Available online at www.sciencedirect.com
ScienceDirect
Theoretical Computer Science 375 (2007) 227–270
Theoretical Computer Science
www.elsevier.com/locate/tcs

A semantics for concurrent separation logic

Stephen Brookes*

School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, United States

Abstract

We present a trace semantics for a language of parallel programs which share access to mutable data. We introduce a resource-sensitive logic for partial correctness, based on a recent proposal of O'Hearn, adapting separation logic to the concurrent setting. The logic allows proofs of parallel programs in which "ownership" of critical data, such as the right to access, update or deallocate a pointer, is transferred dynamically between concurrent processes. We prove soundness of the logic, using a novel "local" interpretation of traces which allows accurate reasoning about ownership. We show that every provable program is race-free.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Concurrency; Pointers; Race condition; Semantics; Logic

1. Introduction

Parallel programs typically involve the concurrent execution of processes which share state and are intended to cooperate to achieve a collective goal. It is notoriously difficult to ensure that process interactions are sufficiently disciplined to preclude undesirable phenomena such as *races*, in which one process changes a piece of state that is simultaneously being used by another process. Races can result in unpredictable, possibly irreproducible, behaviour. In addition to goals expressible as *partial correctness* or *total correctness* properties, we often need to be able to establish *safety* properties, of the form that something bad never happens, and *liveness* properties, of the form that something good happens eventually [35]. Rather than relying on possibly unrealistic assumptions about the granularity of hardware primitives, we would prefer to use proof techniques that guarantee both race-freedom and correctness.

Program design rules based on *resource separation* [22,24,36,37] and the use of synchronization constructs such as conditional critical regions [7,9,8,24] offer the programmer a means to impose discipline. For example, building on an earlier proposal of Hoare [22,24], Owicki and Gries [36,37] introduced a syntax-directed logic for partial correctness of simple shared-memory parallel programs. A key notion behind the success of this approach is its focus on the *critical variables* of a program, characterized as identifiers which may be concurrently written by one process and read or written by another. The programmer is required to partition the critical variables among a fixed collection of *resources*, and to obey a simple syntactic constraint on program structure: each occurrence of a critical variable must be inside a conditional critical region naming the relevant resource. Assuming that resource management is implemented

* Tel.: +1 412 268 8820; fax: +1 412 268 4801.
E-mail address: brookes@cs.cmu.edu.

0304-3975/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2006.12.034

using a suitable low-level synchronization primitive, such as semaphores [19,20], so that at all stages during program execution each resource is held by at most one process, these statically enforceable design rules guarantee mutually exclusive access to the critical variables and therefore freedom from races. The Owicki–Gries inference rules support a modular methodology based on *resource invariants*, in which each process relies on its environment to ensure that whenever a resource is available the corresponding resource invariant holds, and guarantees that whenever a process releases the resource the invariant will hold again. This usage of invariants also serves to simplify the task of program proving, since it abstracts away from what happens "inside" a critical region and focuses instead only on the places where synchronization occurs.

This methodology works well for simple (pointer-free) shared-memory programs, but breaks down when the shared state can contain pointers. The Owicki–Gries rule for parallel composition is unsound for parallel programs that manipulate pointers ("pointer-programs"), because of the possibility of race conditions involving concurrent attempts to deallocate or update a pointer being used by another process. The problems are exacerbated by the possibility of aliasing: syntactically distinct expressions may denote the same pointer value. It is not possible to use purely syntactic constraints to rule out races (and restore soundness) for pointer-programs, because aliasing cannot be detected adequately by static analysis alone.

Pointers require a more sophisticated model of state: a *store* mapping identifiers to values, which may be data values such as integers, or pointer values such as addresses; and a *heap* mapping addresses to values, which again can be data or pointers. For sequential pointer-programs one can give a straightforward denotational or operational semantics based on state transformations, and *separation logic* has been developed as an extension of Hoare-style partial correctness logic to allow reasoning about the store and the heap [41,25]. The key feature of separation logic is a *separating conjunction*, used to specify disjointness constraints. Separation logic has been applied successfully to a range of significant examples [2,4,33,40,44]. The approach suggests a style of *local reasoning* in which one focusses on the "footprint" of a command, i.e. the minimal portion of state actually relevant to the command's execution, and one appeals to a "Frame Rule" whenever necessary to deduce that the command has no effect outside of its footprint [4,25,33,44].

Recently, O'Hearn has proposed using separation logic, together with an adaptation of the Owicki–Gries resource-based methodology, for reasoning about partial correctness of *parallel* pointer-programs [30,31]. Again the shared state is viewed as being partitioned among named resources, each equipped with a resource invariant and a protection list. O'Hearn proposed a methodology based on the following separation hypothesis: at all times the state can be partitioned to yield a *separate* portion for each process, and a *separate* portion, satisfying the relevant resource invariant, for each available resource. It then becomes possible to give a natural *ownership* interpretation of program execution. In particular, the heap portions associated with each process, and with each available resource, are always mutually disjoint. When a process acquires a resource it claims ownership of the state associated with that resource; when releasing the resource it must ensure that the invariant holds again, and returns ownership of the corresponding piece of state. Although the heap portion associated with a resource may vary dynamically, at all stages the Separation Hypothesis ensures that each piece of heap is accessed by at most one process. It thus becomes possible to reason safely about parallel programs in which "ownership" of a pointer, or some fragment of shared state, can be deemed to transfer dynamically between processes, or between a process and a resource: the partitioning of state among resources is not required to stay fixed throughout execution, but may adjust itself dynamically.

The main novelty in O'Hearn's adaptation involves the judicious use of the separating form of conjunction in key places in the pre- and post-conditions of the inference rules which deal with resources. Although this might appear superficially to produce "obvious" variants of the traditional rules, the soundness of the new rules is far from obvious. Indeed, to indicate the difficulties, Reynolds has shown that similar rules (even for *sequential* programs) are unsound if used without restrictions on the formulas allowed as resource invariants [42,31]. Moreover the traditional rules are unsound for pointer-programs, so soundness of the new rules cannot be deduced merely by analogy. O'Hearn provides a series of compelling examples of concurrent programs and informal correctness proofs [31], but (as he remarks) the logic cannot properly be assessed without a suitable semantic model [30].

However, it is not at all obvious how to provide a semantics that permits a formalization of the notions of ownership transfer and race-freedom, and such a semantics is crucial in establishing soundness. Traditional semantic models for shared-memory concurrent languages do not include pointers, and semantic models for pointer-manipulating programs do not typically incorporate concurrency. Most models of shared-memory concurrency do not deal explicitly with race-detection. Furthermore, earlier state-based models of concurrency such as *transition traces* [16,13,38] work with *global states*, which lump together the state shared by all processes, and it is not easy to adapt such models for the kind of local reasoning that is required to track ownership. On the one hand, race-freedom should lead to a semantics in which program behaviour has a sequential flavour *modulo* synchronization through shared resources, but on the other hand we need to properly account for concurrent execution.

In this paper we give a denotational semantics, based on sets of *action traces* [12], that solves these problems. The semantics involves a form of parallel composition that detects race conditions: every parallel program whose components may concurrently read and write the same variable or the same heap cell will produce a runtime error. Our semantics models a potential race condition as catastrophic, since we want to prove the absence of races. This semantic model is worthy of attention in its own right, although our main emphasis here is to demonstrate its utility in proving the soundness of O'Hearn's methodology. We also stress that the semantics applies to all concurrent shared-memory programs, both to race-free programs and to racy programs. The crucial feature of the semantics is that it permits a natural, rigorous and simple characterization of race-freedom.

Our treatment of race conditions leads to a semantic model embodying one of the classic principles of concurrent program design, as originally articulated by Dijkstra [20] and reflected in the design of Owicki–Gries logic and O'Hearn's logic:

> ... processes should be loosely connected, by which 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 reflects this idea in a novel manner, through the interplay between action traces, which describe interleaved behaviour of processes, and an enabling relation that implements the "no interference from outside" notion. This interplay is crucial in permitting a formalization of O'Hearn's intuitive concept of "processes that mind their own business". To the best of the author's knowledge ours is the first semantics in which such a formalization is possible.

O'Hearn, following Owicki and Gries, focused on programs containing a single resource declaration whose scope includes a single parallel composition of sequential commands. We reformulate O'Hearn's rules in a more general manner, allowing nested resource declarations and nested parallel compositions. We introduce a formal definition of *resource contexts*, subject to some natural disjointedness requirements which facilitate modular reasoning, and a class of *resource-sensitive partial correctness* formulas that pins down the syntactic constraints on programs and logical formulas necessary for enforcing the intended resource discipline. Using the trace semantics we give a suitably general (and compositional) notion of validity, and we prove that the proof rules are sound. Our soundness proof demonstrates that a verified program has no race conditions.

A key ingredient in our soundness proof, and another illustration of the benefits of our approach, is a parallel decomposition lemma, again with connections back to early intuitions of Dijkstra. We can summarize this result informally as follows. When $c_1 \| c_2$ is a race-free program, every interleaved computation of $c_1 \| c_2$ can be decomposed into "local" computations of the constituent processes c_1 and c_2 which are interference-free except for interactions with protected resources. This clearly reflects the "loosely connected" assumption for processes and shows how this assumption is crucial in permitting syntax-directed proofs for concurrent programs.

We assume that each resource invariant is *precise*, 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. This does not seem to be a major limitation, since all of O'Hearn's examples involve precise invariants, and a methodology based on precision seems very natural [31]. Moreover, this limitation is sufficient to ensure soundness, and it suffices to avoid the Reynolds counterexample that shows unsoundness when resource invariants are allowed to be arbitrary separation logic formulas.

Since our semantics is trace-based it can be used to support reasoning about safety and liveness properties of concurrent programs, in addition to partial correctness and absence of races. We discuss how to adapt the proof system to deal with total correctness and freedom from deadlock.

We conclude with some comments on related work, a discussion of the limitations of our semantics and the logic, and some suggestions for future research. An Appendix contains some technical details behind some of the key results.

2. Syntax

We use a programming language that combines shared-memory parallelism, resource declarations and conditional critical regions with constructs for manipulating heap pointers.

We use the following meta-variables: r ranges over *resource names*, i over *identifiers*, e over *integer expressions*, b over *boolean expressions*, E over *list expressions, and c over commands*. We omit the syntax for integer expressions and boolean expressions, but we assume that the language includes the usual arithmetic and boolean constructs. The abstract grammar for list expressions is:

$$E \quad ::= \quad (e_0, \ldots, e_n) \quad (n \geq 0)$$

We assume that expressions are *pure*: that is, expressions do not contain notations, such as **cons** and $[-]$, whose semantics refers to the heap, and they do not cause side-effects. The value of an expression therefore depends only on the store.

The syntax for *commands* is defined by the following abstract grammar:

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

There are four assignment-like command constructs, which we distinguish syntactically from each other because of their different semantics. Three have an effect on the store: a traditional assignment $i{:=}e$, a *lookup* $i{:=}[e]$, and an *allocation* $i{:=}\textbf{cons}(E)$. To emphasize this fact we will use the term *assignment* collectively for these forms of command. An allocation also affects the heap. An *update* $[e]{:=}e'$ changes only the heap, as does a *disposal* $\textbf{dispose}(e)$. We will use the term *mutation* to refer to an allocation, update or disposal. Thus assignments affect the store, and mutations affect the heap.

The syntax for commands also includes sequential composition, written $c_1; c_2$, conditional commands, while-loops and parallel composition, which is denoted $c_1 \| c_2$.

A *block* of the form **local** $i = e$ **in** c introduces a *local variable* named i, initialized to the value of e, whose scope is the block body c. Similarly a resource block **resource** r **in** c introduces a *local resource* named r, assumed to be initially *available*, with scope c.

A command of form **with** r **when** b **do** c is called a *conditional critical region* for r, or just a "region" for short. A process attempting to enter a region must wait until the resource r is available, whereupon it may acquire the resource and evaluate the test b: if b is **true** the process executes c then releases the resource; on the other hand, if b is **false** the process releases the resource and waits to try again. Program execution is constrained to ensure that resources are mutually exclusive: a resource can only be acquired when it is available, and can only be held by one process at a time; hence at all stages at most one concurrent process is "inside" a region for r. We impose the natural syntactic constraint that the body c of a region for r must not contain another region for the same resource name r. This decision is made for pragmatic reasons: the only commands ruled out by this constraint would cause deadlock anyway, so their omission is no great cause for concern.

3. Static semantics

We assume given the standard structurally inductive definitions of the sets $\texttt{free}(e)$, $\texttt{free}(b)$, $\texttt{free}(E)$ of identifiers which occur free in an expression. In addition we will define $\texttt{reads}(c)$, the set of identifiers having a free read occurrence in c; $\texttt{writes}(c)$, the set of identifiers having a free write occurrence in c; and $\texttt{res}(c)$, the set of resource names occurring free in c. We only provide the details for a few key cases.

Definition 1. Let $\texttt{reads}(c)$ be the set of identifiers with a free read occurrence in c, given by structural induction. In particular,

$$\begin{array}{l} \texttt{reads}(i{:=}e) = \texttt{free}(e) \\ \texttt{reads}(i{:=}[e]) = \texttt{free}(e) \\ \texttt{reads}(i{:=}\textbf{cons}\, E) = \texttt{free}(E) \\ \texttt{reads}([e]{:=}e') = \texttt{free}(e) \cup \texttt{free}(e') \\ \texttt{reads}(\textbf{dispose}(e)) = \texttt{free}(e) \\ \texttt{reads}(c_1 \| c_2) = \texttt{reads}(c_1) \cup \texttt{reads}(c_2) \\ \texttt{reads}(\textbf{local}\ i = e\ \textbf{in}\ c) = \texttt{free}(e) \cup (\texttt{reads}(c) - \{i\}) \end{array}$$

Definition 2. Let $\texttt{writes}(c)$ be the set of identifiers with a free write occurrence in c, defined by structural induction. In particular:

$$\begin{array}{l}
\texttt{writes}(i{:=}e) = \{i\} \\
\texttt{writes}(i{:=}[e]) = \{i\} \\
\texttt{writes}(i{:=}\mathbf{cons}\, E) = \{i\} \\
\texttt{writes}([e]{:=}e') = \{\} \\
\texttt{writes}(\mathbf{dispose}(e)) = \{\} \\
\texttt{writes}(c_1 \| c_2) = \texttt{writes}(c_1) \cup \texttt{writes}(c_2) \\
\texttt{writes}(\mathbf{local}\ i = e\ \mathbf{in}\ c) = \texttt{writes}(c) - \{i\}
\end{array}$$

For all commands c we then define $\texttt{free}(c) = \texttt{reads}(c) \cup \texttt{writes}(c)$. Note that $\texttt{free}(\mathbf{local}\ i = e\ \mathbf{in}\ c) = \texttt{free}(e) \cup (\texttt{free}(c) - \{i\})$.

Definition 3. Let $\texttt{res}(c)$ be the set of resource names occurring free in c, defined by structural induction. In particular,

$$\begin{array}{l}
\texttt{res}(\mathbf{with}\ r\ \mathbf{when}\ b\ \mathbf{do}\ c) = \texttt{res}(c) \cup \{r\} \\
\texttt{res}(\mathbf{resource}\ r\ \mathbf{in}\ c) = \texttt{res}(c) - \{r\} \\
\texttt{res}(c_1 \| c_2) = \texttt{res}(c_1) \cup \texttt{res}(c_2)
\end{array}$$

4. Dynamic semantics

We give a trace-theoretical 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.[1] 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. We will exploit this feature later, when we show how to 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 in which it is easier to reason about ownership. Another advantage of action traces over the *transition traces* often used to model shared-memory parallel languages is succinctness: an action typically acts the same way on all states, and we can express this implicitly, without enumerating all pairs of states related by the action.

4.1. States and values

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 and V_{addr} be the set of addresses.[2] A *truth value* is either **true** or **false**. Let V_{bool} be the set of truth values. We use t as a meta-variable ranging over truth values.

A *state* σ comprises a *store* s, a *heap* h, and a finite set A of resource names. The *store* maps a finite set of identifiers to values; we let $\mathbf{S}$ be the set of stores, and we write $\texttt{dom}(s) = \{i \mid \exists v.\ (i, v) \in s\}$ for the set of identifiers for which s has a value. The *heap* maps a finite set of addresses to values; we write $\texttt{dom}(h) = \{l \mid \exists v.\ (l, v) \in h\}$ for the set of locations for which h has a value. We will 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 with specific contents. We also use the notation $[s \mid i : v]$ for the store which agrees with s on all identifiers except i, which it maps to v; and the similar notation $[h \mid l : v']$ denotes an updated heap. We also use the notation $h\backslash l$ for the heap obtained from h by deleting l from its domain; clearly $\texttt{dom}(h\backslash l) = \texttt{dom}(h) - \{l\}$.

Since we assume that resources are initially available, an "initial" state will always have the form $(s, h, \{\})$; we will use the abbreviation (s, h) in such a case.

[1] Although we are mainly concerned with partial correctness properties of programs, which depend only on the finite traces of a program, we also want to be able to use our semantics to establish race freedom properties, so we also need to include infinite traces. Consequently it makes sense to build fairness directly into our model.

[2] Actually we treat addresses as integers, so that our semantic model can incorporate address arithmetic, but we maintain the conceptual distinction between integers as values and integers which happen to be addresses in current use.

4.2. Actions

The atomic units in which a program's execution is measured will be called *actions*, and we assume that actions form a simple algebra under concatenation. Actions include reads and writes to individual identifiers, lookups and updates to individual heap addresses, allocations and disposals of heap addresses, and actions involving the acquisition and release of resources. We use λ as a meta-variable ranging over the set of actions.

Definition 4. An action has one of the following forms:

- δ, an idle step
- $i{=}v$, a read of identifier i
- $i{:=}v$, a write to i
- $[l]{=}v$, a lookup of address l
- $[l]{:=}v$, an update to address l
- $alloc(l, L)$, an allocation, where l is an address and L is a finite list of values
- $disp(l)$, a disposal of address l
- $acq(r)$, where r is a resource name
- $rel(r)$, where r is a resource name
- $try(r)$, where r is a resource name
- abort, an error stop.

We will refer to reads and writes as *store actions*, to lookups, updates, allocations and disposals as *heap actions*, and to try, acquire and release actions as *resource actions*.

Each action has a natural intuitive interpretation. For example, an allocate action $alloc(l, [v_0, \dots, v_n])$ allocates a fresh sequence of addresses $l, \dots, l{+}n$ and initializes their contents to $v_0, \dots, v_n$, respectively. A try action represents an unsuccessful attempt to acquire a resource, and an acquire action represents the successful case.

4.3. Effects and enabling

Each action λ is characterized by its its *effect*, which can be defined as a partial function $\overset{\lambda}{\Longrightarrow}$ from states to states (Fig. 1); the domain of this partial function is the set of states from which the action can be executed. To account for runtime errors we use a special "improper" state **abort**.

It is convenient to introduce a more succinct notation that recognizes the facts that: store actions only depend on the store; heap actions only depend on the heap; and resource actions only involve the resource set. Thus when λ is a store action we will treat $\overset{\lambda}{\Longrightarrow}$ as a partial function from stores to stores; when λ is a heap action we may use $\overset{\lambda}{\Longrightarrow}$ as a partial function from heaps to heaps; and when λ is a resource action we may use $\overset{\lambda}{\Longrightarrow}$ as a partial function from resource sets to resource sets.

We extend the definitions of `writes`, `reads`, and `free` to actions:

$$\begin{array}{ll}
\texttt{writes}(i{:=}v) = \{i\} & \texttt{reads}(i{=}v) = \{i\} \\
\texttt{writes}([l]{:=}v) = \{l\} & \texttt{reads}([l]{=}v) = \{l\} \\
\texttt{writes}(disp(l)) = \{l\} & \texttt{reads}(disp(l)) = \{l\} \\
\texttt{writes}(\lambda) = \{\}\quad \text{otherwise} & \texttt{reads}(\lambda) = \{\}\quad \text{otherwise}
\end{array}$$

For all actions λ, we let $\texttt{free}(\lambda) = \texttt{reads}(\lambda) \cup \texttt{writes}(\lambda)$.

For each action λ, $\texttt{reads}(\lambda)$ is the set of identifiers or addresses needed to enable the action, and $\texttt{writes}(\lambda)$ is the set of identifiers or addresses whose current value is changed by the action. Note that allocation actions are given a special treatment: we do not include addresses $l, \dots, l + n$ in the write-set of $alloc(l, [v_0, \dots, v_n])$, because these addresses will be assumed to be fresh (not in current use) whenever the action occurs. We distinguish between this kind of effect (generating a fresh piece of heap) and the effect of a disposal or an update, which modifies or deletes part of the current heap.

- $(s, h, A) \xRightarrow{\delta} (s, h, A)$ always
- $(s, h, A) \xRightarrow{i=v} (s, h, A)$ iff $(i, v) \in s$
- $(s, h, A) \xRightarrow{i=v}$ **abort** iff $i \notin \mathrm{dom}(s)$
- $(s, h, A) \xRightarrow{i:=v} ([s \mid i : v], h, A)$ iff $i \in \mathrm{dom}(s)$
- $(s, h, A) \xRightarrow{i:=v}$ **abort** iff $i \notin \mathrm{dom}(s)$
- $(s, h, A) \xRightarrow{[l]=v} (s, h, A)$ iff $(l, v) \in h$
- $(s, h, A) \xRightarrow{[l]=v}$ **abort** iff $l \notin \mathtt{dom}(h)$
- $(s, h, A) \xRightarrow{[l]:=v} (s, [h \mid l : v], A)$ iff $l \in \mathtt{dom}(h)$
- $(s, h, A) \xRightarrow{[l]:=v}$ **abort** iff $l \notin \mathtt{dom}(h)$
- $(s, h, A) \xRightarrow{alloc(l,[v_0,\ldots,v_n])} (s, [h \mid l : v_0, \ldots, l+n : v_n], A)$
  iff $dom(h) \cap \{l, l+1, \ldots, l+n\} = \{\}$
- $(s, h, A) \xRightarrow{disp(l)} (s, h \backslash l, A)$ iff $l \in \mathtt{dom}(h)$.
- $(s, h, A) \xRightarrow{disp(l)}$ **abort** iff $l \notin \mathtt{dom}(h)$
- $(s, h, A) \xRightarrow{try(r)} (s, h, A)$ iff $r \in A$
- $(s, h, A) \xRightarrow{acq(r)} (s, h, A \cup \{r\})$ iff $r \notin A$
- $(s, h, A) \xRightarrow{rel(r)} (s, h, A - \{r\})$ iff $r \in A$
- $(s, h, A) \xRightarrow{abort}$ **abort** always
- **abort** $\xRightarrow{\lambda}$ **abort** always

Fig. 1. Enabling relations $\xRightarrow{\lambda}$.

For a finite trace α we define $\xRightarrow{\alpha}$ in the obvious way, so that $\sigma \xRightarrow{\lambda_0\ldots\lambda_n} \sigma'$ if there is a sequence of states $\sigma_0, \ldots, \sigma_{n-1}$ such that

$$\sigma \xRightarrow{\lambda_0} \sigma_0 \xRightarrow{\lambda_1} \cdots \xRightarrow{\lambda_{n-1}} \sigma_{n-1} \xRightarrow{\lambda_n} \sigma'.$$

For an infinite trace α we write $(s, h, A) \xRightarrow{\alpha}$ **abort** when there is a finite prefix β of α such that $(s, h, A) \xRightarrow{\beta}$ **abort**. We write $\sigma \xRightarrow{\alpha} \cdot$ when α is enabled from σ. By definition, every trace participating in this kind of enabling is *sequential*. This enabling notion can thus be used to describe the effect of executing a program in isolation, without interference.

4.4. Traces

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. Using the usual pun, we do not distinguish notationally between an action λ and the corresponding trace λ consisting of a single action. (But note that δ is not the same as the empty sequence!)

We write $\alpha_1\alpha_2$ for the trace obtained by concatenating α_1 and α_2; when α_1 is infinite this is just α_1. We assume that *abort* behaves like a left-zero for concatenation, so that $\alpha\, abort\, \beta = \alpha\, abort$, for all traces α and β. We also assume that δ is a unit for concatenation, so that $\alpha\delta\beta = \alpha\beta$ for all traces α and β. Thus, in particular, for all $n > 0$, $\delta^n = \delta$. Note, however, that δ^ω is not (and should not be) equal to δ. Concatenation is associative: for all α_1, α_2 and α_3, $\alpha_1(\alpha_2\alpha_3) = (\alpha_1\alpha_2)\alpha_3$.

4.4.1. Sequential traces

We write $\alpha\lceil i$ for the subsequence of α consisting of reads and writes to identifier i, $\alpha\lceil l$ for the subsequence involving heap cell l, and $\alpha\lceil r$ for the subsequence involving resource r. We say that α is sequential for i from s if $s \xRightarrow{\alpha\lceil i} \cdot$, sequential for l from h if $h \xRightarrow{\alpha\lceil l} \cdot$, and sequential for r from A if $A \xRightarrow{\alpha\lceil r} \cdot$.

A trace which is sequential for i from s describes an execution in which the initial value of i is specified by s and the value of i is not changed by the environment. Such traces will be used to determine the trace set of **local** $i = e$ **in** c, since the scope of the local binding for i includes c but not the environment. For a trace set T we let $T_{[i:v]}$ be the set of traces in T which are sequential for i from $[i : v]$. We define $\alpha\backslash i$ to be the trace obtained from α by replacing every action involving i by δ.

Similarly, a trace which is sequential for r from the empty set describes an execution in which r is initially available and the environment never affects r. This kind of trace will be used to formulate the trace set of **resource** r **in** c, since the scope of the local binding for r only includes c. Since resources are assumed to be initially available we will drop the qualification and call such a trace *sequential for r*. Given a trace set T, let T_r be the subset consisting of the traces in T which are sequential for r. Note that $(T_r)_{r'} = (T_{r'})_r$, so we may write $T_{r,r'}$ for the subset of traces which are sequential both for r and for r', without any ambiguity. We let $\alpha\backslash r$ be the trace obtained from α by replacing each resource action on r by δ.

We say that α is sequential from (s, h, A) if α is sequential for all identifiers from s, for all locations from h, and for all resources from A. An infinite trace is sequential from (s, h, A) if each of its finite prefixes is sequential from (s, h, A).

Sequential traces describe the behaviour of a command when executed in isolation from some given initial store and heap, endowed with a given initial collection of resources. Thus sequential traces provide enough information to determine partial (and total) correctness properties of commands. It is well known that one cannot generally determine the sequential traces of a parallel program solely from the sequential traces of its components. This is a symptom of the usual problem with concurrent programs: in order to obtain a compositional semantics we need to include both sequential and non-sequential traces in the trace set of a command.

4.4.2. *Sequential composition and iteration*

For trace sets T_1 and T_2 we let T_1T_2 be the set of all concatenations $\alpha_1\alpha_2$ with $\alpha_1 \in T_1$ and $\alpha_2 \in T_2$. We also let $\lambda T = \{\lambda\alpha \mid \alpha \in T\}$ and $T\lambda = \{\alpha\lambda \mid \alpha \in T\}$.

For each $n \geq 0$ we define $T^0 = \{\delta\}$, and $T^{n+1} = TT^n = T^nT$. We let $T^* = \bigcup_{n=0}^{\infty} T^n$. We let T^ω be the set of all infinite concatenations of the form $\alpha_1 \ldots \alpha_n \ldots$, where for each $n \geq 1$ we have $\alpha_n \in T$. We let $T^\infty = T^* \cup T^\omega$. Note that $\{\}^*$ is the set $\{\delta\}$ and $\{\}^\omega = \{\}$.

4.4.3. *Parallel composition*

The resource actions permissible for a command will depend on the resources currently held by the command, but also on the resources being used by its environment. These sets of resources will always be disjoint. Accordingly we define the *resource enabling* relation $(A_1, A_2) \xrightarrow{\lambda} (A_1, A_2)$ on disjoint pairs of resource sets, to specify what happens if a program holding resources A_1, in an environment that holds A_2, attempts to perform an action λ. This action may be forbidden because it would acquire a resource already in use by the program or its environment, or because the action would release a resource which the program does not currently hold. If allowed, we specify the action's effect on the resources held by the program:

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

This resource enabling relation generalizes in the obvious way to describe what happens to the resources when the program tries to perform a finite or infinite sequence α of actions. We write $(A_1, A_2) \xrightarrow{\alpha} \cdot$ to indicate that the trace is allowed.

We want to detect *race conditions* caused by an attempt to write to an identifier or address being used concurrently: we will treat such a possibility as a catastrophe. We will write $\lambda_1 \sharp \lambda_2$, pronounced λ_1 *interferes with* λ_2, to indicate when this happens:

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

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

We define, for each pair (A_1, A_2) of disjoint sets of resources, and each pair (α_1, α_2) of finite traces, the set $\alpha_1\,{}_{A_1}\|_{A_2}\,\alpha_2$ of all *mutex fairmerges* of α_1 (with initial resources A_1) and α_2 (with initial resources A_2). The definition is inductive in the lengths of α_1 and α_2, and we include the empty sequence, denoted ϵ, to allow a simpler formulation:

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

Note that the definition only produces interleavings which respect the mutex constraints on resource acquisition.[3]

For example, the set

$$(acq(r)\ x{:=}0\ rel(r))_{\{\}}\|_{\{\}}(acq(r)\ x{=}1\ x{:=}2\ rel(r))$$

contains only

$$acq(r)\ x{:=}0\ rel(r)\ acq(r)\ x{=}1\ x{:=}2\ rel(r)$$

and

$$acq(r)\ x{=}1\ x{:=}2\ rel(r)\ acq(r)\ x{:=}0\ rel(r).$$

We can also give a *coinductive* definition of the mutex fairmerges of two infinite traces, or a finite trace with an infinite trace, starting from a given disjoint pair of resource sets. We need mostly to work with finite traces, given our focus on partial correctness and race-freedom, so we omit the details, which are standard [16].

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 $T_1\|T_2 =_{\text{def}} \bigcup\{\alpha_1\|\alpha_2 \mid \alpha_1 \in T_1\ \&\ \alpha_2 \in T_2\}$. As usual, for all trace sets T_1, T_2 and T_3, $T_1\|(T_2\|T_3) = (T_1\|T_2)\|T_3$, and $T_1\|T_2 = T_2\|T_1$. Moreover, for all trace sets T we have $T\|\{\delta\} = T$.

4.5. Trace semantics of expressions

We do not assume that expression evaluation is atomic, because we want to design a semantics for commands that permits analysis of race conditions, and we do not want to make unrealistic assumptions about granularity.

An expression will denote a set of *evaluation traces* paired with values. Since expression values depend only on the store, the only non-trivial actions participating in such traces will be reads. We will use ρ as a meta-variable ranging over evaluation traces. To allow for the possibility of interference during expression evaluation we will include both non-sequential and sequential evaluation traces. Again the sequential traces describe what happens if an expression is evaluated without interference.

For an integer expression e,

$$[\![e]\!] \subseteq \mathbf{Tr} \times V_{int}$$

is defined to be the set of all (ρ, v) such that e evaluates to v along ρ. For a boolean expression b we define

$$[\![b]\!] \subseteq \mathbf{Tr} \times V_{bool}$$

to be the set of all (ρ, t) such that b evaluates to t along ρ. For a list expression E, we let

$$[\![E]\!] \subseteq \mathbf{Tr} \times V_{int}^*$$

be the set of $(\rho, [v_0, \ldots, v_n])$ such that E evaluates to the value list $[v_0, \ldots, v_n]$ along ρ.

[3] This definition of $(\lambda_1\alpha_1)\,{}_{A_1}\|_{A_2}\,(\lambda_2\alpha_2)$ differs slightly from the one originally proposed and which appears in the earlier versions of this paper [11]. The original definition turns out to lack associativity. Apart from that, all of the results proved in the original paper are valid for both definitions of interleaving. In particular, the new version leads to the same notion of race freedom.

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

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

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

Since expressions are pure, the only non-trivial actions occurring in an expression trace ρ will be reads. Note that $s \overset{\rho}{\Longrightarrow} s$ holds if and only if the reads in ρ are consistent with the store s.

We assume the usual properties. For instance, the value of an expression depends only on the values of its free identifiers, so that in particular whenever $(\rho, v) \in [\![e]\!]$ and stores s_1 and s_2 agree on the values of the identifiers occurring free in e, $s_1 \overset{\rho}{\Longrightarrow} s_1$ holds if and only if $s_2 \overset{\rho}{\Longrightarrow} s_2$ holds. There are analogous properties for boolean expressions and list expressions.

We let $[\![b]\!]_{\mathbf{true}} \subseteq \mathbf{Tr}$ be the set of all ρ such that $(\rho, \mathbf{true}) \in [\![b]\!]$, and likewise $[\![b]\!]_{\mathbf{false}} = \{\rho \mid (\rho, \mathbf{false}) \in [\![b]\!]\}$.

4.6. *Trace semantics of commands*

A command c denotes a set $[\![c]\!] \subseteq \mathbf{Tr}$ of action traces. Again we include both sequential and non-sequential traces.

Definition 5. For all commands c we define the trace set $[\![c]\!] \subseteq \mathbf{Tr}$ inductively by:

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

We hope that the purpose of each semantic clause is evident, and that the reader will readily appreciate the role played in these clauses by the trace constructions discussed earlier. For instance, execution of an assignment command begins with evaluation of the right-hand-side expression and ends with the assignment to the target identifier. (So we do not assume that assignments are atomic.) Similarly, an update command evaluates from left to right, then performs the update action on the relevant heap address. Sequential composition and conditional commands are interpreted using concatenation, and parallel composition is modelled using mutex fairmerge. While-loops correspond, as usual, to iteration, and we include traces representing both terminating and non-terminating executions. A block **local** $i = e$ **in** c begins by evaluating e to obtain a value v, then executes c with i bound locally to v. Similarly a resource block **resource** r **in** c executes c with r bound to a local resource assumed to be initially available.

The iterative structure of the traces of a conditional critical region reflect its characteristic synchronization attributes: 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 that the clause for a critical region allows for the possibility that the body may loop forever or encounter a runtime error, in which case the resource release action will not occur.

Since $[\![\mathbf{true}]\!]_{\mathbf{false}} = \{\}$ and $[\![\mathbf{true}]\!]_{\mathbf{true}} = \{\delta\}$, we can derive a simpler formula for the trace set of **with** r **when true do** c: we will use the syntactic abbreviation **with** r **do** c for this special case, and we have

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

Note that the semantics and the enabling relation allow us to determine, for each command c and state σ, what possible executions of c are enabled from σ, and whether or not execution may encounter a runtime error, such as a dangling pointer, an attempt to read or assign to an uninitialized identifier or a race.

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. Concurrent assignments to the same identifier cause a race. For example $[\![x{:=}x+1 \| x{:=}x+1]\!]$ contains interleavings of traces $x{=}v\ x{:=}v+1$ and $x{=}v'\ x{:=}v'+1$, for all v and v', and also traces that reflect the inherent race condition, such as $x{=}v\ \mathit{abort}$.
3. $[\![\mathbf{with}\ r\ \mathbf{do}\ x{:=}x+1]\!] = \mathit{try}(r)^{*}\,\mathit{acq}(r)\,[\![x{:=}x+1]\!]\,\mathit{rel}(r)\ \cup\ \{\mathit{try}(r)^{\omega}\}$
   This program needs to acquire r before incrementing x, and will wait forever if the resource never becomes available.
4. The trace set $[\![\mathbf{with}\ r\ \mathbf{do}\ x{:=}x+1 \| \mathbf{with}\ r\ \mathbf{do}\ x{:=}x+1]\!]$ contains all traces of the forms:
   - $\mathit{acq}(r)\,\alpha\,\mathit{rel}(r)\,\mathit{acq}(r)\,\beta\,\mathit{rel}(r)$
   - $\mathit{acq}(r)\,\alpha\,\mathit{rel}(r)\,\mathit{try}(r)^{\omega}$
   - $\mathit{try}(r)^{\omega}$

   where $\alpha, \beta \in [\![x{:=}x+1]\!]$. Only the first kind are sequential for r. The trace set also includes traces obtainable from the above forms by inserting (finitely many) additional $\mathit{try}(r)$ steps.
5. It follows from the previous example, focusing on the traces which are sequential for r, that:
   $$\begin{aligned}&[\![\mathbf{resource}\ r\ \mathbf{in}\ (\mathbf{with}\ r\ \mathbf{do}\ x{:=}x+1 \| \mathbf{with}\ r\ \mathbf{do}\ x{:=}x+1)]\!]\\ &\quad= \{\alpha\beta \mid \alpha, \beta \in [\![x{:=}x+1]\!]\}\\ &\quad= [\![x{:=}x+1; x{:=}x+1]\!].\end{aligned}$$
   The parallel assignments to x here are protected by r, and the overall effect is the same as that of two consecutive increments.
6. The command $x{:=}\mathbf{cons}(1) \| y{:=}\mathbf{cons}(2)$ has the trace set
   $$\{\mathit{alloc}(l,[1])\ x{:=}l \mid l \in V_{addr}\} \| \{\mathit{alloc}(l',[2])\ y{:=}l' \mid l' \in V_{addr}\}.$$
   This set includes traces of the form:
   $$\mathit{alloc}(l,[1])\ x{:=}l\ \mathit{alloc}(l,[2])\ y{:=}l,$$
   and other interleavings of $\mathit{alloc}(l,[1])\ x{:=}l$ with $\mathit{alloc}(l,[2])\ y{:=}l$, none of which are sequential for l. The set also includes traces obtained by interleaving $\mathit{alloc}(l,[1])\ x{:=}l$ and $\mathit{alloc}(l',[2])\ y{:=}l'$, where $l \neq l'$; all of these are sequential for l and l'.
7. The command $x{:=}\mathbf{cons}(1) \| \mathbf{dispose}(42)$ has the trace set
   $$\{\mathit{alloc}(l,[1])\ x{:=}l \mid l \in V_{addr}\} \| \{\mathit{disp}(42)\}.$$
   The possible interleavings have one of the forms
   - $\mathit{disp}(42)\ \mathit{alloc}(l,[1])\ x{:=}l$
   - $\mathit{alloc}(l,[1])\ \mathit{disp}(42)\ x{:=}l$
   - $\mathit{alloc}(l,[1])\ x{:=}l\ \mathit{disp}(42)$,

   where $l \in V_{addr}$.

8. The command $\mathbf{dispose}(x) \| \mathbf{dispose}(y)$ has the trace set

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

   including traces of the form $x{=}v\ y{=}v\ abort$ because of the race-detecting clause in the definition of fairmerge. A trace of this form indicates that if the program is executed from a state in which x and y are aliases for the same heap cell v a race condition will occur.
9. To illustrate how the semantic model deals with deadlock, consider

   $$c_1 =_{\mathrm{def}}\ \mathbf{with}\ r_1\ \mathbf{do}\ \mathbf{with}\ r_2\ \mathbf{do}\ x{:=}1$$
   $$c_2 =_{\mathrm{def}}\ \mathbf{with}\ r_2\ \mathbf{do}\ \mathbf{with}\ r_1\ \mathbf{do}\ y{:=}1$$

   We have:

   $$[\![c_1]\!] = try(r_1)^\infty\, acq(r_1)\, try(r_2)^\infty acq(r_2)\, x{:=}1\, rel(r_2)\, rel(r_1)$$
   $$[\![c_2]\!] = try(r_2)^\infty\, acq(r_2)\, try(r_1)^\infty acq(r_1)\, y{:=}1\, rel(r_1)\, rel(r_2).$$

   The trace set of $c_1 \| c_2$ thus includes traces such as

   $$acq(r_1)\, acq(r_2)\, x{:=}1\, rel(r_2)\, rel(r_1)\, acq(r_2)\, acq(r_1)\, y{:=}1\, rel(r_1)\, rel(r_2)$$
   $$acq(r_2)\, acq(r_1)\, y{:=}1\, rel(r_1)\, rel(r_2)\, acq(r_1)\, acq(r_2)\, x{:=}1\, rel(r_2)\, rel(r_1)$$

   which correspond to deadlock-free computations, but also includes traces belonging to the subset

   $$(acq(r_1) \| acq(r_2))\ (try(r_2)^\omega \| try(r_1)^\omega)$$

   which represent the deadlock which occurs if c_1 acquires r_1 and c_2 acquires r_2, whereupon each process is trying to acquire a resource held by the other. Using the above analysis it is easy to see that:

   $$[\![\mathbf{resource}\ r_1, r_2\ \mathbf{in}\ (c_1 \| c_2)]\!] = \{x{:=}1\ y{:=}1,\ y{:=}1\ x{:=}1,\ \delta^\omega\}$$

   and this trace set again records the potential for deadlock.
10. Consider the program c given by:

    $$c =_{\mathrm{def}}\ \mathbf{with}\ r\ \mathbf{do}\ \mathbf{while}\ \mathbf{true}\ \mathbf{do}\ \mathbf{skip}.$$

    We have:

    $$[\![c]\!] = try(r)^*\, acq(r)\, \delta^\omega\ \cup\ try(r)^\omega$$

    so that:

    $$[\![c \| c]\!] \ = \ try(r)^*\, acq(r)\, try(r)^\omega\ \cup\ try(r)^\omega$$

    It follows that:

    $$[\![\mathbf{resource}\ r\ \mathbf{in}\ (c \| c)]\!] = \{\delta^\omega\}$$

    This reflects the expected behaviour of this command: one of the parallel components will acquire the resource and loop forever, while the other waits forever.
11. Let $\mathrm{PUT}(x)$ and $\mathrm{GET}(y)$ be the following code fragments:

    $$\mathrm{PUT}(x) : \mathbf{with}\ buf\ \mathbf{when}\ full = 0\ \mathbf{do}\ (z{:=}x;\ full{:=}1)$$
    $$\mathrm{GET}(y) : \mathbf{with}\ buf\ \mathbf{when}\ full = 1\ \mathbf{do}\ (y{:=}z;\ full{:=}0)$$

    We have:

    $$\begin{array}{rrcl} & [\![\mathrm{PUT}(x)]\!] & = & wait_{\neg full}{}^*\, put\ \cup\ wait_{\neg full}{}^\omega \\ \text{where} & wait_{\neg full} & = & \{acq(buf)\, full{=}1\, rel(buf),\ try(buf)\} \\ & put & = & \{acq(buf)\, put(v)\, rel(buf) \mid v \in V_{int}\} \\ & put(v) & = & full{=}0\, x{=}v\, z{:=}v\, full{:=}1 \end{array}$$

    and

    $$\begin{array}{rrcl} & [\![\mathrm{GET}(y)]\!] & = & wait_{full}{}^*\, get\ \cup\ wait_{full}{}^\omega \\ \text{where} & wait_{full} & = & \{acq(buf)\, full{=}0\, rel(buf),\ try(buf)\} \\ & get & = & \{acq(buf)\, get(v)\, rel(buf) \mid v \in V_{int}\} \\ & get(v) & = & full{=}1\, z{=}v\, y{:=}v\, full{:=}0 \end{array}$$

    The trace set of $\mathrm{PUT}(x) \| (\mathrm{GET}(y); \mathbf{dispose}(y))$ includes traces of the following forms, where v, v', v'' range over V_{int}:

- $acq(buf)\, put(v)\, rel(buf)\, acq(buf)\, get(v')\, rel(buf)\ y{=}v''\ disp(v'')$
- $acq(buf)\, get(v')\, rel(buf)\ ((acq(buf)\, put(v)\, rel(buf)) \| (y{=}v''\ disp(v'')))$

The sequential traces of these forms are:

- $acq(buf)\, put(v)\, rel(buf)\, acq(buf)\, get(v)\, rel(buf)\ y{=}v\ disp(v)$
- $acq(buf)\, get(v')\, rel(buf)\ ((acq(buf)\, put(v)\, rel(buf)) \| (y{=}v'\ disp(v')))$

None of these traces leads to a race.

For $(\mathrm{PUT}(x);\ \mathbf{dispose}(x)) \| \mathrm{GET}(y)$ the trace set includes traces of the forms:

- $acq(buf)\, put(v)\, rel(buf)\ ((x{=}v''\ disp(v'')) \| (acq(buf)\, get(v')\, rel(buf)))$
- $acq(buf)\, get(v')\, rel(buf)\, acq(buf)\, put(v)\, rel(buf)\ x{=}v''\ disp(v'')$

The sequential traces of these forms are:

- $acq(buf)\, put(v)\, rel(buf)\ ((x{=}v\ disp(v)) \| (acq(buf)\, get(v)\, rel(buf)))$
- $acq(buf)\, get(v')\, rel(buf)\, acq(buf)\, put(v)\, rel(buf)\ x{=}v\ disp(v)$

Again there are no races in these traces.

On the other hand, the trace set of $(\mathrm{PUT}(x);\ \mathbf{dispose}(x)) \| (\mathrm{GET}(y);\ \mathbf{dispose}(y))$ includes, for each v, traces of the form:

$$acq(buf)\, put(v)\, rel(buf)\, acq(buf)\, get(v)\, rel(buf)\ (x{=}v\ disp(v)) \| (y{=}v\ disp(v))$$

and hence includes the sequential trace

$$acq(buf)\, put(v)\, rel(buf)\, acq(buf)\, get(v)\, rel(buf)\ x{=}v\ y{=}v\ abort.$$

This indicates the possibility of a race condition, caused in this case by concurrent attempts to dispose the same heap cell. This trace is enabled from any state (s, h) such that $s(full) = 0$, $s(x) = v$, and $y, z \in \mathtt{dom}(s)$.

5. Semantic equivalence

Definition 6. Commands c and c' are said to be semantically equivalent if $[\![c]\!] = [\![c']\!]$.

Since the trace semantics is compositional, semantic equivalence is clearly a congruence: if $[\![c]\!] = [\![c']\!]$ then for all program contexts $C[-]$ we also have $[\![C[c]]\!] = [\![C[c']]\!]$.

We can establish a number of standard laws of semantic equivalences. In particular, sequential composition and parallel composition are associative: for all commands c_1, c_2 and c_3,

$$[\![c_1;\ (c_2;\ c_3)]\!] = [\![(c_1;\ c_2);\ c_3]\!]$$
$$[\![c_1 \| (c_2 \| c_3)]\!] = [\![(c_1 \| c_2) \| c_3]\!]$$

Parallel composition is also commutative: for all c_1 and c_2, $[\![c_1 \| c_2]\!] = [\![c_1 \| c_2]\!]$. Moreover, for all c we have

$$[\![\mathbf{skip};\ c]\!] = [\![c;\ \mathbf{skip}]\!] = [\![c]\!]$$
$$[\![\mathbf{skip} \| c]\!] = [\![c \| \mathbf{skip}]\!] = [\![c]\!]$$

We also obtain the usual loop unrolling law:

$$[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] = [\![\mathbf{if}\ b\ \mathbf{then}\ c;\ \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{else\ skip}]\!].$$

Let $[i'/i]c$ be the command obtained by replacing each free occurrence of i in c by i', changing bound variable names if necessary to avoid capture. If $i' \notin \mathtt{free}(c)$, then

$$[\![\mathbf{local}\ i = e\ \mathbf{in}\ c]\!] = [\![\mathbf{local}\ i' = e\ \mathbf{in}\ [i'/i]c]\!].$$

Simlarly, let $[r'/r]c$ be obtained by replacing every free occurrence of the resource name r in c by r', changing bound resource names if necessary to avoid capture. We use a similar notation $[r'/r]\alpha$ for the trace obtained by replacing each resource action on r in α by the corresponding action on r'. If r' is a "fresh" resource name, so that $r' \notin \mathtt{res}(c)$, the commands $\mathbf{resource}\ r\ \mathbf{in}\ c$ and $\mathbf{resource}\ r'\ \mathbf{in}\ [r'/r]c$ are semantically equivalent, since:

$$\begin{aligned}
[\![\mathbf{resource}\ r'\ \mathbf{in}\ [r'/r]c]\!] &= \{\beta\backslash r' \mid \beta \in [\![[r'/r]c]\!]_{r'}\} \\
&= \{([r'/r]\alpha)\backslash r' \mid \alpha \in [\![c]\!]_r\} \\
&= \{\alpha\backslash r \mid \alpha \in [\![c]\!]_r\} \\
&= [\![\mathbf{resource}\ r\ \mathbf{in}\ c]\!].
\end{aligned}$$

Note also that if r_1 and r_2 are distinct resource names,

$$[\![\textbf{resource } r_1 \textbf{ in resource } r_2 \textbf{ in } c]\!] = \{(\alpha\backslash r_1)\backslash r_2 \mid \alpha \in ([\![c]\!]_{r_1})_{r_2}\}$$
$$[\![\textbf{resource } r_2 \textbf{ in resource } r_1 \textbf{ in } c]\!] = \{(\alpha\backslash r_2)\backslash r_1 \mid \alpha \in ([\![c]\!]_{r_2})_{r_1}\}$$

and since $(\alpha\backslash r_1)\backslash r_2 = (\alpha\backslash r_2)\backslash r_1$ holds for all traces α, and $(T_{r_1})_{r_2} = (T_{r_2})_{r_1}$ holds for all trace sets T, the two commands are semantically equivalent. Accordingly, we may use the convenient syntactic abbreviation

$$\textbf{resource } r_1, r_2 \textbf{ in } c$$

without risk of ambiguity, and we may write:

$$[\![\textbf{resource } r_1, r_2 \textbf{ in } c]\!] = \{\alpha\backslash\{r_1, r_2\} \mid \alpha \in [\![c]\!]_{r_1,r_2}\}.$$

6. Race-free programs

We now formalize, using the trace semantics, a notion of race-freedom for commands. We choose this notion to be strong enough to imply that whenever the program is executed in isolation, without interference, there will be no races, no attempt to access an identifier outside the domain of the store and no attempt to access an address outside of the heap.

Definition 7 (*Race-Free Command*). A command c is race-free from state (s, h) if for all traces $\alpha \in [\![c]\!]$,

$$\neg(s, h) \overset{\alpha}{\Longrightarrow} \textbf{abort}.$$

Examples

1. $x{:=}1 \| y{:=}2$ is race-free from (s, h) if and only if $x, y \in \mathtt{dom}(s)$.
2. $[x]{:=}1 \| [y]{:=}2$ is race-free from (s, h) if and only if $x, y \in \mathtt{dom}(s)$, $s(x) \neq s(y)$, and $s(x), s(y) \in \mathtt{dom}(h)$.
3. $[10]{:=}1 \| [10]{:=}2$ is not race-free from any state.
4. $x{:=}[10] \| y{:=}[10]$ is race-free from all states (s, h) in which $x, y \in \mathtt{dom}(s)$ and $10 \in \mathtt{dom}(h)$. This is because we do not view a concurrent pair of reads as a race condition.
5. $x{:=}1 \| x{:=}1$ is not race-free from any state; similarly $[x]{:=}1 \| x{:=}1$ and $y{:=}[x] \| x{:=}1$ are not race-free.
6. $\textbf{dispose}(x) \| \textbf{dispose}(y)$ is race-free from (s, h) if and only if $x, y \in \mathtt{dom}(s)$, $s(x) \neq s(y)$ and $s(x), s(y) \in \mathtt{dom}(h)$.
7. The command $x{:=}\textbf{cons}(1) \| \textbf{dispose}(42)$ is race-free from any state (s, h) such that $x \in \mathtt{dom}(s)$ and $42 \in \mathtt{dom}(h)$.
8. The command $x{:=}\textbf{cons}(1) \| y{:=}\textbf{cons}(2)$ is race-free from every state (s, h) in which $x, y \in \mathtt{dom}(s)$.
9. The command

   $$x{:=}3 \| \textbf{with } r \textbf{ do } x{:=}x + 1$$

   is not race-free from any state, whereas

   $$\textbf{with } r \textbf{ do } x{:=}3 \| \textbf{with } r \textbf{ do } x{:=}x + 1$$

   is race-free from all states (s, h) with $x \in \mathtt{dom}(s)$.
10. Let $\mathrm{PUT}(x)$ and $\mathrm{GET}(y)$ be the commands introduced earlier. Based on our prior analysis of the traces of these programs, we can deduce that:
    - $\mathrm{PUT}(x) \| (\mathrm{GET}(y); \textbf{dispose}(y))$
      is race-free from (s, h) if and only if: $x, y, z, \mathit{full} \in \mathtt{dom}(s)$ and either $s(\mathit{full}) = 0$ & $s(x) \in \mathtt{dom}(h)$ or $s(\mathit{full}) = 1$ & $s(z) \in \mathtt{dom}(h)$.
    - $(\mathrm{PUT}(x); \textbf{dispose}(x)) \| \mathrm{GET}(y)$
      is race-free from (s, h) if and only if $x, y, z, \mathit{full} \in \mathtt{dom}(s)$ and $s(\mathit{full}) \in \{0, 1\}$ & $s(x) \in \mathtt{dom}(h)$.
    - $(\mathrm{PUT}(x); \textbf{dispose}(x)) \| (\mathrm{GET}(y); \textbf{dispose}(y))$
      is not race-free from any state.
    - $(x{:=}\textbf{cons}(1); \mathrm{PUT}(x)) \| (\mathrm{GET}(y); \textbf{dispose}(y))$
      is race-free from (s, h) if and only if $x, y, z, \mathit{full} \in \mathtt{dom}(s)$ and either $s(\mathit{full}) = 0$, or $s(\mathit{full}) = 1$ & $s(z) \in \mathtt{dom}(h)$.

- $(x{:=}\mathbf{cons}(1); \text{PUT}(x); \mathbf{dispose}(x)) \| \text{GET}(y)$
  is race-free from (s, h) if and only if $x, y, z, \mathit{full} \in \mathtt{dom}(s)$ and $s(\mathit{full}) \in \{0, 1\}$.

We have shown that the trace semantics supports compositional program analysis, and can be used to determine whether or not a command causes a runtime error. The semantics applies to all programs in our programming language, including racy programs as well as race-free programs, but – crucially – we are able to distinguish race-free programs from racy programs. Although it is possible to use the semantic definitions by hand to determine race-freedom in some simple examples, it should be evident from the above analyses that this method is likely to be prohibitively complex for programs on a larger scale.

Now we are ready to introduce a resource-sensitive logic. This logic will be designed to ensure that all provable programs are race-avoiding. Moreover, the logic is designed to abstract away from irrelevant scheduling details and allow attention to be directed more narrowly. The interactions between processes in a well designed parallel program will be amenable to a less taxing analysis that takes advantage of a dynamic form of separation.

7. Separation logic

We begin with the syntax, semantics, and key properties of separation logic formulas, following Reynolds [41].

7.1. Syntax

We use p as a meta-variable ranging over separation logic formulas, given by the following abstract grammar. We let b range over pure boolean expressions, e over pure integer-valued expressions, and E over pure list expressions.

$$p ::= b \mid \mathbf{emp} \mid (e \mapsto e') \mid p_1 * p_2 \mid p_1 \vee p_2 \mid p_1 \wedge p_2 \mid \neg p \mid \exists i.p$$

We also allow inductively defined formulas such as $\mathtt{list}(f)$. We use the usual notation for derived connectives such as implication: $p \Rightarrow q$ is defined to be $(\neg p) \vee q$.

We also use the standard abbreviations, such as $e \mapsto -$ for $\exists i.(e \mapsto i)$ (where i is not free in e) and $e \mapsto E$ for $e \mapsto e_0 * \cdots * (e + n) \mapsto e_n$, when E is $(e_0, \ldots, e_n)$.

Let $\mathtt{free}(p)$ be the set of identifiers occurring free in p, defined as usual by structural induction.

7.2. Semantics

Since the value of a pure expression depends only on the store, we can specify the *atomic semantics* of an integer expression e as a partial function from stores to values. Thus we will write $|e| : \mathbf{S} \rightharpoonup V_{int}$, where $\mathbf{S}$ is the set of stores. Similarly a pure boolean expression b will denote a partial function from stores to truth values, $|b| : \mathbf{S} \rightharpoonup \{\mathbf{true}, \mathbf{false}\}$. And a list expression E denotes a partial function $|E| : \mathbf{S} \rightharpoonup V_{int}^*$ from stores to lists of values. These semantic functions are defined in the traditional, denotational style. For example,

$$|i| = \{(s, v) \mid (i, v) \in s \ \&\ s \in \mathbf{S}\}$$
$$|e_1 + e_2| = \{(s, v_1 + v_2) \mid (s, v_1) \in |e_1| \ \&\ (s, v_2) \in |e_2|\}$$
$$|(e_0, \ldots, e_n)| = \{(s, [v_0, \ldots, v_n]) \mid \forall i.(0 \le i \le n \Rightarrow (s, v_i) \in |e_i|)\}$$

We can connect the atomic semantics and trace semantics of expressions in the following way:

$$(s, v) \in |e| \ \Leftrightarrow\ \exists \rho.\, s \overset{\rho}{\Longrightarrow} s \ \&\ (\rho, v) \in [\![e]\!].$$

The truth value of a separation logic formula p depends on the store and the heap. When $\sigma \models p$ we say that σ *satisfies* p, or that p *holds in* σ.

When $\mathtt{dom}(s) \cap \mathtt{dom}(s') = \{\}$ we say that s and s' are disjoint, written $s \perp s'$, and we write $s \cdot s' = s \cup s'$. Similarly when $\mathtt{dom}(h) \cap \mathtt{dom}(h') = \{\}$ we write $h \perp h'$ and we let $h \cdot h' = h \cup h'$.

Definition 8. The satisfaction relation $(s, h) \models p$ is defined by structural induction on p, for all states (s, h) such that $\texttt{dom}(s) \supseteq \texttt{free}(p)$:

$$\begin{array}{ll}
(s,h) \models b & \text{iff } (s, \mathbf{true}) \in |b| \\
(s,h) \models \mathbf{emp} & \text{iff } h = \{\} \\
(s,h) \models (e \mapsto e') & \text{iff } \exists v, v'.\ (s, v) \in |e| \ \&\ (s, v') \in |e'| \ \&\ h = \{(v, v')\} \\
(s,h) \models p_1 * p_2 & \text{iff } \exists h_1 \perp h_2.\ h = h_1 \cdot h_2 \ \&\ (s, h_1) \models p_1 \ \&\ (s, h_2) \models p_2 \\
(s,h) \models p_1 \wedge p_2 & \text{iff } (s,h) \models p_1 \ \&\ (s,h) \models p_2 \\
(s,h) \models p_1 \vee p_2 & \text{iff } (s,h) \models p_1 \text{ or } (s,h) \models p_2 \\
(s,h) \models \neg p & \text{iff not } (s,h) \models p \\
(s,h) \models \exists i.p & \text{iff } \exists v \in V_{int}.\ ([s \mid i : v], h) \models p
\end{array}$$

We also specify that **abort** $\models p$ is false for all p. We say that a state is *proper* if it is not **abort**; thus $\sigma \models \mathbf{true}$ is true if and only if σ is proper.

We will assume without proof the following agreement theorem, to the effect that the satisfaction of a separation logic formula depends only on the heap and the values of its free identifiers.

Lemma 9 (*Agreement*). *If s_1 agrees with s_2 on $\texttt{free}(p)$ then $(s_1, h) \models p$ if and only if $(s_2, h) \models p$.*

Let $[e/i]p$ be obtained from p by replacing every free occurrence of i by e, renaming bound variables if necessary to avoid capture. The following substitution lemma can be proven by induction on the structure of p.

Lemma 10 (*Substitution*). *For all formulas p, expressions e, identifiers i, and states (s, h),*

$$(s,h) \models [e/i]p \ \Leftrightarrow\ \exists v.\ (s, v) \in |e| \ \&\ ([s \mid i : v], h) \models p.$$

We say that p is *universally valid* if p holds in all (proper) states. Note that an implication $p \Rightarrow q$ is universally valid if and only if every state satisfying p also satisfies q. If $p \Rightarrow q$ and $q \Rightarrow p$ are both universally valid, so that p and q hold in exactly the same states, we say that p and q are logically equivalent.

We say that a formula p holds in a sub-heap of (s, h) if there is a sub-heap $h' \subseteq h$ such that $(s, h') \models p$. We will be particularly concerned with *precise* formulas, which are characterized by the property that in every state there is at most one sub-heap in which the formula holds.

Definition 11. A formula p is precise if, for all states (s, h), there is at most one sub-heap $h' \subseteq h$ such that $(s, h') \models p$.

Note that **emp** and $e \mapsto e'$ are precise, and if R_1 and R_2 are precise, so is $R_1 * R_2$. If b is pure and p_1, p_2 are precise, then $(b \wedge p_1) \vee (\neg b \wedge p_2)$ is precise. If p_1 is precise or p_2 is precise, so is $p_1 \wedge p_2$.

Moreover, if R is precise then, for all p and q, $(p \wedge q) * R$ and $(p * R) \wedge (q * R)$ are logically equivalent.

If R is precise, $(s, h) \models R$, and $h' \subseteq h$, we may refer unambiguously to $(s\lceil\texttt{free}(R), h')$ as the portion of (s, h) *determined by R*.

8. Concurrent separation logic

8.1. Syntax

As in the Owicki–Gries logic, and in O'Hearn's adaptation, we want to prove properties of a parallel program in the context of a collection of assumptions about resources: each resource name occurring in the program is to be associated with a finite set of identifiers (a *protection list*) and a resource invariant. As Owicki remarks, the identifiers chosen to be associated with a particular resource should be "logically" related. Consequently, unlike Owicki–Gries and O'Hearn, we will make this association part of the structure of a logical formula, rather than part of the program itself. We will therefore work with resource-sensitive partial correctness formulas of the form

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

where the pre-condition p and post-condition q are separation logic formulas and Γ is a *resource context* which associates resource names with protection lists and invariants. Each resource invariant is a *precise* separation logic formula.

A typical resource context Γ has the form

$$r_1(X_1) : R_1, \ldots, r_k(X_k) : R_k,$$

in which $k \geq 0$ and for each index $i \in 1 \ldots k$, X_i is the set of identifiers protected by r_i and R_i is the resource invariant for r_i. Let $\mathtt{dom}(\Gamma) = \{r_1, \ldots, r_k\}$ be the set of resource names mentioned in Γ, and $\mathtt{owned}(\Gamma) = \bigcup_{i=1}^{k} X_i$ be the set of identifiers protected by Γ. Let $\mathtt{free}(\Gamma) = \bigcup_{i=1}^{k} \mathtt{free}(R_i)$ be the set of identifiers mentioned in the resource invariants. Let $\mathtt{inv}(\Gamma) = R_1 * \cdots * R_k$ be the separate conjunction of the resource invariants in Γ. In particular, when Γ is empty this is **emp**. Note that 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, we say that:

- 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 specify a connection between the values of protected identifiers and the values of non-critical variables.

The inference rules will be designed to enforce the following additional syntactic constraints[4]:

- Every free write occurrence in c of an identifier used in a resource invariant of Γ is inside a critical region for the corresponding resource.
- Every free occurrence in c of a protected identifier is inside a critical region for the corresponding resource of Γ.
- Every critical identifier of c is protected by a resource.

Resource contexts Γ and Γ' are *disjoint* when $\mathtt{dom}(\Gamma) \cap \mathtt{dom}(\Gamma') = \{\}$ and $\mathtt{owned}(\Gamma) \cap \mathtt{free}(\Gamma') = \{\}$ and $\mathtt{free}(\Gamma) \cap \mathtt{owned}(\Gamma') = \{\}$. We write $\Gamma \perp \Gamma'$ when Γ and Γ' are disjoint, and when this holds we write Γ, Γ' for the union of Γ and Γ'. Note that If Γ and Γ' are well-formed and disjoint, the union context Γ, Γ' is also well-formed.

8.2. Semantics

Intuitively, a resource-sensitive partial correctness formula specifies how the program behaves when executed in an environment which obeys the mutex discipline for resources and respects the protection lists and invariants. Echoing O'Hearn's description of the philosophy behind this methodology, we assume that at all stages the state can be partitioned into the portion owned by the program, the portion owned by its environment, and the portion belonging to the currently available resources. We assume that at all times the *separate conjunction* of the resource invariants holds, for all available resources. The program guarantees to stay within these bounds, provided it can rely on its environment to do likewise. When a process acquires a resource it claims ownership of the protected identifiers and the corresponding (separate) heap portion in which the invariant holds; when releasing the resource it must ensure that the invariant holds again, separately, and yields ownership of the corresponding piece of state.

Based on this intuitive notion of respect, we can now propose an informal notion of validity for resource-sensitive partial correctness formulas.

Proposal 12 (*Informal Notion of Validity*). *A formula $\Gamma \vdash \{p\}c\{q\}$ is valid iff every finite interactive computation of c, from a state satisfying $p * \mathtt{inv}(\Gamma)$ with initial values for $\mathtt{free}(c)$, in an environment that respects Γ, is error-free, respects Γ, and ends in a state satisfying $q * \mathtt{inv}(\Gamma)$.*

The special case when Γ is empty implies conventional partial correctness together with freedom from runtime error: validity of $\{\} \vdash \{p\}c\{q\}$ implies that whenever c is executed from a state satisfying p, with initial values for $\mathtt{free}(c)$, there are no runtime errors, and if execution terminates the final state satisfies q.

We have not yet formulated precisely the notion of an *interactive computation* in an environment that respects Γ. This will be formalized later, but this informal notion of validity should serve as a reasonable guide for now.

We are now ready to present our version of O'Hearn's rules.

[4] We do not use these properties in any of the technical developments that follow, so we will not formalize them or give a proof that they hold in all provable formulas. Nevertheless we state them here since they recall analogous requirements in the Owicki–Gries logic.

8.3. Inference rules

The following are the inference rules of concurrent separation logic. The side conditions of various rules are designed to ensure that every provable formula is well formed. In particular, this means that all resource contexts are well formed, resource invariants are precise, and the pre- and post-condition of a formula do not mention any protected identifiers. Some of the rules have side conditions to ensure that the command obeys the resource discipline, so that protected identifiers, and writes to identifiers occurring in invariants, only appear inside regions. Similar restrictions are made in O'Hearn's paper [31].

- Skip

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

  if $\mathtt{free}(p) \cap \mathtt{owned}(\Gamma) = \{\}$
- Assignment

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

  if $i \notin \mathtt{owned}(\Gamma) \cup \mathtt{free}(\Gamma)$ and $\mathtt{free}(p, e) \cap \mathtt{owned}(\Gamma) = \{\}$
- 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 $i \notin \mathtt{owned}(\Gamma) \cup \mathtt{free}(\Gamma)$
  and $\mathtt{free}(e, e', p) \cap \mathtt{owned}(\Gamma) = \{\}$
- Allocation

$$\frac{}{\Gamma \vdash \{\mathbf{emp}\}i{:=}\mathbf{cons}(E)\{i \mapsto E\}}$$

  if $i \notin \mathtt{free}(E)$ and $i \notin \mathtt{owned}(\Gamma) \cup \mathtt{free}(\Gamma)$ and $\mathtt{free}(E) \cap \mathtt{owned}(\Gamma) = \{\}$
- Update

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

  if $\mathtt{free}(e, e') \cap \mathtt{owned}(\Gamma) = \{\}$
- Disposal

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

  if $\mathtt{free}(e) \cap \mathtt{owned}(\Gamma) = \{\}$
- 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

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

- Local variable

$$\frac{\Gamma \vdash \{p \wedge i = e\}c\{q\}}{\Gamma \vdash \{p\}\mathbf{local}\ i = e\ \mathbf{in}\ c\{q\}}$$

  if $i \notin \mathtt{free}(e, p, q)$ and $\mathtt{free}(p, e, i, q) \cap \mathtt{owned}(\Gamma) = \{\}$

- RENAMING VARIABLE

$$\frac{\Gamma \vdash \{p\}\textbf{local } i' = e \textbf{ in } [i'/i]c\{q\}}{\Gamma \vdash \{p\}\textbf{local } i = e \textbf{ in } c\{q\}}$$

  if $i' \notin \mathtt{free}(c)$
- 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 $\mathtt{free}(p_1, q_1) \cap \mathtt{writes}(c_2) = \mathtt{free}(p_2, q_2) \cap \mathtt{writes}(c_1) = \{\}$
  and $(\mathtt{free}(c_1) \cap \mathtt{writes}(c_2)) \cup (\mathtt{free}(c_2) \cap \mathtt{writes}(c_1)) \subseteq \mathtt{owned}(\Gamma)$
- LOCAL RESOURCE

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

  if $r \notin \mathtt{dom}(\Gamma)$, $X \cap \mathtt{owned}(\Gamma) = \{\}$, $\mathtt{free}(R) \cap \mathtt{owned}(\Gamma) = \{\}$,
  and R is precise.
- 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 $r' \notin \mathtt{res}(c)$
- REGION

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

  if $r \notin \mathtt{dom}(\Gamma)$, $X \cap \mathtt{owned}(\Gamma) = \{\}$, $\mathtt{free}(R) \cap \mathtt{owned}(\Gamma) = \{\}$, R is precise, and $\mathtt{free}(p, q) \cap X = \{\}$
- FRAME

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

  if $\mathtt{free}(I) \cap \mathtt{writes}(c) = \{\}$ and $\mathtt{free}(I) \cap \mathtt{owned}(\Gamma) = \{\}$
- 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,
  and $\mathtt{free}(p', q') \cap \mathtt{owned}(\Gamma) = \{\}$
- EXISTENTIAL

$$\frac{\Gamma \vdash \{p\}c\{q\}}{\Gamma \vdash \{\exists i.p\}c\{\exists i.q\}}$$

  if $i \notin \mathtt{free}(c)$
- AUXILIARY

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

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

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

- DISJUNCTION

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

- EXPANSION

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

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

- CONTRACTION

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

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

8.4. Comments

The rules dealing with the sequential programming constructs of the language are natural adaptations of the corresponding inference rules given by Reynolds, with the incorporation of a resource context and side conditions to ensure well-formedness of the formulas and adherence to the protection policy. For instance, the ASSIGNMENT rule has a side condition to prevent the rule's use when the target identifier is protected or used in a resource invariant, and another side condition to disallow use of a protected identifier on the right-hand side of an assignment. The REGION rule permits such use of protected identifiers inside the body of a critical region for the relevant resource.

The PARALLEL, REGION and RESOURCE rules are based on O'Hearn's proposed 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, but the pre- and post-conditions of the component commands are combined with the separating form of conjunction. The original rule using the standard conjunction is not sound for pointer-programs, as we have already remarked. The well-formedness condition in the RESOURCE and REGION rules require the resource invariant R to be precise. As Reynolds has shown, arbitrary resource invariants cannot be used here without losing soundness.

The AUXILIARY rule similarly adapts the Owicki/Gries rule for auxiliary variables.[5] 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 all assignments to identifiers in X.

The "structural" rules CONJUNCTION and DISJUNCTION are not crucial but can be useful as methodological tools.

The "contextual" rules EXPANSION and CONTRACTION suggest themselves rather naturally as a by-product of our formal development.

We have omitted the obvious structural rules permitting permutation of resource contexts.

9. Examples

To demonstrate the utility of the inference rules, clarify the need for some of the side conditions, and explain what aspects of program behaviour the logic handles, we now present a series of examples. Many of these are more formal versions of examples drawn from O'Hearn's paper, and we include them here to emphasize the virtues of our logical formulation.

[5] Owicki and Gries cite Brinch Hansen [8] and Lauer [29] as having first recognized the need for auxiliary variables in proving correctness properties of concurrent programs.

9.1. Non-termination, deadlock, and runtime errors

Our resource-sensitive notion of partial correctness is designed to support reasoning about the absence of runtime errors. This requires proper account to be taken of the infinite traces of a command, not just its finite traces, in case an infinite trace may lead to a runtime error. The semantics, of course, deals with this possibility appropriately. Nevertheless, our logic is not sensitive to error-free non-termination, and ignores the potential for deadlock. To help clarify the subtleties, consider the following simple commands.

The command **while true do skip** never terminates, and never causes any runtime errors. It is easy to prove the formula

$$\vdash \{\mathbf{true}\}\,\mathbf{while\ true\ do\ skip}\,\{\mathbf{false}\},$$

using the LOOP rule.

On the other hand, the command **while true do dispose**(42) never terminates successfully, and always causes a runtime error. There is no non-trivial formula of the form

$$\vdash \{p\}\,\mathbf{while\ true\ do\ dispose}(42)\ \{q\}$$

that can be proven from our inference rules, since this would require both $p \Rightarrow 42 \mapsto -$ and $\mathbf{emp} \Rightarrow p$ to be universally valid, and this can only happen when p is logically equivalent to **false**.

Finally, let c_1 and c_2 be the following commands:

$$\begin{array}{lll} c_1 & =_{\mathrm{def}} & \mathbf{with}\ r_1\ \mathbf{do\ with}\ r_2\ \mathbf{do}\ x{:=}1 \\ c_2 & =_{\mathrm{def}} & \mathbf{with}\ r_2\ \mathbf{do\ with}\ r_1\ \mathbf{do}\ y{:=}1 \end{array}$$

The formula

$$r_1 : \mathbf{emp}, r_2 : \mathbf{emp} \quad \vdash \quad \{x = 0 \wedge y = 0\}c_1\|c_2\{x = 1 \wedge y = 1\}$$

is provable. Note that the possibility of deadlock, which was evident from the trace set of this program, is ignored by the logic.

From the above formula we can then deduce:

$$\vdash \{x = 0\ \wedge\ y = 0\}\mathbf{resource}\ r_1, r_2\ \mathbf{in}\ (c_1\|c_2)\ \{x = 1 \wedge y = 1\}$$

by RESOURCE and CONSEQUENCE. Again this formula ignores the potential for deadlock.

9.2. Concurrent disposal

Suppose that $p \Rightarrow x \mapsto - * y \mapsto - * q$. We can then construct the following derivation:

- $\vdash \{x \mapsto -\}\mathbf{dispose}(x)\{\mathbf{emp}\}$ by DISPOSAL
- $\vdash \{y \mapsto -\}\mathbf{dispose}(y)\{\mathbf{emp}\}$ by DISPOSAL
- $\vdash \{x \mapsto - * y \mapsto -\}\mathbf{dispose}(x)\|\mathbf{dispose}(y)\{\mathbf{emp} * \mathbf{emp}\}$
  by PARALLEL
- $\vdash \{x \mapsto - * y \mapsto - * q\}\mathbf{dispose}(x)\|\mathbf{dispose}(y)\{\mathbf{emp} * \mathbf{emp} * q\}$
  by FRAME, since $\mathtt{writes}(\mathbf{dispose}(x)\|\mathbf{dispose}(y)) = \{\}$
- $\vdash \{p\}\mathbf{dispose}(x)\|\mathbf{dispose}(y)\{q\}$ by CONSEQUENCE

9.3. Memory manager

Let $list(f)$ be the least predicate satisfying the usual recursive definition:

$$list(f) =_{\mathrm{def}} (f = \mathbf{nil} \wedge \mathbf{emp}) \vee (\exists y. f \mapsto -, y * list(y))$$

This is a precise predicate.

Let $\text{ALLOC}(x)$ and $\text{FREE}(y)$ be the following code fragments:

$$\begin{array}{rcl} \text{ALLOC}(x) & = & \textbf{with}\ mm\ \textbf{do} \\ & & \quad \textbf{if}\ f = \textbf{nil then}\ x{:=}\textbf{cons}(-,-)\ \textbf{else}\ (x{:=}f;\ f{:=}[x+1]) \\[1ex] \text{FREE}(y) & = & \textbf{with}\ mm\ \textbf{do}\ ([y+1]{:=}f;\ f{:=}y) \end{array}$$

The following formula is provable from the rules CONDITIONAL, LOOKUP, SEQUENCE, ALLOCATION and SEQUENTIAL.

$$\begin{array}{lll} \{\} & \vdash & \{\textbf{emp} * list(f)\} \\ & & \textbf{if}\ f = \textbf{nil then}\ x{:=}\textbf{cons}(-,-)\ \textbf{else}\ (x{:=}f;\ f{:=}[x+1]) \\ & & \{x \mapsto -,- * list(f)\} \end{array}$$

Hence, using REGION

$$mm(f) : list(f)\ \vdash \{\textbf{emp}\}\text{ALLOC}(x)\{x \mapsto -,-\}$$

and with the appropriate substitutions we can replay the above derivation to deduce:

$$mm(f) : list(f) \quad \vdash \quad \{\textbf{emp}\}\text{ALLOC}(x_1)\{x_1 \mapsto -,-\}$$

$$mm(f) : list(f) \quad \vdash \quad \{\textbf{emp}\}\text{ALLOC}(x_2)\{x_2 \mapsto -,-\}$$

Using PARALLEL and CONSEQUENCE we get:

$$mm(f) : list(f) \quad \vdash \quad \{\textbf{emp}\}\text{ALLOC}(x_1)\|\text{ALLOC}(x_2)\{x_1 \mapsto -,- * x_2 \mapsto -,-\}$$

Using the RESOURCE rule yields:

$$\begin{array}{lll} \{\} & \vdash & \{list(f)\} \\ & & \textbf{resource}\ mm\ \textbf{in}\ \text{ALLOC}(x_1)\|\text{ALLOC}(x_2) \\ & & \{x_1 \mapsto -,- * x_2 \mapsto -,- * list(f)\} \end{array}$$

Similarly:

$$\begin{array}{rcl} \{\} & \vdash & \{list(f) * y \mapsto -,-\}[y+1]{:=}f;\ f{:=}y\{\textbf{emp} * list(f)\} \\ mm(f) : list(f) & \vdash & \{y \mapsto -,-\}\text{FREE}(y)\{\textbf{emp}\} \end{array}$$

With the appropriate substitutions, we can derive similarly:

$$\begin{array}{rcl} mm(f) : list(f) & \vdash & \{y_1 \mapsto -,-\}\text{FREE}(y_1)\{\textbf{emp}\} \\ mm(f) : list(f) & \vdash & \{y_2 \mapsto -,-\}\text{FREE}(y_2)\{\textbf{emp}\} \end{array}$$

Now using the PARALLEL rule we get:

$$mm(f) : list(f) \quad \vdash \quad \{y_1 \mapsto -,- * y_2 \mapsto -,-\}\text{FREE}(y_1)\|\text{FREE}(y_2)\{\textbf{emp}\}$$

The RESOURCE rule then gives:

$$\begin{array}{lll} \{\} & \vdash & \{y_1 \mapsto -,- * y_2 \mapsto -,- * list(f)\} \\ & & \textbf{resource}\ mm\ \textbf{in}\ \text{FREE}(y_1)\|\text{FREE}(y_2) \\ & & \{list(f)\} \end{array}$$

9.4. Buffer

Let RI be the following (precise) resource invariant:

$$RI :\ (full = 1 \wedge z \mapsto -) \vee (full = 0 \wedge \textbf{emp})$$

Let $\mathrm{PUT}(x)$ and $\mathrm{GET}(y)$ be the following code fragments:

$$\begin{array}{l}\mathrm{PUT}(x) : \textbf{with}\ \mathit{buf}\ \textbf{when}\ \mathit{full} = 0\ \textbf{do}\ (z{:=}x;\ \mathit{full}{:=}1)\\ \mathrm{GET}(y) : \textbf{with}\ \mathit{buf}\ \textbf{when}\ \mathit{full} = 1\ \textbf{do}\ (y{:=}z;\ \mathit{full}{:=}0)\end{array}$$

We can prove:

$$\begin{array}{rcl}\{\} & \vdash & \{(RI * x \mapsto -) \wedge \mathit{full} = 0\}z{:=}x; \mathit{full}{:=}1\{RI * \mathbf{emp}\}\\ \mathit{buf}(z, \mathit{full}) : RI & \vdash & \{x \mapsto -\}\mathrm{PUT}(x)\{\mathbf{emp}\}\end{array}$$

Similarly:

$$\begin{array}{rcl}\{\} & \vdash & \{(RI * \mathbf{emp}) \wedge \mathit{full} = 1\}y{:=}z; \mathit{full}{:=}0\{RI * y \mapsto -\}\\ \mathit{buf}(z, \mathit{full}) : RI & \vdash & \{\mathbf{emp}\}\mathrm{GET}(y)\{y \mapsto -\}\end{array}$$

Hence we can also prove:

$$\begin{array}{l}\mathit{buf}(z, \mathit{full}) : RI \vdash \{\mathbf{emp}\}x{:=}\mathbf{cons}(-);\ \mathrm{PUT}(x)\{\mathbf{emp}\}\\ \mathit{buf}(z, \mathit{full}) : RI \vdash \{\mathbf{emp}\}\mathrm{GET}(y);\ \mathbf{dispose}(y)\{\mathbf{emp}\}\end{array}$$

Using the PARALLEL rule we obtain:

$$\begin{array}{rcl}\mathit{buf}(z, \mathit{full}) : RI & \vdash & \{\mathbf{emp} * \mathbf{emp}\}\\ & & (x{:=}\mathbf{cons}(-);\ \mathrm{PUT}(x))\|(\mathrm{GET}(y);\ \mathbf{dispose}(y))\\ & & \{\mathbf{emp} * \mathbf{emp}\}\end{array}$$

and hence, using CONSEQUENCE,

$$\begin{array}{rcl}\mathit{buf}(z, \mathit{full}) : RI & \vdash & \{\mathbf{emp}\}\\ & & (x{:=}\mathbf{cons}(-);\ \mathrm{PUT}(x))\|(\mathrm{GET}(y);\ \mathbf{dispose}(y))\\ & & \{\mathbf{emp}\}\end{array}$$

Now using the RESOURCE rule we derive:

$$\begin{array}{rcl}\{\} & \vdash & \{RI * \mathbf{emp}\}\\ & & \textbf{resource}\ \mathit{buf}\ \textbf{in}\\ & & \quad (x{:=}\mathbf{cons}(-);\ \mathrm{PUT}(x))\|(\mathrm{GET}(y);\ \mathbf{dispose}(y))\\ & & \{RI * \mathbf{emp}\}\end{array}$$

Again we can simplify via CONSEQUENCE, to obtain:

$$\begin{array}{rcl}\{\} & \vdash & \{RI\}\\ & & \textbf{resource}\ \mathit{buf}\ \textbf{in}\\ & & \quad (x{:=}\mathbf{cons}(-);\ \mathrm{PUT}(x))\|(\mathrm{GET}(y);\ \mathbf{dispose}(y))\\ & & \{RI\}\end{array}$$

Using CONSEQUENCE and the definition of RI we can deduce:

$$\begin{array}{rcl}\{\} & \vdash & \{\mathit{full} = 0 \wedge \mathbf{emp}\}\\ & & \textbf{resource}\ \mathit{buf}\ \textbf{in}\\ & & \quad (x{:=}\mathbf{cons}(-);\ \mathrm{PUT}(x))\|(\mathrm{GET}(y);\ \mathbf{dispose}(y))\\ & & \{(\mathit{full} = 0 \wedge \mathbf{emp})\ \vee\ (\mathit{full} = 1 \wedge z \mapsto -)\}\end{array}$$

Unfortunately the post-condition of this formula does not tell us the whole story, since we expect there to be no heap left over and *full* to be 0. We will revisit this example using auxiliary variables later. Since *full* is a critical variable there's no way to carry around extra information about the value of *full* in the pre- and post-conditions, so we cannot strengthen the formula that way.

9.5. *Ownership is in the eye of the prover*

Suppose we dispose in the first rather than the second process. The program becomes:

$$\begin{array}{l}\textbf{resource}\ \mathit{buf}\ \textbf{in}\\ \quad (x{:=}\textbf{cons}(-);\ \text{PUT}(x);\ \textbf{dispose}(x))\ \|\ \text{GET}(y)\end{array}$$

We must then reason about the program's behaviour under the assumption that no heap locations are deemed to transfer ownership when the resource is acquired or released, so we employ a different resource invariant:

$$RI' =_{\text{def}} (\mathit{full} = 0 \wedge \textbf{emp}) \vee (\mathit{full} = 1 \wedge \textbf{emp})$$

This choice of invariant leads to different specifications for the put and get operations:

$$\begin{array}{rcl}\{\} & \vdash & \{(RI' * x \mapsto -) \wedge \mathit{full} = 0\}z{:=}x;\mathit{full}{:=}1\{RI' * x \mapsto -\}\\ \mathit{buf}(z,\mathit{full}) : RI' & \vdash & \{x \mapsto -\}\text{PUT}(x)\{x \mapsto -\}\end{array}$$

$$\begin{array}{rcl}\{\} & \vdash & \{(RI' * \textbf{emp}) \wedge \mathit{full} = 1\}y{:=}z;\mathit{full}{:=}0\{RI' * \textbf{emp}\}\\ \mathit{buf}(z,\mathit{full}) : RI' & \vdash & \{\textbf{emp}\}\text{GET}(y)\{\textbf{emp}\}\end{array}$$

Hence we can derive:

$$\mathit{buf}(z,\mathit{full}) : RI' \ \vdash\ \{\textbf{emp}\}x{:=}\textbf{cons}(-);\ \text{PUT}(x);\ \textbf{dispose}(x)\{\textbf{emp}\}$$

so the PARALLEL rule gives:

$$\begin{array}{rcl}\mathit{buf}(z,\mathit{full}) : RI' & \vdash & \{\textbf{emp} * \textbf{emp}\}\\ & & (x{:=}\textbf{cons}(-);\ \text{PUT}(x);\ \textbf{dispose}(x))\|\text{GET}(y)\\ & & \{\textbf{emp} * \textbf{emp}\}\end{array}$$

Finishing off with CONSEQUENCE and the RESOURCE rule, we obtain:

$$\begin{array}{rcl}\{\} & \vdash & \{RI'\}\\ & & \textbf{resource}\ \mathit{buf}\ \textbf{in}\\ & & \quad (x{:=}\textbf{cons}(-);\ \text{PUT}(x);\ \textbf{dispose}(x))\|\text{GET}(y)\\ & & \{RI'\}\end{array}$$

Since RI' implies **emp** this post-condition is as strong as can be expected.

We have seen that memory ownership can either be deemed to transfer with a pointer's value, or to stay located in the sending process, depending on what we want to prove. (The distinction is made when we choose a resource invariant.) It is not possible for the ownership to go both ways. For example, there is no resource invariant R that would permit us to prove any non-trivial formula for the program:

$$(x{:=}\textbf{cons}(-);\ \text{PUT}(x);\ \textbf{dispose}(x))\ \|\ (\text{GET}(y);\ \textbf{dispose}(y))$$

in the resource context $\mathit{buf}(z,\mathit{full}) : R$. It is fairly easy to see that for such an invariant R to exist we would have to be able to prove both:

$$\mathit{buf}(z,\mathit{full}) : R \vdash \{\textbf{emp}\}\text{GET}(y)\{y \mapsto -\}$$

and

$$\mathit{buf}(z,\mathit{full}) : R \vdash \{x \mapsto -\}\text{PUT}(x)\{x \mapsto -\}.$$

Thus in turn we would have to be able to prove both:

$$\vdash \{(R * \textbf{emp}) \wedge \mathit{full} = 1\}y{:=}z;\mathit{full}{:=}0\{y \mapsto - * R\}$$

and

$$\vdash \{(R * x \mapsto -) \wedge \mathit{full} = 0\}z{:=}x;\mathit{full}{:=}1\{R * x \mapsto -\}$$

The first requires that $R \wedge \mathit{full} = 1 \Rightarrow z \mapsto -$. But the second requires that $R * x \mapsto -$ holds in the state immediately after setting z to x and *full* to 1. This is impossible since $z = x \wedge (z \mapsto - * x \mapsto -)$ is never true.

9.6. Combining the buffer and the memory manager

Using the notation from before, we had:

$$\begin{array}{l} mm(f) : list(f) \ \vdash \{\mathbf{emp}\}\mathrm{ALLOC}(x)\{x \mapsto -\} \\ mm(f) : list(f) \ \vdash \{y \mapsto -\}\mathrm{FREE}(y)\{\mathbf{emp}\} \end{array}$$

If we let R be the following (precise) resource invariant:

$$R : \ (full = 1 \wedge z \mapsto -, -) \vee (full = 0 \wedge \mathbf{emp})$$

then we can derive the following:

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

The two resource contexts involved here are disjoint, so we can appeal to the EXPANSION rule to obtain:

$$\begin{array}{l} mm(f) : list(f), buf(z, full) : R \ \vdash \{\mathbf{emp}\}\mathrm{ALLOC}(x)\{x \mapsto -, -\} \\ mm(f) : list(f), buf(z, full) : R \ \vdash \{x \mapsto -, -\}\mathrm{PUT}(x)\{\mathbf{emp}\}. \end{array}$$

Hence, using the SEQUENTIAL rule,

$$mm(f) : list(f), buf(z, full) : R \quad \vdash \quad \{\mathbf{emp}\}\mathrm{ALLOC}(x);\ \mathrm{PUT}(x)\{\mathbf{emp}\}$$

Similarly we can derive:

$$mm(f) : list(f), buf(z, full) : R \quad \vdash \quad \{\mathbf{emp}\}\mathrm{GET}(y);\ \mathrm{FREE}(y)\{\mathbf{emp}\}$$

Now the PARALLEL rule yields:

$$\begin{array}{lll} mm(f) : list(f), buf(z, full) : R & \vdash & \{\mathbf{emp}\} \\ & & (\mathrm{ALLOC}(x);\ \mathrm{PUT}(x)) \| (\mathrm{GET}(y);\ \mathrm{FREE}(y)) \\ & & \{\mathbf{emp}\} \end{array}$$

There are two ways to apply the RESOURCE rule, and the rule can be applied twice in either order, yielding:

$$\begin{array}{lll} mm(f) : list(f) & \vdash & \{R\} \\ & & \mathbf{resource}\ buf\ \mathbf{in} \\ & & \quad (\mathrm{ALLOC}(x);\ \mathrm{PUT}(x)) \| (\mathrm{GET}(y);\ \mathrm{FREE}(y)) \\ & & \{R\} \end{array}$$

or

$$\begin{array}{lll} buf(z, full) : R & \vdash & \{list(f)\} \\ & & \mathbf{resource}\ mm\ \mathbf{in} \\ & & \quad (\mathrm{ALLOC}(x);\ \mathrm{PUT}(x)) \| (\mathrm{GET}(y);\ \mathrm{FREE}(y)) \\ & & \{list(f)\} \end{array}$$

followed by:

$$\begin{array}{ll} \vdash & \{R * list(f)\} \\ & \mathbf{resource}\ mm, buf\ \mathbf{in} \\ & \quad (\mathrm{ALLOC}(x);\ \mathrm{PUT}(x)) \| (\mathrm{GET}(y);\ \mathrm{FREE}(y)) \\ & \{R * list(f)\} \end{array}$$

9.7. Using auxiliary variables

Previously we proved the following formula:

$$\begin{array}{lll} \{\} & \vdash & \{\mathit{full} = 0 \wedge \mathbf{emp}\} \\ & & \mathbf{resource}\ \mathit{buf}\ \mathbf{in} \\ & & \quad (x{:=}\mathbf{cons}(-);\ \text{PUT}(x)) \,\|\, (\text{GET}(y);\ \mathbf{dispose}(y)) \\ & & \{(\mathit{full} = 0 \wedge \mathbf{emp})\ \vee\ (\mathit{full} = 1 \wedge z \mapsto -)\} \end{array}$$

and we noted that the post-condition is not strong enough to imply that there is no "memory leak" with this program. In fact the trace set of this command shows that on termination *full* will be 0 and the heap will be empty. However, since *full* is a critical variable, read and written by both processes, there is no way to propagate information about the value of *full* in the logic, except by invoking the resource invariant. We can skirt around this difficulty by using auxiliary variables, as suggested by Owicki and Gries to deal with similar problems in the pointer-free setting.

Let $\text{PUT}'(x)$ and $\text{GET}'(y)$ be the following:

$$\begin{array}{lll} \text{PUT}'(x) & : & \mathbf{with}\ \mathit{buf}\ \mathbf{when}\ \mathit{full} = 0\ \mathbf{do} \\ & & \quad (z{:=}x;\ \mathit{full}{:=}1;\ \mathit{start}{:=}0) \\ \text{GET}'(y) & : & \mathbf{with}\ \mathit{buf}\ \mathbf{when}\ \mathit{full} = 1\ \mathbf{do} \\ & & \quad (y{:=}z;\ \mathit{full}{:=}0;\ \mathit{finish}{:=}1) \end{array}$$

Note that $\text{PUT}'(x)$ and $\text{GET}'(y)$ are obtained from $\text{PUT}(x)$ and $\text{GET}(y)$ by inserting assignments to *start* and *finish*. Since these assignments do not affect the flow of control and have no influence on the values of any other identifiers, or on the heap, *start* and *finish* are indeed auxiliary variables.

Let R' be the (precise) formula:

$$\begin{array}{l} (\mathit{full} = 0 \wedge \mathbf{emp} \wedge (\mathit{start} = 1 \Leftrightarrow \mathit{finish} = 0)) \\ \vee\ (\mathit{full} = 1 \wedge z \mapsto - \wedge \mathit{start} = 0 \wedge \mathit{finish} = 0) \end{array}$$

We can prove the formulas:

$$\begin{array}{lll} \mathit{buf}(z, \mathit{full}) : R' & \vdash & \{\mathit{start} = 1 \wedge \mathbf{emp}\} \\ & & x{:=}\mathbf{cons}(-);\ \text{PUT}'(x) \\ & & \{\mathit{start} = 0 \wedge \mathbf{emp}\} \\ \\ \mathit{buf}(z, \mathit{full}) : R' & \vdash & \{\mathit{finish} = 0 \wedge \mathbf{emp}\} \\ & & \text{GET}'(y);\ \mathbf{dispose}(y) \\ & & \{\mathit{finish} = 1 \wedge \mathbf{emp}\} \\ \\ \mathit{buf}(z, \mathit{full}) : R' & \vdash & \{\mathit{start} = 1 \wedge \mathit{finish} = 0 \wedge \mathbf{emp}\} \\ & & (x{:=}\mathbf{cons}(-);\ \text{PUT}'(x)) \,\|\, (\text{GET}'(y);\ \mathbf{dispose}(y)) \\ & & \{\mathit{start} = 0 \wedge \mathit{finish} = 1 \wedge \mathbf{emp}\} \\ \\ & \vdash & \{\mathit{start} = 1 \wedge \mathit{finish} = 0\ \wedge\ R'\} \\ & & \mathbf{resource}\ \mathit{buf}\ \mathbf{in} \\ & & \quad (x{:=}\mathbf{cons}(-);\ \text{PUT}'(x)) \,\|\, (\text{GET}'(y);\ \mathbf{dispose}(y)) \\ & & \{\mathit{start} = 0 \wedge \mathit{finish} = 1 \wedge R'\} \end{array}$$

We then derive:

$$\begin{array}{ll} \vdash & \{\mathit{full} = 0 \wedge \mathbf{emp}\} \\ & \mathit{start}{:=}1; \\ & \mathit{finish}{:=}0; \\ & \mathbf{resource}\ \mathit{buf}\ \mathbf{in} \\ & \quad (x{:=}\mathbf{cons}(-);\ \text{PUT}'(x)) \,\|\, (\text{GET}'(y);\ \mathbf{dispose}(y)) \\ & \{\mathit{start} = 0 \wedge \mathit{finish} = 1 \wedge R'\} \end{array}$$

Hence, using CONSEQUENCE,

$$
\begin{array}{ll}
\vdash & \{\mathit{full} = 0 \wedge \mathbf{emp}\} \\
& \mathit{start}{:=}1; \\
& \mathit{finish}{:=}0; \\
& \mathbf{resource}\ \mathit{buf}\ \mathbf{in} \\
& \qquad (x{:=}\mathbf{cons}(-);\ \mathrm{PUT}'(x))\,\|\,(\mathrm{GET}'(y);\ \mathbf{dispose}(y)) \\
& \{\mathit{full} = 0 \wedge \mathbf{emp}\}
\end{array}
$$

Since *start* and *finish* are auxiliary variables and do not occur free in the pre-condition or the post-condition, we can use the AUXILIARY rule to deduce:

$$
\begin{array}{ll}
\vdash & \{\mathit{full} = 0 \wedge \mathbf{emp}\} \\
& \mathbf{resource}\ \mathit{buf}\ \mathbf{in} \\
& \qquad (x{:=}\mathbf{cons}(-);\ \mathrm{PUT}(x))\,\|\,(\mathrm{GET}(y);\ \mathbf{dispose}(y)) \\
& \{\mathit{full} = 0 \wedge \mathbf{emp}\},
\end{array}
$$

since the removal of the auxiliary assignments converts $\mathrm{PUT}'(x)$ to $\mathrm{PUT}(x)$ and $\mathrm{GET}'(y)$ to $\mathrm{GET}(y)$.

As desired, this formula expresses the property that this program is error-free and does not leak memory.

10. Towards validity

We now return to the problem of interpretation that we raised earlier but have not yet settled. We wish to establish that every provable resource-sensitive formula is *valid*, but we need to determine precisely what that should mean. Earlier we proposed informally that $\Gamma \vdash \{p\}c\{q\}$ should be regarded as valid if every finite interactive computation of c from a state satisfying $p * \mathtt{inv}(\Gamma)$, in an environment which respects Γ, is error-free, respects Γ, and ends in a state satisfying $q * \mathtt{inv}(\Gamma)$. We might try to formalize this notion of validity in terms of the enabling relation, as in:

for every trace α of c, and all states σ and σ',
if σ satisfies $p * \mathtt{inv}(\Gamma)$ and $\sigma \overset{\alpha}{\Longrightarrow} \sigma'$, then σ' satisfies $q * \mathtt{inv}(\Gamma)$.

This characterization of "validity" would work well for sequential programs. However, it only involves the *sequential* traces of c. As a result it will not suffice for parallel programs: we would be unable to establish soundness of the proof rule for parallel composition. What is missing here is the ability to quantify over traces with *gaps* at resource actions, assuming that the gaps will be filled by actions on protected identifiers performed by an environment which respects invariants and obeys the mutex constraints on resources.

To obtain a suitably general notion of validity we will work with *local* states, so that we can make accurate statements about the portion of the state which is deemed to be "owned" by the program and the pieces of state that are designated to transfer on resource acquisition or release.

10.1. Local states and local enabling

Given a resource context Γ, a process holding resource set A is allowed to access unprotected identifiers, as well as identifiers protected by resources in A, but should be prevented from accessing identifiers protected by other resources. We will therefore 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 the resources in A. Similarly we let $\Gamma\backslash A$ be the rest of Γ. Note that a local state also satisfies $\mathtt{dom}(s) \cap \mathtt{owned}(\Gamma\backslash A) = \{\}$.

We introduce a family of *local enabling relations* (Fig. 2): a step

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

will mean that in the local state (s, h, A) a program is permitted to perform action λ, causing the local state to change to (s', h', A'). This is a *partial relation*, defined only when (s, h, A) is consistent with Γ and the action is enabled in the usual manner; whenever $(s, h, A) \xrightarrow[\Gamma]{\lambda} (s', h', A')$ holds it will follow that (s', h', A') is also consistent with Γ and the action "respects" the resource constraints and the ownership rules. We use the error state **abort** to handle runtime

- $(s, h, A) \xrightarrow[\Gamma]{\delta} (s, h, A)$ and $(s, h, A) \xrightarrow[\Gamma]{abort}$ **abort** always
- $(s, h, A) \xrightarrow[\Gamma]{i=v} (s, h, A)$ iff $(i, v) \in s$
- $(s, h, A) \xrightarrow[\Gamma]{i=v}$ **abort** iff $i \notin \mathrm{dom}(s)$
- $(s, h, A) \xrightarrow[\Gamma]{i:=v} ([s \mid i : v], h, A)$ iff $i \in \mathrm{dom}(s) - \mathtt{free}(\Gamma \backslash A)$
- $(s, h, A) \xrightarrow[\Gamma]{i:=v}$ **abort** iff $i \notin \mathrm{dom}(s)$ or $i \in \mathtt{free}(\Gamma \backslash A)$
- $(s, h, A) \xrightarrow[\Gamma]{[l]=v} (s, h, A)$ iff $(l, v) \in h$
- $(s, h, A) \xrightarrow[\Gamma]{[l]=v}$ **abort** iff $l \notin \mathtt{dom}(h)$
- $(s, h, A) \xrightarrow[\Gamma]{[l]:=v'} (s, [h \mid l : v'], A)$ iff $l \in \mathtt{dom}(h)$
- $(s, h, A) \xrightarrow[\Gamma]{[l]:=v'}$ **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 $dom(h) \cap \{l, l+1, \ldots, l+n\} = \{\}$
- $(s, h, A) \xrightarrow[\Gamma]{disp(l)} (s, h \backslash l, A)$ iff $l \in \mathtt{dom}(h)$
- $(s, h, A) \xrightarrow[\Gamma]{disp(l)}$ **abort** iff $l \notin \mathtt{dom}(h)$
- $(s, h, A) \xrightarrow[\Gamma]{try(r)} (s, h, A)$ iff $r \in A$
- $(s, h, A) \xrightarrow[\Gamma, r(X):R]{acq(r)} (s \cdot s', h \cdot h', A \cup \{r\})$ iff
  $r \notin A$, $h \perp h'$, $\mathtt{dom}(s') = X$ and $(s \cdot s', h') \models R$
- $(s, h, A) \xrightarrow[\Gamma, r(X):R]{rel(r)} (s \backslash X, h - h', A - \{r\})$ iff
  $r \in A$, $h' \subseteq h$, and $(s, h') \models R$
- $(s, h, A) \xrightarrow[\Gamma, r(X):R]{rel(r)}$ **abort** iff $\forall h' \subseteq h.\ \neg(s, h') \models R$

Fig. 2. Local enabling relations on states consistent with Γ.

errors such as races or an attempt to use an identifier or heap address not locally owned, or an attempt to release a resource in a state for which no sub-heap satisfies the corresponding invariant. Thus a step

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

indicates that the action λ is enabled but would cause a runtime error or break the rules. As before it is convenient to extend this enabling relation so that **abort** $\xrightarrow[\Gamma]{\lambda}$ **abort** holds, for all Γ and λ.

The local enabling relations are designed to embody the ownership rules and transfer policy implied by the resource context: each time the program acquires a resource it claims ownership of exactly the store and heap needed to satisfy the relevant invariant, and on releasing a resource it relinquishes ownership of the store and heap determined by the invariant. (The importance of precision here is evident: since resource invariants are precise there will be, for a given global store $s'' \supseteq s$ at most one heap h' such that $(s'', h') \models R$, and hence at most one local transition from (s, h, A) involving the action $acq(r)$ consistent with this global state.)

This leads us to a notion of *local computation* in which the program's claims on heap and protected identifiers are guided by the resource invariants. A local computation can be seen to reflect the *program's view* of the global state during an interactive execution with an environment that respects the resource environment. We write $\sigma \xrightarrow[\Gamma]{\alpha} \sigma'$ when there is a local computation α from σ to σ' respecting Γ. We allow this notation when α is finite, in which case σ' may be a proper state or **abort**; if α is $\lambda_0 \ldots \lambda_n$ we therefore have $\sigma \xrightarrow[\Gamma]{\lambda_0 \ldots \lambda_n} \sigma'$ if there is a sequence of states $\sigma_0, \ldots, \sigma_{n-1}$ such that

$$\sigma \xrightarrow[\Gamma]{\lambda_0} \sigma_0 \xrightarrow[\Gamma]{\lambda_1} \sigma_1 \cdots \xrightarrow[\Gamma]{\lambda_{n-1}} \sigma_{n-1} \xrightarrow[\Gamma]{\lambda_n} \sigma'.$$

We also allow the notation when α is an infinite trace and σ' is **abort**, to handle the case when a program may cause an error part way through a non-terminating computation. Thus $\sigma \xrightarrow[\Gamma]{\alpha}$ **abort** means that a program attempting the trace α from σ aborts, possibly in mid-trace. And we write $\sigma \xrightarrow[\Gamma]{\alpha} \cdot$ to indicate that the trace α is locally enabled, or more informally, that α respects Γ from σ. Note that this notion of local computation makes sense for arbitrary traces, not just for sequential traces.

10.2. Fundamental properties

In preparation for the soundness analysis we build up a series of results expressing basic properties of local computation. Most of the proofs are straightforward, making extensive use of the relevant definitions. We include more details in the Appendix for the more complex cases.

First we show that executing a command with a trivial resource context that never transfers any state is the same as executing the command without interference.

Lemma 13 (*Empty Context Lemma*). *Let α be a finite trace, let $\{r_1, \dots, r_n\}$ be the set of resource names occurring in actions of α, and let Γ_0 be the resource context r_1 :* **emp**, …, *r_n :* **emp**. *Then:*

$$(s, h, A) \overset{\alpha}{\Longrightarrow} \sigma' \text{ if and only if } (s, h, A) \xrightarrow[\Gamma_0]{\alpha} \sigma'.$$

Theorem 14 (*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 15. *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 definition therefore makes sense:

Definition 16. We define $(s, h) \xrightarrow[\Gamma]{\alpha} (s', h')$ if $(s, h, \{\}) \xrightarrow[\Gamma]{\alpha} (s', h', \{\})$.

The effect of a program in a local computation depends only on the heap, the values of its free identifiers, and the values of (non-critical) identifiers occurring free in resource invariants; moreover, a program can only change the value of identifiers which have a free write occurrence.

Theorem 17 (*Agreement*). *Let $\alpha \in [\![c]\!]$ and suppose that s_1 agrees with s_2 on $\mathtt{free}(c, \Gamma)$.*

- *If $(s_1, h) \xrightarrow[\Gamma]{\alpha}$* **abort**, *then $(s_2, h) \xrightarrow[\Gamma]{\alpha}$* **abort**.
- *If $(s_1, h) \xrightarrow[\Gamma]{\alpha} (s_1', h')$ then there is a store s_2' such that*
  $(s_2, h) \xrightarrow[\Gamma]{\alpha} (s_2', h')$ and s_1' agrees with s_2' on $\mathtt{free}(c, \Gamma)$.

If $\alpha \in [\![c]\!]$ and $(s, h) \xrightarrow[\Gamma]{\alpha} (s', h')$ then s' agrees with s' except on $\mathtt{writes}(c)$.

Again this property, together with the Empty Context Lemma, implies the analogous property for sequential traces.

Corollary 18. *If $\alpha \in [\![c]\!]$ and s_1 agrees with s_2 on $\mathtt{free}(c)$, then*

- *$(s_1, h) \overset{\alpha}{\Longrightarrow}$* **abort** *implies $(s_2, h) \overset{\alpha}{\Longrightarrow}$* **abort**
- *$(s_1, h) \overset{\alpha}{\Longrightarrow} (s_1', h')$ implies $(s_2, h) \overset{\alpha}{\Longrightarrow} (s_2, h')$ for some store s_2' that agrees with s_1' on $\mathtt{free}(c)$.*

Moreover, if $\alpha \in [\![c]\!]$ and: $(s, h) \overset{\alpha}{\Longrightarrow} (s', h')$, then s' agrees with s except on $\mathtt{writes}(c)$.

As in the sequential setting, we obtain a frame property.

Theorem 19 (*Frame*). *Let $\alpha \in [\![c]\!]$ and suppose $h_1 \perp h_2$ and $h = h_1 \cdot h_2$.*

- *If $(s, h) \xrightarrow[\Gamma]{\alpha}$* **abort** *then $(s, h_1) \xrightarrow[\Gamma]{\alpha}$* **abort**.

- *If* $(s, h) \xrightarrow[\Gamma]{\alpha} (s', h')$ *then either* $(s, h_1) \xrightarrow[\Gamma]{\alpha}$ **abort**, *or there is a heap* h'_1 *such that* $h'_1 \perp h_2$, $h' = h'_1 \cdot h_2$, *and* $(s, h_1) \xrightarrow[\Gamma]{\alpha} (s', h'_1)$.

Yet again by invoking the Empty Context Lemma we deduce the corresponding property for interference-free executions.

Corollary 20. *Let* $\alpha \in [\![c]\!]$, $h_1 \perp h_2$, *and* $h = h_1 \cdot h_2$.

- *If* $(s, h) \overset{\alpha}{\Longrightarrow}$ **abort** *then* $(s, h_1) \overset{\alpha}{\Longrightarrow}$ **abort**.
- *If* $(s, h) \overset{\alpha}{\Longrightarrow} (s', h')$ *then either* $(s, h_1) \overset{\alpha}{\Longrightarrow}$ **abort**, *or there is a heap* h'_1 *such that* $h'_1 \perp h_2$, $h' = h'_1 \cdot h_2$, *and* $(s, h_1) \overset{\alpha}{\Longrightarrow} (s', h'_1)$.

Using the frame theorem as a basis, we can establish a *parallel decomposition* property relating 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. Suppose $c_1 \| c_2$ has a local computation α from (s, h) to (s', h'), where α is obtained by interleaving $\alpha_1 \in [\![c_1]\!]$ and $\alpha_2 \in [\![c_2]\!]$, and we choose a partition (h_1, h_2) of h. If c_1 and c_2 have successful computations α_1 from $(s \backslash \mathtt{writes}(\alpha_2), h_1)$ and α_2 from $(s \backslash \mathtt{writes}(\alpha_1), h_2)$, the results of these computations fit together, determining (s', h') in a natural manner. On the other hand, if α leads to an error one (or both) of α_1, α_2 must lead to error. The following theorem expresses this intuition more precisely. We include proof details in the Appendix.

Theorem 21 (*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* $(s, h) \xrightarrow[\Gamma]{\alpha}$ **abort** *then*
  $(s \backslash \mathtt{writes}(\alpha_2), h_1) \xrightarrow[\Gamma]{\alpha_1}$ **abort** *or* $(s \backslash \mathtt{writes}(\alpha_1), h_2) \xrightarrow[\Gamma]{\alpha_2}$ **abort**.
- *If* $(s, h) \xrightarrow[\Gamma]{\alpha} (s', h')$ *then*
  $(s \backslash \mathtt{writes}(\alpha_2), h_1) \xrightarrow[\Gamma]{\alpha_1}$ **abort** *or* $(s \backslash \mathtt{writes}(\alpha_1), h_2) \xrightarrow[\Gamma]{\alpha_2}$ **abort**,
  or there are disjoint heaps $h'_1 \perp h'_2$ *such that* $h' = h'_1 \cdot h'_2$ *and:*
  - $(s \backslash \mathtt{writes}(\alpha_2), h_1) \xrightarrow[\Gamma]{\alpha_1} (s' \backslash \mathtt{writes}(\alpha_2), h'_1)$
  - $(s \backslash \mathtt{writes}(\alpha_1), h_2) \xrightarrow[\Gamma]{\alpha_2} (s' \backslash \mathtt{writes}(\alpha_1), h'_2)$

The following property of local computations shows that our definition handles resources sensibly, and provides a way to connect local computations of **resource** r **in** c in resource context Γ with local computations of c in resource context $\Gamma, r(X) : R$. Recall that every trace of **resource** r **in** c has the form $\beta \backslash r$, where β is a trace of c that is sequential for r.

Theorem 22 (*Local Resource Lemma*). *Let* $\beta \in [\![c]\!]_r$ *and suppose* $h_1 \perp h_2$ *and* $(s, h_2) \vdash R$.

- *If* $(s, h_1 \cdot h_2) \xrightarrow[\Gamma]{\beta \backslash r}$ **abort** *then* $(s \backslash X, h_1) \xrightarrow[\Gamma, r(X):R]{\beta}$ **abort**.
- *If* $(s, h_1 \cdot h_2) \xrightarrow[\Gamma]{\beta \backslash r} (s', h')$ *then either* $(s \backslash X, h_1) \xrightarrow[\Gamma, r(X):R]{\beta}$ **abort**, *or there are heaps* $h'_1 \perp h'_2$ *such that* $h' = h'_1 \cdot h'_2$, $(s', h'_2) \models R$, *and* $(s \backslash X, h_1) \xrightarrow[\Gamma, r(X):R]{\beta} (s' \backslash X, h'_1)$.

There is an analogous property relating the local computations of a block **local** $i = e$ **in** c with those of its body.

Theorem 23 (*Local Variable Lemma*). *Let* $\beta \in [\![c]\!]_{[i:v]}$ *and* $i \not\in \mathtt{owned}(\Gamma)$.

- *If* $(s, h) \xrightarrow[\Gamma]{\beta \backslash i}$ **abort** *then* $([s \mid i : v], h) \xrightarrow[\Gamma]{\beta}$ **abort**.
- *If* $(s, h) \xrightarrow[\Gamma]{\beta \backslash i} (s', h')$ *then either* $([s \mid i : v], h) \xrightarrow[\Gamma]{\beta}$ **abort**, *or there is a value* v' *such that* $([s \mid i : v], h) \xrightarrow[\Gamma]{\beta} ([s' \mid i : v'], h')$.

11. Validity

The definition of local enabling was designed to formalize 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. With this definition in hand we can at last give a formal version of validity.

Definition 24. The formula $\Gamma \vdash \{p\}c\{q\}$ is valid if for all traces α of c, all local states (s, h) such that $\mathtt{dom}(s) \supseteq \mathtt{free}(c, \Gamma) - \mathtt{owned}(\Gamma)$, and all σ', if $(s, h) \models p$ and $(s, h) \xrightarrow[\Gamma]{\alpha} \sigma'$ then $\sigma' \models q$.

Note that this definition involves the local enabling relation, so that the quantification ranges over local states (s, h) consistent with Γ, for which $\mathtt{dom}(s) \cap \mathtt{owned}(\Gamma) = \{\}$. Since **abort** does not satisfy q validity implies freedom from race conditions. Furthermore, this notion of validity involves *all* traces of c, not just the sequential traces and not just the finite traces; 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.

It is easy to see from the above definition that, when Γ is the empty context and c has no free resource names, validity of $\{\} \vdash \{p\}c\{q\}$ implies the usual notion of partial correctness together with the guaranteed absence of runtime errors: in every terminating execution of c from a state satisfying p, with values for all free identifiers of c, there is no runtime error and the final state satisfies q . More generally, the same implication holds when $\mathtt{res}(c) = \{r_1, \ldots, r_n\}$ and Γ_0 is the context $r_1 : \mathbf{emp}, \ldots, r_n : \mathbf{emp}$: validity of $\Gamma_0 \vdash \{p\}c\{q\}$ implies the usual notion of partial correctness together with absence of errors.

Again we return to some examples to illustrate validity.

1. $\vdash \{\mathbf{true}\}$ **while true do skip** $\{\mathbf{false}\}$ is valid, because for all states σ there is no state σ' such that $\sigma \xrightarrow[\{\}]{\delta^\omega} \sigma'$.
2. The formula $\vdash \{p\}\mathbf{dispose}(x)\|\mathbf{dispose}(y)\{q\}$ is valid if and only if $p \Rightarrow (x \mapsto -) * (y \mapsto -) * q$ is universally valid.

   Suppose $p \Rightarrow (x \mapsto -) * (y \mapsto -) * q$ is universally valid. Let (s, h) be a state satisfying p and let $s(x) = v, s(y) = v'$. It follows that $v \neq v'$, and $(s, h\backslash\{v, v'\})$ satisfies q. Every trace of $\mathbf{dispose}(x)\|\mathbf{dispose}(y)$ enabled from (s, h) is an interleaving of $x{=}v\, disp(v)$ with $y{=}v'\, disp(v')$, and therefore leads to the state $(s, h\backslash\{v, v'\})$, which satisfies q as required. The converse implication is straightforward.
3. Let Γ be the context $r(x) : x = m + n \wedge \mathbf{emp}$. Note that in this example the resource invariant connects the value of the protected identifier x with the values of two unprotected identifiers m and n. The formula

   $$\Gamma \vdash \{m = 0\}\mathbf{with}\ r\ \mathbf{do}\ (x{:=}x + 1; m{:=}m + 1)\{m = 1\}$$

   is clearly well formed, and also valid. To see this, let (s, h) be a local state such that $\mathtt{dom}(s) \supseteq \{m, n\}, x \notin \mathtt{dom}(s)$, and $(s, h) \models m = 0$, so that $s(m) = 0$ and $s(n) = v$ for some integer v. The only relevant trace of the program, enabled from this state, is:

   $$acq(r)\, x{=}v\, x{:=}v + 1\, m{=}0\, m{:=}1\, rel(r)$$

   and we have:

   $$\begin{array}{lll} (s, h) & \xrightarrow[\Gamma]{acq(r)} & ([s \mid x : v], h, \{r\}) \\ & \xrightarrow[\Gamma]{x=v} & ([s \mid x : v], h, \{r\}) \\ & \xrightarrow[\Gamma]{x:=v+1} & ([s \mid x : v + 1], h, \{r\}) \\ & \xrightarrow[\Gamma]{m=0} & ([s \mid x : v + 1], h, \{r\}) \\ & \xrightarrow[\Gamma]{m:=1} & ([s \mid x : v + 1, m : 1], h, \{r\}) \\ & \xrightarrow[\Gamma]{rel(r)} & ([s \mid m : 1], h), \end{array}$$

   leading to a state satisfying $m = 1$, as required.

   By symmetry the formula

   $$\Gamma \vdash \{n = 0\}\mathbf{with}\ r\ \mathbf{do}\ (x{:=}x + 1; n{:=}n + 1)\{n = 1\}$$

is also well formed and valid. Moreover, the formula

$$\begin{array}{lll} \Gamma & \vdash & \{m = 0 \ \wedge \ n = 0\} \\ & & \quad \textbf{with}\ r\ \textbf{do}\ (x{:=}x + 1; m{:=}m + 1) \\ & & \ \| \ \textbf{with}\ r\ \textbf{do}\ (x{:=}x + 1; n{:=}n + 1) \\ & & \{m = 1 \ \wedge \ n = 1\} \end{array}$$

is well formed, since x is the only critical variable and it is protected by r. This formula is also valid, because when (s, h) is a local state such that $\texttt{dom}(s) \supseteq \{m, n\}$, $x \notin \texttt{dom}(s)$, and $s(m) = s(n) = 0$, every trace of this parallel command enabled from (s, h) leads to the state $([s \mid m : 1, n : 1], h)$.

12. Soundness

Theorem 25 (*Soundness*). *Every provable formula $\Gamma \vdash \{p\}c\{q\}$ is valid.*

Proof. Our inference rules are subject to an implicit well-formedness constraint: only well formed instance of rules are permitted. To prove soundness of the proof system we show that each well formed instance of an inference rule is sound: if the rule's premises and conclusion are well formed, the side conditions hold, and the premises are valid, then the conclusion is valid. It then follows, by induction on the length of the derivation, that every provable formula is valid.

For some of the rules this is fairly easy, although we provide details since the notion of validity is rather subtle. The proofs for Update, Allocation and Disposal are carried out in a similar manner to the proof given here for Lookup; these are all straightforward adaptations of the soundness analysis that can be given for these constructs in the sequential setting. Similarly the rules for Conditional and Loop are straightforward.

- Skip
The formula $\Gamma \vdash \{p\}\textbf{skip}\{p\}$ is clearly valid, because the only computation of **skip** from a state σ satisfying p has the form $\sigma \xrightarrow[\Gamma]{\delta} \sigma$. The well-formedness assumption that $\texttt{free}(p) \cap \texttt{owned}(\Gamma) = \{\}$ simply ensures that if (s, h) satisfies p then so does $(s\backslash\texttt{owned}(\Gamma), h)$.
- Assignment
We verify that the formula $\Gamma \vdash \{[e/i]p\}i{:=}e\{p\}$ is valid when i, $\texttt{free}(e)$, and $\texttt{free}(p)$ are disjoint from $\texttt{owned}(\Gamma)$, and i is not free in any resource invariant of Γ. Let (s, h) be a state satisfying the pre-condition $[e/i]p$ and such that $\texttt{dom}(s) \supseteq \texttt{free}(i{:=}e, \Gamma) - \texttt{owned}(\Gamma)$. This implies that $i \in \texttt{dom}(s)$ and $\texttt{free}(e) \subseteq \texttt{dom}(s)$. Moreover, $(s\backslash\texttt{owned}(\Gamma), h) \models [e/i]p$.
The traces of $i{:=}e$ have the form $\rho\, i{:=}v$, where $(\rho, v) \in [\![e]\!]$. Every local computation of $i{:=}e$ from (s, h) will therefore have the form:
$$(s, h) \xrightarrow[\Gamma]{\rho\, i:=v} ([s \mid i : v], h)$$
where (ρ, v) is an evaluation trace of e enabled from s. Hence $(s, v) \in |e|$ and $(s, h) \xrightarrow[\Gamma]{i:=v} ([s \mid i : v], h)$. Since $(s, h) \models [e/i]p$ and $(s, v) \in |e|$ the Substitution Lemma implies that $([s \mid i : v], h) \models p$, as required.
- Sequential Composition
Suppose that the formulas $\Gamma \vdash \{p_1\}c_1\{p_2\}$, $\Gamma \vdash \{p_2\}c_2\{p_3\}$ are valid and well formed. It is clear that $\Gamma \vdash \{p_1\}c_1; c_2\{p_3\}$ is also well formed. We need to show that $\Gamma \vdash \{p_1\}c_1; c_2\{p_3\}$ is valid.
Suppose $(s, h) \models p_1$ and $\texttt{dom}(s) \supseteq \texttt{free}(c_1; c_2, \Gamma) - \texttt{owned}(\Gamma)$. Every trace of $c_1; c_2$ has the form $\alpha = \alpha_1\alpha_2$ for some traces α_1 of c_1 and α_2 of c_2. Suppose we have a local computation of $c_1; c_2$ of the form
$$(s, h) \xrightarrow[\Gamma]{\alpha_1\alpha_2} \sigma''.$$
We need to show that $\sigma'' \models p_3$. Since $\texttt{dom}(s) \supseteq \texttt{free}(c_1, \Gamma) - \texttt{owned}(\Gamma)$, by validity of $\Gamma \vdash \{p_1\}c_1\{p_2\}$ we know that the computation along α_1 is error-free. If α_1 is infinite (so $\alpha = \alpha_1$) there is no more to prove. Otherwise α_1 is finite and there is a (proper) state (s', h') such that:
$$(s, h) \xrightarrow[\Gamma]{\alpha_1} (s', h') \xrightarrow[\Gamma]{\alpha_2} \sigma''$$
and $(s', h') \models p_2$. By the local respect lemma, $\texttt{dom}(s') = \texttt{dom}(s)$, so we have $\texttt{dom}(s') \supseteq \texttt{free}(c_2, \Gamma) - \texttt{owned}(\Gamma)$. By validity of $\Gamma \vdash \{p_2\}c_2\{p_3\}$ it follows that σ'' satisfies p_3.

- 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 $\mathtt{free}(p_1) \cap \mathtt{writes}(c_2) = \mathtt{free}(p_2) \cap \mathtt{writes}(c_1) = \{\}$ and $(\mathtt{free}(c_1) \cap \mathtt{writes}(c_2)) \cup (\mathtt{writes}(c_1) \cap \mathtt{free}(c_2)) \subseteq \mathtt{owned}(\Gamma)$.
  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 $(s, h_1) \models p_1$, $(s, h_2) \models p_2$. Given the well-formedness assumptions, we also have $(s\backslash\mathtt{writes}(c_2), h_1) \models p_1$ and $(s\backslash\mathtt{writes}(c_1), h_2) \models p_2$.
  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 $\sigma' = \mathbf{abort}$ it would follow by the parallel decomposition lemma that either $(s\backslash\mathtt{writes}(c_2), h_1) \xrightarrow[\Gamma]{\alpha_1} \mathbf{abort}$ or $(s\backslash\mathtt{writes}(c_1), h_2) \xrightarrow[\Gamma]{\alpha_2} \mathbf{abort}$. Neither of these is possible, since they contradict the assumed validity of the premises $\Gamma \vdash \{p_1\}c_1\{q_1\}$ 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 premises, there are heaps $h'_1 \perp h'_2$ such that $h' = h'_1 \cdot h'_2$,

$$(s\backslash\mathtt{writes}(c_2), h_1) \xrightarrow[\Gamma]{\alpha_1} (s'\backslash\mathtt{writes}(c_2), h'_1)$$
$$(s\backslash\mathtt{writes}(c_1), h_2) \xrightarrow[\Gamma]{\alpha_2} (s'\backslash\mathtt{writes}(c_1), h'_2),$$

  and $(s'\backslash\mathtt{writes}(c_2), h'_1) \models q_1$, $(s'\backslash\mathtt{writes}(c_1), h'_2) \models q_2$. Since q_1 does not depend on $\mathtt{writes}(c_2)$ and q_2 does not depend on $\mathtt{writes}(c_1)$ 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.
- REGION
  Suppose we have a well formed and valid instance of the rule's premiss, of the form $\Gamma \vdash \{(p * R) \wedge b\}c\{q * R\}$. We need to show that:

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

  is valid, provided this formula is also well formed. In particular, we assume that $\mathtt{free}(p, q) \cap \mathtt{owned}(\Gamma) = \{\}$ and $\mathtt{free}(p, q) \cap X = \{\}$, and we suppose that $r \notin \mathtt{dom}(\Gamma)$ and R is precise.
  To this end, let (s, h) be a state satisfying p, let α be a trace of $[\![\mathbf{with}\ r\ \mathbf{when}\ b\ \mathbf{do}\ c]\!]$, and assume that $(s, h) \xrightarrow[\Gamma, r(X):R]{\alpha} \sigma'$. We must show that σ' satisfies q. By definition, and ignoring *try* actions, which can be done without loss of generality, α must have the form:

$$acq(r)\ \rho_1\ rel(r)\ \ldots\ acq(r)\ \rho_{n-1}\ rel(r)\ acq(r)\ \rho\ \beta\ rel(r)$$

  where $\rho_1, \ldots, \rho_{n-1} \in [\![b]\!]_{\mathbf{false}}$, $\rho \in [\![b]\!]_{\mathbf{true}}$, and $\beta \in [\![c]\!]$. Each of the ρ_i is a sequence of evaluation actions, having no effect on the state. Since R is precise, the heap portion released at the end of each waiting phase $acq(r)\ \rho_i\ rel(r)$ must be the same as the heap portion acquired at the start of that phase. Hence we have:

$$(s, h) \xrightarrow[\Gamma, r(X):R]{acq(r)\ \rho\ \beta\ rel(r)} \sigma'.$$

  But this requires that there exists a state σ'' such that:

$$(s, h) \xrightarrow[\Gamma, r(X):R]{acq(r)} (s \cdot s_1, h \cdot h_1) \xrightarrow[\Gamma, r(X):R]{\rho\ \beta} \sigma'' \xrightarrow[\Gamma, r(X):R]{rel(r)} \sigma'$$

  for some $s_1 \perp s$, $h_1 \perp h$ such that $\mathtt{dom}(s_1) = X$ and $(s \cdot s_1, h_1) \models R$. Since $\rho \in [\![b]\!]_{\mathbf{true}}$ and b is pure, we must have $(s \cdot s_1, h \cdot h_1) \models b$.
  Since $(s, h) \models p$ and $\mathtt{free}(p) \cap X = \{\}$, it follows that $(s \cdot s_1, h) \models p$. So $(s \cdot s_1, h \cdot h_1) \models (p * R) \wedge b$.
  Since ρ does not change the state we therefore have:

$$(s \cdot s_1, h \cdot h_1) \xrightarrow[\Gamma, r(X):R]{\beta} \sigma'',$$

  and since β cannot contain any acquire or release actions on resource r we also have:

$$(s \cdot s_1, h \cdot h_1) \xrightarrow[\Gamma]{\beta} \sigma''.$$

Validity of the premise$\Gamma \vdash \{(p * R) \wedge b\}c\{q * R\}$ establishes that σ'' is not **abort** and satisfies $q * R$. The final $rel(r)$ action leading from σ'' to σ' must therefore release the unique subheap corresponding to R (retaining the subheap corresponding to q), and remove the store's values for X. Since $\mathtt{free}(q) \cap X = \{\}$ it follows that σ' satisfies q.

Note that the case where β is infinite is handled implicitly in the above proof, and causes no problem.

- RESOURCE

Suppose $\Gamma, r(X) : R \vdash \{p\}c\{q\}$ is well-formed and valid. Thus R is precise, $r \notin \mathtt{dom}(\Gamma)$, $\mathtt{free}(p, q, R) \cap \mathtt{owned}(\Gamma) = \{\}$, $\mathtt{free}(p, q) \cap X = \{\}$, and $X \cap (\mathtt{owned}(\Gamma) \cup \mathtt{free}(\Gamma)) = \{\}$ It is then easy to see that $\Gamma \vdash \{p * R\}$**resource** r **in** $c\{q * R\}$ is well formed. To prove validity of this formula we argue as follows:

Suppose (s, h) satisfies $p * R$, and let α be a trace of **resource** r **in** c such that $(s, h) \xrightarrow[\Gamma]{\alpha} \sigma'$. We must show that σ' satisfies $q * R$.

Choose a trace $\beta \in [\![c]\!]_r$ such that $\beta\backslash r = \alpha$, and heaps $h_1 \perp h_2$ such that $h = h_1 \cdot h_2$, $(s, h_1) \models p$ and $(s, h_2) \models R$. Since $X \cap \mathtt{free}(p) = \{\}$, we also have $(s\backslash X, h_1) \models p$.

By the local resource lemma, if $\sigma' = \textbf{abort}$ we would also have $(s\backslash X, h_1) \xrightarrow[\Gamma, r(X):R]{\beta}$ **abort**, which contradicts our assumption that the premise $\Gamma, r(X) : R \vdash \{p\}c\{q\}$ is valid. So σ' must have the form (s', h'). Again by the local resource lemma and validity of the premise, it follows that there must be heaps $h'_1 \perp h'_2$ such that $h' = h'_1 \cdot h'_2$, $(s', h'_2) \models R$, $(s\backslash X, h_1) \xrightarrow[\Gamma, r(X):R]{\beta} (s'\backslash X, h'_1)$, and $(s'\backslash X, h'_1) \models q$. Since $X \cap \mathtt{free}(q) = \{\}$ we also have $(s', h'_1) \models q$. It then follows that $(s', h') \models q * R$, as required.

- 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' does not occur free in c.

This rule is sound because if $r' \notin \mathtt{res}(c)$ the commands **resource** r **in** c and $\mathtt{resource}$ r' **in** $[r'/r]c$ are semantically equivalent. Since they have the same traces they have the same computations.

- LOOKUP

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

provided i not free in e or e' and the formula is well formed, i.e. $\mathtt{free}(p, e, e') \cap \mathtt{owned}(\Gamma) = \{\}$.

Suppose $(s, h) \models [e'/i]p \wedge e \mapsto e'$. Let $v = |e|s$ and $v' = |e'|s$, so that we have $([s \mid i : v'], h) \models p$ by the substitution theorem, and $h = \{(v, v')\}$. The only traces of $i{:=}[e]$ relevant here have the form $\rho\, [v]{=}v'\, i{:=}v'$, and it is obvious that we have:

$$(s, h) \xrightarrow[\Gamma]{\rho\, [v]=v'\, i:=v'} ([s \mid i : v'], h).$$

Since $i \notin \mathtt{free}(e, e')$ we have $|e|[s \mid i : v'] = |e|s$ and $|e'|[s \mid i : v'] = |e'|s$, so $([s \mid i : v'], h) \models p \wedge e \mapsto e'$, as required.

- UPDATE

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

provided $\mathtt{free}(e) \cap \mathtt{owned}(\Gamma) = \{\}$ and $\mathtt{free}(e') \cap \mathtt{owned}(\Gamma) = \{\}$.

Suppose $(s, h) \models e \mapsto -$. Thus there are values v and v_0 such that $(s, v) \in |e|$ and $h = \{(v, v_0)\}$. Every trace of $[e]{:=}e'$ enabled from (s, h) has the form $\rho\, \rho'\, [v]{:=}v'$, where $(s, v') \in |e'|$. And we have

$$(s, h) \xrightarrow[\Gamma]{\rho\, \rho'\, [v]:=v'} (s, [h \mid v : v']).$$

Clearly $[h \mid v : v'] = \{(v, v')\}$. Since e and e' are pure we also have $(s, \{(v, v')\}) \models e \mapsto e'$, as required.

- LOCAL VARIABLE

Suppose $\Gamma \vdash \{p \wedge i = e\}c\{q\}$ is valid and $i \notin \mathtt{free}(e, p, q)$, $i \notin \mathtt{owned}(\Gamma)$, $\mathtt{free}(e) \cap \mathtt{owned}(\Gamma) = \{\}$, and $\mathtt{free}(p, q) \cap \mathtt{owned}(\Gamma) = \{\}$.

We must show that $\Gamma \vdash \{p\}$**local** $i = e$ **in** $c\{q\}$ is valid. (It is obvious that this formula is also well formed.)

Suppose $(s, h) \models p$. Every trace of **local** $i = e$ **in** c has the form $\rho\,(\alpha\backslash i)$ where $(\rho, v) \in [\![e]\!]$ and $\alpha \in [\![c]\!]_{[i:v]}$. If $(s, h) \xrightarrow[\Gamma]{\rho\,(\alpha\backslash i)} \sigma'$ then $(s, v) \in |e|$ and $(s, h) \xrightarrow[\Gamma]{\alpha\backslash i} \sigma'$. If $\sigma' =$ **abort** then $([s \mid i : v], h) \xrightarrow[\Gamma]{\alpha}$ **abort**, by the Local Variable Lemma. But we already know that the state $([s \mid i : v], h)$ satisfies $p \wedge i = e$ and α is a trace of c, so this would contradict validity of the premise. Hence σ' has the form (s', h'), and the local variable lemma implies that $([s \mid i : v], h) \xrightarrow[\Gamma]{\alpha} ([s' \mid i : v'], h')$ for some v'. Since the premise is valid we have $([s' \mid i : v'], h') \models q$, and since i is not free in q we obtain $(s', h') \models q$, as required.

- EXPANSION

  Let $\Gamma \vdash \{p\}c\{q\}$ be valid and well formed, $\texttt{free}(c) \cap \texttt{owned}(\Gamma') = \{\}$, and $\texttt{writes}(c) \cap \texttt{free}(\Gamma') = \{\}$. Suppose that $\Gamma, \Gamma' \vdash \{p\}c\{q\}$ is well formed. We need to prove that this formula is valid.

  By well-formedness Γ and Γ' are mutually disjoint. By assumption, c does not read or write any identifier protected by Γ', and c does not write to any identifier mentioned in the resource invariants of Γ'. Hence, if $\alpha \in [\![c]\!]$ and $\sigma \xrightarrow[\Gamma,\Gamma']{\alpha} \sigma'$, then $\sigma \xrightarrow[\Gamma]{\alpha} \sigma'$. The result follows easily.

- CONTRACTION

  Let $\Gamma, \Gamma' \vdash \{p\}c\{q\}$ be well formed and valid. In particular Γ and Γ' are mutually disjoint. Suppose that $\texttt{res}(c) \subseteq \texttt{dom}(\Gamma)$. We must show that $\Gamma \vdash \{p\}c\{q\}$ is valid. Let $\alpha \in [\![c]\!]$, $\sigma \models p$, and $\sigma \xrightarrow[\Gamma]{\alpha} \sigma'$. Since α cannot contain any actions involving the resources of Γ', we also get $\sigma \xrightarrow[\Gamma,\Gamma']{\alpha} \sigma'$. So by validity of $\Gamma, \Gamma' \vdash \{p\}c\{q\}$ it follows that $\sigma' \models q$, as required.

- EXISTENTIAL

  Suppose $\Gamma \vdash \{p\}c\{q\}$ is valid and well formed, $i \notin \texttt{free}(\Gamma) \cup \texttt{owned}(\Gamma)$, and $i \notin \texttt{free}(c)$. We must show that $\Gamma \vdash \{\exists i.p\}c\{\exists i.q\}$ is valid.

  Assume that $(s, h) \models \exists i.p$, so that $([s \mid i : v_0], h) \models p$ for some value v_0. Let α be a trace of c. Since i is not free in c we can use the agreement theorem to deduce that $(s, h) \xrightarrow[\Gamma]{\alpha}$ **abort** if and only if, for all v, $([s \mid i : v], h) \xrightarrow[\Gamma]{\alpha}$ **abort**. Similarly, $(s, h) \xrightarrow[\Gamma]{\alpha} (s', h')$ if and only if, for all v, $([s \mid i : v], h) \xrightarrow[\Gamma]{\alpha} ([s' \mid i : v], h')$. Since $\Gamma \vdash \{p\}c\{q\}$ is valid and $([s \mid i : v_0], h) \models p$ it follows that for all (s', h') such that: $(s, h) \xrightarrow[\Gamma]{\alpha} (s', h')$ we have $([s' \mid i : v_0], h') \models q$, and hence $(s', h') \models \exists i.q$.

- AUXILIARY

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

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

  Recall that a set X is auxiliary for c if every free occurrence in c of an identifier from X is in an assignment whose target belongs to X. The command $c\backslash X$ is obtained from c by deleting all assignments to identifiers in X.

  Suppose X is auxiliary for c, $\Gamma \vdash \{p\}c\{q\}$ is well formed and valid, and $X \cap \texttt{free}(p, q) = \{\}$. We must show that $\Gamma \vdash \{p\}c\backslash X\{q\}$ is valid. So let $\beta \in [\![c\backslash X]\!]$, and suppose that $(s, h) \models p$ and $(s, h) \xrightarrow[\Gamma]{\beta} \sigma'$. We need to show that $\sigma' \models q$.

  Since X is auxiliary for c, if $(s, h) \xrightarrow[\Gamma]{\beta}$ **abort**, there is a trace α of c and a store $\hat{s}$ such that $\hat{s}$ agrees with s except on X and $(\hat{s}, h) \xrightarrow[\Gamma]{\alpha}$ **abort**. But since $X \cap \texttt{free}(p) = \{\}$ we also have $(\hat{s}, h) \models p$, so this contradicts the validity of $\Gamma \vdash \{p\}c\{q\}$. Hence σ' must be a proper state of the form (s', h'). Similarly, since we now have $(s, h) \xrightarrow[\Gamma]{\beta} (s', h')$, and X is auxiliary, there are stores $\hat{s}$ and $\hat{s'}$ that agree with s and s' (respectively) on X, and a trace α of c, such that $(\hat{s}, h) \xrightarrow[\Gamma]{\alpha} (\hat{s'}, h')$. Again we have $(\hat{s}, h) \models p$, so by validity of $\Gamma \vdash \{p\}c\{q\}$ it follows that $(\hat{s'}, h') \models q$. Since $X \cap \texttt{free}(q) = \{\}$ we deduce that $(s', h') \models q$, as required.

- FRAME

  Assume $\Gamma \vdash \{p\}c\{q\}$ is well formed and valid, $\texttt{free}(I) \cap \texttt{writes}(c) = \{\}$ and $\texttt{free}(I) \cap \texttt{owned}(\Gamma) = \{\}$. We must show that $\Gamma \vdash \{p * I\}c\{q * I\}$ is valid.

  Let $\alpha \in [\![c]\!]$ and let (s, h) be a state satisfying $p * I$. Let $h = h_1 \cdot h_2$ with $h_1 \perp h_2$, $(s, h_1) \models p$, $(s, h_2) \models I$. Suppose $(s, h) \xrightarrow[\Gamma]{\alpha} \sigma'$. By the Frame Property and validity of $\Gamma \vdash \{p\}c\{q\}$, there is a state (s', h_1') such that $h_1' \perp h_2$, $(s, h_1) \xrightarrow[\Gamma]{\alpha} (s', h_1')$, $(s', h_1') \models q$, and σ' has the form $(s', h_1' \cdot h_2)$. Since α does not write to $\texttt{free}(I)$, we also have $(s', h_2) \models I$ by the agreement theorem. Hence $\sigma' \models q * I$, as required.

Thus we have established soundness of the inference rules with respect to a "local" enabling relation that keeps track of protection lists and resource invariants. It remains to connect this result with the global enabling the relation introduced earlier. In fact, we can now show that validity, defined on the basis of local computations, implies the weaker notion of validity that was discussed earlier.

13. Provability implies no races

As we mentioned earlier, the global state can be regarded as combining the local states of each process and the state portions determined by the available resources. A global state (s, h, A) corresponds in the obvious manner to the local state $(s{\downarrow}A, h, A)$, where we define $s{\downarrow}A = s\backslash\texttt{owned}(\Gamma\backslash A)$.

Lemma 26 (*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 $(s, h, A) \xRightarrow{\lambda}$ **abort** then $(s{\downarrow}A, h_1, A) \xrightarrow[\Gamma]{\lambda}$ **abort**.*
- *If $(s, h, A) \xRightarrow{\lambda} (s', h', A')$ 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 $(s{\downarrow}A, h_1, A) \xrightarrow[\Gamma]{\lambda} (s'{\downarrow}A', h_1', A')$.*

Proof. Case analysis using the definitions of the two enabling relations.

- For δ, $i{=}v$ and **abort** the results hold trivially.
- For $i{:=}v$ note that the side condition justifying a successful local step is sufficient to ensure that the change to i has no effect on the relevant invariants.
- For $[l] = v'$, $[l]{:=}v'$, $alloc(l, L)$, $disp(l)$ the proof is straightforward.
- For $acq(r)$ let $r(X) : R \in \Gamma$. Note that if $(s, h, A) \xRightarrow{acq(r)} (s, h, A \cup \{r\})$ and $r \notin A$, then we can split h_2 into disjoint pieces h_r, h_3 such that $(s, h_r) \models R$ and $(s, h_3) \models \texttt{inv}(\Gamma\backslash(A \cup \{r\}))$. Hence we also have:

$$(s{\downarrow}A, h_1, A) \xrightarrow[\Gamma]{acq(r)} ((s{\downarrow}A) \cdot (s\lceil X), h_1 \cdot h_r, A \cup \{r\}).$$

  Clearly $(s{\downarrow}A) \cdot (s\lceil X) = s{\downarrow}(A \cup \{r\})$ and $(h_1 \cdot h_r) \cdot h_3 = h$, and the result follows.
- For $rel(r)$ the proof is similar.

The connection property obviously generalizes to a finite sequence of transitions, i.e. to a finite trace α instead of a single action λ. This is easy to prove by induction on the length of α using the above lemma as the base case. By putting $A = A' = \{\}$ we obtain the following corollary:

Corollary 27. *Let α be a trace. Let (s, h) be a state, $h = h_1 \cdot h_2$, and $(s, h_2) \models \texttt{inv}(\Gamma)$.*

- *If $(s, h) \xRightarrow{\alpha}$ **abort** then $(s\backslash\texttt{owned}(\Gamma), h_1) \xrightarrow[\Gamma]{\alpha}$ **abort**.*
- *If $(s, h) \xRightarrow{\alpha} (s', h')$ 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 $(s\backslash\texttt{owned}(\Gamma), h_1) \xrightarrow[\Gamma]{\alpha} (s'\backslash\texttt{owned}(\Gamma), h_1')$.*

Theorem 28 (*Valid Implies Race-free*). *If $\Gamma \vdash \{p\}c\{q\}$ is valid and well formed, then c is race-free from every state satisfying $p * \texttt{inv}(\Gamma)$. In fact, for all states σ, σ' and all traces $\alpha \in [\![c]\!]$, if $\sigma \models p * \texttt{inv}(\Gamma)$ and $\sigma \xRightarrow{\alpha} \sigma'$ then $\sigma' \models q * \texttt{inv}(\Gamma)$.*

Proof. If $\Gamma \vdash \{p\}c\{q\}$ is well formed then $\texttt{free}(p, q) \cap \texttt{owned}(\Gamma) = \{\}$. Let α be a trace of c. Suppose (s, h) satisfies $p * \texttt{inv}(\Gamma)$, with $h_1 \perp h_2$, $h = h_1 \cdot h_2$ and $(s, h_1) \models p$, and $(s, h_2) \models \texttt{inv}(\Gamma)$. Note that $s{\downarrow}\{\} = s\backslash\texttt{owned}(\Gamma)$ and that the stores s and $s\backslash\texttt{owned}(\Gamma)$ agree on $\texttt{free}(p, q)$. Hence we also have $(s\backslash\texttt{owned}(\Gamma), h_1) \models p$.

By the connection corollary above, and validity of $\Gamma \vdash \{p\}c\{q\}$, we cannot have $(s, h) \xRightarrow{\alpha}$ **abort**, i.e. every computation of α from (s, h) is error-free.
Similarly, for every computation of form $(s, h) \xRightarrow{\alpha} (s', h')$ there is a corresponding local computation $(s\backslash\texttt{owned}(\Gamma), h_1) \xrightarrow[\Gamma]{\alpha} (s'\backslash\texttt{owned}(\Gamma), h_1')$, and a subset h_2' of h' such that $(s', h_2') \models \texttt{inv}(\Gamma)$, $h' = h_1' \cdot h_2'$. By validity of $\Gamma \vdash \{p\}c\{q\}$, it follows that $(s'\backslash\texttt{owned}(\Gamma), h_1') \models q$. Since s' and $s'\backslash\texttt{owned}(\Gamma)$ agree on $\texttt{free}(q)$ we also

have $(s', h'_1) \models q$. Hence $(s', h') \models q * \mathtt{inv}(\Gamma)$, as required.
Since the transition relation handles races by aborting, this is enough to ensure absence of races when the program is run from an initial (global) state satisfying $p * \mathtt{inv}(\Gamma)$,

Corollary 29. *If* $\vdash \{p\}c\{q\}$ *is provable then for all* σ, σ' *such that* $\sigma \models p$ *and all traces* $\alpha \in [\![c]\!]$*, if* $\sigma \overset{\alpha}{\Longrightarrow} \sigma'$ *then* $\sigma' \models q$*. Hence* c *is race-free from every state satisfying* p*.*

Using the above result we now have another way to demonstrate race-freedom for the example programs discussed earlier. In each case the logic confirms our previous semantic analysis.

For instance, let Γ be the following resource context:

$$\mathit{buf}(c, \mathit{full}) : (\mathit{full} = 1 \wedge z \mapsto -) \vee (\mathit{full} = 0 \wedge \mathbf{emp})$$

We showed that the formula

$$\begin{array}{lll} \Gamma & \vdash & \{\mathbf{emp}\} \\ & & (x{:=}\mathbf{cons}(1);\ \mathrm{PUT}(x)) \| (\mathrm{GET}(y);\ \mathbf{dispose}(y)) \\ & & \{\mathbf{emp}\} \end{array}$$

is provable. Hence the formula is also valid. By the above result, it follows that the program is race-free from any state satisfying

$$(\mathit{full} = 1 \wedge z \mapsto -) \vee (\mathit{full} = 0 \wedge \mathbf{emp}).$$

14. Conclusions

We have given a trace-based denotational semantics for a language of parallel programs operating on shared mutable data. The semantics employs a form of fair parallel composition that detects, and views as catastrophic, the potential for race conditions. The semantics supports compositional reasoning about partial correctness and the absence of races, and we used the semantics as the basis for a proof of soundness for resource-sensitive concurrent separation logic. In doing this we formulated a novel "local" semantics that permits reasoning about the dynamic transfer of heap ownership that may occur during program execution.

It is already known that concurrent separation logic can be used to reason about a wide range of examples, including parallel mergesort and a simple memory allocator [31,30]. In view of the newness of the logic and the freshness of the methodology there is still room for further exploration of the benefits, power and utility of this framework. We plan to tackle a series of challenging examples from the literature, with the expectation that *concurrent separation logic* will facilitate more streamlined proofs. The semantic framework should help to formalize and better understand intuitive concepts such as transfer of ownership, and help to generalize such notions as appropriate.

We have assumed so far that each resource invariant is *precise*, so that a resource context defines what might be called a *precise ownership policy*: when a program acquires or releases a resource there is a uniquely determined portion of the heap whose ownership can be deemed to transfer. This has not seemed to be a major limitation so far, and a methodology based on precision seems very natural. Moreover this limitation is sufficient to ensure soundness. But the question remains if there is a more general class of formulas, suitable as resource invariants, for which the rules remain sound (possibly with the additional imposition of further side conditions restricting the kind of pre- and post-condition allowed in rules dealing with resources).

One cannot simply drop the precision constraint completely and allow arbitrary resource invariants. This is shown by the following problematic formula, due to John Reynolds:

$$r : \mathbf{true} \vdash \{\mathbf{emp} \vee \mathbf{one}\}\mathbf{with}\ r\ \mathbf{do}\ \mathbf{skip}\{\mathbf{emp}\},$$

where **one** is a separation logic formula that holds only in heaps of size one. This formula is derivable if we allow the REGION rule as before but without insisting that the resource invariant be precise. This formula is not *valid*, according to our notion of validity, adapted to the imprecise setting in the obvious way. Nor is there a reasonable variation on the notion of validity that would make this formula valid and still accurately reflect the program's computational behaviour: when executed in a heap of size 1 the program obviously has a computation in which it ends in the same heap, which certainly does not satisfy the specified post-condition.

Our semantic model can be used to prove that the methodology is still sound under a *parsimonious* ownership policy, characterized as follows: when a process acquires a resource it claims ownership of the *smallest heap* portion that suffices, and when releasing the resource cedes ownership of the minimal relevant heap portion. Technically this involves the use of *supported* resource invariants with compensatory adjustments in the rules for regions and resource declarations to require that their pre- and post-conditions be *intuitionistic*. A supported formula [41,42] has the characteristic property that in any state there is at most one *minimal* sub-heap for which the formula holds. If an intuitionistic formula [41,42] holds in a sub-heap of the state then it holds in all larger sub-heaps. With these adjustments, the inference rules would then be:

- Region

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

  if R supported, p and q intuitionistic
- Resource

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

  if R supported, p and q intuitionistic

The key lemmas used in the soundness proof, notably the parallel decomposition lemma and the local resource lemma, can be adapted to this setting, and the soundness proof goes through as before, with appropriate adjustments in the case analysis for these two rules.

This seems an intuitively natural generalization of the approach using precise ownership policies. The use of intuitionistic and supported formulas suggests, by analogy with results from sequential separation logic, that this kind of reasoning may be useful for concurrent programs operating on data structures that involve structure sharing, such as overlapping linked lists [33,44,42]. Another example in which such formulas arise naturally is parallel mergesort. It would also be interesting to see if any other natural ownership policies are useful and can be fit into this framework.

The trace semantics was designed to detect races. We did not include a pair of concurrent reads as a race, since this kind of passive interaction is usually regarded as benign. However, the use of separating conjunction in the Parallel rule requires that the processes in a provable program operate on disjoint portions of the heap, even if part of the heap is treated as "read-only" by all processes and could safely be shared without racing. For example, there is no way to prove the obviously valid formula

$$\vdash \{z \mapsto 1\}x{:=}[z] \| y{:=}[z]\{x = y = 1 \wedge z \mapsto 1\},$$

since the logic requires both processes to "need" to own the heap cell denoted by z, separately. Nevertheless, the trace semantics handles this issue (and this example) correctly, so here is a place where the semantics is ahead of the logic.

We believe that it may be possible to solve this passivity problem by introducing a further class of formulas of the form $\Gamma \vdash_R \{p\}c\{q\}$, decorated with a separation logic formula R describing a "read-only" part of the heap, together with suitably designed inference rules. It is not yet clear if this approach can be pushed through completely, or if it is necessary to restrict the kind of formula allowed as read-only annotation, perhaps to the class of precise formulas. Another possibility might be to try to adapt Boyland's ideas on *fractional permissions* [6], perhaps by designing a semantics in which partial permissions are attached to resource actions and managed in an appropriate manner upon resource acquisition and release, instead of all-or-nothing transfer. The trace semantic framework should help to provide a rigorous test-bed for checking the soundness of such proposed extensions.

Our focus so far has been limited to partial correctness. It should also be possible to develop resource-sensitive inference rules for *total correctness*, leading to a logic in which every provable program is both race-free and deadlock-free. One natural idea is to take a more intentional view of the structure of resource contexts, so that a context designates a *sequence* rather than a *set* of resources, conveying an *acquisition order* for resource names. One can then re-phrase the side conditions in the inference rules so that in every provable program all processes acquire resources in the order in which they occur in the list Γ. When all processes obey the same acquisition order we will be able to rule out "cyclic" deadlocks. For example, the program

$$(\textbf{with } r_1 \textbf{ do with } r_2 \textbf{ do } x{:=}1) \| (\textbf{with } r_2 \textbf{ do with } r_1 \textbf{ do } y{:=}1)$$

can either deadlock or terminate successfully, depending on the scheduling. However, there is no resource context Γ for which both c_1 and c_2 respect the precedence order, and hence $c_1 \| c_2$ has no provable formulas. This idea, that precedence rules can prevent deadlock, appears to be a well known folk theorem.

The trace semantics presented here makes distinctions between programs based on the order in which they perform actions, and hence fails to be fully abstract for partial correctness. Moreover our repertoire of actions assumes that reads and writes to individual variables and heap addresses are executable indivisibly. But the partial correctness properties of a race-free program should not depend on whether assignments, or reads and writes to a variable, are atomic [43]. For example, the trace sets denoted by the programs $x{:=}1;\, y{:=}1$ and $y{:=}1;\, x{:=}1$ are distinct, but the two programs clearly satisfy the same partial correctness formulas (and the same race-freedom properties) in all program contexts. Of course, the fact that our semantics is compositional implies the usual half of full abstraction: if two programs have the same trace set then they satisfy the same partial correctness formulas in all contexts. It would be interesting to devise a semantic model more abstract than ours, abstracting away from granularity, in which (for example) the above programs would be given the same meaning. One possibility is to work with a form of "big step" transition trace in which the actions between successive resource actions are conflated (by a form of "mumbling") into one big state transformation [16,13]. Such a semantics would ascribe identical meaning to all pairs of commands which are indistinguishable in this sense. John Reynolds has recently proposed an alternative semantics with similar aims but different structure. However appealing this prospect is, we leave this as a topic for future research.

Turning the above argument on its head, we might equally well argue that the trace semantics makes the right kinds of distinctions between programs to support reasoning about safety and liveness properties, since these properties depend on the sequences of states through which a program may pass during a fair execution. Using temporal logics an enormous variety of safety and liveness properties can be expressed [39]. We plan to explore a combination of separation logic with the modal operators of temporal logic to obtain a *temporal separation logic*. As a first step in this direction it may be possible to adapt *rely/guarantee* methodology [26,28, 27] to our setting. Indeed our description of ownership transfer policy clearly has both "rely" and "guarantee" aspects.

The idea of using traces of some kind to model processes is widespread and attests to the utility of the general concept, but the word "trace" means different things to different people. Hoare proposed a form of action trace in which each action represents a potential to send or receive a value on a channel, and used such traces in an early model of CSP which ignored deadlock (and ignored state). The failures model of CSP augmented such traces with "refusal sets" to permit proper treatment of deadlock. The failures/divergences model further incorporated "divergence traces" to permit a limited form of liveness analysis. In retrospect these models can be seen as early pre-cursors of the action trace framework that we currently advocate: our notion of action trace encompasses both state (including mutable state with embedded pointers) and communication. Transition traces are descended from the foundational work of David Park, who used similar traces to model shared-variable programs. The main difference is that in Park's semantics each step represents the effect (again on the global shared state) of a single atomic action, so that Park's model failed to be fully abstract, for instance distinguishing unnecessarily between **skip** and **skip**; **skip**. A similar motivation was behind our built-in assumption that δ is a unit for concatenation of actions.

Acknowledgements

Throughout the development of this work, I have had the distinct pleasure of being able to interact on a regular basis with John Reynolds. We have had numerous discussions, ranging from deep technical concerns to more high-level philosophical issues. I am happy to acknowledge John's influence and guidance. John has prompted me in more ways than I can remember: to justify my approach, to find simpler ways to explain concepts, to deal with counterexamples, to seek maximum generality. I am especially pleased that this paper will appear as part of the Reynolds Festschrift. John has been, and continues to be, a shining example to us all and a source of inspiration.

Many thanks to Peter O'Hearn for proposing the methodology for resource-based reasoning about concurrent programs on which this work is based. It was in response to Peter's ideas that this work emerged, as my attempt to underpin the ideas formally. I have benefitted immensely from discussions with Peter, from start to finish. Thanks also to Josh Berdine, for a series of detailed discussions during his visit to CMU; his insights have led to several improvements in the structure of this paper.

Appendix

We include here some technical lemmas leading to the proof of the parallel decomposition lemma, which was used crucially in the soundness proof for the PARALLEL rule.

Lemma 1 (*Agreement Property for Traces*). *For all resource contexts Γ and all traces α:*

1. *If s_1 agrees with s_2 on Y and $Y \supseteq \mathtt{free}(\alpha, \Gamma)$, then:*
   - *If $(s_1, h, A) \xrightarrow[\Gamma]{\alpha}$* **abort**, *then $(s_2, h, A) \xrightarrow[\Gamma]{\alpha}$* **abort**.
   - *If $(s_1, h, A) \xrightarrow[\Gamma]{\alpha} (s_1', h', A')$ then there is a store s_2' such that $(s_2, h, A) \xrightarrow[\Gamma]{\alpha} (s_2', h', A')$ and s_1' agrees with s_2' on Y.*
2. *If $(s, h, A) \xrightarrow[\Gamma]{\alpha} (s', h', A')$ then $s\backslash\mathtt{owned}(\Gamma)$ agrees with $s'\backslash\mathtt{owned}(\Gamma)$ except on $\mathtt{writes}(\alpha)$.*

Proof of (1). By induction on the length of α.
The base case (when α is a single action λ) is a straightforward case analysis using the definition of the transition relations $\xrightarrow[\Gamma]{\lambda}$, and the inductive step is easy. Here are the base cases for resource actions:

- For λ of the form $acq(r)$, suppose that s_1 and s_2 agree on $Y \supseteq \mathtt{free}(\Gamma)$.
  - If $(s_1, h, A) \xrightarrow[\Gamma]{acq(r)}$ **abort** then $r \in A$, so we also have $(s_2, h, A) \xrightarrow[\Gamma]{acq(r)}$ **abort**.
  - If $(s_1, h, A) \xrightarrow[\Gamma]{acq(r)} (s_1 \cdot s', h \cdot h', A \cup \{r\})$ where $\mathtt{dom}(s') = X$, $h \perp h'$, and $(s_1 \cdot s', h') \models R$, since s_1 and s_2 agree on $\mathtt{free}(R)$ by assumption, it follows that we also have $(s_2 \cdot s', h') \models R$. Hence: $(s_2, h, A) \xrightarrow[\Gamma]{acq(r)} (s_2 \cdot s', h \cdot h', A \cup \{r\})$. It follows easily that $s_1 \cdot s'$ and $s_2 \cdot s'$ agree on Y, as required.
- For λ of form $rel(r)$, suppose that s_1 and s_2 agree on $Y \supseteq \mathtt{free}(\Gamma)$.
  - If $(s_1, h, A) \xrightarrow[\Gamma]{rel(r)}$ **abort** then either $r \notin A$, or $r(X) : R \in \Gamma$ and for all $h' \subseteq h$ we have $(s_1, h') \models \neg R$. In the first case it is obvious that we also have $(s_2, h, A) \xrightarrow[\Gamma]{rel(r)}$ **abort**. Otherwise $r \in A$, and since s_1 and s_2 agree on $\mathtt{free}(R)$, we also have $(s_2, h') \models \neg R$ for all $h' \subseteq h$, so again $(s_2, h, A) \xrightarrow[\Gamma]{rel(r)}$ **abort**.
  - Otherwise assume that: $(s_1, h, A) \xrightarrow[\Gamma]{rel(r)} (s_1\backslash X, h - h', A - \{r\})$ where $r \in A$, $r(X) : R \in \Gamma$, $h' \subseteq h$ and $(s_1, h') \models R$. Since s_1 and s_2 agree on $\mathtt{free}(R)$ we also have $(s_2, h') \models R$, so that $(s_2, h, A) \xrightarrow[\Gamma]{rel(r)} (s_2\backslash X, h - h', A - \{r\})$. Clearly $s_1\backslash X$ and $s_2\backslash X$ agree on Y, as required.

Proof of (2). Again by induction on the length of α.
The base case uses the definition of $\xrightarrow[\Gamma]{\lambda}$ and the inductive step is easy. Here are the base cases for resource actions.

- If $(s, h, A) \xrightarrow[\Gamma]{acq(r)} (s \cdot s', h \cdot h', A \cup \{r\})$ we have $\mathtt{writes}(acq(r)) = \{\}$, and $\mathtt{dom}(s') \subseteq \mathtt{owned}(\Gamma)$, so $(s \cdot s')\backslash\mathtt{owned}(\Gamma) = s\backslash\mathtt{owned}(\Gamma)$, as required.
- If $(s, h, A) \xrightarrow[\Gamma]{rel(r)} (s\backslash X, h - h', A - \{r\})$ we have $X \subseteq \mathtt{owned}(\Gamma)$ and $(s\backslash X)\backslash\mathtt{owned}(\Gamma) = s\backslash\mathtt{owned}(\Gamma)$. Since $\mathtt{writes}(rel(r)) = \{\}$ the result holds.

Lemma 2 (*Frame Property for Actions*). *Suppose $h_1 \perp h_2$, $A_1 \perp A_2$, and $h = h_1 \cdot h_2$, $A = A_1 \cdot A_2$. Assume that $(A_1, A_2) \xrightarrow{\lambda} (A_1', A_2)$.*

- *If $(s, h, A) \xrightarrow[\Gamma]{\lambda}$* **abort**, *then $(s\backslash\mathtt{owned}(\Gamma\lceil A_2), h_1, A_1) \xrightarrow[\Gamma]{\lambda}$* **abort**.
- *If $(s, h, A) \xrightarrow[\Gamma]{\lambda} (s', h', A')$ then either $(s\backslash\mathtt{owned}(\Gamma\lceil A_2), h_1, A_1) \xrightarrow[\Gamma]{\lambda}$* **abort**, *or there is a heap h_1' such that $h_1' \perp h_2$, $h' = h_1' \cdot h_2$, $A' = A_1' \cdot A_2$, and*

$$(s\backslash\mathtt{owned}(\Gamma\lceil A_2), h_1, A_1) \xrightarrow[\Gamma]{\lambda} (s'\backslash\mathtt{owned}(\Gamma\lceil A_2), h_1', A_1').$$

Proof. Case analysis for each form of action. Most cases are straighforward. Here are the cases for resource actions. Let $s{\downarrow}A_1 = s\backslash\mathtt{owned}(\Gamma\lceil A_2)$.

- For $\lambda = acq(r)$, since $(A_1, A_2) \xrightarrow{acq(r)} (A_1', A_2)$ we have $r \not\in A_1 \cdot A_2$ and $A_1' = A_1 \cup \{r\}$.
  - Obviously we also have $r \not\in A_1$, so the **abort** case is vacuous.
  - If $(s, h, A) \xrightarrow[\Gamma]{acq(r)} (s \cdot s'', h \cdot h'', A \cup \{r\})$ where $r(X) : R \in \Gamma$, $s'' \perp s$, $\mathtt{dom}(s'') = X$, $h'' \perp h$, and $(s \cdot s'', h'') \models R$, we argue as follows.
    Since $r \not\in A$, we have $\mathtt{free}(R) \cap \mathtt{owned}(\Gamma\lceil A) = \{\}$. Hence the stores $s \cdot s''$ and $(s{\downarrow}A_1) \cdot s''$ agree on $\mathtt{free}(R)$, so that we also get

    $$((s{\downarrow}A_1) \cdot s'', h'') \models R.$$

    It follows that $(s{\downarrow}A_1, h_1, A_1) \xrightarrow[\Gamma]{acq(r)} ((s{\downarrow}A_1) \cdot s'', h_1 \cdot h'', A_1 \cup \{r\})$. Clearly $(h_1 \cdot h'') \perp h_2$ and $(h_1 \cdot h'') \cdot h_2 = h \cdot h''$. By the disjointness properties of Γ, $(s \cdot s''){\downarrow}(A_1 \cup \{r\}) = (s{\downarrow}A_1) \cdot s''$. The result thus holds for this case.
- For $\lambda = rel(r)$ since $(A_1, A_2) \xrightarrow[\Gamma]{rel(r)} (A_1', A_2)$ we have $r \in A_1$ and $A_1' = A_1 - \{r\}$. Hence $r \in A$. Let $r(X) : R \in \Gamma$.
  - If $(s, h, A) \xrightarrow[\Gamma]{rel(r)}$ **abort** then (since $r \in A$) there is no subset h' of h such that $(s, h') \models R$. Since $r \not\in A_2$ the stores s and $s{\downarrow}A_1$ agree on $\mathtt{free}(R)$. It follows that there is no subset h' of h_1 such that $(s{\downarrow}A_1, h') \models R$, and hence that $(s{\downarrow}A_1, h_1, A_1) \xrightarrow[\Gamma]{rel(r)}$ **abort**.
  - On the other hand, if

    $$(s, h, A) \xrightarrow[\Gamma]{rel(r)} (s\backslash X, h - h', A - \{r\}),$$

    where $(s, h') \models R$ and $h' \subseteq h$, we argue as follows.
    Recall that $r \in A_1$. By the disjointness properties of Γ and the assumption that $r \not\in A_2$, the stores $s{\downarrow}A_1$ and s agree on $\mathtt{free}(R)$. Hence $(s{\downarrow}A_1, h') \models R$. If h' is not also a subset of h_1 we clearly get

    $$(s{\downarrow}A_1, h_1, A_1) \xrightarrow[\Gamma]{rel(r)} \textbf{abort}.$$

    Otherwise, $h' \subseteq h_1$ and we therefore get

    $$(s{\downarrow}A_1, h_1, A_1) \xrightarrow[\Gamma]{rel(r)} ((s{\downarrow}A_1)\backslash X, h_1 - h', A_1 - \{r\}).$$

    Since $h_1 \perp h_2$ and $h = h_1 \cdot h_2$ we also have $h - h' = (h_1 - h') \cdot h_2$, $(h_1 - h') \perp h_2$, and $A' - \{r\} = (A_1 - \{r\}) \cdot A_2$. The result follows, since $(s\backslash X){\downarrow}(A_1 - \{r\}) = (s{\downarrow}A_1)\backslash X$.

The generalization to traces is an obvious induction.

Lemma 3 (*Frame Property for Traces*). *Suppose* $h_1 \perp h_2$, $A_1 \perp A_2$, *and* $h = h_1 \cdot h_2$, $A = A_1 \cdot A_2$.
Assume that: $(A_1, A_2) \xrightarrow{\alpha} (A_1', A_2)$.

- *If* $(s, h, A) \xrightarrow[\Gamma]{\alpha}$ **abort**, *then* $(s\backslash\mathtt{owned}(\Gamma\lceil A_2), h_1, A_1) \xrightarrow[\Gamma]{\alpha}$ **abort**.
- *If* $(s, h, A) \xrightarrow[\Gamma]{\alpha} (s', h', A')$ *then either* $(s\backslash\mathtt{owned}(\Gamma\lceil A_2), h_1, A_1) \xrightarrow[\Gamma]{\alpha}$ **abort**, *or there is a heap* h_1' *such that* $h_1' \perp h_2$, $h' = h_1' \cdot h_2$, $A' = A_1' \cdot A_2$, *and*

  $$(s\backslash\mathtt{owned}(\Gamma\lceil A_2), h_1, A_1) \xrightarrow[\Gamma]{\alpha} (s'\backslash\mathtt{owned}(\Gamma\lceil A_2), h_1', A_1').$$

In the statement of the following lemma let $\mathtt{free}(\alpha_1)$, $\mathtt{writes}(\alpha_2)$ and so on refer to the set of free identifiers, and the set of free write identifiers, respectively, of a trace. (We do not include the heap cells read or written by the trace, since the lemma concerns the effect of the trace on the identifiers protected by Γ.)

Lemma 4 (*Parallel Decomposition for Traces*). *Assume* $(\mathtt{free}(\alpha_1) \cap \mathtt{writes}(\alpha_2)) \cup (\mathtt{writes}(\alpha_1) \cap \mathtt{free}(\alpha_2)) \subseteq \mathtt{owned}(\Gamma)$ *and* $\alpha \in \alpha_1\ {}_{A_1}\|_{A_2}\ \alpha_2$. *Suppose* $h_1 \perp h_2$, $A_1 \perp A_2$, *and* $h = h_1 \cdot h_2$, $A = A_1 \cdot A_2$.
Let $s_1 = s\backslash\mathtt{writes}(\alpha_2)\backslash\mathtt{owned}(\Gamma) \cup s\lceil\mathtt{owned}(\Gamma\lceil A_1)$, *and* $s_2 = s\backslash\mathtt{writes}(\alpha_1)\backslash\mathtt{owned}(\Gamma) \cup s\lceil\mathtt{owned}(\Gamma\lceil A_2)$.

- *If* $(s, h, A) \xrightarrow[\Gamma]{\alpha}$ **abort** *then either* $(s_1, h_1, A_1) \xrightarrow[\Gamma]{\alpha_1}$ **abort**, *or* $(s_2, h_2, A_2) \xrightarrow[\Gamma]{\alpha_2}$ **abort**.

- *If* $(s, h, A) \xrightarrow[\Gamma]{\alpha} (s', h', A')$ *then either* $(s_1, h_1, A_1) \xrightarrow[\Gamma]{\alpha_1}$ **abort**, *or* $(s_2, h_2, A_2) \xrightarrow[\Gamma]{\alpha_2}$ **abort**, *or there are disjoint heaps* h'_1, h'_2, *and disjoint resource sets* A'_1, A'_2, *such that* $h' = h'_1 \cdot h'_2$, $A' = A'_1 \cdot A'_2$, $(s_1, h_1, A_1) \xrightarrow[\Gamma]{\alpha_1} (s'_1, h'_1, A'_1)$, *and* $(s_2, h_2, A_2) \xrightarrow[\Gamma]{\alpha_2} (s'_2, h'_2, A'_2)$, *where* $s'_1 = (s' \backslash \mathtt{writes}(\alpha_2) \backslash \mathtt{owned}(\Gamma)) \cup (s' \lceil \mathtt{owned}(\Gamma \lceil A'_1))$ *and* $s'_2 = (s' \backslash \mathtt{writes}(\alpha_1) \backslash \mathtt{owned}(\Gamma)) \cup (s' \lceil \mathtt{owned}(\Gamma \lceil A'_2))$.

Proof. By induction on the lengths of α_1 and α_2.

- Base case: when one of the traces is empty.
  Without loss of generality, assume that $\alpha_2 = \epsilon$ and $\alpha \in \alpha_1 \;{}_{A_1}\|_{A_2}\; \epsilon$, so that $(A_1, A_2) \xrightarrow[\Gamma]{\alpha} (A'_1, A_2)$ for some $A'_1 \perp A_2$, and $\alpha = \alpha_1$. Note that $s_1 = s \downarrow A_1$ and $s_2 = s \backslash \mathtt{writes}(\alpha) \backslash \mathtt{owned}(\Gamma) \cup s \lceil \mathtt{owned}(\Gamma \lceil A_2)$.
  - If $(s, h, A) \xrightarrow[\Gamma]{\alpha}$ **abort** then $(s \downarrow A_1, h_1, A_1) \xrightarrow[\Gamma]{\alpha}$ **abort** by the Frame Property. Hence $(s_1, h_1, A_1) \xrightarrow[\Gamma]{\alpha_1}$ **abort**, as required.
    (The other base case, when α_1 is empty, is symmetric; we would get $(s_2, h_2, A_2) \xrightarrow[\Gamma]{\alpha_2}$ **abort** here instead.)
  - If $(s, h, A) \xrightarrow[\Gamma]{\alpha} (s', h', A')$ we use the frame property again. Let $s'_1 = s' \downarrow A'_1$ and $s'_2 = s' \backslash \mathtt{writes}(\alpha_1) \backslash \mathtt{owned}(\Gamma) \cup s' \lceil \mathtt{owned}(\Gamma \lceil A_2)$. There are two possibilities.
    - ∗ Either $(s \downarrow A_1, h_1, A_1) \xrightarrow[\Gamma]{\alpha}$ **abort**, and we can argue as above to show that $(s_1, h_1, A_1) \xrightarrow[\Gamma]{\alpha_1}$ **abort**.
    - ∗ Or $(s \downarrow A_1, h_1, A_1) \xrightarrow[\Gamma]{\alpha_1} (s' \downarrow A'_1, h'_1, A'_1)$ with $h'_1 \perp h_2$ and $h' = h'_1 \cdot h_2$. Hence $(s_1, h_1, A_1) \xrightarrow[\Gamma]{\alpha_1} (s'_1, h'_1, A'_1)$. Trivially we also have $(s_2, h_2, A_2) \xrightarrow[\Gamma]{\epsilon} (s_2, h_2, A_2)$. By the Agreement Property $s' \backslash \mathtt{owned}(\Gamma)$ agrees with $s \backslash \mathtt{owned}(\Gamma)$ except on $\mathtt{writes}(\alpha)$, and by definition of the enabling relation $\mathtt{writes}(\alpha)$ must be disjoint from $\mathtt{owned}(\Gamma \lceil A_2)$, so it is easy to see that $s'_2 = s_2$. The result follows.
- Inductive case: $\alpha_1 = \lambda_1 \alpha'_1$ and $\alpha_2 = \lambda_2 \alpha'_2$, $\alpha \in \alpha_1 \;{}_{A_1}\|_{A_2}\; \alpha_2$.
  If α is *abort* because λ_1 and λ_2 interfere, they must involve a concurrent write to a critical identifier or to a heap cell. Since critical identifiers are protected and $A_1 \cap A_2 = \{\}$, and $\mathtt{dom}(h_1) \cap \mathtt{dom}(h_2) = \{\}$, it follows that either $(s_1, h_1, A_1) \xrightarrow[\Gamma]{\lambda_1}$ **abort** or $(s_2, h_2, A_2) \xrightarrow[\Gamma]{\lambda_2}$ **abort**. The result then follows.

  Otherwise, without loss of generality, assume that: $(A_1, A_2) \xrightarrow{\lambda_1} (A''_1, A_2)$, $\alpha = \lambda_1 \alpha''$, $\alpha'' \in \alpha'_1 \|_{A''_1, A_2} \alpha_2$. (Again the other case is symmetrical.)
  Let $s_1 = s \backslash \mathtt{writes}(\alpha_2) \backslash \mathtt{owned}(\Gamma) \cup s \lceil \mathtt{owned}(\Gamma \lceil A_1)$,
  and $s_2 = s \backslash \mathtt{writes}(\alpha_1) \backslash \mathtt{owned}(\Gamma) \cup s \lceil \mathtt{owned}(\Gamma \lceil A_2)$.
  - If $(s, h, A) \xrightarrow[\Gamma]{\alpha}$ **abort** then either $(s, h, A) \xrightarrow[\Gamma]{\lambda_1}$ **abort**, or there is a state (s'', h'', A'') such that $(s, h, A) \xrightarrow[\Gamma]{\lambda_1} (s'', h'', A'') \xrightarrow[\Gamma]{\alpha''}$ **abort**.

    In the first subcase the frame property for λ_1 implies that $(s \downarrow A_1, h_1, A_1) \xrightarrow[\Gamma]{\lambda_1}$ **abort**. But $s \downarrow A_1$ and s_1 agree on $\mathtt{free}(\alpha_1)$, so $(s_1, h_1, A_1) \xrightarrow[\Gamma]{\lambda_1}$ **abort** and $(s_1, h_1, A_1) \xrightarrow[\Gamma]{\alpha_1}$ **abort**, as required.
    In the second subcase by the Frame Property for λ_1 there is a heap $h''_1 \perp h_2$ such that $h'' = h''_1 \cdot h_2$, and

    $$(s \downarrow A_1, h_1, A_1) \xrightarrow[\Gamma]{\lambda_1} (s'' \downarrow A''_1, h''_1, A''_1).$$

    Let $s''_1 = s'' \backslash \mathtt{writes}(\alpha_2) \backslash \mathtt{owned}(\Gamma) \cup s'' \lceil \mathtt{owned}(\Gamma \lceil A''_1)$,
    and $s''_2 = s'' \backslash \mathtt{writes}(\alpha_1) \backslash \mathtt{owned}(\Gamma) \cup s'' \lceil \mathtt{owned}(\Gamma \lceil A_2)$.
    By the Agreement Properties, s''_2 agrees with s_2 on $\mathtt{free}(\alpha_2, \Gamma)$, and

    $$(s_1, h_1, A_1) \xrightarrow[\Gamma]{\lambda_1} (s''_1, h_1, A_1).$$

    We also have, by assumption,

    $$(s'', h'', A'') \xrightarrow[\Gamma]{\alpha''} \textbf{abort}.$$

    By the induction hypothesis for α'', we must have:
    - ∗ either $(s''_1, h''_1, A''_1) \xrightarrow[\Gamma]{\alpha'_1}$ **abort** and hence $(s_1, h_1, A_1) \xrightarrow[\Gamma]{\alpha_1}$ **abort**;

  * or $(s_2'', h_2, A_2) \xrightarrow[\Gamma]{\alpha_2}$ **abort**, in which case since s_2 agrees with s_2'' on $\mathtt{free}(\alpha_2, \Gamma)$ it also follows that $(s_2, h_2, A_2) \xrightarrow[\Gamma]{\alpha_2}$ **abort**.

- If $(s, h, A) \xrightarrow[\Gamma]{\alpha} (s', h', A')$ then there must be a state (s'', h'', A'') such that

$$(s, h, A) \xrightarrow[\Gamma]{\lambda_1} (s'', h'', A'') \xrightarrow[\Gamma]{\alpha''} (s', h', A').$$

Use the frame property for the first step.
If $(s{\downarrow}A_1, h_1, A_1) \xrightarrow[\Gamma]{\lambda_1}$ **abort** we get$(s_1, h_1, A_1) \xrightarrow[\Gamma]{\alpha_1}$ **abort** as above.
Otherwise, we must have $(s{\downarrow}A_1, h_1, A_1) \xrightarrow[\Gamma]{\lambda_1} (s''{\downarrow}A_1'', h_1'', A_1'')$ with $h_1'' \perp h_2, h'' = h_1'' \cdot h_2$. Using the Agreement Properties as above it follows that $(s_1, h_1, A_1) \xrightarrow[\Gamma]{\lambda_1} (s_1'', h_1'', A_1'')$, where $s_1'' = s''\backslash\mathtt{writes}(\alpha_2)\backslash\mathtt{owned}(\Gamma) \cup s''\lceil\mathtt{owned}(\Gamma\lceil A_1'')$.
Let $s_2'' = s''\backslash\mathtt{writes}(\alpha_1')\backslash\mathtt{owned}(\Gamma) \cup s''\lceil\mathtt{owned}(\Gamma\lceil A_2)$.
The induction hypothesis for α'' implies that

  * either $(s_1'', h_1'', A_1'') \xrightarrow[\Gamma]{\alpha_1'}$ **abort**, so that $(s_1, h_1, A_1) \xrightarrow[\Gamma]{\alpha_1}$ **abort**;
  * or $(s_2'', h_2, A_2) \xrightarrow[\Gamma]{\alpha_2}$ **abort**, and since s_1'' agrees with s_2 on $\mathtt{free}(\alpha_2, \Gamma)$ we get $(s_2, h_2, A_2) \xrightarrow[\Gamma]{\alpha_2}$ **abort**;
  * or $(s_1'', h_1'', A_1'') \xrightarrow[\Gamma]{\alpha_1'} (s_1', h_1', A_1')$ and $(s_2'', h_2, A_2) \xrightarrow[\Gamma]{\alpha_2} (s_2''', h_2', A_2')$, where $h_1' \perp h_2', h' = h_1' \cdot h_2'$, $A_1' \perp A_2', A_1' \cdot A_2' = A'$, and
  $s_1' = s'\backslash\mathtt{writes}(\alpha_2)\backslash\mathtt{owned}(\Gamma) \cup s'\lceil\mathtt{owned}(\Gamma\lceil A_1')$,
  $s_2''' = s'\backslash\mathtt{writes}(\alpha_1')\backslash\mathtt{owned}(\Gamma) \cup s'\lceil\mathtt{owned}(\Gamma\lceil A_2')$.
  Hence $(s_1, h_1, A_1) \xrightarrow[\Gamma]{\lambda_1} (s_1'', h_1'', A_1'') \xrightarrow[\Gamma]{\alpha_1'} (s_1', h_1', A_1')$ and

$$(s_1, h_1, A_1) \xrightarrow[\Gamma]{\alpha_1} (s_1', h_1', A_1').$$

  Since s_2'' agrees with s_2 except on $\mathtt{writes}(\lambda_1) - \mathtt{owned}(\Gamma)$ we also get

$$(s_2, h_2, A_2) \xrightarrow[\Gamma]{\alpha_2} (s_2', h_2', A_2'),$$

  where $s_2' = s'\backslash\mathtt{writes}(\alpha_1)\backslash\mathtt{owned}(\Gamma) \cup s'\lceil\mathtt{owned}(\Gamma\lceil A_2')$.
  That completes the proof.

Corollary 5 (*Parallel Decomposition*). *Assume* $(\mathtt{free}(c_1) \cap \mathtt{writes}(c_2)) \cup (\mathtt{writes}(c_1) \cap \mathtt{free}(c_2)) \subseteq \mathtt{owned}(\Gamma)$ *and* $\alpha \in \alpha_1 \| \alpha_2$, *where* $\alpha_1 \in [\![c_1]\!]$ *and* $\alpha_2 \in [\![c_2]\!]$. *Suppose that* $h_1 \perp h_2$ *and* $h = h_1 \cdot h_2$. *Let* $s_1 = s\backslash\mathtt{writes}(\alpha_2)$ *and* $s_2 = s\backslash\mathtt{writes}(\alpha_1)$.

- *If* $(s, h) \xrightarrow[\Gamma]{\alpha}$ **abort** *then* $(s_1, h_1) \xrightarrow[\Gamma]{\alpha_1}$ **abort** *or* $(s_2, h_2) \xrightarrow[\Gamma]{\alpha_2}$ **abort**.
- *If* $(s, h) \xrightarrow[\Gamma]{\alpha} (s', h')$ *then* $(s_1, h_1) \xrightarrow[\Gamma]{\alpha_1}$ **abort** *or* $(s_2, h_2) \xrightarrow[\Gamma]{\alpha_2}$ **abort**, *or there are disjoint heaps* $h_1' \perp h_2'$ *such that* $h' = h_1' \cdot h_2'$ *and* $(s_1, h_1) \xrightarrow[\Gamma]{\alpha_1} (s_1', h_1')$, $(s_2, h_2) \xrightarrow[\Gamma]{\alpha_2} (s_2', h_2')$, *where* $s_1' = s'\backslash\mathtt{writes}(\alpha_2)$ *and* $s_2' = s'\backslash\mathtt{writes}(\alpha_1)$.

Proof. Let $A = \{\}$ in the previous lemma.

References

[1] G. Andrews, Concurrent Programming: Principles and Practice, Benjamin/Cummings, 1991.

[2] L. Birkedal, N. Torp-Smith, J.C. Reynolds, Local reasoning about a copying garbage collector, in: Proc. POPL Conference, Venice, January 2004, pp. 220–231.

[3] R. Bornat, C. Calcagno, P.W. O'Hearn, M. Parkinson, Permission accounting in separation logic, in: Proc. POPL 2005, pp. 59–70.

[4] R. Bornat, C. Calcagno, P.W. O'Hearn, Local reasoning, separation, and aliasing, in: Proc. 2nd ACM/SIGPLAN Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management, SPACE 2004, January 2004.

[5] C. Boyapati, R. Lee, M. Rinard, Ownership types for safe programming: Preventing data races and deadlocks, in: Proc. OOPSLA 2002, 2002, pp. 211–230.

[6] J. Boyland, Checking interference with fractional permissions, in: R. Cousot (Ed.), Proc. 10th Symposium on Static Analysis, in: Springer LNCS, vol. 2694, 2003, pp. 55–72.

[7] P. Brinch Hansen, Structured multiprogramming, Communications of the ACM 15 (7) (1972) 574–578.
[8] P. Brinch Hansen, Concurrent programming concepts, ACM Computing Surveys 5 (4) (1973) 223–245.
[9] P. Brinch Hansen, Operating System Principles, Prentice Hall, 1973.
[10] P. Brinch Hansen, The programming language Concurrent Pascal, IEEE Transactions on Software Engineering SE-1 (2) (1975) 196–206.
[11] S. Brookes, A semantics for concurrent separation logic, in: Invited Paper, CONCUR 2004, London, in: Springer LNCS, vol. 3170, 2004.
[12] S. Brookes, Traces, pomsets, fairness and full abstraction for communicating processes, in: Proc. CONCUR 2002, Brno, in: Springer LNCS, vol. 2421, 2002, pp. 466–482.
[13] S. Brookes, The essence of parallel algol, in: Proc. 11th Symposium on Logic in Computer Science, IEEE Computer Society Press, 1996, pp. 164–173; Information Computation 179 (1) (2002) 118–149.
[14] 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.
[15] S. Brookes, Idealized CSP: Combining procedures with communicating processes, in: 13th MFPS Conference, Pittsburgh, March 1997, in: Electronic Notes in Theoretical Computer Science, vol. 6, Elsevier, 1997.
[16] S. Brookes, Full abstraction for a shared-variable parallel language, in: Proc. 8th IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press, 1993, pp. 98–109; Information and Computation 127 (2) (1996) 145–163.
[17] S. Brookes, A.W. Roscoe, Deadlock analysis in networks of communicating processes, Distributed Computing 4 (1991) 209–230.
[18] E.W. Dijkstra, Hierarchical ordering of sequential processes, Acta Informatica 1 (2) (1972) 115–138.
[19] E.W. Dijkstra, The structure of the 'THE' multiprogramming system, Communications of the ACM 11 (5) (1968) 341–346.
[20] E.W. Dijkstra, Cooperating sequential processes, in: F. Genuys (Ed.), Programming Languages, Academic Press, 1968, pp. 43–112.
[21] C.A.R. Hoare, A structured paging system, Computer Journal 16 (3) (1973) 209–215.
[22] C.A.R. Hoare, Parallel programming: An axiomatic approach, Computer Languages 1 (1975) 151–160.
[23] C.A.R. Hoare, Monitors: An operating system structuring concept, Communications of the ACM 17 (10) (1974) 549–557.
[24] C.A.R. Hoare, Towards a Theory of Parallel Programming, in: C.A.R. Hoare, R.H. Perrot (Eds.), Operating Systems Techniques, Academic Press, 1972, pp. 61–71.
[25] S. Isthiaq, P.W. O'Hearn, BI as an assertion language for mutable data structures, in: Proc. 28th POPL Conference, London, January 2001, pp. 36–49.
[26] C.B. Jones, Development methods for computer programs including a notion of interference, Ph.D. Thesis, Oxford University, June 1981. Technical Monograph PRG-25, Programming Research Group, Oxford University Computing Laboratory.
[27] C.B. Jones, Specification and design of (parallel) programs, in: Proc. IFIP Conference, 1983.
[28] J. Misra, M. Chandy, Proofs of networks of processes, IEEE Transactions on Software Engineering 7 (1981) 417–426.
[29] H.C. Lauer, Correctness in operating systems. Ph.D. Thesis, Carnegie Mellon University, 1973.
[30] P.W. O'Hearn, Notes on separation logic for shared-variable concurrency, January 2002, Unpublished manuscript.
[31] P.W. O'Hearn, Resources, concurrency, and local reasoning, in: CONCUR 2004 (Invited Paper), London, August 2004, in: Springer LNCS, vol. 3170, Theoretical Computer Science 375 (1–3) (2007) 271–307.
[32] P.W. O'Hearn, H. Yang, J.C. Reynolds, Separation and information hiding, in: Proc. 31st POPL Conference, ACM Press, Venice, January 2004, pp. 268–280.
[33] P.W. O'Hearn, J.C. Reynolds, H. Yang, Local reasoning about programs that alter data structures, in: Proc. 15th Conference of the European Association for Computer Science Logic, in: Springer LNCS, vol. 2142, 2001, pp. 1–19.
[34] P.W. O'Hearn, D.J. Pym, The logic of bunched implications, Bulletin of Symbolic Logic 5 (2) (1999) 215–244.
[35] S. Owicki, L. Lamport, Proving liveness properties of concurrent programs, ACM TOPLAS 4 (3) (1982) 455–495.
[36] S. Owicki, D. Gries, An axiomatic proof technique for parallel programs I, Acta Informatica 6 (1976) 319–340.
[37] S. Owicki, D. Gries, Verifying properties of parallel programs: An axiomatic approach, Communications of the ACM 19 (5) (1976) 279–285.
[38] D. Park, On the semantics of fair parallelism, in: Abstract Software Specifications, in: LNCS, vol. 86, Springer-Verlag, 1979, pp. 504–526.
[39] A. Pnueli, The temporal semantics of concurrent programs, Theoretical Computer Science 13 (1) (1981) 45–60.
[40] J.C. Reynolds, Intuitionistic reasoning about shared mutable data structure, in: J. Davies, A.W. Roscoe, J. Woodcock (Eds.), Millenium Perspectives in Computer Science, Palgrave, 2000, pp. 303–321.
[41] J.C. Reynolds, Separation logic: A logic for shared mutable data structures, in: Invited Paper. Proc. 17th IEEE Conference on Logic in Computer Science, LICS 2002, IEEE Computer Society, 2002, pp. 55–74.
[42] J.C. Reynolds, Lecture notes on separation logic (15-819A3), Department of Computer Science, Carnegie-Mellon University, Spring 2003. Revised May 23, 2003, page 178 (Chapter 8).
[43] J.C. Reynolds, Towards a grainless semantics for shared-variable concurrency, in: Slides from Invited Lecture, 31st POPL concerence, Venice, January 2004.
[44] H. Yang, An example of local reasoning in BI pointer logic: The Schorr–Waite graph marking algorithm, in: Proc. SPACE 2001 Workshop on Semantics, Program Analysis and Computing Environments for Memory Management IT University of Copenhagen, 2001, pp. 41–68.