Published in *Theoretical Computer Science 375(1-3)*, pp271-307, May 2007.

Resources, Concurrency and Local Reasoning*

Peter W. O'Hearn
Queen Mary, University of London

To John C. Reynolds for his 70th birthday.

Abstract

In this paper we show how a resource-oriented logic, separation logic, can be used to reason about the usage of resources in concurrent programs.

1 Introduction

Resource has always been a central concern in concurrent programming. Often, a number of processes share access to system resources such as memory, processor time, or network bandwidth, and correct resource usage is essential for the overall working of a system. In the 1960s and 1970s Dijkstra, Hoare and Brinch Hansen attacked the problem of resource control in their basic works on concurrent programming [17, 18, 23, 24, 8, 9]. In addition to the use of synchronization mechanisms to provide protection from inconsistent use, they stressed the importance of *resource separation* as a means of controlling the complexity of process interactions and reducing the possibility of time-dependent errors. This paper revisits their ideas using the formalism of separation logic [43].

Our initial motivation was actually rather simple-minded. Separation logic extends Hoare's logic to programs that manipulate data structures with embedded pointers. The main primitive of the logic is its separating conjunction, which allows local reasoning about the mutation of one portion of state, in a way that automatically guarantees that other portions of the system's state remain unaffected [34]. Thus far separation logic has been applied to sequential code but, because of the way it breaks state into chunks, it seemed as if the formalism might be well suited to shared-variable concurrency, where one would like to assign different portions of state to different processes.

Another motivation for this work comes from the perspective of general resource-oriented logics such as linear logic [19] and BI [35]. Given the development of these logics it might seem natural to try to apply them to the problem of reasoning about resources in concurrent programs. This paper is one attempt to do so – separation logic's assertion language is an instance of BI – but it is certainly not a final story. Several limitations and directions for further work will be discussed at the end of the paper.

There are a number of approaches to reasoning about imperative concurrent programs (e.g., [38, 40, 26]), but the ideas in an early paper of Hoare on concurrency, "Towards a Theory of Parallel Programming [23]" (henceforth, TTPP), fit particularly well with the viewpoint of separation

*A preliminary version of this paper appeared in the Proceedings of the 15th International Conference on Concurrency Theory (CONCUR'04), P. Gardner and N. Yoshida editors, LNCS 3170, pp49–76.

logic. The approach there revolves around a concept of "spatial separation" as a way to organize thinking about concurrent processes, and to simplify reasoning. Based on compiler-enforceable syntactic constraints for ensuring separation, Hoare described formal proof rules for shared-variable concurrency that were beautifully modular: one could reason locally about a process, and simple syntactic checks ensured that no other process could tamper with its state in a way that invalidated the local reasoning.

So, the initial step in this work was just to insert the separating conjunction in appropriate places in the TTPP proof rules, or rather, the extension of these rules studied by Owicki and Gries [39]. Although the mere insertion of the separating conjunction was straightforward, it turned out that the resulting proof rules contained two surprises, one positive and the other negative.

The negative surprise was a counterexample due to John Reynolds, which showed that the rules were unsound if used without restriction. Difficulties showing soundness had delayed the publication of the proof rules, which were first circulated in an unpublished note of the author from August of 2001. After the counterexample came to light the author, Hongseok Yang and Reynolds resolved a similar issue in a sequential programming language [37] by requiring certain assertions in the proof rules to be "precise"; assertions are those that unambiguously pick out an area of storage. In the present work precise assertions can again be used to salvage soundness, as has been demonstrated by Brookes [12]. Reynolds's counterexample, and the notion of precise assertion, will be presented in Section 11.

The positive surprise was that we found we could handle a number of daring, though valuable, programming idioms, and this opened up a number of unexpected (for us) possibilities. Typically, the idioms involve the transfer of the ownership of, or right to access, a piece of state from one process to another. This behaviour is common in systems programs, where limited resources are shared amongst a collection of competing or cooperating processes. We say more on the nature of the daring programs in Section 2.2 below.

The least expected result, which was never a conscious aim at the beginning, was that the method turned out not to depend essentially on having a structured notation encapsulating high level uses of synchronization constructs. For instance, we found we were able to reason in a modular way about some semaphore idioms. The key to this is what we call the *resource reading of semaphores*, where a semaphore is (logically) attached to a piece of state, and where pieces of state flow dynamically between the semaphores and surrounding code when the `P` and `V` operations are performed. The resource reading is described informally in Section 2.3, and more formally in Sections 6 and 10.

The ability to deal with ownership transfer is a consequence of using a logical connective, the separating conjunction, to describe resource partitions that change over time. This is in contrast to the fixed partitioning that can be described by more static techniques such as scoping constraints or typing.

This paper is very much about fluency with the logic – how to reason with it – rather than its metatheory. A number of examples are given in some detail; proving particular programs is the only way I know to probe whether a logic connects to, or alternatively is detached from, reasoning intuition. A very simple example, parallel mergesort, is given in Section 3. It shows the basic idea of independent reasoning about separate portions of memory, where the partition between the portions is dynamically determined. A binary semaphore example is presented in Section 6, where the global property that several sempahore values never add up to more than one (that we have a "split binary semaphore") is a consequence of purely local remarks about state attached to each semaphore; we do not need to maintain a global invariant describing the relationship between the different semaphore values. Sections 7 and 8 contain examples, a pointer-transferring buffer and a toy memory manager, where ownership transfer is part of the interface specification of a data abstraction. The buffer and the memory manager are then combined in Section 9) to bring home the modularity aspect. Finally, the unbounded buffer of [17] is considered in Section 10. It exhibits a stream of different pieces of state flowing through a single (counting) semaphore, and also the possibility of working concurrently at both ends of the buffer.

The point we will attempt to demonstrate in each example is that the specification for each program component is "local" or "self contained", in that it concentrates on only the state used by

the component, not mentioning at all the states of other processes or resources [34]. This sense of local reasoning is essential if we are ever to have reasoning methods that scale; of course, readers will have to judge for themselves whether the specifications meet this aim.

We refer the reader to the companion paper by Stephen Brookes for a thorough theoretical analysis [12]. Brookes gives a denotational model of the programming language with respect to which he proves soundness of the proof rules. In addition to soundness, the analysis justifies many of the remarks we make about the proof system during the course of the paper, such as remarks about race conditions. The relation to Brookes's work is detailed in Section 2.4 below.

2 Basic Principles

In the introduction above we acknowledge that this work unfolded not according to a fixed plan but by the unexpected discovery of useful reasoning idioms. In this section we bring some system to the discussion by describing the underlying ideas and issues in a non-technical way. As this section is mainly about providing context for the work that follows some readers may wish to skip or skim it, and refer back as necessary.

2.1 Racy and Race-free Programs

To begin with we assume a programming model where a number of concurrent processes share access to a common state. The state may include program variables as well as a behind-the-scenes heap component not all of whose addresses are named by program variables.

(Aside: since the processes we are talking about share the same address space they would be called "threads" in Unix jargon.)

When processes share the same address space there is the possibility that they compete for access to a portion of state.

> A program is *racy* if two concurrent processes attempt to access the same portion of state at the same time. Otherwise the program is *race-free*.

An example of a racy program is

$$x := y + x \parallel x := x \times z$$

The meaning of this statement is dependent not only on the orderings of potential interleavings, but even on the level of granularity of the operations.[1]

Race avoidance is an issue when processes compete for system resources, if these resources are to be used in a consistent way. In this paper we will concentrate on competition for program state, but the point is equally valid for other kinds of resource such as files, CPU timeslices, or network bandwidth.

Most theories of concurrency do not attach any special significance to the disinction between racy and race-free programs. But it is understood as significant by programmers. Races can lead to irreproducible program behaviour which makes testing difficult. Stated more positively, race-freedom frees us from thinking about minute details of interleavings, or even granularity, of sequential programming constructs. For example, sequentially equivalent programs that can be distinguished by concurrency are $x := x + 1; x := x + 1$ and $x := x + 2$. But the only way we can see their inequivalence is through interference from another process, that is, by racing.

It is not that it is impossible, in principle, to describe the minute details of interleavings. Rather, the aim of a program design is often to ensure that we don't have to think about these minute details. This is essentially the point of Dijkstra's criterion of *speed independence* in his principles for concurrent program design [17].

[1] In the Conclusion we discuss the relative nature of raciness, and how it is related tol granularity. See also John Reynolds's recent work which proposes that race-free programs can enjoy a semantics that is "grainless" [45].

2.2 Cautious and Daring Concurrency

We make some further assumptions. We suppose that there is a way to identify groupings of mutual exclusion. A "mutual exclusion group" is a class of commands whose elements (or their occurrences) are required not to overlap in their executions. Notice that there is no requirement of atomicity; execution of commands not in the same mutual exclusion group might very well overlap. In monitor-based concurrency each monitor determines a mutual exclusion group, consisting of all calls to the monitor procedures. When programming with semaphores each semaphore s determines a group, the pair of the semaphore operations $P(s)$ and $V(s)$. In TTPP the collection of conditional critical regions `with` r `when` B `do` C with common resource name r forms a mutual exclusion group.

With this terminology we may now state one of the crucial distinctions in the paper.

> A program is *cautious* if, whenever concurrent processes access the same piece of state, they do so only within commands from the same mutual exclusion group. Otherwise, the program is *daring*.

The simplicity and modularity of the TTPP proof rules is achieved by syntactic restrictions which ensure caution. It is possible to program cautiously with pointers as well – for example, by using a linked structure rather than an array to represent a queue – and then separation logic can help. But the more significant way that separation logic will go beyond TTPP will be into the realm of daring programs.

Examples of daring programs are many. Indeed, almost all semaphore programs are daring. Brinch Hansen states the point with characteristic clarity:

> Since a semaphore can be used to solve arbitrary synchronizing problems, a compiler cannot conclude that a pair of *wait* and *signal* operations [`P` and `V` operations] on a given semaphore initialized to one delimits a critical region, nor that a missing member of such a pair is an error [10].

Now, semaphores are often used in a

```
P(mutex)
critical piece of code
V(mutex)
```

form, where the `P`'s and `V`'s for a given semaphore match in a properly nested fashion. In contrast, for illustrative purposes, we will consider an unmatching idiom, where two processes share access to address 10 and send signals to one another via two semaphores. (Recall that a semaphore is represented by a non-negative integer, where `V`(s) bumps the value of s up by 1, and `P`(s) decrements s by 1 whenever it is greater than 0, waiting otherwise.)

```
semaphore free := 1; busy := 0
⋮                              ⋮
P(free);                       P(busy);
[10] := m;            ||       n := [10];
V(busy);                       V(free);
⋮                              ⋮
```

The overall effect is that the left process assigns a message m to address 10 and then signals the second process, which reads the message and then signals back to the sending process, perhaps to send another message.[2]

[2] Our use of an address, 10, here rather than a program variable is something of an embarrassment. The reason is that separation logic treats the heap more flexibly than ordinary variables in Hoare logic, so that a heap address, but not a variable, can be transferred between threads. We say more on this in Remark 4 at the end of Section 6 and in Section 12.

This is a daring program according to our classification above because the accesses to address 10 are not *within* the units of mutual exclusion, which are the semaphores themselves. Of course, the semaphore idiom is intended to implement mutual exclusion of the accesses to 10, but this exclusion is not enforced on a language level. For example, we could initialize the semaphores wrongly by, say, starting both at 1 in which case both processes would speed through their $\texttt{P}(\cdot)$ operations and race for 10. (The issue here is analogous to how one can implement a procedure call/return idiom in assembly language, where correct use of the idiom is not enforced by the language.)

The skeptical reader might complain at this point that we could rewrite this code using conditional critical regions or monitors, that it is, morally, cautious concurrency. While not wanting to apologize for semaphores we would partially agree, but would also make two further points.

First, there are examples where the daring aspect – which has to do with accessing shared state *outside* a unit of exclusion such as a critical region – is part of the very purpose of the program. These arise when a pointer or other form of identification is passed into and out of a grouping of mutual exclusion. A typical example is in efficient message passing, where a pointer is transferred from one process to another in order to avoid copying large pieces of data, and one has to be careful to avoid dereferencing the pointer at the wrong time. An example of this form is given in Section 7.

Perhaps more vividly, resource managers of essentially any kind are daring. For instance, interaction with a memory manager results in pieces of storage transferring between the manager and its clients as allocation and deallocation operations are performed. Typically, a piece of state will be used by one process, not within a unit of mutual exclusion, then returned to the manager where it is temporarily protected by synchronization, and subsequently reallocated to another process where it can again be accessed outside of a unit of mutual exclusion. If the first process accesses the state after it has passed it back to the manager then unpredictable results may ensue. An example of this form is given in Section 8.

Indeed, concurrent systems programs, such as microkernel OS designs or middleware code, are almost always daring.

But to be daring is to court danger: If processes access the same portion of state outside units of mutual exclusion then they just might do so at the same time, and we can very well get inconsistent results. How might we be daring and yet consistent? If we can answer this question then we might be able to reason about daring programs without accounting explicitly for the possibility of interference or considering the minute details of possible interleavings.

2.3 Ownership and Separation

Our approach revolves around an interplay between two ideas: ownership and separation.

> *Ownership Hypothesis*. A code fragment can access only those portions of state that it owns.

This is a hypothesis in the sense that we are supposing that there is a notion of ownership for which the statement makes sense. Ownership will be explained by example below, and described technically by assertions in separation logic. We stress that ownership is not here required to be part of the runtime model of a system, but rather is an additional notion we put on top of the model, as an aid to reasoning.

The real point of the Ownership Hypothesis is that is enables us to state the

> *Separation Property*. At any time, the state can be partitioned into that owned by each process and each grouping of mutual exclusion.

The Separation Property is what allows independent reasoning about program components. The job of our proof system will be to ensure that any program that gets past the proof rules satisfies it.

A crucial point is that the Separation Property does not presume a static, once and for all, partition of the state; it allows the partition to change over time. This can be made clear using

the snippet of daring semaphore code above (as long as the semaphores are properly initialized, *free* to 1 and *busy* to 0). Here is the code again, this time annotated with assertions.

$$
\begin{array}{ll}
\vdots & \vdots \\
\{\texttt{emp}\} & \{\texttt{emp}\} \\
\texttt{P}(\mathit{free}); & \texttt{P}(\mathit{busy}); \\
\{10 \mapsto -\} & \{10 \mapsto -\} \\
[10] := m; \quad \| & n := [10]; \\
\{10 \mapsto -\} & \{10 \mapsto -\} \\
\texttt{V}(\mathit{busy}); & \texttt{V}(\mathit{free}); \\
\{\texttt{emp}\} & \{\texttt{emp}\} \\
\vdots & \vdots
\end{array}
$$

We will explain the assertions later but first give an informal narrative about this code.

The key to our discussion is what we call the *resource reading of semaphores*. In this reading a semaphore is not just a device for counting or sending signals, but is additionally a resource owner. Portions of state are *attached* to semaphores, in a logical way part of what they mean. At any given time a semaphore owns a portion of state and the `P` and `V` operations transfer ownership between the semaphore and the surrounding code. It is this (logical) attaching of resource to semaphores, combined with ownership transfer, that will allow us to to reason modularly about them.

The assertions in the code do not describe the linkage between each semaphore and address 10, the attached piece of state; the assertions describe instead local states from point of view of the processes. Technically, in the proof theory there are invariants which describe when each semaphore owns address 10. We do not want to descend into these technicalities at the moment (the full proof will be given in Section 6), but instead want to concentrate on the intuitive understanding of this code in terms of ownership and separation. So we continue with our narrative.

In the code above the job of a $\texttt{P}(s)$ operation for a semaphore s is to release ownership of 10 into the code that follows, while $\texttt{V}(s)$ acquires ownership of 10, swallowing it into s. (Notice the curious inversion of the usual intuitive description of operations on binary semaphores as acquiring and releasing locks.') In this example the ownership for a semaphore is all-or-nothing, it owns 10 or it doesn't, and it is always the same piece of state that is owned or not. In subtler examples (e.g., Section 10) the semaphores can release and swallow up portions of state a little at a time and different pieces of state can be owned at different times.

Returning to the example code above we will want to preserve the Separation Property throughout our narrative; in particular, we will ensure that

> address 10 is owned by exactly one of the left process, the right process, the *free* semaphore or the *busy* semaphore

holds at all times. We assume that the property holds at the beginning, that 10 is owned by *free*. In fact, both semaphores will satisfy an invariant which says "either my value is 1 and I own address 10, or my value is 0 and I own nothing".

Looking at the annotated code the command $\texttt{P}(\mathit{free})$ in the left process can be executed only when the semaphore *free* owns address 10. On completion of the operation the semaphore gives up ownership, releasing it in to the code that follows. In this way, ownership of 10 is assumed by the left process before the command $[10] := m$. In fact, 10 must be owned at this point if we are to be consistent with the Ownership Hypothesis. And this transfer of ownership from *free* into the left process preserves the Separation Property. Continuing with the command $\texttt{V}(\mathit{busy})$, ownership of 10 is then given up by the left process and swallowed up by the *busy* semaphore.

A similar narrative can be given for the righthand process. The point is that the $\texttt{V}(\mathit{busy})$ command has the effect of *busy* assuming ownership of address 10, after which $\texttt{P}(\mathit{busy})$ can release it before the address is read in the rightmost process. Similarly, $\texttt{V}(\mathit{free})$ in the right process swallows ownership of 10, after which it can again be released into the leftmost process using $\texttt{P}(\mathit{free})$.

This narrative explains the sense in which ownership of 10 moves around, passing between processes and semaphores, in a way that satisfies the Separation Property. Of course, there is nothing operational which says that "ownership moves"; the semaphore operations themselves do not mention 10 at all. This is the sense in which ownership is a conceptual add-on, which we put on top of the runtime model for the purposes of understanding.

Now we turn our attention to the assertion annotations. We use the "ownership" or "permission" reading of separation logic, where an assertion P at a program point implies that "I have the right to dereference the cells in P here", or more briefly, "I own P" [37]. According to this reading the assertion $10 \mapsto -$ says "I own the cell at address 10" (and I don't own anything else). The assertion `emp` does not say that the global state is empty, but rather that "I don't own any heap cells, here".

With this reading of assertions the annotated code then mirrors our narrative. The triple $\{\texttt{emp}\}\texttt{P}(\mathit{free})\{10 \mapsto -\}$ corresponds to how ownership of 10 is released into the code in the left process, and the use of `emp` in the postcondition of $\{10 \mapsto -\}\texttt{V}(\mathit{busy})\{\texttt{emp}\}$ corresponds to the idea that the first process gives up ownership of 10 when it executes the `V` operation.

We have now reached a point where our explanations can go no further without studying the proof rules themselves. We hope, however, to have displayed the basic idea: if we can account for transfer of ownership between program components then we can guarantee consistency of daring programs.

2.4 Relation to Brookes's Analysis

In addition to soundness, Brookes's analysis in [12] justifies the remarks made in this section. In particular, he shows that

> if $\{P\}C\{Q\}$ can be proven then any execution of C starting from a state satisfying P will not result in a race condition or an attempt to access a dangling pointer.

In our definition of racy programs we referred to processes accessing a portion of state at the "same time"; we hope that the reader can understand this in an intuitive sense. We stress, though, that it does not indicate a commitment to true concurrency. Indeed, Brookes's formulation uses an interleaving model where "same time" means one after the other without intervening synchronization operations.

Brookes establishes an anologue of the Separation Property as part of his proof of soundness, using a semantics that explicitly partitions heap cells. We warn the reader, however, that this is a meta-property in the sense that we will never write a formula that describes it. Rather, the proof rules are organized in a way that guarantees the property in an implicit way, as a result of the way that proofs are constructed. The assertions we use for reasoning about processes will always be local, in that they refer to the state of a resource or an individual process, whereas the Separation Property is a global fact that is a consequence of the reasoning.

The most remarkable part of Brookes's analysis is an interplay between an interleaving semantics based on traces of actions and a "local enabling" relation that "executes" a trace in a portion of state owned by a process. The enabling relation skips over intermediate states and works by presuming that there is no "interference from the outside". This presumption is vital for the soundness of the sequential proof rules. For instance, we can readily derive

$$\{10 \mapsto 3\}x := [10]; x := [10]\{(10 \mapsto 3) \wedge x = 3\}$$

in separation logic, but if there was outside interference, say altering the contents of 10 between the first and second commands, then the triple would not be true. This presumption is also pertinent to the discussion at the end of Section 2.1, about the usual inequivalence of $x := x + 1; x := x + 1$ and $x := x + 2$, a distinction that arises only from outside interference.

In the definitions of racy programs and the Separation Property we have not distinguished read from write access; the informal definitions could be weakened to allow two processes to read the same portion of state at the same time. In one part of Brookes's analysis (the "global" semantics)

he allows for sharing of heap cells, while in another part (the "local" semantics) he does not. This issue is related to the definition of the separating conjunction, which requires strict disjointness of heap cells. The issue of shared read access to the heap is the subject of current research in separation logic.

3 Disjoint Concurrency

Before considering process interaction it will be useful to have a warm-up session, with the special case of disjoint concurrency. The presentation in this section will not be completely formal. It is assumed, though, that the reader is familiar with the basics of separation logic. (The precise syntax and semantics of assertions will be given in table 2 and in the appendix.) For now we only remind the reader of two main points. First, $P * Q$ means that the (current, or owned) heap can be split into two components, one of which makes P true and the other of which makes Q true. Second, to reason about a dereferencing operation we must know that a cell exists in a precondition. For instance, if $\{P\}[10] := 42\{Q\}$ holds, where the command mutates address 10, then P must imply the assertion $10 \mapsto - * \texttt{true}$ that 10 not be dangling.

The rule for disjoint concurrency is

$$\frac{\{P\}C\{Q\} \quad \{P'\}C'\{Q'\}}{\{P * P'\}C \parallel C'\{Q * Q'\}}$$

where C does not modify any variables free in P', C', Q', and conversely.

Consider the following putative triple:

$$\{10 \mapsto -\}\, [10] := 42 \;\parallel\; [10] := 6\, \{???\}.$$

The precondition says that address 10 points to something (it is not dangling) and, incidentally, that there are no other cells in the heap. The two assignment commands mutate address 10, and thus consitute a race condition.

We claim that there is no assertion we can find to fill in for the ??? and make this a derivable triple. The reason is that, in separation logic we must know that 10 is not dangling to reason about either of the two assignment commands. So, to reason about the constituent commands we have to give $10 \mapsto -$ to each in its individual precondition. We cannot do that because we would then have to use $*$ to put these preconditions together, and that would result in falsity.

Stated more formally, to prove the program we would first have to split the precondition $10 \mapsto -$ into two assertions P_1, P_2 where

$$10 \mapsto - \;\Rightarrow\; P_1 * P_2.$$

Then we could use the rule of consequence to get the precondition into the form required by the concurrency rule. But, to prove anything about the parallel commands, we must have the implications

$$P_1 \;\Rightarrow\; 10 \mapsto - * \texttt{true} \qquad P_2 \;\Rightarrow\; 10 \mapsto - * \texttt{true}$$

because of the requirement that 10 not be dangling prior to an access to it. There is no way to choose such assertions P_1, P_2 because, if we could, then chaining together the implications we would obtain

$$10 \mapsto - \;\Rightarrow\; (10 \mapsto - * \texttt{true}) * (10 \mapsto - * \texttt{true})$$

which simplifies to

$$10 \mapsto - \;\Rightarrow\; \texttt{false}$$

and this final implication cannot hold.

This discussion is just a special case of a general fact established by Brookes, on race avoidance for all proven programs.

$$
\begin{array}{rcl}
\mathit{array}(a,i,j) & \triangleq & \forall k.\, i \le k \le j \Rightarrow a+i \hookrightarrow - \\
\mathit{sorted}(a,i,j) & \triangleq & \forall k.\, i \le k < j \Rightarrow [a+k] \le [a+k+1] \\
[E] \le [F] & \triangleq & \exists ab.E \hookrightarrow a \wedge F \hookrightarrow b \wedge a \le b \\
E \hookrightarrow F & \triangleq & E \mapsto F * \mathtt{true}
\end{array}
$$

Table 1: Predicates for Mergesort

For a more positive example, we can prove a program that has a potential race, as long as that race is ruled out by the precondition.

$$
\frac{\{x \mapsto 3\}\,[x] := 4\,\{x \mapsto 4\} \qquad \{y \mapsto 3\}\,[y] := 5\,\{y \mapsto 5\}}{\{x \mapsto 3 \;*\; y \mapsto 3\}\,[x] := 4 \parallel [y] := 5\,\{x \mapsto 4 \;*\; y \mapsto 5\}}
$$

Here, the $*$ in the precondition guarantees that x and y are not aliases.

It will be helpful to have an annotation notation for (the binary case of) the parallel composition rule. We will use an annotation form where the overall precondition and postcondition come first and last, vertically, and are broken up for the annotated constituent processes; so the just-given proof is pictured

$$
\begin{array}{ccc}
 & \{x \mapsto 3 \;*\; y \mapsto 3\} & \\
\{x \mapsto 3\} & & \{y \mapsto 3\} \\
[x] := 4 & \parallel & [y] := 5 \\
\{x \mapsto 4\} & & \{y \mapsto 5\} \\
 & \{x \mapsto 4 \;*\; y \mapsto 5\} &
\end{array}
$$

Disjoint concurrency is restrictive, in that it does not allow for process interaction. But it is not without its uses. Consider a parallel version of mergesort:

$\{\mathit{array}(a,i,j)\}$
`procedure` $\mathtt{ms}(a,i,j)$
`newvar` $m := (i+j)/2$;
`if` $i < j$ `then`
  $(\mathtt{ms}(a,i,m) \parallel \mathtt{ms}(a,m+1,j))$;
  $\mathtt{merge}(a,i,m+1,j)$;
$\{\mathit{sorted}(a,i,j)\}$

For simplicity this specification just says that the final array is sorted, not that it is a permutation of the initial array.

Array programs can be translated into separation logic by viewing $a[i]$ as sugar for $[a+i]$. Note that on this view a component assignment $a[i] := E$ does *not* modify the variable a; rather, it modifies address $a+i$. As a result, the `merge` and `ms` procedures do not alter any variables, they only alter heap cells.

The assertion $\mathit{array}(a,i,j)$ says that (at least) the segment from $a+i$ to $a+j$ is owned (in the domain of the current heap), and $\mathit{sorted}(a,i,j)$ says that that segment is sorted. The definitions of these predicates may be found in Table 1.

The crucial part of the proof of the body is the following proof figure for the parallel composition.

SYNTAX

$$
\begin{array}{lll}
E, F & ::= & x, y, ... \mid 0 \mid 1 \mid E + F \mid E \times F \mid E - F \quad \text{(Integer Expressions)} \\
B & ::= & \texttt{false} \mid B \Rightarrow B \mid E = F \mid E < F \quad \text{(Boolean Expressions)} \\
P, Q, R & ::= & B \mid \texttt{emp} \mid E \mapsto F \mid P * Q \mid \quad \text{(Assertions)} \\
 & & P \Rightarrow Q \mid \forall x.P \mid \cdots
\end{array}
$$

ABBREVIATIONS

$$\neg P \;=\; P \Rightarrow \texttt{false};\ \texttt{true} \triangleq \neg(\texttt{false});\ P \vee Q \triangleq (\neg P) \Rightarrow Q;\ P \wedge Q \triangleq \neg(\neg P \vee \neg Q);$$
$$\exists x.\ P \triangleq \neg\forall x.\ \neg P$$

$$
\begin{array}{lll}
E \mapsto F_0, ..., F_n & \triangleq & (E \mapsto F_0) * \cdots * (E + n \mapsto F_n) \\
E \mapsto - & \triangleq & \exists y. E \mapsto y \qquad\qquad (y \notin \mathrm{Free}(E))
\end{array}
$$

Table 2: Assertions

$$
\begin{array}{ccc}
 & \{array(a, i, m) * array(a, m + 1, j)\} & \\
\{array(a, i, m)\} & & \{array(a, m + 1, j)\} \\
\texttt{ms}(a, i, m) & \| & \texttt{ms}(a, m + 1, j) \\
\{sorted(a, i, m)\} & & \{sorted(a, m + 1, j)\} \\
 & \{sorted(a, i, m) * sorted(a, m + 1, j)\} &
\end{array}
$$

We have used the overall specification of `ms` as an hypothesis, as is usual when reasoning about recursion. The verified property sets us up for a call to `merge`; with an appropriate specification of it, we could easily complete the proof of the program using the standard rules for recursive procedures [22] and the rules of sequential separation logic [43].

It is worth noting that this program cannot be proven using the original rule for disjoint concurrency from TTPP:

$$\frac{\{P\}C\{Q\} \quad \{P'\}C'\{Q'\}}{\{P \wedge P'\}C \parallel C'\{Q \wedge Q'\}}$$

where C does not modify any variables free in P', C', Q', and conversely. The reason is that traditional Hoare logic treats array-component assignment globally, where an assignment to $a[i]$ is viewed as an assignment to the entire array

$$\{P[(a|i : E)/a]\}\, a[i] := E\, \{P\}$$

In this view the two parallel calls to `ms` are judged to be altering the *same* variable, a, and so the original disjoint concurrency rule from TTPP cannot apply.

In contrast, the axioms of separation logic treat a component assignment, rendered as $[a+i] := E$, in a local fashion. This, combined with the fact that the separating conjunction lets us divide two different segments of the same array in a way that depends on the state (the values of i and j), leads to a simple program proof.

I am grateful to Tony Hoare for suggesting this example (actually, the suggestion was parallel quicksort, but the point is the same).

4 Process Interaction

The proof rules below refer to assertions from separation logic. A grammar of assertions is in Table 2; assertions include the points-to relation $E \mapsto F$, all of classical logic, and the spatial connectives `emp` and $*$. The use of $\cdots$ in the grammar means we are being open-ended, in that we allow for the

possibility of other forms such as the $-\!\!*$ connective from BI or a predicate for describing linked lists, as in Section 8. A semantics for these assertions has been included in the appendix. There we also relate the form of the semantics to the "permissions" reading of assertions mentioned in Section 2.3.

The expressions E that we use are simply standard integer expressions. In the model given in the appendix the heap is a finite partial mapping from non-negative integers to all integers, so that whenever $E \mapsto F$ holds we will know that E is non-negative. We can regard the negative integers as non-addressible values, or atoms. (For example, in the predicates for linked lists and list segments in Sections 8 and 10 we can regard `nil` as being represented by -1.)

The programming language of this study uses conditional critical regions (CCRs) for process interaction. Our choice to use CCRs is based on theoretical pragmatism. On one hand CCRs can represent semaphores, thus assuring their expressiveness as regards synchronization. On the other hand CCRs are considerably more flexible; see in particular the remarks on "super-semaphores" below. Another possibility would have been to found our study on monitors, but this would require us to include a procedure mechanism and it is theoretically simpler not to do so.

The presentation of the programming language and the proof rules in this section and the next follows that of Owicki and Gries [39], with alterations to account for the heap. As there, we will concentrate on programs of a special form, where we have a single resource declaration, possibly prefixed by a sequence of assignments to variables, and a single parallel composition of sequential commands.

init;
`resource` r_1(variable list), ..., r_m(variable list)
$C_1 \parallel \cdots \parallel C_n$

It is possible to consider nested resource declarations and parallel compositions, but the basic case will allow us to describe variable side conditions briefly in an old-fashioned, wordy, style. We restrict to this basic case mainly to get more quickly to examples and the main point of this paper, which is exploration of idioms (fluency). We refer to [12] for a more thorough treatment of the programming language which does not observe this restricted form.

The grammar for the sequential processes is as follows.

$$\begin{array}{rcl} C & ::= & x := E \mid x := [E] \mid [E] := F \mid x := \texttt{cons}(E_1, ..., E_n) \mid \texttt{dispose}(E) \\ & \mid & \texttt{skip} \mid C; C \mid \texttt{if } B \texttt{ then } C \texttt{ else } C \mid \texttt{while } B \texttt{ do } C \\ & \mid & \texttt{with } r \texttt{ when } B \texttt{ do } C \texttt{ endwith} \end{array}$$

Sequential processes have the constructs of `while` programs as well as operators for accessing a program heap. These include operations $[E] := F$ and $x := [E]$ for mutating and reading heap cells, and operations $x := \texttt{cons}(E_1, ..., E_n)$ and $\texttt{dispose}(E)$ for allocating and deleting cells.

The command for accessing a resource is the conditional critical region:

`with` r `when` B `do` C `endwith`.

Here, B ranges over (heap independent) boolean expressions and C over commands. The `with` command is a unit of mutual exclusion; two `with` commands for the same resource cannot be executed simultaneously. Execution of `with` r `when` B `do` C `endwith` can proceed if no other region for r is currently executing, and if the boolean condition B is true; otherwise, it must wait until the conditions for it to proceed are fulfilled. Thus, the collection of commands `with` r `when` B `do` C `endwith` forms a mutual exclusion group, in the sense of Section 2.2.

The term "resource" is used because the view is that the constructs give a way to protect system resources from inconsistent use, by grouping together a number of critical regions that access the state of the resource with mutual exclusion. This way of representing resources can be seen particularly in examples of Brinch Hansen and Hoare, often using the monitor notation which is a descendent of conditional critical regions [9, 10, 24].

[Aside: There is a possibility of terminological inconsistency in the use of "resource", summed up as follows by Richard Bornat, writing in [5].

> "Hoare called resource bundles simply resources, but I want that word to apply to the items – heap locations and variables at least, perhaps time and stack space and whatever else we can manage – ... A resource bundle contains a bundle of resources described by an invariant formula – hence the nomenclature."

Bornat prefers to call them simply "bundles". That makes sense, but I will continue with "resource" for consistency with previous work.]

A typical use of critical regions is to protect the storage in a buffer; here is code for accessing a one-place buffer, together with two parallel processes in a producer-consumer relationship:

$$
\begin{array}{lcl}
 & \mathit{full} := \texttt{false}; & \\
 & \texttt{resource}\ \mathit{buf}(c, \mathit{full}) & \\
\vdots & & \vdots \\
\mathit{produce}\, m; & \parallel & \texttt{get}(n); \\
\texttt{put}(m); & & \mathit{consume}\, n; \\
\vdots & & \vdots
\end{array}
$$

where the `get` and `put` operations expand into `with` commands as follows,

$$
\begin{array}{lcl}
\texttt{put}(m) & \triangleq & \texttt{with}\ \mathit{buf}\ \texttt{when}\ \neg \mathit{full}\ \texttt{do} \\
 & & \quad c := m;\ \mathit{full} := \texttt{true} \\
 & & \texttt{endwith} \\
 & & \\
\texttt{get}(n) & \triangleq & \texttt{with}\ \mathit{buf}\ \texttt{when}\ \mathit{full}\ \texttt{do} \\
 & & \quad n := c;\ \mathit{full} := \texttt{false} \\
 & & \texttt{endwith};
\end{array}
$$

For presentational convenience, we are using definitions of the form

$$\texttt{name}(\vec{x}) \triangleq \texttt{with}\ r\ \texttt{when}\ B\ \texttt{do}\ C\ \texttt{endwith}$$

to encapsulate operations on a resource. In this we are not introducing a procedure mechanism, but are merely using $\texttt{name}(\vec{x})$ as an abbreviation.

There is a standard encoding of semaphores in terms of CCRs. We use a semaphore name s to double as a program variable and a resource name, and then define

$$
\begin{array}{lcl}
\texttt{P}(s) & = & \texttt{with}\ s\ \texttt{when}\ s > 0\ \texttt{do}\ s := s - 1\ \texttt{endwith} \\
\texttt{V}(s) & = & \texttt{with}\ s\ \texttt{when}\ \texttt{true}\ \texttt{do}\ s := s + 1\ \texttt{endwith}.
\end{array}
$$

This use of CCRs to represent semaphores is not reasonable from an implementation point of view – one would rather use semaphores to implement CCRs – but it is useful from a logical point of view. The CCR encoding directly represents the exclusion aspect of semaphores, where operations on the same semaphore must exclude each other in time. We can use CCR proof rules to reason about semaphores and, furthermore, the CCR notation gives us a simple technical way to treat "super-semaphores", semaphores that come with manipulations of auxiliary variables in order to keep track of certain quantities for the purposes of reasoning. The super-semaphore operations generally have the form

$$
\begin{array}{lcl}
\texttt{P}'(s) & = & \texttt{with}\ s\ \texttt{when}\ s > 0\ \texttt{do}\ \text{auxiliary assignments};\ s := s - 1\ \texttt{endwith} \\
\texttt{V}'(s) & = & \texttt{with}\ s\ \texttt{when}\ \texttt{true}\ \texttt{do}\ \text{auxiliary assignments};\ s := s + 1\ \texttt{endwith}.
\end{array}
$$

We stress that these auxiliary assignments will never affect control flow, and thus super-semaphores do not need to be implemented differently than ordinary semaphores; the auxiliary variables are a reasoning device, solely. Indeed, the proof rule for auxiliary variables in the next section allows these assignments to be deleted, once an overall property of a program has been proven. An example using super-semaphores will be given in Section 10.

Programs are subject to variable conditions for their well-formedness (from [39]). We say that a variable *belongs to* resource r if it is in the associated variable list in a resource declaration. We require that

1. a variable belongs to at most one resource;
2. if variable x belongs to resource r, it cannot appear in a parallel process except in a critical region for r; and
3. if variable x is changed in one process, it cannot appear in another unless it belongs to a resource.

For the third condition note that a variable x is changed by an assignment command $x := -$, but not by $[x] := E$; in the latter it is a heap cell, rather than a variable, that is altered.

The first two of these requirements mean that variables owned by resources are *protected*; they can only be accessed when within critical regions, which must be executed with mutual exclusion. The local variables of processes, those that don't belong to resources, are not protected by synchronization, but interference with or by them is ruled out by the third condition.

For example, the parallel compositions

$$x := 3 \parallel x := x + 1$$

and

$$x := 3 \parallel \texttt{with}\ r\ \texttt{when true do}\ x := x + 1$$

are both ruled out by the variable restrictions. Both of these are racy programs.

In the presence of pointers these syntactic restrictions are not enough to avoid interference. For example, in the legal program

$$[x] := 3 \parallel [y] := 4$$

if x and y denote the same integer in the starting state then they will be aliases, while if x and y are not aliases then there will be no race. In general, whether or not there is a race condition depends on the state, and we will use assertions to pick out states which guarantee absence of races.

As it happens, the restrictions do not even rule out more obvious race conditions where we use integer constants as addresses, such as

$$[10] := 3 \parallel [10] := 4.$$

Although it is difficult for a compiler to rule out interference in pointer programs, our proof rules will ensure that interference is precluded in any program we verify. Syntactic restrictions control interference through variables, while $*$ controls interference through heap cells.

5 Proof Rules

Now we give the proof rules for resource declarations, concurrency, and critical regions. To reason about a program

init;
`resource` r_1(variable list), ..., r_m(variable list)
$C_1 \parallel \cdots \parallel C_n$

we first specify a formula RI_{r_i}, the resource invariant, for each resource r_i. These formulae must satisfy

- any command $x := \cdots$ changing a variable x which is free in RI_{r_i} must occur within a critical region for r_i.

Owicki and Gries used a stronger condition, requiring that each variable free in RI_{r_i} belong to resource r_i. The weaker condition is due to Brookes, and allows a resource invariant to connect the value of a protected variable with the value of an unprotected one.

Also, for soundness, we require that each resource invariant is "precise". The definition of precise predicates is postponed until Section 11; we will never use any assertion that is not precise in any examples, so all examples will adhere to this restriction on invariants.

The inference rule is

$$\frac{\{P\}\mathit{init}\{RI_{r_1} * \cdots * RI_{r_m} * P'\} \qquad \{P'\}C_1 \parallel \cdots \parallel C_n\{Q\}}{\begin{array}{l}\{P\}\\ \mathit{init};\\ \texttt{resource}\ r_1(\text{variable list}), ..., r_m(\text{variable list})\\ C_1 \parallel \cdots \parallel C_n\\ \{RI_{r_1} * \cdots * RI_{r_m} * Q\}\end{array}}$$

Here, we show that the resource invariants are *separately* established by the initialization sequence, together with an additional portion of state that is given to the parallel processes for access outside of critical regions. These resource invariants are then removed from the pieces of state accessed directly by processes, and reestablished on conclusion.

The inference rule for parallel composition is

$$\frac{\{P_1\}\, C_1\, \{Q_1\}\ \cdots\ \{P_n\}\, C_n\, \{Q_n\}}{\{P_1 * \cdots * P_n\}\, C_1 \parallel \cdots \parallel C_n\, \{Q_1 * \cdots * Q_n\}}$$

where

- no variable free in P_i or Q_i is changed in C_j when $j \neq i$.

This rule is similar in form to the disjoint concurrency rule. The difference is that the reasoning that establishes the triples $\{P_i\}C_i\{Q_i\}$ for sequential processes is done in the context of an assignment of the resource invariants RI_{r_i} to resources r_i. This contextual assumption is used in the rule for critical regions.

$$\frac{\{(P * RI_r) \wedge B\}\, C\, \{Q * RI_r\}}{\{P\}\, \texttt{with}\ r\ \texttt{when}\ B\ \texttt{do}\ C\ \texttt{endwith}\, \{Q\}} \quad \begin{array}{l}\text{No other process modifies}\\ \text{variables free in } P \text{ or } Q\end{array}$$

The idea of this rule is that when inside a critical region the code gets to see the state associated with the resource as well as that local to the process it is part of, while when outside a region command reasoning proceeds without knowledge of the resource's state.

The side condition "No other process..." refers to the form of a program as composed of a fixed number of processes $C_1 \parallel \cdots \parallel C_n$. An occurrence of a `with` command will be in one of these processes C_i, and the condition means that there are no assignment commands $x := -$ in process C_j for $i \neq j$, whenever x is free in P or Q.

Brookes gives a more thorough, modern, presentation of the proof rules [12] . In particular, the contextual assumption that resource invariants have been chosen for each resource when reasoning about parallel processes is treated there using sequents

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

where Γ is a function mapping resource names to their invariants. This allows for a smooth treatment of nested resource declarations.

Finally, we will need a rule for deleting assignments to certain variables whose only purpose is to aid in reasoning.

> A set X of variables is auxiliary in a program if every occurrence of a variable $x \in X$ in the program is in a command $y := \cdots$ whose target y is in X. (The use of $\cdots$ here covers all of the forms $y := E$, $y := [E]$ and $y := \texttt{cons}(E_1, ..., E_n)$.)

Because an auxiliary variable never appears within a boolean expression or within an expression to the right of := when the left is non-auxiliary, an assignment to it cannot affect the flow of control. This gives rise to an inference rule for removing assignments to auxiliary variables [39].

> Suppose that X is an auxiliary variable set for program $\texttt{prog}'$ and that $\texttt{prog}$ is obtained from $\texttt{prog}'$ by removing all commands $x := \cdots$ to variables $x \in X$. Suppose further that no variable in X appears freely in assertions P and Q. Then
>
> $$\frac{\{P\}\texttt{prog}'\{Q\}}{\{P\}\texttt{prog}\{Q\}}$$

When this rule is applied to a program that uses super-semaphores the deletion of operations on auxiliary variables results in the super-semaphore operations $\texttt{P}'$ and $\texttt{V}'$ being mapped to the standard operations P and V.

Besides these proof rules we allow all of sequential separation logic; see the appendix. In treating examples we will not be completely formal in reasoning about the sequential commands, concentrating on the effect of the concurrency rules. But we will insert the most important intermediate assertions which show key steps in reasoning. After seeing the first example in Section 6 the reasoning steps might appear straightforward, and some readers may wish to skip over some of the detailed proof outlines later in the paper.

The soundness of proof rules for sequential constructs is delicate in the presence of concurrency. For instance, we can readily derive

$$\{10 \mapsto 3\}x := [10]; x := [10]\{(10 \mapsto 3) \wedge x = 3\}$$

in separation logic, but if there was interference from another process, say altering the contents of 10 between the first and second statements, then the triple would not be true.

The essential point is that proofs in our system build in the assumption that there is "no interference from the outside", in that processes only affect one another at explicit synchronization points. This mirrors a classic program design principle of Dijkstra, that "apart from the (rare) moments of explicit intercommunication, the individual processes are to be regarded as completely independent of each other" [17]. It allows us to ignore the minute details of potential interleavings of sequential programming constructs, thus greatly reducing the number of process interactions that must be accounted for in a verification.

The Ownership Hypothesis from Section 2.3 is essential in all of this, because attempting to access a piece of state you don't own might result in you trampling on the state of another process. The formal cousin of the Ownership hypothesis – that proven processes only dereference those cells that they own, those known to exist in a precondition for a program point – ensures that *well specified processes mind their own business*. This, combined with the use of $*$ to partition program states, implements Dijkstra's principle.

These intuitive statements about interference and ownership receive formal underpinning in Brookes's semantic model [12]. The most remarkable part of his analysis is an interplay between an interleaving semantics based on traces of actions and a "local enabling" relation that "executes" a trace in a portion of state owned by a process. The enabling relation skips over intermediate states and explains the "no interference from the outside" idea.

6 Binary Semaphore Example

As our first example of interacting processes we consider the semaphore code from Section 2.2. The proofs we do will be using rules derived from the standard encoding of semaphore operations in terms of CCRs given in Section 4.

Here again is the annotated code

$$
\begin{array}{lcl}
\vdots & & \vdots \\
\{\texttt{emp}\} & & \{\texttt{emp}\} \\
\texttt{P}(\mathit{free}); & & \texttt{P}(\mathit{busy}); \\
\{10 \mapsto -\} & & \{10 \mapsto -\} \\
[10] := m; & \| & n := [10]; \\
\{10 \mapsto -\} & & \{10 \mapsto -\} \\
\texttt{V}(\mathit{busy}); & & \texttt{V}(\mathit{free}); \\
\{\texttt{emp}\} & & \{\texttt{emp}\} \\
\vdots & & \vdots
\end{array}
$$

To verify the indicated pre and postconditions for $\texttt{P}(\cdot)$ and $\texttt{V}(\cdot)$ we must specify the semaphore invariants. For s being either *free* or *busy* the invariant is

$$RI_s \quad = \quad (s = 0 \wedge \texttt{emp}) \vee (s = 1 \wedge 10 \mapsto -).$$

In prose this invariant says "either s is 0 and no storage is owned, or s is 1 and the semaphore owns 10". Then, we have semaphore proof rules

$$\frac{\{(A * RI_s) \wedge s > 0\}\ s := s - 1\ \{A' * RI_s\}}{\{A\}\,\texttt{P}(s)\{A'\}}$$

$$\frac{\{A * RI_s\}\ s := s + 1\ \{A' * RI_s)\}}{\{A\}\,\texttt{V}(s)\{A'\}}$$

which are derived from the rule for CCRs.

Here is how to obtain the triple $\{\texttt{emp}\}\texttt{P}(\mathit{free})\{10 \mapsto -\}$, which appears above in the annotated code for the left process.

$$
\begin{array}{l}
\{(\texttt{emp} * ((\mathit{free} = 0 \wedge \texttt{emp}) \vee (\mathit{free} = 1 \wedge 10 \mapsto -))) \wedge \mathit{free} > 0\} \\
\{\mathit{free} = 1 \wedge 10 \mapsto -\} \\
\ \mathit{free} := \mathit{free} - 1 \\
\{\mathit{free} = 0 \wedge 10 \mapsto -\} \\
\{10 \mapsto - * (\mathit{free} = 0 \wedge \texttt{emp})\} \\
\{10 \mapsto - * ((\mathit{free} = 0 \wedge \texttt{emp}) \vee (\mathit{free} = 1 \wedge 10 \mapsto -))\}
\end{array}
$$

Consecutive assertions correspond to uses of the rule of consequence. The implication from the first to the second assertion holds because when we know that $\mathit{free} > 0$ then this, paired with the resource invariant, implies that the second disjunct of the invariant holds. Thus, the semaphore owns address 10. After *free* is decremented we must re-establish the invariant. We are forced to choose the first disjunct of the invariant if we are to obtain $10 \mapsto -$ as the postcondition of the operation; the reason is that since 10 is not dangling in the A' part of what we have to show, there is no way that the right disjunct in the invariant can hold. Thus, the separation in $*$ forces the semaphore to give up ownership of 10.

In the verification of $\texttt{V}(\mathit{free})$ the transfer happens in the other direction.

$$
\begin{array}{l}
\{10 \mapsto - * ((\mathit{free} = 0 \wedge \texttt{emp}) \vee (\mathit{free} = 1 \wedge 10 \mapsto -))\} \\
\{10 \mapsto - * (\mathit{free} = 0 \wedge \texttt{emp})\} \\
\ \mathit{free} := \mathit{free} + 1 \\
\{10 \mapsto - * (\mathit{free} = 1 \wedge \texttt{emp})\} \\
\{\texttt{emp} * (\mathit{free} = 1 \wedge 10 \mapsto -)\} \\
\{\texttt{emp} * ((\mathit{free} = 0 \wedge \texttt{emp}) \vee (\mathit{free} = 1 \wedge 10 \mapsto -))\}
\end{array}
$$

Here, in the first step since 10 is owned to the left of $*$ we know that the right disjunct of the resource invariant cannot hold, and thus *free* is 0. On conclusion we again re-establish the invariant in the only way open to us.

$\{10 \mapsto -\}$
$free := 1,\ busy := 0;$
$\{(free = 1 \wedge 10 \mapsto -) * (busy = 0 \wedge \texttt{emp})\}$
$\{RI_{free} * RI_{busy} * \texttt{emp} * \texttt{emp}\}$
`resource` $free(free),\ busy(busy);$
$\{\texttt{emp} * \texttt{emp}\}$

$\{\texttt{emp}\}$		$\{\texttt{emp}\}$
`while true do`		`while true do`
$\{\texttt{emp} \wedge \texttt{true}\}$		$\{\texttt{emp} \wedge \texttt{true}\}$
$\{\texttt{emp}\}$		$\{\texttt{emp}\}$
produce m		P($busy$);
$\{\texttt{emp}\}$		$\{10 \mapsto -\}$
P($free$);		$n := [10];$
$\{10 \mapsto -\}$		$\{10 \mapsto -\}$
$[10] := m;$	$\|$	V($free$);
$\{10 \mapsto -\}$		$\{\texttt{emp}\}$
V($busy$);		*consume* n
$\{\texttt{emp}\}$		$\{\texttt{emp}\}$
$\{\texttt{emp} \wedge \neg\texttt{true}\}$		$\{\texttt{emp} \wedge \neg\texttt{true}\}$
$\{\texttt{false}\}$		$\{\texttt{false}\}$

$\{\texttt{false} * \texttt{false}\}$
$\{RI_{free} * RI_{busy} * \texttt{false}\}$
$\{\texttt{false}\}$

Table 3: Producer/Consumer via Split Binary Semaphore

The verifications for the other semaphore operations are similar, and this completes our proof of the annotated code.

We have reasoned about the parallel processes themselves, but not yet their concurrent composition. To tie the two processes together using the concurrency rule we need to fill in the surrounding code. One way to do it is to wrap both code segments in infinite loops, together with assumed code that produces message m and that consumes n.

`semaphore` $free := 1,\ busy := 0;$		
`while true do`		`while true do`
produce m		P($busy$);
P($free$);		$n := [10];$
$[10] := m;$	$\|$	V($free$);
V($busy$);		*consume* n

Here, the `semaphore` declaration can be readily desugared as a `resource` declaration preceded by an initialization.

$free := 1,\ busy := 0;$
`resource` $free(free),\ busy(busy);$

We will explicitly use this desugaring in the proof figure below.

To prove a property of this program we assume that *produce* and *consume* do not require any use of the heap; this is reflected in their preconditions and postconditions below. (If *produce* and *consume* did require heap, other than address 10, we could use the frame rule to interface their specifications with the triples we have shown for the buffer-managing code.)

The verification is in Table 3. In writing annotated programs we generally include assertions at program points to show the important properties that hold; to formally connect to the proof theory we would sometimes have to apply an axiom followed by the Hoare rule of consequence

or other structural rules. Also, it should be clear how the nesting in this proof figure indicates where to apply various rules of inference. We first establish the resource invariants, and then remove them in the precondition for the parallel commands, as required by the rule for complete programs. Then we use the rule for parallel compositions, and in each process we use the properties of semaphore operations already established. In both processes `emp` is the chosen loop invariant.

At this point we can make some of the assumptions about our skeletal code clearer. First, for there not to be a race condition it is necessary for the semaphores to be initialized properly. Consider if we initialized both to 1. This faulty initialization is blocked by the proof rules, because we must establish the $RI_{free} * RI_{busy}$ part of the assertion before the parallel command, and this cannot hold when both *free* and *busy* are 1 because no address can be owned on two sides of $*$. Second, it is assumed that the address 10 not be dereferenced either immediately before or after the code snippets. In the logic, such dereferencing is ruled out by the assertion `emp`. For example, if we were to place $[10] := 42$ immediately after the $\texttt{V}(busy)$ command then we would have a race condition, but this code could not get past the proof rules because 10 must be pointing to something in the precondition of $[10] := 42$ in order to find a valid postcondition.

Remarks.

1. Notice that a semaphore invariant talks only about itself, not the other semaphore or the processes. This local point of view is a consequence of the resource reading of semaphores, where semaphores are (logically) attached to pieces of state.

2. *free* and *busy* together constitute a "split binary semaphore", where their sum is never greater than 1. Interestingly, we did not have to refer to the split property at all in our assertions and, in particular, we did not need to include $0 \leq free + busy \leq 1$ as an explicit global invariant, the truth of which was checked at each step. This is not because it is unimportant, but rather because the relevant information can be gleaned locally, from ownership. For instance, in the verification of $\texttt{P}(busy)$ when *busy* is greater than 0 we inferred, from the resource invariant, that *busy* owned 10. This then implies that *free* cannot own 10 and so, because of its resource invariant, it must at that point be 0.

3. Similarly, we do not check that the program satisfies the Separation Property at each step, and neither do we check that it satisfies the Ownership Hypothesis. Rather, the system is arranged to achieve these properties implicitly, for any program that gets past the proof rules.

4. We have proven $\{10 \mapsto -\}\texttt{program}\{\texttt{false}\}$ for our final program. At first this might seem a strange thing to do in a partial correctness formalism, but it is not vacuous: it ensures not just that the program loops, but also that there is no race condition and that no dangling pointers are dereferenced.

5. We would have preferred to use a variable instead of a heap address in this example; that is, normally one would have assignment commands $c := m$ and $n := c$ in place of the accesses to [10]. The problem is that this code using variables is disallowed by the variable conditions, because the accesses to c are not within the critical regions (the semaphore operations). Although CCRs can implement semaphores, the syntactic conditions required for soundness of the CCR proof rules in [23, 39, 1] rule out almost all semaphore programs, as they would commonly be written.

   The use of a heap address 10 in the semaphore code was, then, a cheeky choice on our part; we sneak in a heap address in place of a variable, and then we use the dynamic nature of $*$ to track the address as its ownership passes between the variables and the semaphores. This was to get around the fact that variables are treated in an entirely static manner by the side conditions used in Hoare logic; these result in scoping constraints that are set down before a program runs, and do not change as a program runs.

This raises the question of whether we could treat variables in the same way as heap addresses, splitting them with $*$, and letting their ownership transfer dynamically. This question has not yet been resolved to complete satisfaction – we will discuss the issue further in Section 12 – and for now we will push on and continue exploring the proof system, as formulated.

7 Pointer-transferring Buffer

The use of the idea of ownership transfer in the previous section might initially seem like a technical trick, to work around the unstructured nature of semaphore programs. Of course, we would claim that the view of a semaphore as a holder of resource (apart from the semaphore variable itself), with `P` and `V` operations as transformers of resource ownership, is intuitive. However, as we mentioned earlier, transfer is sometimes part of the very idea of how an entire program component works, not just an explanation of some of its internal constructs. In this section and the next we consider two such examples.

For efficient message passing it is often better to pass a pointer to a value from one process to another, rather than passing the value itself; this avoids unneeded copying of data. For example, in packet processing systems a packet is written to storage by one process, which then inserts a pointer to the packet into a message queue. The receiving process, after finishing with the packet, returns the pointer to a pool for subsequent reuse. Similarly, if a large file is to be transmitted from one process to another it can be better to pass a pointer than to copy its contents.

This section considers a pared-down version of this scenario, using a one-place buffer. We give a "structural integrity" proof which shows ownership properties of pointers, that the right parts of the code own a pointer at the right time. We do not attempt to specify full correctness. Then we give further examples to bring out some of the finer points of the proof system, having to do with the relative nature of ownership transfer and the use of auxiliary variables. These finer points, in Sections 7.2 and 7.3, may be safely skipped without a danger of loss of continuity.

In this and following sections we use operations `cons` and `dispose` for allocating and deleting *binary* cons cells. (To be more literal, `dispose`(E) in this section would be expanded into `dispose`(E); `dispose`$(E+1)$ in the syntax of Section 4.) We also use a dot notation for field selection rather than explicit address arithmetic, using $x.1$ and $x.2$ for $[x]$ and $[x+1]$. So, for example, we write $f := x.2$ instead of $f := [x+1]$.

7.1 A Structural Integrity Proof

We repeat the buffer code from earlier, where we have

$full :=$ `false`;
`resource` $buf(c, full)$;

and standard code for putting a value into the buffer and for reading it out.

`put`$(x) \triangleq$ `with` buf `when` $\neg full$ `do`
$c := x$; $full :=$ `true`
`endwith`

`get`$(y) \triangleq$ `with` buf `when` $full$ `do`
$y := c$; $full :=$ `false`
`endwith`;

We focus on the following code.

$x :=$ `cons`(a, b);	$\|$	`get`(y);
`put`(x);		$use(y)$;
		`dispose`(y);

This creates a new pointer in one process, which points to a binary cons cell containing values a and b. To transmit these values to the other process, instead of copying both a and b the pointer itself is placed in the buffer. The second process reads the pointer out, uses it in some way, and finally disposes it. To reason about the `dispose` operation in the second process, we must ensure that $y \mapsto -,-$ holds beforehand.

As it stands this code is completely sequential: the left process must go first. Since we have assumed that *full* has been initialized to `false` above, the code for $\texttt{get}(y)$ cannot proceed until $\texttt{put}(x)$ has been done. Like in the preceding section we can have surrounding code, including these code snippets in loops, as in

$$
\begin{array}{l}
\{\texttt{emp}\} \\
\mathit{full} := \texttt{false}; \\
\texttt{resource}\ \mathit{buf}(c, \mathit{full}) \\
\begin{array}{lcl}
\texttt{while}\ \texttt{true}\ \texttt{do} & & \texttt{while}\ \texttt{true}\ \texttt{do} \\
\mathit{produce}(a,b); & & \texttt{get}(y); \\
x := \texttt{cons}(a,b); & \| & \mathit{use}(y); \\
\texttt{put}(x); & & \texttt{dispose}(y);
\end{array} \\
\{\texttt{false}\}
\end{array}
$$

It should be clear how move to a proof of this program from the properties we display of the snippets within the loops (as we did in Table 3), so in what follows we will just concentrate on proving properties of the snippets.

The resource invariant for the buffer is

$$RI_{buf}: \quad (\mathit{full} \wedge c \mapsto -,-) \vee (\neg \mathit{full} \wedge \texttt{emp}).$$

This invariant says that the buffer owns the binary cons cell associated with c when *full* is true, and otherwise it owns no heap cells.

Here is a proof for the body of the region command in $\texttt{put}(x)$.

$$
\begin{array}{l}
\{(RI_{buf} * x \mapsto -,-) \wedge \neg \mathit{full}\} \\
\{(\neg \mathit{full} \wedge \texttt{emp}) * x \mapsto -,-\} \\
\{x \mapsto -,-\} \\
\ c := x;\ \mathit{full} := \texttt{true} \\
\{\mathit{full} \wedge c \mapsto -,-\} \\
\{RI_{buf}\} \\
\{RI_{buf} * \texttt{emp}\}
\end{array}
$$

The rule for region commands then gives us

$$\{x \mapsto -,-\}\texttt{put}(x)\{\texttt{emp}\}.$$

A crucial point in the proof of the body is the implication

$$\mathit{full} \wedge c \mapsto -,- \ \Rightarrow\ RI_{buf},$$

which is applied in the penultimate step. This step reflects the idea that the knowledge "x points to something" flows out of the user program and into the resource. On exit from the critical region x does indeed point to something in the global state, but this information cannot be recorded in the postcondition of `put`. The reason is that we used $c \mapsto -,-$ to re-establish the resource invariant; having $x \mapsto -,-$ as the postcondition would be tantamount to asserting $(x \mapsto -,-) * (c \mapsto -,-)$ at the end of the body of the `with` command, and this assertion is necessarily false when c and x are equal, as they are at that point.

The flipside of the first process giving up ownership is the second's assumption of it:

$\{(RI_{buf} * \texttt{emp}) \wedge full\}$
$\{full \wedge c \mapsto -,-\}$
$y := c$; $full := \texttt{false}$
$\{y \mapsto -,- \wedge \neg full\}$
$\{(\neg full \wedge \texttt{emp}) * y \mapsto -,-\}$
$\{RI_{buf} * \ y \mapsto -,-\}$,

which gives us

$\{\texttt{emp}\}\texttt{get}(y)\{y \mapsto -,-\}$.

We can then prove the parallel processes as follows, assuming that $use(y)$ satisfies the indicated triple.

$\{\texttt{emp} * \texttt{emp}\}$

$\{\texttt{emp}\}$		$\{\texttt{emp}\}$
$x := \texttt{cons}(a, b);$	$\|$	$\texttt{get}(y);$
$\{x \mapsto -,-\}$		$\{y \mapsto -,-\}$
$\texttt{put}(x);$		$use(y);$
$\{\texttt{emp}\}$		$\{y \mapsto -,-\}$
		$\texttt{dispose}(y)$
		$\{\texttt{emp}\}$

$\{\texttt{emp} * \texttt{emp}\}$
$\{\texttt{emp}\}$

Then using the fact that the initialization establishes the resource invariant in a way that gets us ready for the parallel rule

$\{\texttt{emp}\}$
$full := \texttt{false}$
$\{\neg full \wedge \texttt{emp}\}$
$\{RI_{buf}\}$
$\{RI_{buf} * \texttt{emp} * \texttt{emp}\}$

we obtain the following triple for the complete program `prog`:

$\{\texttt{emp}\}\texttt{prog}\{RI_{buf}\}$.

In this verification we have concentrated on tracking ownership, using assertions that are type-like in nature: they say what kind of data exists at various program points, but not the identities of the data. For instance, because the assertions use $-,-$ they do not track the flow of the values a and b from the left to the right process. So, we have proven a "structural integrity" property of the code, which implies that there is no race condition, but we have not shown real correctness.

In this simple example we could track the values a and b as they flow through the buffer by changing the resource invariant to have $c \mapsto a, b$ instead of $c \mapsto -,-$. In more realistic examples, when the code snippets are parts of loops, we would have to use auxiliary variables [39]. Another use of auxiliary variables is given below in Section 7.3.

7.2 Ownership is in the Eye of the Asserter

Transfer of ownership is not something that is determined operationally. Whether we transfer the storage associated with a single address, a segment, a linked list, or a tree depends on what we want to prove. Of course, what we can prove is constrained by code outside of critical regions. For example, in the program that used the buffer code

$x := \texttt{cons}(a, b);$	$\|$	$\texttt{get}(y);$
$\texttt{put}(x);$		$use(y);$
		$\texttt{dispose}(y);$

our hand was forced by the $\texttt{dispose}(y)$ command in the second parallel process.

As an extreme case, if we dispose in the first rather than the second process, and we don't attempt to use y in the second process, then we have to ensure that no storage transfers with the pointer's value.

$$\begin{array}{lcl} x := \texttt{cons}(a,b); & \| & \texttt{get}(y); \\ \texttt{put}(x); & & \\ \texttt{dispose}(x); & & \end{array}$$

This is a silly program, but we should be able to prove it, and we can by choosing a different resource invariant:

$$RI: \quad (\mathit{full} \wedge \texttt{emp}) \vee (\neg \mathit{full} \wedge \texttt{emp}).$$

The use of `emp` in the left disjunct prevents ownership of the storage associated with x from flowing into the buffer.

This new resource invariant leads to different specifications of `get` and `put`:

$$\{x \mapsto -,-\}\texttt{put}(x)\{x \mapsto -,-\}$$
$$\{\texttt{emp}\}\texttt{get}(x)\{\texttt{emp}\}.$$

With the new resource invariant these specifications can be proven using the rule for `with` commands. We omit the details, and just give the proof figure for the parallel composition.

$$\begin{array}{lcl} & \{\texttt{emp}\} & \\ & \{\texttt{emp} * \texttt{emp}\} & \\ \{\texttt{emp}\} & & \{\texttt{emp}\} \\ x := \texttt{cons}(a,b); & \| & \texttt{get}(y); \\ \{x \mapsto -\} & & \{\texttt{emp}\} \\ \texttt{put}(x); & & \\ \{x \mapsto -,-\} & & \\ \texttt{dispose}(x); & & \\ \{\texttt{emp}\} & & \\ & \{\texttt{emp} * \texttt{emp}\} & \\ & \{\texttt{emp}\} & \end{array}$$

We have seen that memory ownership can either transfer with a pointer's value, or stay located in the sending process, depending on what we want to prove. A final point is that it is not possible for the ownership to go both ways. For example, we cannot find a resource invariant that lets us prove

$$\begin{array}{lcl} x := \texttt{cons}(a,b); & \| & \texttt{get}(y); \\ \texttt{put}(x); & & \texttt{dispose}(y); \\ \texttt{dispose}(x); & & \end{array}$$

as is fortunate since this program attempts to dispose the same pointer in both processes.

The reason why there is no resource invariant letting us prove this program is that the separating nature of $*$ will not let ownership of a pointer *both* flow into the buffer *and* stay located in the sending process. More formally, since Brookes has shown that any program that gets past our proof rules is race-free, and since this program has a race, it cannot be verified (except with `false` as the precondition).

7.3 Avoiding Memory Leak using Auxiliary Variables

Returning to the end of Section 7.1, we proved there a weaker property than might have been expected. That is, we only obtained RI in the postcondition, and the resource invariant in that section was a disjunction, one of whose components included $c \mapsto -$. But, given the way the

program works, we know the heap will be empty on termination; there is a small memory leak in the specification. To fix this, we would like to get `emp` in the postcondition.

The problem is that we have lost the information that the buffer is empty when we exit the critical region of the second process. And we cannot include $\neg full$ in the postcondition, because this would violate the variable restriction that is essential for the soundness of the `with` rule.

To handle the problem of memory leak, we must first verify a more complex program that uses additional variables to record control information. Then we use the Auxiliary Variable Rule to delete those variables to infer a more exact property for the original program.

The new program has two additional variables, *start* and *finish*, and begins with the following initialization sequence and `resource` command.

$full := \texttt{false}$; $start := \texttt{true}$; $finish := \texttt{false}$;
`resource` $buf(c, full, start, finish)$;

We assign to the extra variables within the critical regions for buffer management.

$$\begin{array}{lll} \texttt{put}'(x) & \triangleq & \texttt{with } buf \texttt{ when } \neg full \texttt{ do} \\ & & \quad c := x;\ full := \texttt{true};\ start := \texttt{false} \\ & & \texttt{endwith} \end{array}$$

$$\begin{array}{lll} \texttt{get}'(y) & \triangleq & \texttt{with } buf \texttt{ when } full \texttt{ do} \\ & & \quad y := c;\ full := \texttt{false};\ finish := \texttt{true} \\ & & \texttt{endwith}; \end{array}$$

The new resource invariant is

$$RI: \quad \begin{array}{l} (\neg full \wedge start \wedge \neg finish \wedge \texttt{emp}) \\ \vee (full \wedge \neg start \wedge \neg finish \wedge c \mapsto -,-) \\ \vee (\neg full \wedge \neg start \wedge finish \wedge \texttt{emp}). \end{array}$$

The three disjuncts correspond to what will be three control points: before `put'` begins; after `put'` ends and before `get'` begins; after `get'` ends.

Now, we can reason about the parallel composition as follows, where the triples for `put'` and `get'` are established in Table 4

$$\begin{array}{lcl} & \{\texttt{emp} \wedge start \wedge \neg finish\} & \\ & \{(\texttt{emp} \wedge start) * (\texttt{emp} \wedge \neg finish)\} & \\ \{\texttt{emp} \wedge start\} & & \{\texttt{emp} \wedge \neg finish\} \\ x := \texttt{cons}(-,-); & \| & \texttt{get}'(y); \\ \{x \mapsto -,- \wedge start\} & & \{y \mapsto -,- * (\texttt{emp} \wedge finish)\} \\ \texttt{put}'(x); & & use(y); \\ \{\texttt{emp} \wedge \neg start\} & & \{y \mapsto -,- * (\texttt{emp} \wedge finish)\} \\ & & \texttt{dispose}(y); \\ & & \{\texttt{emp} \wedge finish\} \\ & \{(\texttt{emp} \wedge \neg start) * (\texttt{emp} \wedge finish)\} & \\ & \{\texttt{emp} \wedge \neg start \wedge finish\} & \end{array}$$

Next, the initialization sequence establishes $\{RI * (\texttt{emp} \wedge start \wedge \neg finish)\}$, so if we denote by `prog'` the initialization sequence and resource declaration of this section, followed by the parallel composition, we we use the rule for complete programs to obtain

$$\{\texttt{emp}\}\ \texttt{prog'}\ \{RI * (\texttt{emp} \wedge \neg start \wedge finish)\}.$$

In the postcondition since $\neg start \wedge finish$ holds we know that the third of the disjuncts in RI must be true, so by the rule of consequence we get

$$\{\texttt{emp}\}\ \texttt{prog'}\ \{\texttt{emp}\}.$$

PROOF ESTABLISHING $\{x \mapsto -,- \wedge \mathit{start}\}\texttt{put}'(x)\{\texttt{emp} \wedge \neg \mathit{start}\}$:

$\{(RI * (x \mapsto -,- \wedge \mathit{start})) \wedge \neg \mathit{full}\}$
$\{(\neg \mathit{full} \wedge \mathit{start} \wedge \neg \mathit{finish} \wedge \texttt{emp}) * x \mapsto -,-\}$
$c := x$; $\mathit{full} := \texttt{true}$; $\mathit{start} := \texttt{false}$
$\{(\mathit{full} \wedge \neg \mathit{start} \wedge \neg \mathit{finish} \wedge \texttt{emp}) * c \mapsto -,-\}$
$\{(\mathit{full} \wedge \neg \mathit{start} \wedge \neg \mathit{finish} \wedge c \mapsto -,-) * (\texttt{emp} \wedge \neg \mathit{start})\}$
$\{RI * (\texttt{emp} \wedge \neg \mathit{start})\}$

PROOF ESTABLISHING $\{\texttt{emp} \wedge \neg \mathit{finish}\}\texttt{get}'(y)\{(y \mapsto -,-) * (\texttt{emp} \wedge \mathit{finish})\}$:

$\{(RI * (\texttt{emp} \wedge \neg \mathit{finish})) \wedge \mathit{full}\}$
$\{\mathit{full} \wedge \neg \mathit{start} \wedge \neg \mathit{finish} \wedge c \mapsto -,-\}$
$y := c$; $\mathit{full} := \texttt{false}$; $\mathit{finish} := \texttt{true}$
$\{\neg \mathit{full} \wedge \neg \mathit{start} \wedge \mathit{finish} \wedge y \mapsto -,-\}$
$\{(\neg \mathit{full} \wedge \neg \mathit{start} \wedge \mathit{finish} \wedge \texttt{emp}) * (y \mapsto -,-) * (\texttt{emp} \wedge \mathit{finish})\}$
$\{RI * (y \mapsto -,-) * (\texttt{emp} \wedge \mathit{finish})\}$

Table 4: Proofs for put' and get'

The variables *start* and *finish* occur only in commands of the form $\mathit{start} := E$ or $\mathit{finish} := E$. They thus do not affect the flow of control, and are auxiliary variables in the sense of [39]. Further, neither the pre nor postcondition of prog' mentions *start* or *finish*. Therefore, we can use the Auxiliary Variable Rule to remove all commands involving these variables in prog', as well as their positions in the resource declaration, and we obtain

$$\{\texttt{emp}\}\,\texttt{prog}\,\{\texttt{emp}\}$$

where prog is the program from Section 7.1. We have finally proven the exact property we were after.

8 Memory Manager

The binary semaphore and pointer-transferring buffer examples are all-or-nothing: a piece of storage is either completely owned by a protected resource, or completely owned in user code. We now consider an example where portions are added to and broken off from the protected resource, a little at a time.

A resource manager keeps track of a pool of resources, which are given to requesting processes, and received back for reallocation. As an example of this we consider a toy manager, where the resources are memory chunks of size two. The manager maintains a free list, which is a singly-linked list of binary cons cells. The free list is pointed to by f, which is part of the declaration

$\texttt{resource}\ \mathit{mm}(f)$.

The invariant for *mm* is just that f points to a singly-linked list without any dangling pointers in the link fields:

$RI_{mm}:\ \ \mathit{list}\ f$.

The *list* predicate is the least satisfying the following recursive specification.

$$\mathit{list}\,x \iff^{\Delta} (x = \texttt{nil} \wedge \texttt{emp}) \vee (\exists y.\, x \mapsto -, y * \mathit{list}\,y)$$

PROOF ESTABLISHING $\{\texttt{emp}\}\texttt{alloc}(x,a,b)\{x \mapsto a,b\}$:

$\{\texttt{emp} * \mathit{list}\, f\}$
$\{\mathit{list}\, f\}$
 $\texttt{if}\, f = \texttt{nil}\, \texttt{then}$
  $\{\mathit{list}\, f \wedge f = \texttt{nil}\}$
  $\{f = \texttt{nil} \wedge \texttt{emp}\}$
  $x := \texttt{cons}(a,b)$
  $\{(x \mapsto a,b) * (f = \texttt{nil} \wedge \texttt{emp})\}$
  $\{(x \mapsto a, b * \mathit{list}\, f\}$
 $\texttt{else}$
  $\{\mathit{list}\, f \wedge f \neq \texttt{nil}\}$
  $\{\exists y.\, f \mapsto -, y * \mathit{list}\, y\}$
  $x := f;\ f := x.2;\ x.1 := a;\ x.2 := b$
  $\{(x \mapsto a,b) * \mathit{list}\, f\}$
$\{(x \mapsto a,b) * \mathit{list}\, f\}$

PROOF ESTABLISHING $\{y \mapsto -,-\}\texttt{dealloc}(y)\{\texttt{emp}\}$:

$\{(y \mapsto -,-) * \mathit{list}\, f\}$
$y.2 := f;$
$f := y;$
$\{\mathit{list}\, f\}$
$\{\texttt{emp} * \mathit{list}\, f\}$

Table 5: Proofs for Alloc and Dealloc

When a user program asks for a new cell, *mm* gives it a pointer to the first element of the free list, if the list is nonempty. In case the list is empty the *mm* calls `cons` to get an extra element.

$$\begin{aligned}\texttt{alloc}(x,a,b) \triangleq\ & \texttt{with}\ \mathit{mm}\ \texttt{when}\ \texttt{true}\ \texttt{do}\\ & \texttt{if}\ f = \texttt{nil}\ \texttt{then}\ x := \texttt{cons}(a,b)\\ & \texttt{else}\ x := f;\ f := x.2;\ x.1 := a;\ x.2 := b\end{aligned}$$

$$\begin{aligned}\texttt{dealloc}(y) \triangleq\ & \texttt{with}\ \mathit{mm}\ \texttt{when}\ \texttt{true}\ \texttt{do}\\ & y.2 := f;\\ & f := y;\end{aligned}$$

(We remind the reader that in this code we are using the notation specified at the beginning of Section 7, where we are using field selection instead of address arithmetic; e.g., $y.2 := f$ means the same as $[y+1] := f$ in the RAM model.

Using the rule for `with` commands (Table 5) we obtain the following "interface specifications":

$$\{\texttt{emp}\}\texttt{alloc}(x,a,b)\{x \mapsto a,b\} \qquad \{y \mapsto -,-\}\texttt{dealloc}(y)\{\texttt{emp}\}.$$

The specification of $\texttt{alloc}(x,a,b)$ illustrates how ownership of a pointer materializes in the user code, for subsequent use. Conversely, the specification of `dealloc` requires ownership to be given up. The proofs of the bodies of these operations using the `with` rule describe ownership transfer in much the same way as in the previous section, and are omitted.

Since we have used a critical region to protect the free list from corruption, it should be possible to have parallel processes that interact with *mm*. A tiny example of this is just two processes, each of which allocates, mutates, then deallocates.

$$
\begin{array}{lcl}
 & \{\texttt{emp} * \texttt{emp}\} & \\
\{\texttt{emp}\} & & \{\texttt{emp}\} \\
\texttt{alloc}(x,a,b); & & \texttt{alloc}(y,a',b'); \\
\{x \mapsto a,b\} & & \{y \mapsto a',b'\} \\
x.1 := 4; & \| & y.1 := 7; \\
\{x \mapsto 4,b\} & & \{y \mapsto 7,b'\} \\
\texttt{dealloc}(x); & & \texttt{dealloc}(y); \\
\{\texttt{emp}\} & & \{\texttt{emp}\} \\
 & \{\texttt{emp} * \texttt{emp}\} & \\
 & \{\texttt{emp}\} &
\end{array}
$$

This little program is an example of one that is daring but still safe. To see the daring aspect, consider an execution where the left process goes first, right up to completion, before the right one begins. Then the statements mutating $x.1$ and $y.1$ will in fact alter the same cell, and these statements are not within critical regions. However, although there is potential aliasing between x and y, the program proof tells us that there is no possibility of racing in *any* execution.

On the other hand, if we were to insert a command $x.1 := 8$ immediately following $\texttt{dealloc}(x)$ in the leftmost process then we would indeed have a race. However, the resulting program would not get past the proof rules, because the postcondition of $\texttt{dealloc}(x)$ is `emp`.

Unsafe daring examples arise when when one process calls `dealloc` while another uses `alloc`, and when there is other dereferencing.

$$
\begin{array}{lcl}
\vdots & & \vdots \\
x.1 := 42; & \| & \texttt{alloc}(y,a,b); \\
\texttt{dealloc}(x); & & y.1 := 7; \\
\vdots & & \vdots
\end{array}
$$

In this case if the left process goes first the allocated cell might very well be the same address just freed, in which case we have daring concurrency. However, although there is potential aliasing between x and y, with the specifications of `dealloc` and `alloc` we rule out all possibility of racing.

$$
\begin{array}{lcl}
\vdots & & \vdots \\
\{x \mapsto -,-\} & & \{\texttt{emp}\} \\
x.1 := 42; & \| & \texttt{alloc}(y,a,b); \\
\{x \mapsto -,-\} & & \{y \mapsto a,b\} \\
\texttt{dealloc}(x); & & y.1 := 7; \\
\{\texttt{emp}\} & & \{y \mapsto 1,b\} \\
\vdots & & \vdots
\end{array}
$$

In particular, note that in a command immediately following $\texttt{dealloc}(x)$ we could not dereference x, because the postcondition of the command is `emp`.

We finish this part with an example that is excluded by our proof rules. Consider a parallel composition of size one, which tries to corrupt the memory manager.

$$
\begin{array}{l}
\texttt{alloc}\ (z); \\
\texttt{dealloc}\ (z); \\
z.2 := z;
\end{array}
$$

This code makes use of knowledge of the way `dealloc` works; after z is put back on the front of the free list, the program creates a cycle.

It is impossible to verify this code in our formalism with any precondition other than `false`. The reason is that the proof rule for region commands would require us to reason about the body of `dealloc` as follows

$\{list\ f * z \mapsto -, -\}$
$z.2 := f;$
$f := z;$
$\{list\ f * A * (z.2 \mapsto -)\}$

for some A, where $z.2 \mapsto -$ means that we own the cdr of cons cell z (in the RAM, this is $z+1 \mapsto -$). This is because if the command $z.2 := z$ is to verified, we must have $z.2 \mapsto -$ in its precondition (unless that precondition is `false`). But after $z.2 := f$; $f := z$ the formula $\{list\ f * A * (z.2 \mapsto -)$ cannot hold, since $f = z$.

The same principle applies with less malicious, but still erroneous, programs. A program might, through using a disposed pointer or simply through an address arithmetic error, corrupt the free list. Such an error would in principle show up in a failed verification attempt, since such corruption is ruled out by our formalism.

Finally, the issues discussed in this section are not exclusive to memory managers. When using a connection pool or a thread pool in a web server, for example, once a handle is returned to the pool the returning process must make sure not to use it again, or inconsistent results may ensue.

9 Combining the Buffer and Memory Manager

We now show how to put the treatment of the buffer together with the homegrown memory manager *mm*, using `alloc` and `dealloc` instead of `cons` and `dispose`. The aim is to show different resources interacting in a modular way.

We presume now that we have the resource declarations for both *mm* and *buf*, and their associated resource invariants. Here is the proof for the parallel processes in Section 7 done again, this time using `mm`.

$\{\texttt{emp} * \texttt{emp}\}$

$\{\texttt{emp}\}$		$\{\texttt{emp}\}$
$\texttt{alloc}(x, a, b);$	$\|\|$	$\texttt{get}(y);$
$\{x \mapsto -, -\}$		$\{y \mapsto -, -\}$
$\texttt{put}(x);$		$use(y);$
$\{\texttt{emp}\}$		$\{y \mapsto -, -\}$
		$\texttt{dealloc}(y);$
		$\{\texttt{emp}\}$

$\{\texttt{emp} * \texttt{emp}\}$
$\{\texttt{emp}\}$

In this code, a pointer's ownership is first transferred out of the *mm* resource into the lefthand user process. It is then sent into the *buf* resource, from where it is subsequently taken out by the righthand process and promptly returned to *mm*.

The initialization sequence and resource declaration now have the form

$full := \texttt{false};$
`resource` $buf(c, full),\ mm(f);$

and we have the triple

$\{\texttt{list}\ f\}\ full := \texttt{false}\ \{RI_{buf} * RI_{mm} * \texttt{emp} * \texttt{emp}\}$

which sets us up for reasoning about the parallel composition. We can use the rule for complete programs to obtain a property of the complete program.

The point is that we did not have to change any of the code or verifications done with *mm* or with *buf* inside the parallel processes; we just used the same preconditions and postconditions for `get`, `put`, `alloc` and `dealloc`, as given to us by the proof rule for CCRs. The crucial point is that the rule for CCRs does not include the resource invariant in the "interface specification" described by the conclusion of the rule. As a result, a proof using these specifications does not

need to be repeated, even if we change the implementation and internal resource invariant of a module. Effective resource separation allows us to present a localized view, where the state of a resource is hidden from user programs (when outside critical regions).

10 Counting Semaphore Example

Our final example uses counting semaphores to implement an unbounded buffer. Before translating it into our formalism it will be helpful to first picture the code as originally presented in [17].

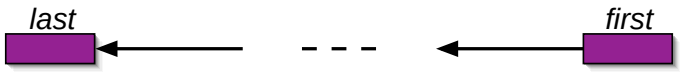
```
begin integer number of queuing portions;
          number of queuing portions := 0;
          parbegin
          producer: begin
                    again 1: produce the next portion;
                             add portion to buffer;
                             V(number of queuing positions);
                             goto again 1;
                    end;
           consumer: begin
                    again 2: P(number of queuing positions);
                             take portion from buffer;
                             process portion taken;
                             goto again 2;
                    end;
           parend;
end;
```

In this code the P(·) command in the second process stops the consumer from accessing the buffer when it is empty. It is possible, though, for the producer to get ahead of the consumer by putting a number of portions into the buffer while the consumer is, say, either delayed or busy processing a previously taken portion.

We will program the buffer manipulations using pointers. The buffer will be represented by a linked list, adding a portion to a buffer will involve allocation at one end of the list, and removing an element will involve reading and then deallocating an element from the other end of the list.

In order for the `add portion` and `take portion` operations to work correctly we must be careful that they not race. In [17] this is achieved by wrapping these operations in mutex semaphore operations, using the matching P(*mutex* code V(*mutex*) idiom, thus ensuring mutual exclusion. In the implementation that we will give, mutual exclusion is stronger than necessary: when there is at least one element in the queue, it will be okay for the operations to proceed concurrently. In doing this we want to be careful that there are no races.

The data structure we use will be a non-empty linked list from *first* to *last*, which implements a fifo queue.



The list is pictured in a backwards-going direction because the producer process, which will be written to the left of ‖, accesses the *last* element, to put a message on the end of the queue, while the consumer process, on the right of ‖, takes the *first* element off. The element *last* is a dummy node. Adding a portion to the buffer places that portion in the car of *last*, then calls an allocator to get a new cell, and links that cell into the list making it the last (and dummy). Removing a portion results in a value being read from the car of the *first* node, giving this node back to the allocator, and moving *first* along the list by one. When the list is of length one *last* and *first* will

PREDICATE DEFINITION

$$ls[x\,n\,z] \quad \stackrel{\Delta}{\Longleftrightarrow} \quad (x = z \wedge n = 0 \wedge \texttt{emp}) \\ \vee \quad (x \neq z \wedge n > 0 \wedge \exists y.\, x \mapsto -, y * ls[y\,(n-1)\,z]$$

RESOURCE INVARIANT

$$RI \stackrel{\Delta}{=} ls[f\ number\,l]$$

PROOF ESTABLISHING $\{f = first \wedge \texttt{emp}\}\texttt{P}'(number)\{first \mapsto -, f\}$:

$\{(f = first \wedge \texttt{emp}) * ls[f\ number\,l] \wedge number > 0\}$
$\{\exists y.\, first \mapsto -, y * ls[y\ (number - 1)\ l]\}$
$f := first.2;$
$\{first \mapsto -, f * ls[f\ (number - 1)\,l)\}$
$number := number - 1;$
$\{first \mapsto -, f * ls[f\ number\,l]\}$

PROOF ESTABLISHING $\{l \mapsto -, last * last \mapsto -, -\}\texttt{V}'(number)\{l = last \wedge last \mapsto -, -\}$:

$\{l \mapsto -, last * last \mapsto -, - * ls[f\ number\,l]\}$
$\{last \mapsto -, - * ls[f\ number\,l] * l \mapsto -, last\}$
$\{(last \mapsto -, -) * ls[f\ (number + 1)\ last]\}$
$\{(last = last \wedge last \mapsto -, -) * ls[f\ (number + 1)\ last]\}$
$l := last;$
$\{(l = last \wedge last \mapsto -, -) * ls[f\ (number + 1)\ l]\}$
$number := number + 1$
$\{(l = last \wedge last \mapsto -, -) * ls[f\ number\,l]\}$

Table 6: Proofs for Super-Semaphore Operations

be equal, and this will correspond to an empty buffer; it is in this case that synchronization is necessary to protect from races.

The relevant parts of the code are as follows.

```
    semaphore number := 0;
⋮                                   ⋮
produce(m)                          P(number);
last.1 := m;                        n := first.1; t := first;
last.2 := cons(–, –);       ||      first := first.2
last := last.2                      dispose(t);
V(number)                           consume(n)
⋮                                   ⋮
```

In this code the commands in the two processes do not necessarily exclude one another in time when *number* is greater than 0. So, for example, we might be executing both $last.1 := m$ and $n := first.1$, and we must beware of aliasing in order to avoid a race condition. Even more dangerous is the concurrency of $last.2 := \texttt{cons}(-, -)$ and $\texttt{dispose}(t)$ which, in the case of aliasing, would possibly corrupt the free list.

To describe the state owned by *number* requires the use of two auxiliary variable, f and l. These variables are updated in super-semaphore operations which, for the purpose of reasoning,

replace the standard operations in the code.

$$\begin{array}{lcl} \mathtt{P}'(\mathit{number}) & \triangleq & \mathtt{with}\ \mathit{number}\ \mathtt{when}\ \mathit{number} > 0\ \mathtt{do}\ f := \mathit{first}.2;\ \mathit{number} := \mathit{number} - 1 \\ \mathtt{V}'(\mathit{number}) & \triangleq & \mathtt{with}\ \mathit{number}\ \mathtt{when}\ \mathtt{true}\ \mathtt{do}\ l := \mathit{last};\ \mathit{number} := \mathit{number} + 1. \end{array}$$

The idea of the assignment $f := \mathit{first}.2$ is that the semaphore cannot own *first* after the $\mathtt{P}'$ operation completes, because ownership will be released into the following code for reading and then disposal. When ownership of this single cell is released, what *number* owns moves along by one in the list. Note that, since *first* is altered in the following code, it cannot be used to describe the resource invariant. Similarly, the assignment $l := \mathit{last}$ takes a snapshot of a quantity that will be altered after $\mathtt{V}'$ is exited.

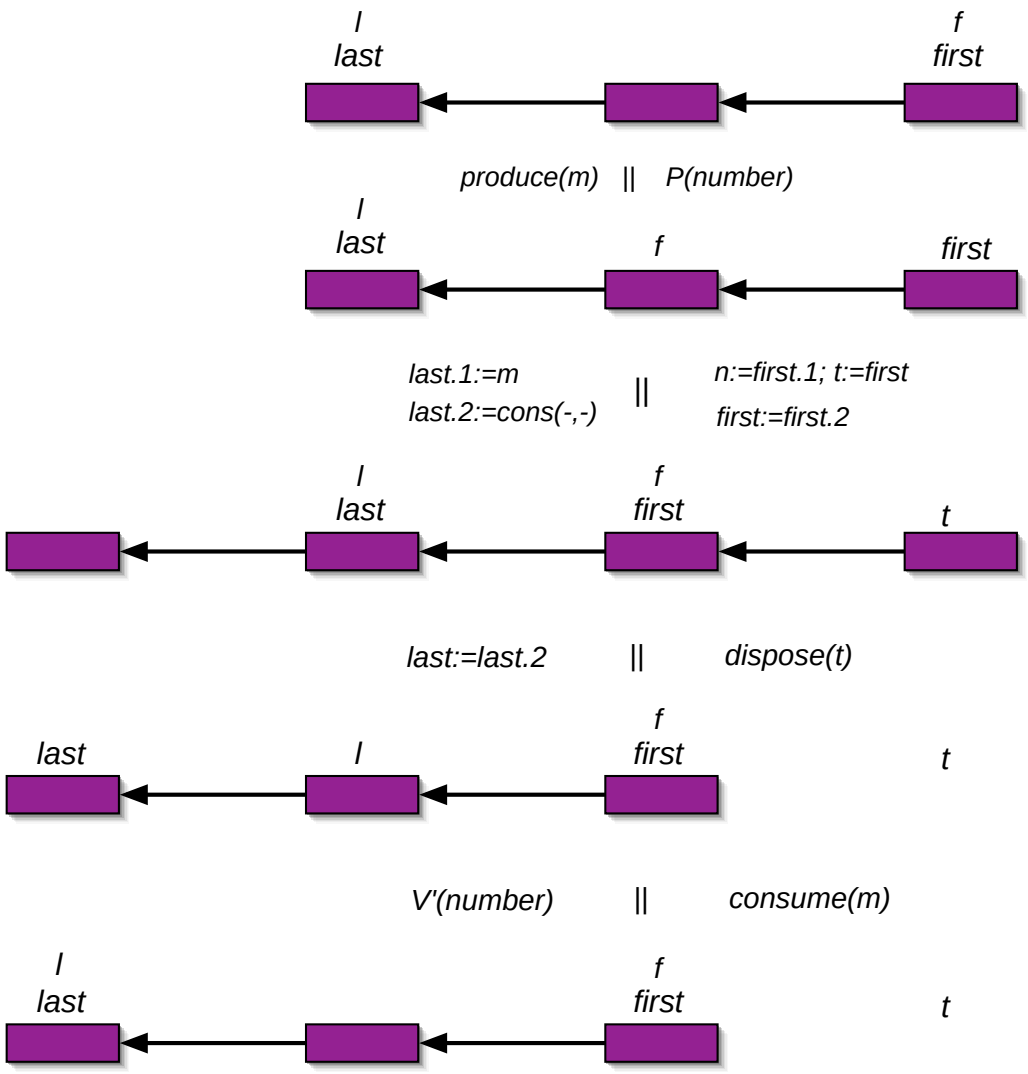
The resource invariant says that *number* owns a linked list segment of size *number* running from f to l (but not including l). The definition of this predicate, and proof outlines for $\mathtt{P}'$ and $\mathtt{V}'$, are in Table 6; since $\mathtt{P}'$ and $\mathtt{V}'$ are represented as CCRs, we are just appealing to the proof rule for `with`. Some of the steps using the rule of consequence in the table require induction to show.

The proofs in Table 6 provide the correct pre and postconditions for the occurrences of $\mathtt{P}'$ and $\mathtt{V}'$ in the following.

$$\begin{array}{l} \{l = \mathit{last} \wedge \mathit{last} \mapsto -,-\} \\ \mathit{produce}(m) \\ \{l = \mathit{last} \wedge \mathit{last} \mapsto -,-\} \\ \mathit{last}.1 := m; \\ \{l = \mathit{last} \wedge \mathit{last} \mapsto -,-\} \\ \mathit{last}.2 := \mathtt{cons}(-,-); \\ \{l = \mathit{last} \wedge \exists g.(\mathit{last} \mapsto -, g * g \mapsto -,-)\} \\ \mathit{last} :;= \mathit{last}.2 \\ \{l \mapsto -, \mathit{last} * \mathit{last} \mapsto -,-)\} \\ \mathtt{V}'(\mathit{number}) \\ \{l = \mathit{last} \wedge \mathit{last} \mapsto -,-\} \end{array} \quad \| \quad \begin{array}{l} \{f = \mathit{first} \wedge \mathtt{emp}\} \\ \mathtt{P}'(\mathit{number}); \\ \{\mathit{first} \mapsto -, f\} \\ n := \mathit{first}.1;\ t := \mathit{first} \\ \{t = \mathit{first} \wedge t \mapsto -, f\} \\ \mathit{first} := \mathit{first}.2 \\ \{f = \mathit{first} \wedge t \mapsto -, f\} \\ \mathtt{dispose}(t); \\ \{f = \mathit{first} \wedge \mathtt{emp}\} \\ \mathit{consume}(n) \\ \{f = \mathit{first} \wedge \mathtt{emp}\} \end{array}$$

From this annotated code we can see that *last*, the dummy, is always owned by the left process, though its value may change. In some cases the left process owns two cells, one of which gets swallowed up by the $\mathtt{V}'$ operation. The right process owns at most one (binary) cell, and *number* always owns the linked list from f to l.

It is helpful to picture a concurrent exection trace; here is a trace when $\mathit{number} = 2$;



In each step in the trace the semaphore owns the list running from f to l, but not including l. So, for instance, in the first list the semaphore owns the two rightmost cells. The left process always owns the cell l, and anything to the left of it, while the right process owns anything to the right of f.

Finally, to mimic the uses of `goto` in the original code pictured at the beginning of this section, we wrap the specified code segments in `while` loops; the resulting code is in Table 7. The most interesting there is the sequence of initialization commands, which establishes the resource invariant and the preconditions for the two processes. Reasoning about the `while` loops can be done with the precondition chosen as the invariant in each case; our reasoning above has already shown that the bodies are invariant.

With this code we can employ the Auxiliary Variable Rule to delete all the uses of the variables f and l, in which case we get

$$\{\texttt{emp}\}\texttt{prog}\{\texttt{false}\}$$

for a program that uses the standard semaphore operations `P` and `V` rather than the super-semaphore versions `P′` and `V′` with auxiliary variables. Thus, we have shown that this sempahore program has no races.

$\{\texttt{emp}\}$
$number := 0;\ last := \texttt{cons}(-,-);\ l := last;\ f := last;\ first := last$
$\{(number = 0 \wedge \texttt{emp} \wedge f = l) * (l = last \wedge last \mapsto -,-) * (f = first \wedge \texttt{emp})\}$
$\{RI * (l = last \wedge last \mapsto -,-) * (f = first \wedge \texttt{emp})\}$
$\texttt{resource}\ number(number);$
$\{(l = last \wedge last \mapsto -,-) * (f = first \wedge \texttt{emp})\}$

$\{l = last \wedge last \mapsto -,-\}$		$\{f = first \wedge \texttt{emp}\}$
$\texttt{while true do}$		$\texttt{while true do}$
$produce(m)$		$\texttt{P}'(number);$
$last.1 := m;$		$n := first.1;\ t := first$
$last.2 := \texttt{cons}(-,-);$	$\|\|$	$first := first.2$
$last := last.2$		$\texttt{dispose}(t);$
$\texttt{V}'(number);$		$consume(n);$
$\{(l = last \wedge last \mapsto -,-) \wedge \neg\texttt{true}\}$		$\{(f = first \wedge \texttt{emp}) \wedge \neg\texttt{true}\}$
$\{\texttt{false}\}$		$\{\texttt{false}\}$

$\{\texttt{false} * \texttt{false}\}$
$\{RI * \texttt{false}\}$
$\{\texttt{false}\}$

Table 7: Counting Semaphore Program

Remarks.

1. This use of counting semaphores to allow the `add portion` and `take portion` operations to proceed concurrently is similar to the circular buffer program in [20]. The original used a circular linked list, but it is most often presented using an array and modular arithmetic to implement the cycling [1]. Reynolds considers the array-based circular buffer class notes from a course on separation logic [44]. We used the non-circular version here because it is simpler for expository purposes; in particular, it uses only one semaphore where the circular buffer uses two. Reynolds also proves a stronger correctness property than here, in a verification that keeps track of buffer contents as well as ownership.

2. One area of potential lack of concurrency is in the uses of `cons` and `dispose` in the different processes. If the memory manager protects these operations via critical regions, then they could not both operate at the same time. However, this kind of simplistic protection of the memory manager, like we did with *mm* in Section 8, can be greatly improved upon.

   For the example of this section, it would make sense to use a manager implemented as a queue rather than a list terminated with `nil`, where `cons` takes an element from one end and `dispose` puts an element on the other. That is, take the idea in the implementation of the buffer in this section, and use it for the resource manager as well. This strategy works well only when the manager deals with elements all the same size, because otherwise the manager should search a list (or lists) to find a good place to insert or remove blocks. But this strategy is a good one for a buffer program as in this section. And when dealing with variable-sized messages we could still use the same strategy, by employing a pointer-transferring version of the unbounded buffer, borrowing the ideas from Section 7.

3. It is possible to write a cautious program that allows for concurrent access to the two ends of a buffer by associating a different critical region or monitor with each buffer element. This would take us outside the formalism as presented in this paper since it would require holding resource names in the heap, and dynamic allocation and deallocation of them. More to the point, though, it is hard to argue that this cautious way of programming the example is

superior to the daring version, because it requires an unbounded number of cautious units of mutual exclusion where one daring semaphore will do.

4. The need to use auxiliary variables in examples such as this one is an impediment to automation. We have been using lightweight assertions that describe integrity rather than full correctness properties. Ideally, one would like to have a form of checking such assertions automatically, by analogy with typechecking. However, given a piece of code we sometimes would have to in some way guess auxiliary variables and assignments to them, just to prove safety.

11 The Reynolds Counterexample and its Consequences

The following counterexample, due to John Reynolds, shows that the concurrency proof rules are incompatible with the usual Hoare logic rule of conjunction (unless the resource invariants are required to be precise)

$$\frac{\{P\}C\{Q\} \quad \{P'\}C\{Q'\}}{\{P \wedge P'\}C\{Q \wedge Q'\}} \ .$$

The example uses a resource declaration

$\texttt{resource}\ r()$

with invariant

$RI_r \ = \ \texttt{true}.$

Let `one` stand for the assertion $10 \mapsto -$. First, we have the following derivation using the axiom for `skip`, the rule of consequence, and the rule for critical regions.

$$\frac{\dfrac{\{\texttt{true}\}\texttt{skip}\{\texttt{true}\}}{\{(\texttt{emp} \vee \texttt{one}) * \texttt{true}\}\texttt{skip}\{\texttt{emp} * \texttt{true}\}}}{\{\texttt{emp} \vee \texttt{one}\}\texttt{with}\ r\ \texttt{when true do skip endwith}\{\texttt{emp}\}}$$

Then, from the conclusion of this proof, we can construct two derivations:

$$\frac{\dfrac{\dfrac{\{\texttt{emp} \vee \texttt{one}\}\texttt{with}\ r\ \texttt{when true do skip endwith}\{\texttt{emp}\}}{\{\texttt{emp}\}\texttt{with}\ r\ \texttt{when true do skip endwith}\{\texttt{emp}\}}}{\{\texttt{emp} * \texttt{one}\}\texttt{with}\ r\ \texttt{when true do skip endwith}\{\texttt{emp} * \texttt{one}\}}}{\{\texttt{one}\}\texttt{with}\ r\ \texttt{when true do skip endwith}\{\texttt{one}\}}$$

and

$$\frac{\{\texttt{emp} \vee \texttt{one}\}\texttt{with}\ r\ \texttt{when true do skip endwith}\{\texttt{emp}\}}{\{\texttt{one}\}\texttt{with}\ r\ \texttt{when true do skip endwith}\{\texttt{emp}\}}$$

Both derivations begin with the rule of consequence, using the implications $\texttt{emp} \Rightarrow \texttt{emp} \vee \texttt{one}$ and $\texttt{one} \Rightarrow \texttt{emp} \vee \texttt{one}$. The first derivation continues with an application of the ordinary frame rule, with invariant `one`, and one further use of consequence.

The conclusions of these two derivations are incompatible with one another. The first says that ownership of the single cell is kept by the user code, while the second says that it is swallowed up by the resource. One application of the conjunction rule with these two conclusions gives us the premise of the following which, using the rule of consequence, leads to an inconsistency.

$$\frac{\{\texttt{one} \wedge \texttt{one}\}\texttt{with}\ r\ \texttt{when true do skip endwith}\{\texttt{emp} \wedge \texttt{one}\}}{\{\texttt{one}\}\texttt{with}\ r\ \texttt{when true do skip endwith}\{\texttt{false}\}}$$

The last triple would indicate that the program diverges, where it clearly does not.

The fact that the resource invariant `true` does not precisely say what storage is owned conspires together with the nondeterministic nature of $*$ to fool the proof rules.

The way out of this problem is to insist that resource invariants precisely nail down a definite area of storage [37]. In the semantic notation of the appendix,

> an assertion P is *precise* if for all states (s, h) there is at most one subheap $h' \sqsubseteq h$ where $s, h' \models P$.

The subheap h' here is the area of storage that a precise predicate identifies. Furthermore, if P is precise then there can be at most one heap splitting that can be chosen to make $P * Q$ true, whether or not Q is precise. So precision short-circuits the non-deterministic nature of $*$.

The assertions `emp` and $E \mapsto F$ and $*$-combinations of precise predicates are all precise. A conjunction $P \wedge Q$ is precise if one of P or Q is. This form was used in the resource invariant for the pointer-transferring buffer. A disjunction of the form $(B \wedge P) \vee (B' \wedge Q)$ is precise if P and Q are and B and B' are pure boolean expressions that are exclusive. This form was used in the resource invariants for binary semaphores, with B of the form $s = 0$ and B' of the form $s = 1$.

We can now indicate the main soundness result of [12]:

> **Theorem** (Brookes): the proof rules are sound if all resource invariants are precise predicates.

This rules out Reynolds's counterexample because `true` is not a precise predicate. All the resource invariants in the examples of this paper are precise.

In [37] a similar issue was addressed in a sequential setting, and there soundness was achieved *either* by restricting resource invariants to be precise, *or* by restricting preconditions of procedure specifications to be precise. The latter restriction puts the "blame" for unsoundness on the assertion `emp` $\vee$ `one`, which is indeed a strange assertion, rather than on the resource invariant `true`.

So, by analogy with [37], we might expect that the proof system here would be sound under a restriction on the `with` rule:

> **False Conjecture** : the proof rules are sound if the precondition in the (conclusion of the) rule for region commands is restricted to precise predicates.

This is false because we can get to the two incompatible conclusions above via different routes, using the axiom for `skip`, the rule of consequence, and the rule for critical regions.

$$\dfrac{\dfrac{\{\texttt{one} * \texttt{true}\}\ \texttt{skip}\ \{\texttt{one} * \texttt{true}\}}{\{(\texttt{one} * \texttt{true}) \wedge \texttt{true}\}\ \texttt{skip}\ \{\texttt{one} * \texttt{true}\}}}{\{\texttt{one}\}\texttt{with}\ r\ \texttt{when true do skip endwith}\{\texttt{one}\}}$$

$$\dfrac{\dfrac{\dfrac{\{\texttt{one} * \texttt{true}\}\ \texttt{skip}\ \{\texttt{one} * \texttt{true}\}}{\{\texttt{one} * \texttt{true}\}\ \texttt{skip}\ \{\texttt{true}\}}}{\{(\texttt{one} * \texttt{true}) \wedge \texttt{true}\}\ \texttt{skip}\ \{\texttt{emp} * \texttt{true}\}}}{\{\texttt{one}\}\texttt{with}\ r\ \texttt{when true do skip endwith}\{\texttt{emp}\}}$$

We could then again use the conjunction rule to obtain an inconsistency.

Remarks.

1. The reason why we are worried about the rule of conjunction in this section is that it seems that it should be true in a reasonable semantics; indeed, it does hold in Brookes's semantics. Now, there *are* models of logics for sequential programs which invalidate the conjunction law; these models use predicate transformers that do not satisfy condition of conjunctivity

(preservation of finite conjunctions). It may be that there is a model of our concurrency rules where there is no need for restricting the invariants to precise predicates, but where the conjunction law is invalid.

In sequential program logics some have argued in favour of non-conjunctive predicate transformers, and so against the conjunction law. We would not feel comfortable making such an argument here. The intuitive statements about ownership in this paper make sense when assertions are precise, a point validated by Brookes's analysis. It is less clear that the statements make sense when assertions are imprecise. Nonetheless, the possibility that there might be a model with no restriction on predicates, but that invalidates the conjunction rule, is at least worth noting.

2. For readers familiar with [37], the false conjecture above might at first suggest some incompatibility with the results there on information-hiding modules. However, Hongseok Yang has explained why there is actually no incompatibility. He states the following moral of the second counterexample.

   > We should not give two different interface specs of a single module procedure if the resource invariant of the module is not precise. Even when the preconditions of those two specs are precise, we might get an inconsistency.

   Yang has shown how an extension of the proof rules in [37], which allows a procedure to have two specifications instead of only one, leads to a variant on the second counterexample of this section.

3. The predicates $array(a, i, j)$ and $sorted(a, i, j)$ used for parallel mergesort in Section 3 are not precise. This is not in contradiction of Brookes's result, because parallel mergesort does not use CCRs.

   The predicates can be made precise, using a construction $P \wedge \neg(P * \neg\mathtt{emp})$ which focuses attention on the minimum heap satisfying P, where P is either $array(a, i, j)$ or $sorted(a, i, j)$. In fact, these two predicates are *supported* in the sense of [37], and Brookes's soundness theorem works as well for supported resource invariants, when the preconditions and postconditions of `with` commands are intuitionistic (closed under heap extension). This is by analogy with a result of [37].

12 Reservations and Limitations

The first, and most obvious, limitation of this work is our focus on partial correctness specifications, so our results pertain to safety but not liveness. It would be interesting to attempt to recast the ideas in a temporal logic form. The main question in doing this is whether the modular, or local reasoning, aspect of our approach can be maintained. Initially, it may seem that liveness properties require a form of global reasoning, as they involve intricate details about how a number of processes interact; we are not sure, however, if this global aspect is necessary.

A second limitation is that we have not allowed shared read access to heap cells. An interesting approach to this issue of *passivity* has recently been put forward in Boyland's work on fractional permissions [7]. Permissions have been adapted to separation logic in [5], extending the work here and leading to proofs of some concurrent programs such as readers-and-writers that are beyond the current paper. Although progress has been made, there is more to be learnt before a final solution to the problem of passivity can be claimed.

The restriction to precise resource invariants was initially an irritation, but precision seems to be interesting in its own right. We wonder whether there is a general calculus of precise predicates, perhaps extending ideas in [3].

One reservation is that the logic is formulated at a particular level of abstraction. Here we have used a RAM model, and other models are possible, but it would be desirable to be able to transit between levels of abstraction that take different points of view on what "the heap"

means. This would mean being able to do refinement proofs that start from an abstract program and specification, rather operating exclusively on the level of the final programs. We decided to present our ideas on a particular level for the very simple reason that the right way to incorporate refinement into our logic is not clear.

Finally, the presence of the variable side conditions in the frame rule and the concurrency proof rules is a continuing source of embarrassment. This is a general problem in Floyd-Hoare logic, having nothing to do with separation logic *per se*. But it is striking that heap locations are treated here more flexibly than variables, allowing their dynamic movability. It is tempting to just disallow variable alteration, confining mutation to the heap, in the style of ML. While simple to do in a language design, if we do that while maintaining the pure (heap independent) expressions of separation logic, then the assertions used to specify example programs quickly become unwieldy; one trades simplicity for the logician (simpler looking proof theory) for complexity for the programmer. So that is no solution. It seems that we either need to find a way to describe a decent logic of complex, heap-dependent expressions, or find a way to treat variables more like locations. We refer to the recent work of Bornat [4] for an attempt in the second direction and for further discussion.

13 Conclusions and Related Work

The essential idea in this paper is the interaction between a notion of "mutual exclusion group" and the separating conjunction $*$. Each group is viewed as a guardian of resource, where an associated invariant describes its attached state, and the use of $*$ to partition the state amongst processes and exclusion groups enables distinct program components to be reasoned about independently. These ideas are a development of those of Hoare in [23], with the main addition being the use of the separating conjunction rather than static variable constraints to extend the method to a larger class of programs, including programs that use pointers and programs that exhibit what we have labelled "daring concurrency" (Section 2.2).

The notion of mutual exclusion group (Section 2.2) is intimately related to raciness. In fact, we should stress that raciness is only relative to a chosen level of granularity. As an extreme, if you surround every single read and write to a shared resource by a critical region, putting everything in the same mutual exclusion group, then there are no races. In the language and proof system of this paper we say nothing about the granularity of basic commands such as $x := y + x$ (commands that don't have subcommands), and we regard them as not part of any exclusion group (though they might appear within, not as, a command in a group). In more detail, the notion of raciness is that is appropriate is that two processes race if two *basic statements* that are *not in the same exclusion group* may attempt to touch the same portion of store at the same time. According to this definition, the notion of race is relative to what one considers to be the mutual exclusion groups.

For this reason the implications of Brookes's "no races" theorem should be interpreted with care. If we vary our notion of exclusion group then we can in principle reason about programs that are normally considered racy. For example, one might take the position that certain very basic commands, such as $x := 1$ and $x := x + 1$, that access a given piece of state can be considered atomic, and we might regard them as being in the same mutual exclusion group.[3] This kind of stipulation is natural when reasoning about certain interfering programs. The choice one makes of whether two basic commands interfere or race is not fundamental to the approach in this paper, but rather simply constitutes one of a range of choices of what the mutual exclusion groups are. Once such a choice is laid down (we took one in the paper), the reasoning methods described here apply.

We have used a particular language and model to illustrate our methods, but they appear to apply more generally. For instance, in light of this discussion on the relative nature of raciness

[3]This is independent of whether we use the syntactic mechanisms of resource names and CCRs to express exclusion groups: the pertinent point is that we would associate an invariant on x (and perhaps other variables) to describe the outcomes of on-the-surface racy behaviour.

it is conceivable that the methods could be applied to programs that are normally thought racy. But we would want a method that applies well, not only in principle. A good challenge would be to give convincing specifications for data structures that use fine-grained locking schemes to allow a high degrees of concurrency (e.g., [29]). Another would be to reason about the extremely subtle non-blocking algorithms which allow races but recover from them rather than prohibiting them (e.g. [30]).

Speaking of non-blocking concurrency, it is natural to wonder whether an implementation of CCRs using the "transaction" idea [21] might satisfy the same or similar proof rules to the ones given here. A transaction attempts to complete, but if another transaction attempted to access the same cells in the meantime then it tries again; transactions purposely do not enforce literal mutual exclusion. It seems, though, that transactional concurrency is "as if race free" when used cautiously (when shared state is accessed only within a transaction). There are also daring idioms that one would probably like to account for, when access to non-transactional memory gets transferred from one process to another.

We have been speaking about shared-variable concurrency, but it may seem as if the intuitive points about separation made in this paper do not depend on it; it would be interesting to attempt to provide modular methods for reasoning about process calculi using resource-oriented logics. In CSP the concepts of resource separation and sharing have been modelled in a much more abstract way than in this paper [25]. And the π-calculus is based on very powerful primitives for name manipulation [31], which are certainly reminiscent of pointers in imperative programs. In both cases it is natural to wonder whether one could have a logic which allows names to be successively owned by different program components, while maintaining the resource separation that is often the basis of system designs. However, the ultimately right way of extending the ideas here to process calculi is not obvious to this author.

An important step has been taken in work of Pym and Tofts on a process calculus and associated modal logic using BI-style technology [41]. They explicitly distinguish resources from processes, associate resource transformers to action names, and include explicit primitives for resource transfer While the exact connection to the work here is unclear – for one, they use the synchronous calculus SCCS as their basis – the overall viewpoint is conceptually consistent with our approach. In particular, the basic decision to target the interaction between resources and processes appears to have some promise.

A line of work that bears a formal similarity to ours is that of Caires, Cardelli and Gordon on logics for process calculi [15, 14]. Like here, they use a mixture of substructural logic and ordinary classical logic and, like here, they consider concurrency. But independence between processes has not been emphasized in their work – there is no analogue of what we called the Separation Property – and they do not distinguish between resources and processes. Their focus is instead on the expression of what they call "intensional" properties, such as the number of connections between two processes; several illuminating examples are given in [13]. Although similar in underlying logical technology, their approach uses this technology in a very different way.

The theme of ownership transfer was the subject of previous work [37], which used methods similar to those of this paper in a sequential setting, to approach modules. The modularity aspect is present here in the way that the rule for critical regions uses a resource invariant only when reasoning about the body of a region, with the separating conjunction prohibiting tampering with the invariant when outside of critical regions. David Naumann has suggested that a new attack is opening up on an old problem, what he calls "imperative modules", which hide mutable state as well as types and code. With Barnett he has described an interesting approach to the problem which utilizes auxiliary variables in a novel way [33].

There idea of ownership is, as one might expect, central in work on Ownership Types [16], and also in the semantics of objects [2]. The typing schemes were originally developed to account for encapsulation, but were then adapted as well to the problem of ruling out race conditions [6]. The main limitation of the ownership typing schemes is that they do not deal naturally with changing partitions, especially when the partitions depend on the values of program variables as is the case in the program for the buffer with concurrent access. Perhaps more promising in this direction

are typing schemes based on separation or permissions (e.g., [7, 46]).

Indeed, in this paper we have mainly used assertions that are type-like in the sense that they describe ownership at program points, but not specific properties of the contents of a data structure. This raises the question of whether a method of checking lightweight separation logic assertions might be used in tandem with a verification condition generator to give a form of *logical control of interference*, as a more flexible version of the syntactic control of interference [23, 9, 42].

Stepping back in time, one of the important early works on reasoning about imperative concurrent programs was that of Owicki and Gries [38]. The Owicki-Gries method involves explicit checking of non-interference between program components, while our system rules out interference in an implicit way, by the nature of the way that proofs are constructed. The result is that the method here is more modular.

This last claim is not controversial; it just echoes a statement of Owicki and Gries. There are in fact two classic Owicki-Gries works, one [39] which extends the approach of Hoare in TTPP, and another [38] which is more powerful but which involves explicit non-interference checking. They candidly acknowledge that "the proof process becomes much longer" in their more powerful method; one way to view the present work is as an attempt to extend the more modular of the two approaches, where the proof process is shorter, to a wider variety of programs.

The non-compositional nature of the more powerful of the Owicki-Gries methods has lead to considerable further work. The most prominent is the rely/guarantee method of Jones [26] (and the related [32]), which is rightly celebrated for providing a compositional approach to reasoning about concurrency. The examples in this paper could probably be proven, in principle, with rely/guarantee. But the specifications and proofs would be much longer.

Take parallel mergesort. We used a pre/post spec

$$\{array(a,i,j)\}\ \mathtt{ms}(a,i,j)\ \{sorted(a,i,j)\}$$

and then proved the recursive calls directly using the disjoint concurrency rule (Section 3). In contrast, in the rely/guarantee formalism we would have to include additional remarks about the interaction with the environment: stated informally,

- **Rely**: No other process touches my array segment $array(a,i,j)$;
- **Guarantee**: I do not touch any sotrage outside my segment $array(a,i,j)$.

The problem here is not just the cost for individual steps of reasoning, but rather that the rely and guarantee conditions, which are present to deal with subtle issues of interference, complicate the specification itself, even when no interference is present. This seems to me to be clear evidence that by using a resource-oriented logic it is possible, at least in some cases, to provide much simpler and more modular specifications and proofs than with existing approaches.

Of course, this little comparison should be taken with a grain of salt – the example is maximally oriented to separation logic's strong points. But again, the criticism that we make is not controversial. In his writings Cliff Jones has taken a refreshingly critical look at rely/guarantee [27, 28]. He is looking for methods to combat *interference flooding*, where complex specification forms used to account for subtle concurrent behaviour flood a specfification development, into areas where interference does not arise and where simpler specification forms suffice.

We are in agreement with many of the remarks Jones makes, particularly on the challenges facing the development of truly modular specification and reasoning methods for concurrent processes, even if we have used different techniques. We repeat, however, that this paper is but a first attempt at bringing separation logic to concurrency. We focussed on CCRs and resource invariants because it eased the attempt; perhaps a marriage between separation logic and rely-guarantee is also possible. Generally, though, we believe that resource-oriented logics offer considerable promise for modular reasoning about concurrent programs, as we hope to have demonstrated in the form of proofs and specifications given in this paper.

Acknowledgements. The influence of John Reynolds on this and all of my work is plain to see, and it is a special honour to be able to dedicate this paper to him. John has always stressed how

accounting for non-interference simplifies reasoning about programs; I hope that this paper might be considered one illustration of his general point. As to the specifics of this work, in addition to the example showing subtleties as regards soundness, the view of semaphore operations as ownership transformers really took flight during a visit to CMU, when we together sketched the treatment of a circular buffer with concurrent access.

Thanks go to Steve Brookes for the remarkable analysis in his companion paper on the semantics, without which this paper would not have been possible.

It is a pleasure to acknowledge the influence of Per Brinch Hansen who, during numerous discussions in Syracuse in the 1990s, impressed upon me the ideas from early works in concurrent programming, especially the importance of resource separation.

The viewpoint in this work owes much to David Pym's insistence on the need for a genuine theory of resource (we are not there yet), and on the semantics he developed for BI. In Pym's semantics the possible worlds correspond to "portions of state"; that they are portions, and not the whole state, is reflected in the "permissions" or "ownership" reading of assertions.

Comments from Tony Hoare and Robin Milner helped me to sharpen my explanations of the relative nature of raciness and mutual exclusion groups.

Finally, thanks to Josh Berdine, Richard Bornat, Cristiano Calcagno and Hongseok Yang for countless discussions on the ideas in this work, especially for daring me to embrace the daring programming idioms.

This research was supported by the EPSRC.

References

[1] G. Andrews. *Concurrent programming: principles and practice.* Benjamin/Cummings, 1991.

[2] A. Banerjee and D. A. Naumann. Representation independence, confinement and access control. In *29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.

[3] J. Berdine, C. Calcagno, and P. O'Hearn. A decidable fragment of separation logic. In *Proceedings of FSTTCS*, pages 97–109, 2004. LNCS 3328.

[4] R. Bornat. Variables as resource in Separation Logic. *Proceedings of the 21st Conference on Mathematical Foundations of Computer Science*, Springer LNCS, to appear, 2005.

[5] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *32nd POPL*, pages 59–70, 2005.

[6] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. OOPSLA, 2002.

[7] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: 10th International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72, Berlin, Heidelberg, New York, 2003. Springer.

[8] P. Brinch Hansen. The nucleus of a multiprogramming system. *Comm. ACM*, 13(4):238–250, 1970.

[9] P. Brinch Hansen. Structured multiprogramming. *Comm. ACM*, 15(7):574–578, 1972. Reprinted in [11].

[10] P. Brinch Hansen. *Operating System Principles.* Prentice Hall, 1973.

[11] P. Brinch Hansen, editor. *The Origin of Concurrent Programming.* Springer-Verlag, 2002.

[12] S. D. Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, this Volume. Preliminary version appeared in *Proceedings of the 15th CONCUR (2004)*, LNCS 3170, pp16-34., 2005.

[13] L. Caires. Behavioral and spatial observations in a logic for the pi-calculus. *Proceedings of FOSSACS, LNCS 2987*, 2004.

[14] L. Cardelli and L Caires. A spatial logic for concurrency. In *4th International Symposium on Theoretical Aspects of Computer Science*, LNCS 2255:1–37, Springer, 2001.

[15] L. Cardelli and A. D. Gordon. Anytime, anywhere. Modal logics for mobile ambients. In *27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 365–377, 2000.

[16] D. Clarke, J. Noble, and J. Potter. Simple ownership types for object containment. *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 53-76, Springer LNCS 2072, 2001.

[17] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, 1968. Reprinted in [11].

[18] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1 2:115–138, October 1971. Reprinted in [11].

[19] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[20] A. N. Habermann. Synchronization of communicating processes. *Comm. ACM*, 15(3):171–176, 1972.

[21] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of OOPSLA*, pages 388–402, 2003.

[22] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In E. Engler, editor, *Symposium on the Semantics of Algebraic Languages*, pages 102–116. Springer, 1971. Lecture Notes in Math. 188.

[23] C. A. R. Hoare. Towards a theory of parallel programming. In Hoare and Perrot, editors, *Operating Systems Techniques*. Academic Press, 1972. Reprinted in [11].

[24] C. A. R. Hoare. Monitors: An operating system structuring concept. *Comm. ACM*, 17(10):549–557, 1974. Reprinted in [11].

[25] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[26] C. B. Jones. Specification and design of (parallel) programs. *IFIP Conference*, 1983.

[27] C. B. Jones. Interference revisited. In J.E. Nicholls, editor, *Proceedings of the 5th Annual Z User Meeting: Z User Workshop, Oxford, UK*, pages 58–73, 1991. Springer-Verlag.

[28] C. B. Jones. Wanted: A compositional approach to concurrency. In A. McIver and C. Morgan, editors, *Programming Methodology*, pages 1–15, 2003. Springer-Verlag.

[29] H. T. Kung and P. L. Lehman. Concurrent manipulation of binary search trees. *ACM Trans. Database Systems* ,5, 354-382, 2005.

[30] M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.

[31] R. Milner. The polyadic pi-calculus: a tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag, 1993.

[32] J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Trans. Software Eng.*, 7(4):417–426, 1981.

[33] D. A. Naumann and M. Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state. Manuscript, 2 February, 2004.

[34] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of 15th Annual Conference of the European Association for Computer Science Logic*, LNCS, pages 1–19. Springer-Verlag, 2001.

[35] P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, June 99.

[36] P. W. O'Hearn and R. D. Tennent, editors. *Algol-like Languages*. Two volumes, Birkhauser, Boston, 1997.

[37] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 268–280, Venice, January 2004.

[38] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, (19):319–340, 1976.

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

[40] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13(1), 45–60, 1981.

[41] D.J. Pym and C. Tofts. A calculus and logic of resources and processes. *HP Labs Technical Report HPL-2004-170*, 2004.

[42] J. C. Reynolds. Syntactic control of interference. In *5th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 39–46, Tucson, Arizona, January 1978. ACM, New York. Also in [36], vol 1.

[43] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. Invited Paper, *Proceedings of the 17th IEEE Symposium on Logic in Computer Science*, pages 55-74, 2002.

[44] J. C. Reynolds. Shared variable concurrency. Chapter 6 of class notes from CS-819 C, CMU, 15 March, 2002.

[45] J. C. Reynolds. Towards a grainless semantics for shared variable concurrency. In *Proceedings of FSTTCS*, pages 35–48, 2004. LNCS 3328.

[46] F. Smith, D. Walker, and G. Morrisett. Alias types. Proceedings of ESOP'99.

Appendix: Sequential Separation Logic

Reasoning about atomic commands is based on the "small axioms" where x, m, n are assumed to be distinct variables.

$$\{E \mapsto -\}\,[E] := F\,\{E \mapsto F\}$$

$$\{E \mapsto -\}\,\mathtt{dispose}(E)\,\{\mathtt{emp}\}$$

$$\{x = m \wedge \mathtt{emp}\}x := \mathtt{cons}(E_1, ..., E_k)\{x \mapsto E_1[m/x], ..., E_k[m/x]\}$$

$$\{x = m \wedge \mathtt{emp}\}\,x := E\,\{x = (E[m/x]) \wedge \mathtt{emp}\}$$

$$\{x = m \wedge E \mapsto n\}\,x := [E]\,\{x = n \wedge E[m/x] \mapsto n\}$$

Typically, the effects of these "small" axioms can be extended using the frame rule:

$$\frac{\{P\}\,C\,\{Q\}}{\{P * R\}\,C\,\{Q * R\}} \quad \begin{array}{l} C \text{ doesn't change} \\ \text{variables free in } R \end{array}$$

In addition to the above we have the usual proof rules of standard Hoare logic.

$$\frac{\{P \wedge B\}C\{P\}}{\{P\}\texttt{while}\,B\,\texttt{do}\,C\{P \wedge \neg B\}} \qquad \frac{P \Rightarrow P' \quad \{P'\}C\{Q'\} \quad Q' \Rightarrow Q}{\{P\}C\{Q\}}$$

$$\{P\}\texttt{skip}\{P\} \qquad \frac{\{P\}C_1\{Q\} \quad \{Q\}C_2\{R\}}{\{P\}C_1;C_2\{R\}}$$

$$\frac{\{P \wedge B\}\,C\,\{Q\} \quad \{P \wedge \neg B\}\,C'\,\{Q\}}{\{P\}\,\texttt{if}\,B\,\texttt{then}\,C\,\texttt{else}\,C'\{Q\}}$$

Also, although we have not stated them, there is a substitution rule and a rule for introducing existential quantifiers, as in [34].

In order to apply the rule of consequence we have to have a way to determine whether an implication $P \Rightarrow Q$ is true. We do so in this paper using a semantics of the assertion language, which might be viewed as an oracle in the proof system. We can use $P \Rightarrow Q$ in the consequence rule when $s, h \models P \Rightarrow Q$ holds for all s and h in the semantics below (when the domain of s contains the free variables of P and Q.)

A state consists of two components, the stack $s \in S$ and the heap $h \in H$, both of which are finite partial functions as indicated in the following domains.

$$\begin{array}{rcl@{\qquad}rcl} \texttt{Variables} & \triangleq & \{x, y, ...\} & \texttt{Nats} & \triangleq & \{0, 1, 2...\} \\ \texttt{Ints} & \triangleq & \{..., -1, 0, 1, ...\} & H & \triangleq & \texttt{Nats} \rightharpoonup_{\text{fin}} \texttt{Ints} \\ S & \triangleq & \texttt{Variables} \rightharpoonup_{\text{fin}} \texttt{Ints} & \texttt{States} & \triangleq & S \times H \end{array}$$

Integer and boolean expressions are determined by valuations

$$[\![E]\!]s \in \texttt{Ints} \qquad [\![B]\!]s \in \{true, false\}$$

where the domain of $s \in S$ includes the free variables of E or B. We use the following notations in the semantics of assertions.

1. $dom(h)$ denotes the domain of definition of a heap $h \in H$, and $dom(s)$ is the domain of $s \in S$;
2. $h\#h'$ indicates that the domains of h and h' are disjoint;
3. $h * h'$ denotes the union of disjoint heaps (i.e., the union of functions with disjoint domains);
4. $(f \mid i \mapsto j)$ is the partial function like f except that i goes to j.

The satisfaction judgement $s, h \models P$ which says that an assertion holds for a given store and heap. (This assumes that $\text{Free}(P) \subseteq dom(s)$, where $\text{Free}(P)$ is the set of variables occurring freely in P.)

$$\begin{array}{lll} s, h \models B & \text{iff} & [\![B]\!]s = true \\ s, h \models E \mapsto F & \text{iff} & \{[\![E]\!]s\} = dom(h) \text{ and } h([\![E]\!]s) = [\![F]\!]s \\ s, h \models P \Rightarrow Q & \text{iff} & \text{if } s, h \models P \text{ then } s, h \models Q \\ s, h \models \forall x.P & \text{iff} & \forall v \in \texttt{Ints}.\, [s \mid x \mapsto v], h \models P \\ s, h \models \texttt{emp} & \text{iff} & h = [\,] \text{ is the empty heap} \\ s, h \models P * Q & \text{iff} & \exists h_0, h_1.\, h_0\#h_1,\ h_0 * h_1 = h,\ s, h_0 \models P \text{ and } s, h_1 \models Q \end{array}$$

Notice that the semantics of $E \mapsto F$ is "exact", where it is required that E is the only active address in the current heap. Using $*$ we can build up descriptions of larger heaps. For example, $(10 \mapsto 3) * (11 \mapsto 10)$ describes two adjacent cells whose contents are 3 and 10.

The "permissions" reading of assertions is intimately related to the way the semantics above works with "portions" of the heap. Consider, for example, a formula

$$\texttt{list}(f) * x \mapsto -, -$$

as was used in the memory manager example. A heap h satisfying this formula must have a partition $h = h_0 * h_1$ where h_0 contains the free list (and nothing else) and h_1 contains the binary cell pointed to by x. It is evident from this that we cannot regard an assertion P on its own as describing the entire state, because it might be used within another assertion, as part of a $*$ conjunct.

The reading of triples in separation logic then takes this point one step further, by integrating the "portions of state" view with the idea of access. The precondition describes the portion of state needed by the command to run without an access violation (a memory fault, or a race) or, put another way, the portion of state that the command needs to own before it is run.