Resources, Concurrency and Local Reasoning

Peter W. O'Hearn

Queen Mary, University of London

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 [8, 9, 11, 12, 1, 2]. 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 [22].

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 [16]. 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 [10] and BI [17]. 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 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., [19, 21, 14]), but the ideas in an early paper of Hoare on concurrency, "Towards a Theory of Parallel Programming [11]" (henceforth, TTPP), fit particularly well with the viewpoint of separation 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 partial-correctness 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 [20]. Although the mere insertion of the separating conjunction was straightforward, we found we could handle a number of daring, though valuable, programming idioms, and this opened up a number of unexpected (for us) possibilities.

To describe the nature of the daring programs we suppose that there is a way in the programming language to express 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 from a mutual exclusion group might very well overlap with execution of a command not in that group. 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 $\texttt{with}\, r\, \texttt{when}\, B\, \texttt{do}\, C$ 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*.

Obviously, the nature of mutual exclusion is to guarantee that cautious programs are not *racy*, where concurrent processes attempt to access the same portion of state at the same time without explicit synchronization. The simplicity and modularity of the TTPP proof rules is achieved by syntactic restrictions which ensure caution; a main contribution of this paper is to take the method into the realm of daring programs, while maintaining its modular nature.

Daring programs are many. Examples include: double-buffered I/O, such as where one process renders an image represented in a buffer while a second process is filling a second buffer, and the two buffers are switched when an image changes; efficient message passing, where a pointer is passed from one process to another to avoid redundant copying of large pieces of data; memory managers and other resource managers such as thread and connection pools, which are used to avoid the overhead of creating and destroying threads or connections to databases. Indeed, almost all concurrent systems programs are daring, such as microkernel OS designs, programs that manage network connectivity and routing, and even many application programs such as web servers.

But to be daring is to court danger: If processes access the same portion of state outside a common mutual exclusion grouping then they just might do so at the same time, and we can very well get inconsistent results. Yet it is possible to be safe, and to know it, when a program design observes a principle of resource separation.

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

When combined with the principle that a program component only accesses state that it owns, separation implies race-freedom.

Our proof system will be designed to ensure that any program that gets past the proof rules satisfies the Separation Property. And because we use a logical connective (the separating conjunction) rather than scoping constraints to express separation, we are able to describe dynamically changing state partitions, where ownership (the right to access) transfers between program components. It is this that takes us safely into the territory of daring programs.

This paper is very much about fluency with the logic – how to reason with it – rather than its metatheory; we refer the reader to the companion paper by Stephen Brookes for a thorough theoretical analysis [4]. In addition to soundness, Brookes shows that any proven program will not have a race in an execution starting from a state satisfying its precondition.

After describing the proof rules we give two examples, one of a pointer-transferring buffer and the other of a toy memory manager. These examples are then combined to illustrate the modularity aspect. The point we will attempt to demonstrate is that the specification for each program component is "local" or "self contained", in the sense that assertions make local remarks about the portions of state used by program components, instead of global remarks about the entire system state. Local specification and 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.

This is a preliminary paper. In the long version we include several further examples, including two semaphore programs and a proof of parallel mergesort.

2 The Programming Language

The presentation of the programming language and the proof rules in this section and the next follows that of Owicki and Gries [20], 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)
$C_1 \parallel \cdots \parallel C_n$

$$
\begin{array}{lll}
C & ::= & x := E \mid x := [E] \mid [E] := F \mid x := \mathtt{cons}(E_1, ..., E_n) \mid \mathtt{dispose}(E) \\
 & \mid & \mathtt{skip} \mid C; C \mid \mathtt{if}\ B\ \mathtt{then}\ C\ \mathtt{else}\ C \mid \mathtt{while}\ B\ \mathtt{do}\ C \\
 & \mid & \mathtt{with}\ r\ \mathtt{when}\ B\ \mathtt{do}\ C \\
E, F & ::= & x, y, ... \mid 0 \mid 1 \mid E + F \mid E \times F \mid E - F \\
B & ::= & \mathtt{false} \mid B \Rightarrow B \mid E = F \mid E < F
\end{array}
$$

Table 1. Sequential Commands

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 [4] for a more modern presentation of the programming language, which does not observe this restricted form.

A grammar for the sequential processes is included in Table 1. They include constructs for `while` programs as well as operators for accessing a program heap. The operations $[E] := F$ and $x := [E]$ are for mutating and reading heap cells, and the commands $x := \mathtt{cons}(E_1, ..., E_n)$ and $\mathtt{dispose}(E)$ are for allocating and deleting cells. Note that the integer expressions E are pure, in that they do not themselves contain any heap dereferencing $[\cdot]$. Also, although expressions range over arbitrary integers, the heap is addressed by non-negative integers only; the negative numbers can be used to represent data apart from the addresses, such as atoms and truth values, and we will do this without comment in examples like in Section 4 where we include `true`, `false` and `nil` amongst the expressions E (meaning, say, -1, -2 and -3).

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

$$\mathtt{with}\ r\ \mathtt{when}\ B\ \mathtt{do}\ C\ .$$

Here, B ranges over (heap independent) boolean expressions and C over commands. Each resource name determines a mutual exclusion group: two `with` commands for the same resource name cannot overlap in their executions. Execution of $\mathtt{with}\ r\ \mathtt{when}\ B\ \mathtt{do}\ C$ 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.

It would have been possible to found our study on monitors rather than CCRs, but this would require us to include a procedure mechanism and it is theoretically simpler not to do so.

Programs are subject to variable conditions for their well-formedness (from [20]). 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;

SYNTAX

$$P, Q, R ::= B \mid \texttt{emp} \mid E \mapsto F \mid P * Q \mid \texttt{false} \mid P \Rightarrow Q \mid \forall x.P \mid \cdots$$

ABBREVIATIONS

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

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

Table 2. Assertions

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.

These conditions ensure that any variables accessed in two concurrent processes must be protected by synchronization. For example, the racy program

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

is ruled out by the conditions. In the presence of pointers these syntactic restrictions are not enough to avoid all races. 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 and we will have a race, while if x and y are unequal then there will be no race.

3 Proof Rules

The proof rules below refer to assertions from separation logic; see Table 2. The assertions include the points-to relation $E \mapsto F$, the separating conjunction $*$, the empty-heap predicate `emp`, and all of classical logic. 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 5. A semantics for these assertions has been included in the appendix.

Familiarity with the basics of separation logic is assumed [22]. 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 statement mutates address 10, then P must imply the assertion $(10 \mapsto -) * \texttt{true}$ that 10 not be dangling. Thus, a precondition confers the right to access certain cells, those that it guarantees are not dangling; this provides the connection between program logic and the intuitive notion of "ownership" discussed in the introduction.

To reason about a program

$$\begin{array}{l} \mathit{init}; \\ \texttt{resource}\ r_1(\text{variable list}), ..., r_m(\text{variable list}) \\ C_1 \parallel \cdots \parallel C_n \end{array}$$

we first specify a formula RI_{r_i}, the resource invariant, for each resource name 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 need to require that each resource invariant is "precise". The definition of precision, and an example of Reynolds showing the need to restrict the resource invariants, is postponed to Section 7; for now we will just say that the invariants we use in examples will adhere to the restriction.

In a complete program the resource invariants must be *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. The resource invariants are then removed from the pieces of state accessed directly by processes. This is embodied in the

Rule for Complete Programs

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

For a parallel composition we simply give each process a separate piece of state, and separately combine the postconditions for each process.

Parallel Composition Rule

$$\frac{\{P_1\}\, C_1\, \{Q_1\} \;\cdots\; \{P_n\}\, C_n\, \{Q_n\}}{\{P_1 * \cdots * P_n\}\, C_1 \parallel \cdots \parallel C_n\, \{Q_1 * \cdots * Q_n\}} \quad \begin{array}{l}\text{no variable free in } P_i \text{ or } Q_i \\ \text{is changed in } C_j \text{ when } j \neq i\end{array}$$

Using this proof rule we can prove a program that has a potential race, as long as that race is ruled out by the precondition.

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

Here, the $\ast$ 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

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

The reasoning that establishes the triples $\{P_j\}C_j\{Q_j\}$ for sequential processes in the parallel rule is done in the context of an assignment of invariants RI_{r_i} to resource names r_i. This contextual assumption is used in the

Critical Region Rule

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

The idea of this rule is that when inside a critical region the code gets to see the state associated with the resource name as well as that local to the process it is part of, while when outside the region 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 $C_1 \parallel \cdots \parallel C_n$, where an occurrence of a `with` command will be in one of these processes C_j.

Besides these proof rules we allow all of sequential separation logic; see the appendix. 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" [8]. 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.

In sloganeering terms we might say that *well specified processes mind their own business*: proven processes only dereference those cells that they own, those known to exist in a precondition for a program point. 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 [4]. 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.

4 Example: Pointer-transferring Buffer

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.

In this section 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 2.)

The initialization and resource declaration are

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

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

```
put($x$) $\stackrel{\Delta}{=}$ with $buf$ when $\neg full$ do
                 $c := x$; $full$ := true;

get($y$) $\stackrel{\Delta}{=}$ with $buf$ when $full$ do
                 $y := c$; $full$ := false;
```

For presentational convenience we are using definitions of the form

$$\texttt{name}(\boldsymbol{x}) \stackrel{\Delta}{=} \texttt{with}\ r\ \texttt{when}\ B\ \texttt{do}\ C$$

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

We focus on the following code.

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

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. At the end of the section we will place these code snippets into loops, as part of a producer/consumer iidiom, but for now will concentrate on the snippets themselves.

The resource invariant for the buffer is

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

To understand this invariant it helps to 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" [18]. According to this reading the assertion $c \mapsto -, -$ says "I own binary cons cell c" (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". Given this reading the resource 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 `with` command in $\texttt{put}(x)$.

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

The rule for `with` commands then gives us

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

The postcondition indicates that the sending process gives up ownership of pointer x when it is placed into the buffer, even though the value of x is still held by the sender.

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

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

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 buffer 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:

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

which gives us

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

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

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

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

$$
\begin{array}{l}
\{\texttt{emp}\} \\
\mathit{full} := \texttt{false} \\
\{\neg \mathit{full} \wedge \texttt{emp}\} \\
\{RI_{buf} * \texttt{emp} * \texttt{emp}\}
\end{array}
$$

we obtain the triple $\{\texttt{emp}\}\texttt{prog}\{RI_{buf}\}$ for the complete program prog.

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. For instance, in the left process above we used $x \mapsto -,-$ as the postcondition of $x := \texttt{cons}(a, b)$; to get there from the "official" postcondition $x \mapsto a, b$ we just observe that it implies $x \mapsto -,-$. We will often omit mention of little implications such as this one.

```
                    {emp}
                    full := false;
                    {emp ∧ ¬full}
                    {RI_buf * emp * emp}
                    resource buf(c, full)
                        {emp * emp}
{emp}                                   {emp}
while true do                           while true do
    {emp}                                   {emp}
    produce(a, b);                          get(y);
    x := cons(a, b);        ||              use(y);
    put(x);                                 dispose(y);
    {emp}                                   {emp}
{false}                                 {false}
                       {false * false}
                    {RI_buf * false}
                    {false}
```

Table 3. Pointer-passing Producer/Consumer Program

The verification just given also shows that if we were to add a command, say $x.1 := 3$, that dereferences x after the `put` command in the left process then we would not be able to prove the resulting program. The reason is that `emp` is the postcondition of `put`(x), while separation logic requires that x point to something (be owned) in the precondition of any operation that dereferences x.

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 do not speak of 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. To show stronger correctness properties, which track buffer contents, we would generally need to use auxiliary variables [20].

As it stands the code we have proven is completely sequential: the left process must go first. Using the properties we have shown it is straightforward to prove a producer/consumer program, where these code snippets are parts of loops, as in Table 3. In the code there `emp` is the invariant for each loop, and the overall property proven ensures that there is no race condition.

5 Example: Memory Manager

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

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

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

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

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

$$\mathit{list}\,x \overset{\Delta}{\Longleftrightarrow} (x = \texttt{nil} \wedge \texttt{emp}) \vee (\exists y.\, x \mapsto -, y \ast \mathit{list}\, y)$$

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.

```
alloc(x, a, b) ≜ with mm when true do
                    if $f = \texttt{nil}$ then $x := \texttt{cons}(a, b)$
                    else $x := f;\ f := x.2;\ x.1 := a;\ x.2 := b$
dealloc(y) ≜ with mm when true do
                 $y.2 := f;$
                 $f := y;$
```

The command $f := x.2$ reads the cdr of binary cons cell x and places it into f. We can desugar $x.2$ as $[x + 1]$ in the RAM model of separation logic, and similarly we will use $x.1$ for $[x]$ to access the car of a cons cell.

Using the rule for `with` commands 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.

```
                 {emp * emp}
{emp}                            {emp}
alloc(x, a, b);                  alloc(y, a′, b′);
{x ↦ a, b}                       {y ↦ a′, b′}
x.1 := 4             ‖           y.1 := 7
{x ↦ 4, b}                       {y ↦ 7, b′}
dealloc(x);                      dealloc(y);
{emp}                            {emp}
                 {emp * emp}
                 {emp}
```

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 `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 `dealloc`(x) is `emp`.

The issue here is 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.

6 Combining the Buffer and Memory Manager

We now show how to put the treatment of the buffer together with the home-grown 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 4 done again, this time using `mm`.

```
          {emp * emp}
{emp}                     {emp}
alloc(x, a, b);    ||     get(y);
{x ↦ –, –}                {y ↦ –, –}
put(x);                   use(y);
{emp}                     {y ↦ –, –}
                          dealloc(y);
                          {emp}
          {emp * emp}
          {emp}
```

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

The initialization sequence and resource declaration now have the form

```
full := 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).

7 The Reynolds Counterexample

The following counterexample, due to John Reynolds, shows that the concurrency proof rules are incompatible with the usual Hoare logic rule of conjunction

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

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.

$$\cfrac{\cfrac{\{\texttt{true}\}\texttt{skip}\{\texttt{true}\}}{\{(\mathsf{emp} \vee \mathsf{one}) * \texttt{true}\}\texttt{skip}\{\mathsf{emp} * \texttt{true}\}}}{\{\mathsf{emp} \vee \mathsf{one}\}\texttt{with}\ r\ \texttt{when true do skip}\ \{\mathsf{emp}\}}$$

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

$$\cfrac{\cfrac{\cfrac{\{\mathsf{emp} \vee \mathsf{one}\}\texttt{with}\ r\ \texttt{when true do skip}\ \{\mathsf{emp}\}}{\{\mathsf{emp}\}\texttt{with}\ r\ \texttt{when true do skip}\ \{\mathsf{emp}\}}}{\{\mathsf{emp} * \mathsf{one}\}\texttt{with}\ r\ \texttt{when true do skip}\ \{\mathsf{emp} * \mathsf{one}\}}}{\{\mathsf{one}\}\texttt{with}\ r\ \texttt{when true do skip}\ \{\mathsf{one}\}}$$

and

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

Both derivations begin with the rule of consequence, using the implications $\mathtt{emp} \Rightarrow \mathtt{emp} \vee \mathtt{one}$ and $\mathtt{one} \Rightarrow \mathtt{emp} \vee \mathtt{one}$. 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. An 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.

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

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 nondeteministic nature of $*$ to fool the proof rules. A way out of this problem is to insist that resource invariants precisely nail down a definite area of storage [18]. In the semantic notation of the appendix,

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

The subheap h' here is the area of storage that a precise predicate identifies.

The Reynolds counterexample was discovered in August of 2002, a year after the author had described the proof rules and given the pointer-transferring buffer example in an unpublished note. Realizing that the difficulty in the example had as much to do with information hiding as concurrency, the author, Yang and Reynolds studied a version of the problem in a sequential setting, where precise resource invariants were used to describe the internal state of a module [18]. The more difficult concurrent case was then settled by Brookes [4]; his main result is

> **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. And the resource invariants in the one-place buffer and the toy memory manager are both precise.

8 Conclusion

It may seem as if the intuitive points about separation made in this paper should apply more generally than to shared-variable concurrency; in particular, 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 [13]. And the π-calculus is based on very powerful primitives for name manipulation [15], 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 right way of extending the ideas here to process calculi is not obvious.

A line of work that bears a formal similarity to ours is that of Caires, Cardelli and Gordon on logics for process calculi [6, 5]. 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 neither have they considered the impact of race conditions. Their focus is instead on the expression of what they call "intensional" properties, such as the number of connections between two processes. So, although similar in underlying logical technology, their approach uses this technology in a very different way.

The idea of ownership is, as one might expect, central in work on Ownership Types [7]. It would be interesting to attempt to describe a formal connection.

Stepping back in time, one of the important early works on reasoning about imperative concurrent programs was that of Owicki and Gries [19]. A difference with the work here is that our system rules out racy programs, while theirs does not. However, they handle racy programs by assuming a fixed level of granularity, where if we were to make such an assumption explicit (using a critical region) such programs would not be, in principle, out of reach of our methods. More importantly, 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 [20] which extends the approach of Hoare in TTPP, and another [19] 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 this 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.

There are a number of immediate directions for future work. One is the incorporation of passivity, which would allow read-only sharing of heap cells between processes. Another is proof methods that do not require complete resource separation, such as the rely-guarantee method [14, 23], where the aim would be to use separation logic's local nature to cut down the sizes of rely and guarantee conditions. A third is the incorporation of temporal features. Generally, however, we believe that the direction of resource-oriented logics offers promise for reasoning about concurrent systems, as we hope to have demonstrated in the form of proofs and specifications given in this paper.

ACKNOWLEDGEMENTS. I am grateful to Per Brinch Hansen, David Pym and John Reynolds for discussions on resource and concurrency that impacted the form of this work, to Steve Brookes for the theoretical analysis without which it could not be presented, and to Josh Berdine, Richard Bornat, Cristiano Calcagno and Hongseok Yang for daring me to embrace the daring programming idioms. This research was supported by the EPSRC.

References

1. P. Brinch Hansen. The nucleus of a multiprogramming system. *Comm. ACM*, 13(4):238–250, 1970.
2. P. Brinch Hansen. Structured multiprogramming. *Comm. ACM*, 15(7):574–578, 1972. Reprinted in [3].
3. P. Brinch Hansen, editor. *The Origin of Concurrent Programming.* Springer-Verlag, 2002.
4. S. D. Brookes. A semantics for concurrent separation logic. This Volume, Springer LNCS, *Proceedings of the 15th CONCUR*, London. August, 2004.
5. 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.
6. 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.
7. 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.
8. E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, 1968. Reprinted in [3].
9. E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1 2:115–138, October 1971. Reprinted in [3].
10. J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
11. C. A. R. Hoare. Towards a theory of parallel programming. In Hoare and Perrot, editors, *Operating Systems Techniques.* Academic Press, 1972. Reprinted in [3].
12. C. A. R. Hoare. Monitors: An operating system structuring concept. *Comm. ACM*, 17(10):549–557, 1974. Reprinted in [3].
13. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
14. C. B. Jones. Specification and design of (parallel) programs. *IFIP Conference*, 1983.
15. 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.
16. 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.
17. P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, June 99.
18. 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.

19. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, (19):319–340, 1976.
20. S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Comm. ACM*, 19(5):279–285, 1976.
21. A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13(1), 45–60, 1981.
22. 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.
23. C. Stirling. A generalization of the Owicki-Gries Hoare logic for a concurrent while language. *Theoretical Computer Science*, 58:347–359, 1988.
24. H. Yang and P. O'Hearn. A semantic basis for local reasoning. In *Foundations of Software Science and Computation Structures*, Springer LNCS 2303., 2002.

Appendix: Sequential Separation Logic

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

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

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

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

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

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

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

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

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 [16].

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.) Thus, the semantics is, in this paper, used as an oracle by the proof system.

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.

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

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. $h \cdot h'$ 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 stack and heap. (This assumes that Free$(P) \subseteq dom(s)$, where Free(P) is the set of variables occurring freely in P.)

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

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.