ c \mid \\ & \mathbf{resource}\ r\ \mathbf{in}\ c \mid \mathbf{with}\ r\ \mathbf{when}\ b\ \mathbf{do}\ c \end{aligned}$$

Expressions are *pure*, so evaluation has no side-effect and the value of an expression depends only on the *store*. An *assignment* command $i{:=}e$ affects only the store; *allocation* $i{:=}\mathbf{cons}(e_0,\ldots,e_n)$, *lookup* $i{:=}[e]$, *update* $[e]{:=}e'$, and *disposal* $\mathbf{dispose}(e)$ involve the *heap*. A command of form $\mathbf{resource}\ r\ \mathbf{in}\ c$ introduces a local resource name r, whose scope is c. A command of form $\mathbf{with}\ r\ \mathbf{when}\ b\ \mathbf{do}\ c$ is a *conditional critical region* for resource r. A process attempting to enter a region must wait until the resource is available, acquire the resource and evaluate b: if b is **true** the process executes c then releases the resource; if b is **false** the process releases the resource and waits to try again. A resource can only be held by one process at a time. We use the abbreviation $\mathbf{with}\ r\ \mathbf{do}\ c$ when b is **true**.

Let $\texttt{free}(c)$ be the set of identifiers occurring free in c, with similar notation for expressions. Let $\texttt{writes}(c)$ be the set of identifiers having a free write occurrence in c, and $\texttt{res}(c)$ be the set of resource names occurring free in c. These sets are defined as usual, by structural induction. For instance, $\texttt{res}(\mathbf{with}\ r\ \mathbf{when}\ b\ \mathbf{do}\ c) = \texttt{res}(c) \cup \{r\}$, $\texttt{res}(\mathbf{resource}\ r\ \mathbf{in}\ c) = \texttt{res}(c) - \{r\}$.

3 Semantics

We give a trace-theoretic semantics for expressions and commands. The meaning of an expression will be a set of trace-value pairs, and the meaning of a command will be a set of traces. The trace set denoted by a program describes in abstract terms the possible interactive computations that the program may perform when executed fairly, in an environment which is also capable of performing actions.

We interpret sequential composition as concatenation of traces, and parallel composition as a resource-sensitive form of interleaving of traces that enforces mutually exclusive access to each resource. By presenting traces as sequences of *actions* we can keep the underlying notion of *state* more or less implicit.[2] We will exploit this feature later, when we use the semantics to prove soundness of a concurrent separation logic. We start by providing an interpretation of actions using a global notion of state; later we will set up a more refined local notion of state with which it is easier to reason about ownership.

Our semantics is designed to support reasoning about partial correctness and the absence (or potential presence) of runtime errors. The semantics also models deadlock, as a form of infinite waiting, and allows reasoning about safety and liveness properties. The semantics assumes that parallel processes are executed under the control of a *weakly fair* scheduler [16], so that each process that has not yet terminated will eventually be scheduled for execution.

States, actions, and traces

A *value* is either an integer, or an address. We use v to range over values, l over addresses. Let V_{int} be the set of integers, V_{addr} be the set of addresses[3], and V_{bool} be the set of truth values. A *resource set* is a finite set of resource names. A *state* σ comprises a *store* s, a *heap* h, and a resource set A. The *store* maps a finite set of identifiers to values; the *heap* maps a finite set of addresses to values. We use notations such as $[i_1 : v_1, \ldots, i_k : v_k]$ and $[l_1 : v_1, \ldots, l_n : v_n]$ to denote stores and heaps, and $[s \mid i : v]$ and $[h \mid l : v]$ for updated stores and heaps. We write $s\backslash X$ for the store obtained by removing the identifiers in X from the domain of s, and $h\backslash l$ for the heap obtained from h by deleting l from its domain. When heaps h_1 and h_2 have disjoint domains we write $h_1 \perp h_2$, and we let $h_1 \cdot h_2$ denote their union. We use a similar notation for stores. An "initial" state will have the form $(s, h, \{\})$; we may use the abbreviation (s, h) in such a case.

We will describe a program's behavior in terms of *actions*. These include *store actions*: reads $i{=}v$ and writes $i{:=}v$ to identifiers; *heap actions*: lookups $[l]{=}v$, updates $[l]{:=}v$, allocations $alloc(l, [v_0, \ldots, v_n])$, and disposals $disp(l)$ of addresses; and *resource actions*: $try(r)$, $acq(r)$, $rel(r)$ involving resource names. We also include an *idle* action δ, and an error action *abort*. We use λ to range over the set of actions.

Each action λ is characterized by its *effect*, a partial function $\overset{\lambda}{\Longrightarrow}$ from states to states. This partial function describes the set of states in which the action is enabled, and the state change caused by executing the action. Note that an action may cause a runtime error, for which we employ the error state **abort**.

[2] An advantage of action traces [4, 3] over the *transition traces* [5] often used to model shared-memory parallel languages is succinctness: an action typically acts the same way on many states, and we can express this implicitly, without enumerating all pairs of states related by the action.

[3] Actually we treat addresses as integers, so that our semantic model can incorporate address arithmetic, but for moral reasons we distinguish between integers as values and integers which happen to be addresses in current use.

Definition 1 *The effect* $\xRightarrow{\lambda}$ *of an action* λ *is given by the following clauses:*

$$\begin{array}{ll}
(s,h,A) \xRightarrow{\delta} (s,h,A) & \textit{always} \\
(s,h,A) \xRightarrow{i=v} (s,h,A) & \textit{iff } (i,v) \in s \\
(s,h,A) \xRightarrow{i=v} \textbf{abort} & \textit{iff } i \notin \mathtt{dom}(s) \\
(s,h,A) \xRightarrow{i:=v} ([s \mid i:v],h,A) & \textit{iff } i \in \mathtt{dom}(s) \\
(s,h,A) \xRightarrow{i:=v} \textbf{abort} & \textit{iff } i \notin \mathtt{dom}(s) \\
(s,h,A) \xRightarrow{[l]=v} (s,h,A) & \textit{iff } (l,v) \in h \\
(s,h,A) \xRightarrow{[l]=v} \textbf{abort} & \textit{iff } l \notin \mathtt{dom}(h) \\
(s,h,A) \xRightarrow{[l]:=v} (s,[h \mid l:v],A) & \textit{iff } l \in \mathtt{dom}(h) \\
(s,h,A) \xRightarrow{[l]:=v} \textbf{abort} & \textit{iff } l \notin \mathtt{dom}(h) \\
(s,h,A) \xRightarrow{alloc(l,[v_0,\ldots,v_n])} (s,[h \mid l{:}v_0,\ldots,l+n{:}v_n],A) & \textit{iff } \forall m \leq n.\ l+m \notin dom(h) \\
(s,h,A) \xRightarrow{disp(l)} (s,h\backslash l,A) & \textit{iff } \ l \in \mathtt{dom}(h) \\
(s,h,A) \xRightarrow{disp(l)} \textbf{abort} & \textit{iff } l \notin \mathtt{dom}(h) \\
(s,h,A) \xRightarrow{try(r)} (s,h,A) & \textit{iff } r \in A \\
(s,h,A) \xRightarrow{acq(r)} (s,h,A \cup \{r\}) & \textit{iff } r \notin A \\
(s,h,A) \xRightarrow{rel(r)} (s,h,A - \{r\}) & \textit{iff } r \in A \\
(s,h,A) \xRightarrow{abort} \textbf{abort} & \textit{always} \\
\textbf{abort} \xRightarrow{\lambda} \textbf{abort} & \textit{always}
\end{array}$$

It is obvious from the above definition that store actions depend only on the store, heap actions depend only on the heap, and resource actions depend only on the resource set. In general an action is either *enabled* or *stuck* in a given state. For example, if $s(x) = 0$ the action x=0 is enabled, but the action x=1 is stuck. The stuck cases play only a minor role in the development.

Note that a $try(r)$ action is allowed, from a state (s,h,A) in which $r \in A$, to model the case where one parallel component of the program has already acquired r but another component process wants to acquire it and must wait until the resource is released. A process can only acquire a resource that it does not already possess, and can only release a resource that it currently holds.

The clause defining the effect of an allocation action is non-deterministic, to model our assumption that storage management is governed by a mutual exclusion discipline and ensures the use of "fresh" heap cells. A given state (s,h,A) enables all allocation actions of the form $alloc(l,[v_0,\ldots,v_n])$ for which the heap cells $l, l+1, \ldots, l+n$ are all outside of $\mathtt{dom}(h)$. We assume that the storage allocator never chooses to allocate a heap cell in current use, so we do not need to include an error case for allocate actions. On the other hand, since disposals are done by the program we include an error case for disposal actions to account for the possibility of a dangling pointer.

A *trace* is a non-empty finite or infinite sequence of actions. Let **Tr** be the set of all traces. We use α, β as meta-variables ranging over the set of traces, and

T_1, T_2 range over trace sets. We write $\alpha\beta$ for the trace obtained by concatenating α and β; when α is infinite this is just α. We assume that $\alpha\, abort\, \beta = \alpha\, abort$, and $\alpha\delta\beta = \alpha\beta$, for all traces α and β.

For trace sets T_1 and T_2 we let $T_1\,T_2 = \{\alpha_1\alpha_2 \mid \alpha_1 \in T_1 \;\&\; \alpha_2 \in T_2\}$, and we use the usual notations T^* and T^ω for the finite and infinite concatenations of traces from the set T. We let $T^\infty = T^* \cup T^\omega$.

We define the effect $\overset{\alpha}{\Longrightarrow}$ of a trace α in the obvious way, by composing the effects of the actions occurring in the trace. When $(s, h, A) \overset{\alpha}{\Longrightarrow} (s', h', A')$ the trace α can be executed from (s, h, A) without the need for interference from outside; we call such a trace *sequential*[4]. As is well known, the sequential traces of $c_1 \| c_2$ cannot generally be determined from the sequential traces of c_1 and c_2, so we need to include non-sequential traces in order to achieve a compositional semantics.

Parallel composition

The behavior of a command depends on resources: those held by the command and those being used by its environment. These sets of resources start empty and will always be disjoint. Accordingly we define for each action λ a *resource enabling* relation $(A_1, A_2) \xrightarrow{\lambda} (A_1', A_2)$ on disjoint pairs of resource sets, to specify when a process holding resources A_1, in an environment that holds A_2, can perform this action, and the action's effect on the resources held by the program:

$$\begin{array}{ll}
(A_1, A_2) \xrightarrow{try(r)} (A_1, A_2) & \\
(A_1, A_2) \xrightarrow{acq(r)} (A_1 \cup \{r\}, A_2) & \text{if } r \notin A_1 \cup A_2 \\
(A_1, A_2) \xrightarrow{rel(r)} (A_1 - \{r\}, A_2) & \text{if } r \in A_1 \\
(A_1, A_2) \xrightarrow{\lambda} (A_1, A_2) & \text{if } \lambda \text{ is not a resource action}
\end{array}$$

Clearly if A_1 and A_2 are disjoint and $(A_1, A_2) \xrightarrow{\lambda} (A_1', A_2')$ then $A_2 = A_2'$ and A_1' is disjoint from A_2.

This resource enabling notion generalizes in the obvious way to a sequence of actions; we write $(A_1, A_2) \xrightarrow{\alpha} \cdot$ to indicate that a process holding resources A_1 in an environment holding A_2 can perform the trace α.

We want to detect *race conditions* caused by an attempt to write to an identifier or address being used concurrently. This can be expressed succinctly as follows. Extend the definitions of `free` and `writes` to actions:

$$\begin{array}{ll}
\texttt{free}(i{:=}v) = \{i\} & \texttt{writes}(i{:=}v) = \{i\} \\
\texttt{free}(i{=}v) = \{i\} & \texttt{writes}(i{=}v) = \{\} \\
\texttt{free}([l]{:=}v) = \{l\} & \texttt{writes}([l]{:=}v) = \{l\} \\
\texttt{free}([l]{=}v) = \{l\} & \texttt{writes}([l]{=}v) = \{\} \\
\texttt{free}(disp(l)) = \{l\} & \texttt{writes}(disp(l)) = \{l\} \\
\texttt{free}(alloc(l, [v_0, \ldots, v_n]) = \{\} & \texttt{writes}(alloc(l, [v_0, \ldots, v_n]) = \{\} \\
\texttt{free}(\lambda) = \{\} & \texttt{writes}(\lambda) = \{\} \quad \text{otherwise}
\end{array}$$

[4] Technically we say that α is sequential if and only if $\overset{\alpha}{\Longrightarrow} \neq \{\}$.

Informally, $\texttt{free}(\lambda)$ contains the identifiers or addresses whose current values are needed to enable the action, and $\texttt{writes}(\lambda)$ contains the identifiers or addresses whose values in the current state are affected by the action. We do not include addresses $l, \dots, l{+}n$ in the free- or write-set of $\mathit{alloc}(l, [v_0, \dots, v_n])$, because these addresses are assumed to be fresh when the action occurs.

We write $\lambda_1 \sharp \lambda_2$ (λ_1 *interferes with* λ_2) when λ_1 and λ_2 represent a race:

$$\lambda_1 \sharp \lambda_2 \;\Leftrightarrow\; \texttt{free}(\lambda_1) \cap \texttt{writes}(\lambda_2) \neq \{\} \vee \texttt{writes}(\lambda_1) \cap \texttt{free}(\lambda_2) \neq \{\}.$$

Notice that we do not regard two concurrent reads as a disaster.

We now define, for each pair (A_1, A_2) of disjoint resource sets and each pair (α_1, α_2) of action sequences, the set $\alpha_1 {}_{A_1}\|_{A_2} \alpha_2$ of all *mutex fairmerges* of α_1 using A_1 with α_2 using A_2. The definition is inductive[5] in the lengths of α_1 and α_2, and we include the empty sequence ϵ, to allow a simpler formulation:

$$\begin{aligned}
\alpha_1 \,{}_{A_1}\|_{A_2}\, \epsilon &= \{\alpha_1 \mid (A_1, A_2) \xrightarrow{\alpha_1} \cdot\} \\
\epsilon \,{}_{A_1}\|_{A_2}\, \alpha_2 &= \{\alpha_2 \mid (A_2, A_1) \xrightarrow{\alpha_2} \cdot\} \\
(\lambda_1\alpha_1) \,{}_{A_1}\|_{A_2}\, (\lambda_2\alpha_2) &= \{\mathit{abort}\} \quad \text{if } \lambda_1 \sharp \lambda_2 \\
&= \{\lambda_1\beta \mid \ (A_1, A_2) \xrightarrow{\lambda_1} (A_1', A_2) \ \&\ \beta \in \alpha_1 \,{}_{A_1'}\|_{A_2}(\lambda_2\alpha_2)\} \\
&\quad \cup \{\lambda_2\beta \mid \ (A_2, A_1) \xrightarrow{\lambda_2} (A_2', A_1) \ \&\ \beta \in (\lambda_1\alpha_1) \,{}_{A_1}\|_{A_2'}\, \alpha_2\} \\
&\qquad\qquad \text{otherwise}
\end{aligned}$$

For traces α_1 and α_2, let $\alpha_1 \| \alpha_2$ be defined to be $\alpha_1 {}_{\{\}}\|_{\{\}} \alpha_2$. For trace sets T_1 and T_2 we define $T_1 \| T_2 = \bigcup\{\alpha_1 \| \alpha_2 \mid \alpha_1 \in T_1 \ \&\ \alpha_2 \in T_2\}$.

Semantics of expressions

An expression will denote a set of *evaluation traces paired with values*: we define $[\![e]\!] \subseteq \mathbf{Tr} \times V_{int}$ for an integer expression e, and $[\![b]\!] \subseteq \mathbf{Tr} \times V_{bool}$ for a boolean expression b. Since expression values depend only on the store, the only non-trivial actions participating in such traces will be reads. To allow for the possibility of interference during expression evaluation we include both non-sequential and sequential evaluation traces. Again the sequential traces describe what happens if an expression is evaluated without interference.

The semantic functions are given, by structural induction, in the usual way. For example:

$$\begin{aligned}
&[\![10]\!] = \{(\delta, 10)\} \\
&[\![i]\!] = \{(i{=}v, v) \mid v \in V_{int}\} \\
&[\![e_1 + e_2]\!] = \{(\rho_1\rho_2, v_1 + v_2) \mid (\rho_1, v_1) \in [\![e_1]\!] \ \&\ (\rho_2, v_2) \in [\![e_2]\!]\} \\
&[\![(e_0, \dots, e_n)]\!] = \{(\rho_0 \cdots \rho_n, [v_0, \dots, v_n]) \mid \forall j.\ 0 \leq j \leq n \Rightarrow (\rho_j, v_j) \in [\![e_j]\!]\}.
\end{aligned}$$

The use of concatenation in these semantic clauses assumes that sum expressions and lists are evaluated in left-right order. This assumption is not crucial; it would

[5] We can also give a *coinductive* definition of the mutex fairmerges of two infinite traces, starting from a given disjoint pair of resource sets. We need mostly to work here with finite traces, so we omit the details.

be just as reasonable to assume parallel evaluation for such expressions. With an appropriately modified semantic definition, this adjustment can be made without affecting the ensuing development.

Let $[\![b]\!]_{\mathbf{true}} \subseteq \mathbf{Tr}$ be the set of traces ρ such that $(\rho, \mathbf{true}) \in [\![b]\!]$, and $[\![b]\!]_{\mathbf{false}}$ be the set of traces ρ such that $(\rho, \mathbf{false}) \in [\![b]\!]$.

Semantics of commands

A command c denotes a set $[\![c]\!] \subseteq \mathbf{Tr}$ of action traces, defined by structural induction.

Definition 2
The trace set $[\![c]\!]$ of a command c is defined by the following clauses:

$$
\begin{array}{l}
[\![\mathbf{skip}]\!] = \{\delta\} \\
[\![i{:=}e]\!] = \{\rho\, i{:=}v \mid (\rho, v) \in [\![e]\!]\} \\
[\![i{:=}[e]]\!] = \{\rho\, [v]{=}v'\, i{:=}v' \mid (\rho, v) \in [\![e]\!]\} \\
[\![i{:=}\mathbf{cons}\,(e_0, \ldots, e_n)]\!] = \{\rho\, alloc(l, L)\, i{:=}l \mid (\rho, L) \in [\![(e_0, \ldots, e_n)]\!]\} \\
[\![[e]{:=}e']\!] = \{\rho\, \rho'\, [v]{:=}v' \mid (\rho, v) \in [\![e]\!]\ \&\ (\rho', v') \in [\![e']\!]\} \\
[\![\mathbf{dispose}(e)]\!] = \{\rho\, disp(l) \mid (\rho, l) \in [\![e]\!]\} \\
[\![c_1; c_2]\!] = [\![c_1]\!]\, [\![c_2]\!] = \{\alpha_1\alpha_2 \mid \alpha_1 \in [\![c_1]\!]\ \&\ \alpha_2 \in [\![c_2]\!]\} \\
[\![\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2]\!] = [\![b]\!]_{\mathbf{true}}\, [\![c_1]\!]\ \cup\ [\![b]\!]_{\mathbf{false}}\, [\![c_2]\!] \\
[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] = ([\![b]\!]_{\mathbf{true}}\, [\![c]\!])^*\, [\![b]\!]_{\mathbf{false}}\ \cup\ ([\![b]\!]_{\mathbf{true}}\, [\![c]\!])^\omega \\
[\![c_1 \| c_2]\!] = [\![c_1]\!] \| [\![c_2]\!] \\
[\![\mathbf{with}\ r\ \mathbf{when}\ b\ \mathbf{do}\ c]\!] = wait^*\ enter\ \cup\ wait^\omega \\
\qquad\qquad \text{where } wait = acq(r)\, [\![b]\!]_{\mathbf{false}}\, rel(r)\ \cup\ \{try(r)\} \\
\qquad\qquad \text{and } \quad enter = acq(r)\, [\![b]\!]_{\mathbf{true}}\, [\![c]\!]\, rel(r) \\
[\![\mathbf{resource}\ r\ \mathbf{in}\ c]\!] = \{\alpha \backslash r \mid \alpha \in [\![c]\!]_r\}
\end{array}
$$

In the above semantic clauses we have prescribed a left-to-right sequential evaluation order for $i{:=}\mathbf{cons}(e_0, \ldots, e_n)$ and $[e]{:=}e'$, reflected in the use of concatenation on the traces of sub-expressions; again this assumption is not crucial, and it is straightforward to adapt the ensuing development to allow for parallel evaluation of sub-expressions.

The iterative structure of the traces of a conditional critical region reflect its use to achieve synchronization: waiting until the resource is available and the test condition is true, followed by execution of the body command while holding the resource, and finally releasing the resource. Note the possibility that the body may loop forever or encounter a runtime error, in which case the resource will not get released. Since $[\![\mathbf{true}]\!]_{\mathbf{false}} = \{\}$ and $[\![\mathbf{true}]\!]_{\mathbf{true}} = \{\delta\}$ it is easy to derive a simpler formula for the trace set of **with** r **do** c: we have

$$[\![\mathbf{with}\ r\ \mathbf{do}\ c]\!] = try(r)^*\ acq(r)\, [\![c]\!]\, rel(r)\ \cup\ \{try(r)^\omega\}.$$

Since the command **resource** r **in** c introduces a local resource named r, whose scope is c, its traces are obtained from traces of c in which r is assumed initially available and the actions involving r are executed without interference.

We let $[\![c]\!]_r$ be the set of traces of c which are *sequential for* r in this manner[6]. We let $\alpha\backslash r$ be the trace obtained from α by replacing each action on r by δ.

Examples

1. $[\![x{:=}x+1]\!] = \{x{=}v\; x{:=}v+1 \mid v \in V_{int}\}$
   This program always terminates, when executed from a state in which x has a value; its effect is to increment the value of x by 1.
2. $[\![x{:=}x+1 \| x{:=}x+1]\!] = \{x{=}v\; abort \mid v \in V_{int}\}$
   Concurrent assignments to the same identifier cause a race, no matter what the initial value of x is.
3. $[\![\textbf{with } r \textbf{ do } x{:=}x+1]\!] = try(r)^*\; acq(r)\; [\![x{:=}x+1]\!]\; rel(r) \;\cup\; \{try(r)^\omega\}$
   This program needs to acquire r before incrementing x, and will wait forever if the resource never becomes available.
4. $[\![\textbf{with } r \textbf{ do } x{:=}x+1 \| \textbf{with } r \textbf{ do } x{:=}x+1]\!]$ contains traces of the forms $acq(r)\,\alpha\, rel(r)\; acq(r)\,\beta\, rel(r)$, $acq(r)\,\alpha\, rel(r)\; try(r)^\omega$, and $try(r)^\omega$, where $\alpha, \beta \in [\![x{:=}x{+}1]\!]$, as well as traces of similar form containing additional $try(r)$ steps. Only the first kind are sequential for r. It follows that

   $$\begin{gathered}[\![\textbf{resource } r \textbf{ in } (\textbf{with } r \textbf{ do } x{:=}x+1 \| \textbf{with } r \textbf{ do } x{:=}x+1)]\!] \\ = \{\alpha\beta \mid \alpha, \beta \in [\![x{:=}x+1]\!]\} = [\![x{:=}x+1; x{:=}x+1]\!].\end{gathered}$$

   The overall effect is the same as that of two consecutive increments.
5. The command $x{:=}\textbf{cons}(1) \| y{:=}\textbf{cons}(2)$ has the trace set

   $$\{alloc(l,[1])\; x{:=}l \mid l \in V_{addr}\} \| \{alloc(l',[2])\; y{:=}l' \mid l' \in V_{addr}\}.$$

   This set includes traces of the form

   $$alloc(l,[1])\; x{:=}l\; alloc(l,[2])\; y{:=}l,$$

   and other interleavings of $alloc(l,[1])\, x{:=}l$ with $alloc(l,[2])\, y{:=}l$, none of which are sequential. The set also includes traces obtained by interleaving $alloc(l,[1])\, x{:=}l$ and $alloc(l',[2])\, y{:=}l'$, where $l \neq l'$; all of these are sequential.
6. The command $\textbf{dispose}(x) \| \textbf{dispose}(y)$ has trace set

   $$\{x{=}v\; disp(v) \mid v \in V_{addr}\} \| \{y{=}v'\; disp(v') \mid v' \in V_{addr}\}.$$

   This includes traces of the form $x{=}v\; y{=}v\; abort$ because of the race-detecting clause in the definition of fairmerge. If this command is executed from a state in which x and y are aliases a race will occur, with both processes attempting to dispose the same heap cell: if $s(x) = s(y) = v$ and $v \in \texttt{dom}(h)$ we have $(s, h, \{\}) \xRightarrow{x=v\; y=v\; abort} \textbf{abort}$.

[6] Technically, we say that α is sequential for r if $(\{\}, \{\}, \{\}) \xRightarrow{\alpha\lceil r} \cdot$ holds, where $\alpha\lceil r$ is the subsequence of α consisting of actions on resource r. This expresses formally the requirement that α represents an execution in which r is initially available and r is never acquired (or released) by the environment. Equivalently, $\alpha\lceil r$ is a prefix of a trace in the set $(acq(r)\; try(r)^\infty\; rel(r))^\infty$.

4 Concurrent separation logic

Separation logic [17] provides a class of formulas for specifying properties of stores and heaps. The syntax includes separating conjunction, denoted $p_1 * p_2$, and formulas **emp** and $e \mapsto e'$ specifying an empty heap and a singleton heap. We write $(s, h) \models p$ when(s, h) satisfies p. In particular, $(s, h) \models p_1 * p_2$ if and only if there are disjoint heaps h_1, h_2 such that $h = h_1 \cdot h_2$, $(s, h_1) \models p_1$, and $(s, h_2) \models p_2$. Reynolds [17] provides a Hoare-style partial correctness logic for sequential pointer-programs in which the pre- and post-conditions are separation logic formulas.

We now introduce resource-sensitive partial correctness formulas of the form $\Gamma \vdash \{p\}c\{q\}$, where p and q are separation logic formulas, c is a parallel pointer-program, and Γ is a *resource context* $r_1(X_1) : R_1, \ldots, r_k(X_k) : R_k$ associating resource names r_j with protection lists X_j and resource invariants R_j. Each protection list represents a finite set of identifiers. We require each resource invariant to be a *precise* separation logic formula. A separation logic formula p is *precise* [17] if for all s and h, there is at most one $h' \subseteq h$ such that $(s, h') \models p$.

Let $\mathtt{dom}(\Gamma) = \{r_1, \ldots, r_k\}$ be the set of resource names in Γ, $\mathtt{owned}(\Gamma) = \bigcup_{j=1}^{k} X_j$ be the set of identifiers protected by Γ, and $\mathtt{free}(\Gamma) = \bigcup_{j=1}^{k} \mathtt{free}(R_j)$ be the set of identifiers mentioned in the invariants. Let $\mathtt{inv}(\Gamma) = R_1 * \cdots * R_k$ be the separating conjunction of the resource invariants in Γ. In particular, when Γ is empty this is **emp**. Since each resource invariant is precise it follows that $\mathtt{inv}(\Gamma)$ is precise.

We will impose some syntactic *well-formedness* constraints on contexts and formulas, designed to facilitate modularity. Specifically:

- Γ is well-formed if its entries are disjoint, in that if $i \neq j$ then $r_i \neq r_j$, $X_i \cap X_j = \{\}$, and $\mathtt{free}(R_i) \cap X_j = \{\}$.
- $\Gamma \vdash \{p\}c\{q\}$ is well-formed if Γ is well-formed, and p and q do not mention any protected identifiers, i.e. $\mathtt{free}(p, q) \cap \mathtt{owned}(\Gamma) = \{\}$.

Thus in a well-formed context each identifier belongs to at most one resource. We do *not* require that the free identifiers in a resource invariant be protected, i.e. that $\mathtt{free}(R_i) \subseteq X_i$. This allows us to use a resource invariant to connect the values of protected identifiers and the values of non-critical variables.

The inference rules will enforce the following syntactic constraints on commands, relative to the relevant resource context[7]:

- Every critical identifier is protected by a resource.
- Every free occurrence of a protected identifier is within a region for the corresponding resource.
- Every free write occurrence of an identifier mentioned in a resource invariant is within a region for the corresponding resource.

[7] We will not formalize these properties or give a proof that they hold in all provable formulas. We state them explicitly since they recall analogous requirements in the Owicki-Gries logic and in O'Hearn's rules.

Intuitively, a resource-sensitive partial correctness formula specifies how a program behaves when executed in an environment that respects the resource context, assuming that at all times the *separating conjunction* of the resource invariants holds, for the currently available resources. The program guarantees to stay within these bounds, provided it can rely on its environment to do likewise. This informal notion of validity for formulas should help provide intuition for the structure of the following inference rules. Later we will give a formal definition of validity.

We allow all well-formed instances of the following inference rules. Some of the rules have side conditions to ensure well-formedness and the syntactic requirements given above, as in [12].

– Skip

$$\frac{}{\Gamma \vdash \{p\}\mathbf{skip}\{p\}}$$

– Assignment

$$\frac{}{\Gamma \vdash \{[e/i]p\}i{:=}e\{p\}}$$

if $i \notin \mathtt{owned}(\Gamma) \cup \mathtt{free}(\Gamma)$

– Lookup

$$\frac{}{\Gamma \vdash \{[e'/i]p \wedge e \mapsto e'\}i{:=}[e]\{p \wedge e \mapsto e'\}}$$

if $i \notin \mathtt{free}(e, e')$ and $i \notin \mathtt{owned}(\Gamma) \cup \mathtt{free}(\Gamma)$

– Allocation

$$\frac{}{\Gamma \vdash \{\mathbf{emp}\}i{:=}\mathbf{cons}(e_0, \ldots, e_n)\{i \mapsto e_0 * \cdots * i + n \mapsto e_n\}}$$

if $i \notin \mathtt{free}(e_0, \ldots, e_n)$ and $i \notin \mathtt{owned}(\Gamma) \cup \mathtt{free}(\Gamma)$

– Update

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

– Disposal

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

– Sequential

$$\frac{\Gamma \vdash \{p_1\}c_1\{p_2\} \quad \Gamma \vdash \{p_2\}c_2\{p_3\}}{\Gamma \vdash \{p_1\}c_1; c_2\{p_3\}}$$

– Conditional

$$\frac{\Gamma \vdash \{p \wedge b\}c_1\{q\} \quad \Gamma \vdash \{p \wedge \neg b\}c_2\{q\}}{\Gamma \vdash \{p\}\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\{q\}}$$

– Loop

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

– Parallel

$$\frac{\Gamma \vdash \{p_1\}c_1\{q_1\} \quad \Gamma \vdash \{p_2\}c_2\{q_2\}}{\Gamma \vdash \{p_1 * p_2\}c_1 \| c_2\{q_1 * q_2\}}$$

if $\quad \texttt{free}(p_1, q_1) \cap \texttt{writes}(c_2) = \texttt{free}(p_2, q_2) \cap \texttt{writes}(c_1) = \{\}$
and $(\texttt{free}(c_1) \cap \texttt{writes}(c_2)) \cup (\texttt{free}(c_2) \cap \texttt{writes}(c_1)) \subseteq \texttt{owned}(\Gamma)$

– Resource

$$\frac{\Gamma, r(X) : R \vdash \{p\}c\{q\}}{\Gamma \vdash \{p * R\}\textbf{resource } r \textbf{ in } c\{q * R\}}$$

– Renaming resource

$$\frac{\Gamma \vdash \{p\}\textbf{resource } r' \textbf{ in } [r'/r]c\{q\}}{\Gamma \vdash \{p\}\textbf{resource } r \textbf{ in } c\{q\}}$$

if $r' \notin \texttt{res}(c)$

– Region

$$\frac{\Gamma \vdash \{(p * R) \wedge b\}c\{q * R\}}{\Gamma, r(X) : R \vdash \{p\}\textbf{with } r \textbf{ when } b \textbf{ do } c\{q\}}$$

– Frame

$$\frac{\Gamma \vdash \{p\}c\{q\}}{\Gamma \vdash \{p * I\}c\{q * I\}}$$

if $\texttt{free}(I) \cap \texttt{writes}(c) = \{\}$

– Consequence

$$\frac{p' \Rightarrow p \quad \Gamma \vdash \{p\}c\{q\} \quad q \Rightarrow q'}{\Gamma \vdash \{p'\}c\{q'\}}$$

provided $p' \Rightarrow p$ and $q \Rightarrow q'$ are universally valid

– Auxiliary

$$\frac{\Gamma \vdash \{p\}c\{q\}}{\Gamma \vdash \{p\}c \backslash X\{q\}}$$

if X is auxiliary for c, and $X \cap \texttt{free}(p, q) = \{\}$.

– Conjunction

$$\frac{\Gamma \vdash \{p_1\}c\{q_1\} \qquad \Gamma \vdash \{p_2\}c\{q_2\}}{\Gamma \vdash \{p_1 \wedge p_2\}c\{q_1 \wedge q_2\}}$$

– Expansion

$$\frac{\Gamma \vdash \{p\}c\{q\}}{\Gamma, \Gamma' \vdash \{p\}c\{q\}}$$

if $\texttt{writes}(c) \cap \texttt{free}(\Gamma') = \{\}$ and $\texttt{free}(c) \cap \texttt{owned}(\Gamma') = \{\}$

– Contraction

$$\frac{\Gamma, \Gamma' \vdash \{p\}c\{q\}}{\Gamma \vdash \{p\}c\{q\}}$$

if $\texttt{res}(c) \subseteq \texttt{dom}(\Gamma)$

Comments

1. The rules dealing with the sequential program constructs are natural adaptations of the rules given by Reynolds [17], with the incorporation of a resource context and side conditions to ensure well-formedness and adherence to the protection policy. The Frame and Consequence rules similarly generalize analogous rules from the sequential setting.
2. The Parallel, Region and Resource rules are based on O'Hearn's adaptations of Owicki-Gries inference rules. A side condition in the Parallel rule enforces the requirement that each critical variable must be associated with a resource, just as in the original Owicki-Gries rule.
3. The Auxiliary rule similarly adapts the Owicki/Gries rule for auxiliary variables[8]. As usual, a set of identifiers X is said to be *auxiliary* for c if every free occurrence in c of an identifier from X is in an assignment that only affects the values of identifiers in X. In particular, auxiliary identifiers cannot occur in conditional tests or loop tests, and do not influence the control flow of the program. The command $c\backslash X$ is obtained from c by deleting assignments to identifiers in X.
4. In the Resource renaming rule we write $[r'/r]c$ for the command obtained from c by replacing each free occurrence of r by r'.
5. We have omitted the obvious structural rules permitting permutation of resource contexts.

5 Validity

We wish to establish that every provable resource-sensitive formula is *valid*, but we need to determine precisely what that should mean. Adapting the notion of validity familiar from the sequential setting, we might try to interpret validity of $\Gamma \vdash \{p\}c\{q\}$ as the property that every finite computation of c from a state satisfying $p * \texttt{inv}(\Gamma)$ is error-free and ends in a state satisfying $q * \texttt{inv}(\Gamma)$. However, this notion of "sequential validity" is not compositional for parallel programs; although it expresses a desirable property we need a notion of validity that takes account of process interaction.

Informally we might say that the formula $\Gamma \vdash \{p\}c\{q\}$ is valid if every finite interactive computation of c from a state satisfying $p * \texttt{inv}(\Gamma)$, in an environment that respects Γ, is error-free, also respects Γ, and ends in a state satisfying $q * \texttt{inv}(\Gamma)$. However, such a formulation would be incomplete, since it does not properly specify what "respect" for Γ entails. To obtain a suitably formal (and compositional) notion of validity we need to keep track of the portions of the state deemed to be "owned" by a process, its environment, and the available resources.

With respect to a resource context Γ, a process holding resource set A should be allowed to access identifiers protected by resources in A, but not identifiers

[8] Owicki and Gries cite Brinch Hansen [2] and Lauer [10] as having first recognized the need for auxiliary variables in proving correctness properties of concurrent programs.

protected by other resources. We say that (s, h, A) is a *local state* consistent with Γ if $\mathtt{dom}(s) \cap \mathtt{owned}(\Gamma) = \mathtt{owned}(\Gamma\lceil A)$, where $\Gamma\lceil A$ is the subset of Γ involving resources in A. We let $\Gamma\backslash A$ be the rest of Γ. We introduce *local enabling relations* between local states: a step

$$(s, h, A) \xrightarrow[\Gamma]{\lambda} (s', h', A')$$

means that in state (s, h, A) a process can perform action λ, causing its local state to change to (s', h', A') and respecting the resource invariants and protection rules. We use the error state **abort** to handle runtime errors and logical errors such as an attempt to release a resource in a state for which no sub-heap satisfies the corresponding invariant, or a write to an identifier mentioned in a resource invariant without first acquiring the resource.

Definition 3 *The local enabling relations* $\xrightarrow[\Gamma]{\lambda}$ *are given by the following clauses, in which* (s, h, A) *ranges over local states consistent with* Γ*:*

$(s, h, A) \xrightarrow[\Gamma]{\delta} (s, h, A)$	*always*
$(s, h, A) \xrightarrow[\Gamma]{abort} \textbf{abort}$	*always*
$(s, h, A) \xrightarrow[\Gamma]{i=v} (s, h, A)$	*iff* $(i, v) \in s$
$(s, h, A) \xrightarrow[\Gamma]{i=v} \textbf{abort}$	*iff* $i \notin \mathtt{dom}(s)$
$(s, h, A) \xrightarrow[\Gamma]{i:=v} ([s \mid i : v], h, A)$	*iff* $i \in \mathtt{dom}(s) - \mathtt{free}(\Gamma\backslash A)$
$(s, h, A) \xrightarrow[\Gamma]{i:=v} \textbf{abort}$	*iff* $i \notin \mathtt{dom}(s)$ *or* $i \in \mathtt{free}(\Gamma\backslash A)$
$(s, h, A) \xrightarrow[\Gamma]{[l]=v} (s, h, A)$	*iff* $(l, v) \in h$
$(s, h, A) \xrightarrow[\Gamma]{[l]=v} \textbf{abort}$	*iff* $l \notin \mathtt{dom}(h)$
$(s, h, A) \xrightarrow[\Gamma]{[l]:=v'} (s, [h \mid l : v'], A)$	*iff* $l \in \mathtt{dom}(h)$
$(s, h, A) \xrightarrow[\Gamma]{[l]:=v'} \textbf{abort}$	*iff* $l \notin \mathtt{dom}(h)$
$(s, h, A) \xrightarrow[\Gamma]{alloc(l,[v_0,\ldots,v_n])} (s, [h \mid l{:}v_0, \ldots, l + n{:}v_n], A)$	*iff* $\forall m \leq n.\ l + m \notin dom(h)$
$(s, h, A) \xrightarrow[\Gamma]{disp(l)} (s, h\backslash l, A)$	*iff* $l \in \mathtt{dom}(h)$
$(s, h, A) \xrightarrow[\Gamma]{disp(l)} \textbf{abort}$	*iff* $l \notin \mathtt{dom}(h)$
$(s, h, A) \xrightarrow[\Gamma]{try(r)} (s, h, A)$	*always*
$(s, h, A) \xrightarrow[\Gamma]{acq(r)} (s \cdot s', h \cdot h', A \cup \{r\})$	*iff* $r(X){:}R \in \Gamma,\ r \notin A,\ h \perp h',$ $\mathtt{dom}(s') = X,$ *and* $(s \cdot s', h') \models R$
$(s, h, A) \xrightarrow[\Gamma]{rel(r)} (s\backslash X, h - h', A - \{r\})$	*iff* $r(X){:}R \in \Gamma,\ r \in A,\ h' \subseteq h,\ (s, h') \models R$
$(s, h, A) \xrightarrow[\Gamma]{rel(r)} \textbf{abort}$	*iff* $r(X){:}R \in \Gamma$ *and* $\forall h' \subseteq h.\ \neg(s, h') \models R$

The clauses for $acq(r)$ and $rel(r)$ deal with ownership transfer: when a process acquires a resource its local state grows to include the identifiers protected by the resource and the heap portion in which the resource invariant holds; when a process releases a resource its local state ceases to include the protected identifiers and the heap associated with the resource invariant; a "logical" error occurs if the invariant is not suitably satisfied. Since resource invariants are assumed to be precise formulas in each case there is a uniquely determined portion of heap associated with the relevant invariant.

We write $\sigma \xrightarrow[\Gamma]{\alpha} \sigma'$ when there is a local computation α from σ to σ'.

Note that non-sequential traces play a non-trivial role in the local enabling relation, and in a local computation external interference can occur only at a resource acquisition step. Thus the local enabling relation provides a formalization of "loosely connected" processes in the spirit of Dijkstra.

The following result connects the local enabling relations $\xrightarrow[\Gamma]{\alpha}$, which model interactive execution in an environment that respects a resource context, and the effect relations $\overset{\alpha}{\Longrightarrow}$, which represent interference-free executions, when α is a sequential trace.

Lemma 1 (Empty Transfer Lemma)
Let α be a finite trace, let $\{r_1, \ldots, r_n\}$ be the set of resource names occurring in actions of α, and let Γ_0 be the resource context $r_1(\{\}) : \mathbf{emp}, \ldots, r_n(\{\}) : \mathbf{emp}$. Then $(s, h, A) \overset{\alpha}{\Longrightarrow} \sigma'$ if and only if $(s, h, A) \xrightarrow[\Gamma_0]{\alpha} \sigma'$.

Theorem 2 (Respect for resources)
If $\alpha \in [\![c]\!]$ and $(s, h, A) \xrightarrow[\Gamma]{\alpha} (s', h', A')$, then $\mathtt{dom}(s') = \mathtt{dom}(s)$ and $A = A'$.

Note that these results imply the corresponding property for sequential traces.

Corollary 3
If $\alpha \in [\![c]\!]$ and $(s, h, A) \overset{\alpha}{\Longrightarrow} (s', h', A')$, then $\mathtt{dom}(s) = \mathtt{dom}(s')$ and $A = A'$.

The following *parallel decomposition* property relates a local computation of a parallel program to local computations of its components. If the critical identifiers of c_1 and c_2 are protected by resources in Γ, a local computation of $c_1 \| c_2$ can be "projected" into a local computation of c_1 and a local computation of c_2. In stating this property we use (s, h) as an abbreviation for $(s, h, \{\})$.

Theorem 4 (Parallel Decomposition)
Suppose $(\mathtt{free}(c_1) \cap \mathtt{writes}(c_2)) \cup (\mathtt{writes}(c_1) \cap \mathtt{free}(c_2)) \subseteq \mathtt{owned}(\Gamma)$ and $\alpha \in \alpha_1 \| \alpha_2$, where $\alpha_1 \in [\![c_1]\!]$ and $\alpha_2 \in [\![c_2]\!]$. Suppose $h_1 \perp h_2$ and $h = h_1 \cdot h_2$.

- *If $(s, h) \xrightarrow[\Gamma]{\alpha} \mathbf{abort}$ then*
  $(s \backslash \mathtt{writes}(c_2), h_1) \xrightarrow[\Gamma]{\alpha_1} \mathbf{abort}$ or $(s \backslash \mathtt{writes}(c_1), h_2) \xrightarrow[\Gamma]{\alpha_2} \mathbf{abort}$.
- *If $(s, h) \xrightarrow[\Gamma]{\alpha} (s', h')$ then*
  $(s \backslash \mathtt{writes}(c_2), h_1) \xrightarrow[\Gamma]{\alpha_1} \mathbf{abort}$ or $(s \backslash \mathtt{writes}(c_1), h_2) \xrightarrow[\Gamma]{\alpha_2} \mathbf{abort}$,
  or there are disjoint heaps $h'_1 \perp h'_2$ such that $h' = h'_1 \cdot h'_2$ and
  - $(s \backslash \mathtt{writes}(c_2), h_1) \xrightarrow[\Gamma]{\alpha_1} (s' \backslash \mathtt{writes}(c_2), h'_1)$
  - $(s \backslash \mathtt{writes}(c_1), h_2) \xrightarrow[\Gamma]{\alpha_2} (s' \backslash \mathtt{writes}(c_1), h'_2)$

The definition of local enabling formalizes the notion of a computation by a process, in an environment that respects resources, and "minds its own business" by obeying the ownership policy of a given resource context. This leads us to the following rigorous formulation of validity. Again we write (s, h) for $(s, h, \{\})$.

Definition 5
The formula $\Gamma \vdash \{p\}c\{q\}$ is valid if for all traces α of c, all local states (s,h) such that $\texttt{dom}(s) \supseteq \texttt{free}(c,\Gamma) - \texttt{owned}(\Gamma)$, and all σ', if $(s,h) \models p$ and $(s,h) \xrightarrow[\Gamma]{\alpha} \sigma'$ then $\sigma' \neq \textbf{abort}$ and $\sigma' \models q$.

This definition uses the local enabling relation, so that the quantification ranges over local states (s,h) consistent with Γ, for which $\texttt{dom}(s) \cap \texttt{owned}(\Gamma) = \{\}$. Furthermore, this notion of validity involves *all* traces of c, not just the sequential traces and not just the finite traces[9].

When Γ is the empty context and $\texttt{res}(c) = \{\}$, validity of $\{\} \vdash \{p\}c\{q\}$ implies the usual notion of partial correctness together with the guaranteed absence of runtime errors. More generally, the same implication holds when $\texttt{res}(c) = \{r_1, \ldots, r_n\}$ and Γ is the context $r_1(\{\}) : \textbf{emp}, \ldots, r_n(\{\}) : \textbf{emp}$.

We now come to the main result of this paper: soundness of concurrent separation logic.

Theorem 6 (Soundness)
Every provable formula $\Gamma \vdash \{p\}c\{q\}$ is valid.

Proof:
Show that each well formed instance of an inference rule is sound: if the rule's premisses and conclusion are well formed, the side conditions hold, and the premisses are valid, then the conclusion is valid. It then follows, by induction on the length of the derivation, that every provable formula is valid.

We give details for the Parallel rule.

- Parallel Composition
  Suppose that $\Gamma \vdash \{p_1\}c_1\{q_1\}$ and $\Gamma \vdash \{p_2\}c_2\{q_2\}$ are well formed and valid, and that $\texttt{free}(p_1,q_1) \cap \texttt{writes}(c_2) = \texttt{free}(p_2,q_2) \cap \texttt{writes}(c_1) = \{\}$ and $(\texttt{free}(c_1) \cap \texttt{writes}(c_2)) \cup (\texttt{writes}(c_1) \cap \texttt{free}(c_2)) \subseteq \texttt{owned}(\Gamma)$.
  It is clear that $\Gamma \vdash \{p_1 * p_2\}c_1\|c_2\{q_1 * q_2\}$ is well formed. We must show that $\Gamma \vdash \{p_1 * p_2\}c_1\|c_2\{q_1 * q_2\}$ is valid.
  Let $(s,h) \models p_1 * p_2$, and suppose $h_1 \perp h_2$, $h = h_1 \cdot h_2$, and $(s,h_1) \models p_1$, $(s,h_2) \models p_2$. Since $\texttt{free}(p_1) \cap \texttt{writes}(c_2) = \texttt{free}(p_2) \cap \texttt{writes}(c_1) = \{\}$ we also have $(s\backslash\texttt{writes}(c_2), h_1) \models p_1$ and $(s\backslash\texttt{writes}(c_1), h_2) \models p_2$.
  Let $\alpha \in [\![c_1\|c_2]\!]$, and $(s,h) \xrightarrow[\Gamma]{\alpha} \sigma'$. Choose traces $\alpha_1 \in [\![c_1]\!]$ and $\alpha_2 \in [\![c_2]\!]$ such that $\alpha \in \alpha_1\|\alpha_2$. If $\sigma' = \textbf{abort}$ the Parallel Decomposition Lemma would imply that $(s\backslash\texttt{writes}(c_2), h_1) \xrightarrow[\Gamma]{\alpha_1} \textbf{abort}$ or $(s\backslash\texttt{writes}(c_1), h_2) \xrightarrow[\Gamma]{\alpha_2} \textbf{abort}$. Neither of these is possible, since they contradict the assumed validity of the premisses $\Gamma \vdash \{p_1\}c_1\{q_1\}$ and $\Gamma \vdash \{p_2\}c_2\{q_2\}$. If α is infinite that is all we need. Otherwise α is finite, and σ' has the form (s',h'). Again by the Parallel Decomposition Lemma and validity of the premisses, there are heaps $h_1' \perp h_2'$ such that $h' = h_1' \cdot h_2'$,

$$\begin{aligned}(s\backslash\texttt{writes}(c_2), h_1) &\xrightarrow[\Gamma]{\alpha_1} (s'\backslash\texttt{writes}(c_2), h_1')\\ (s\backslash\texttt{writes}(c_1), h_2) &\xrightarrow[\Gamma]{\alpha_2} (s'\backslash\texttt{writes}(c_1), h_2'),\end{aligned}$$

[9] The infinite traces only really matter in the no-**abort** requirement, since we never get $\sigma \xrightarrow[\Gamma]{\alpha} \sigma'$ when α is infinite and σ' is a proper state.

and $(s'\backslash\texttt{writes}(c_2), h_1') \models q_1$, $(s'\backslash\texttt{writes}(c_1), h_2') \models q_2$. Since q_1 does not depend on $\texttt{writes}(c_2)$ and q_2 does not depend on $\texttt{writes}(c_1)$ we also have $(s', h_1') \models q_1$ and $(s', h_2') \models q_2$, from which it follows that $(s', h') \models q_1 * q_2$, as required.

6 Provability implies no races

For a process holding resource set A and a corresponding global state (s, h, A), let $s{\downarrow}A = s\backslash\texttt{owned}(\Gamma\backslash A)$. This is the "local" portion of the global store "visible" to the process by virtue of its current resource set.

The following result shows how the local effect of an action relates to its global effect, modulo the protection policy imposed by the resource context, assuming that the process performing the action owns resources in A and the global heap contains a sub-heap in which the resource invariants for the available resources hold, separately.

Lemma 7 (Connection Property)
Let (s, h, A) be a global state and suppose $h = h_1 \cdot h_2$ with $(s, h_2) \models \texttt{inv}(\Gamma\backslash A)$.

- *If $(s, h, A) \overset{\lambda}{\Longrightarrow}$ **abort** then $(s{\downarrow}A, h_1, A) \xrightarrow[\Gamma]{\lambda}$ **abort**.*
- *If $(s, h, A) \overset{\lambda}{\Longrightarrow} (s', h', A')$ then*
  - *either $(s{\downarrow}A, h_1, A) \xrightarrow[\Gamma]{\lambda}$ **abort***
  - *or there are heaps $h_1' \perp h_2'$ such that $h' = h_1' \cdot h_2'$, $(s', h_2') \models \texttt{inv}(\Gamma\backslash A')$, and $(s{\downarrow}A, h_1, A) \xrightarrow[\Gamma]{\lambda} (s'{\downarrow}A', h_1', A')$*

We can then deduce the following result for all commands c, letting $A = A' = \{\}$ and using induction on trace structure.

Corollary 8
Let $\alpha \in [\![c]\!]$. Suppose $h = h_1 \cdot h_2$, and $(s, h_2) \models \texttt{inv}(\Gamma)$.

- *If $(s, h) \overset{\alpha}{\Longrightarrow}$ **abort** then $(s\backslash\texttt{owned}(\Gamma), h_1) \xrightarrow[\Gamma]{\alpha}$ **abort**.*
- *If $(s, h) \overset{\alpha}{\Longrightarrow} (s', h')$ then*
  - *either $(s\backslash\texttt{owned}(\Gamma), h_1) \xrightarrow[\Gamma]{\alpha}$ **abort**,*
  - *or there are heaps $h_1' \perp h_2'$ such that $h' = h_1' \cdot h_2'$, $(s', h_2') \models \texttt{inv}(\Gamma)$, and $(s\backslash\texttt{owned}(\Gamma), h_1) \xrightarrow[\Gamma]{\alpha} (s'\backslash\texttt{owned}(\Gamma), h_1')$.*

Finally, combining this with the definition of validity we obtain a link with the earlier notion of "sequential validity", which we can express rigorously in terms of the interference-free enabling relations $\overset{\alpha}{\Longrightarrow}$.

Theorem 9 (Valid implies race-free)
*If $\Gamma \vdash \{p\}c\{q\}$ is valid and well formed, then c is error-free from every global state satisfying $p * \texttt{inv}(\Gamma)$. More specifically, for all states σ, σ' and all traces $\alpha \in [\![c]\!]$, if $\sigma \models p * \texttt{inv}(\Gamma)$ and $\sigma \overset{\alpha}{\Longrightarrow} \sigma'$ then $\sigma' \neq$ **abort** and $\sigma' \models q * \texttt{inv}(\Gamma)$.*

Combining this result with the Soundness Theorem, it follows that provability of $\Gamma \vdash \{p\}c\{q\}$ implies that c is race-free from all states satisfying $p * \texttt{inv}(\Gamma)$.

7 Acknowledgements

I have benefitted immensely from discussions with Peter O'Hearn, whose ideas from [11] prompted this work; John Reynolds, who suggested treating races catastrophically; and Josh Berdine, whose insights led to technical improvements. The anonymous referees also made helpful suggestions.

References

1. P. Brinch Hansen. *Structured multiprogramming.* Comm. ACM, 15(7):574-578, July 1972.
2. P. Brinch Hansen. *Concurrent programming concepts.* ACM Computing Surveys 5(4):223-245, December 1973.
3. S. Brookes, *Traces, pomsets, fairness and full abstraction for communicating processes.* Proc. CONCUR 2002, Brno. Springer LNCS vol. 2421, pp. 466-482. August 2002.
4. S. Brookes. *Communicating Parallel Processes: Deconstructing CSP.* In: **Millenium Perspectives in Computer Science**, Proc. 1999 Oxford-Microsoft Symposium in honour of Sir Tony Hoare. Palgrave, 2000.
5. S. Brookes. *Full abstraction for a shared-variable parallel language.* Inf. Comp., vol 127(2):145-163, Academic Press, June 1996.
6. E. W. Dijkstra. *Cooperating sequential processes.* In: **Programming Languages**, F. Genuys (editor), pp. 43-112. Academic Press, 1968.
7. C.A.R. Hoare, *Towards a Theory of Parallel Programming.* In **Operating Systems Techniques**, C. A. R. Hoare and R. H. Perrott, editors, pp. 61-71, Academic Press, 1972.
8. S. Isthiaq and P. W. O'Hearn. *BI as an assertion language for mutable data structures.* Proc. 28^{th} POPL conference, pp. 36-49, January 2001.
9. C.B. Jones. *Specification and design of (parallel) programs.* Proc. IFIP Conference, 1983.
10. H.C. Lauer. *Correctness in operating systems.* Ph. D. thesis, Carnegie Mellon University, 1973.
11. P. W. O'Hearn. *Notes on separation logic for shared-variable concurrency.* Unpublished manuscript, January 2002.
12. P.W. O'Hearn. *Resources, Concurrency, and Local Reasoning.* This volume, Springer LNCS, CONCUR 2004, London, August 2004.
13. P.W. O'Hearn, H. Yang, and J.C. Reynolds. *Separation and Information Hiding.* Proc. 31^{st} POPL conference, pp 268-280, Venice. ACM Press, January 2004.
14. P. W. O'Hearn and D. J. Pym. *The logic of bunched implications.* Bulletin of Symbolic Logic, 5(2):215-244, June 1999.
15. S. Owicki and D. Gries, *Verifying properties of parallel programs: An axiomatic approach*, Comm. ACM. 19(5):279-285, May 1976.
16. D. Park, *On the semantics of fair parallelism.* In: **Abstract Software Specifications**, Springer-Verlag LNCS vol. 86, 504–526, 1979.
17. J.C. Reynolds, *Separation logic: a logic for shared mutable data structures*, Invited paper. Proc. 17^{th} IEEE Conference on Logic in Computer Science, LICS 2002, pp. 55-74. IEEE Computer Society, 2002.
18. J. C. Reynolds. Lecture notes on separation logic (15-819A3), chapter 8, page 178. Department of Computer Science, Carnegie-Mellon University, Spring 2003. Revised May 23, 2003.