1

Separation logic and concurrency

Richard Bornat

School of Computing Science, Middlesex University, UK R.Bornat@mdx.ac.uk

Summary. Concurrent separation logic is a development of Hoare logic adapted to deal with pointers and concurrency. Since its inception it has been enhanced with a treatment of permissions, to enable sharing of data between threads, and a treatment of variables as resource alongside heap cells as resource. An introduction to the logic is given, with several examples of proofs, culminating in a treatment of Simpson's 4-slot algorithm, an instance of racy non-blocking concurency.

1.1 Introduction

Computing is the offspring of an historic collision, starting in the 1930s and continuing to this day, between mechanical calculation and formal logic. Mechanical calculation gave us hardware, formal logic gave us programming.

Programming is hard, probably just because it is formal,[1] and never seems to get any easier. Each advance in formalism, for example when high-level languages were invented, or types were widely introduced, and each methodological improvement, for example structured programming, or Dijkstra's treatment of concurrent programming discussed below, is taken by programmers as an opportunity to extend their range rather than improve their accuracy. Advances in formalism and improvements in methodology are intended to help to prevent bugs and to make the programmer's life easier, but programmers' response has always been to write bigger and more complicated programs with about the same number of bugs as their previous smaller, simpler programs. There may be grounds to say that programming is just exactly as hard as we dare make it (but, of course, no harder!).

For some time now, perhaps ever since the late 1960s, concurrent programming has been the hardest kind of programming. At the moment there is a great deal of scientific interest in the subject. New approaches – in particular,

[1] Our programs are executed by machines which do not and could never understand our intentions. The executions of our programs are entirely determined by the arrangement of symbols we write down. This is a very severe discipline indeed.

non-blocking algorithms [20] and transactional memory [14] – are being tried out. New theoretical advances are beginning to give formal treatments to old problems. And hardware designers, in their relentless search for speed, are providing us with multiprocessor chips, forcing us to confront the problems of concurrent and parallel programming in our program designs. At once we are being pressed to solve a problem that we haven't been able to solve before, at the same time as we are developing enthusiasm that we might be able to make a dent in it. These are exciting times!

In the strange way that computer science has always developed, theoreticians are both behind and ahead of the game. This paper looks at some classic problems – the unbounded buffer, readers and writers, and Simpson's 4-slot algorithm – which are beginning to yield to pressure.

1.2 Background

The difficulties of concurrent programming are easily illustrated. Suppose that there are two programs executing simultaneously on two processors, making two concurrent processes, and we intend them to communicate. One way to do so is via a shared store. Process A might write a message into a shared area of the store, for example, for process B to read. If this is the means of communication, B can't know when A has communicated – notification is communication! – and so must read the store every now and then to see if A's said anything new. Then if A writes at about the same time that B reads, they may trip over each other: depending on the order that the various parts of the message are written and read, B may read part of what A is currently writing, and part of what it wrote previously, thus retrieving a garbled message. Miscommunication is clearly possible, and really happens: you can read the wrong time, for example, if the hardware clock writes a two-word value into the store and you aren't careful about the way you read it.

The problem is just as acute if the processes are running in a multiprogramming system, not actually simultaneously but with parts of their execution interleaved: A runs for a while, then B, then A, and so on. Unless we carefully control communications, A can be interrupted part way through sending a message, or B part way through sending one, and miscommunication can as easily result as before. And we can get tied into just the same knots if we consider threads within a single process.

Once, back in the prehistory of concurrent programming, people thought that they would have to reason about the speed of concurrent processes and prove that communication collisions couldn't happen. That's effectively impossible in the case of multiprogramming pseudo-concurrency, and – not for the last time – Dijkstra had to put us right:

> "We have stipulated that processes should be connected loosely; by this we mean that apart from the (rare) moments of explicit intercom-

> munication, the individual processes themselves are to be regarded as completely independent of each other." [12]

He then provided us with the first effective mechanism for concurrent programming, using semaphores and critical sections, but he didn't provide us with a formalism in which we could reason about concurrent programs. His injunction provides us with the basic plank of a programming framework, though: we have to separate consideration of communication from consideration of what happens between communications.

Once we recognise that we need some special mechanism to communicate reliably, all kinds of new difficulties arise: deadlock (everybody waiting for somebody to move first); livelock (everybody rushing about, constantly missing each other); and problems of efficiency. Those are interesting difficulties questions, but my focus will be on safety, showing that if communication does happen, it happens properly.

The problem is eased, but not solved, by going for more tractable means of communication, like message-sending. Message-sending has to be implemented in terms of things that the hardware can do, which means worrying about who does what when with which physical communications medium. And the other issues, of deadlock, livelock and efficiency, somehow manage to keep poking through.

1.2.1 Related work

Concurrent separation logic, as we shall see, is about partial-correctness specifications of threads executed concurrently. The seminal work in this area was Susan Owicki's. In her thesis [26] she tackled head-on the problems of 'interference', where two processes compete for access to the same resource, and she dealt with some significant examples, including the bounded buffer, a version of readers-and-writers, dining philosophers, and several semaphore programs. Sometimes referred to as the Owicki-Gries method, following the papers she wrote with her supervisor [23, 24], it proved capable of verifying really difficult algorithms [13]. But it wasn't without its drawbacks: it didn't deal well with pointers, and in general each instruction in every thread had to be shown not to interfere with any of the assertions expressed in any other thread. Her RPL language, using conditional critical regions (described in section 1.5), was more tractable, but restricted.

Jones's rely-guarantee [19] is the best-known of the later developments. Jones introduced thread-global 'rely' and 'guarantee' conditions in addition to pre- and postconditions. Rely conditions state what a thread can assume about the state of shared data, essentially constraints on the action of other threads. Guarantee conditions state what a thread must maintain in the state of shared data. Whilst more modular than Owicki's method, it is still the case that a proof must check the maintenance of guarantee conditions by each instruction of a thread.

There's also temporal logic, for example [29, 1, 25]. Temporal logic can deal with liveness; separation logic, as we shall see, can deal only with safety. (Owicki characterised the difference as "safety is that bad things don't happen; liveness is that good things do happen".) But temporal logic approaches so far aren't as modular as separation logic.

1.3 Hoare logic

Separation logic uses Hoare triples, a mechanism of Hoare logic [15]. In Hoare logic you make classical-logic assertions about the state of the computer's store, and you reason about programs in terms of their effect on that store: the triple $\{Q\}\,C\,\{R\}$ is a specification of the command (program) C: if you execute C in a store which satisfies precondition Q, then it is certain to terminate, leaving a store which satisfies R. Hoare logic works with a programming language which is rather like Pascal or Algol 60: assignments to variables and arrays, sequences of commands, choices (if-then-else), loops (while-do), and procedure calls. Its greatest glory is the variable-assignment axiom $\{R[E/x]\}\,x := E\,\{R\}$, which converts difficult semantic arguments about stores into simple substitution of expression E for occurrences of variable x in the postcondition formula R, thus reducing large parts of formal program verifications to syntactic calculation. Its most evident complexity is the use of invariants to describe loops but, as we have learned to accept, that's not a fault of the logic – loops really are that difficult to understand.

One difficulty in Hoare logic is *aliasing*. It arises in a minor way with the assignment axiom – if you have more than one name for a program variable, then you can't use simple substitution for the single name used in the assignment. That difficulty could perhaps be overcome, or variable aliasing could be outlawed as it was in Ada, but there's a worse problem in array-element aliasing. To make array-element assignment work in the logic, arrays are treated as single variables containing an indexed sequence. The array-element assignment $a[I] := E$ is treated as if it writes a new sequence value into the variable a which is the same as the sequence before the assignment, except that at position I it contains E. That's all very well, but if, for example, $i + 2 = j$, then elements $a[i+1]$ and $a[j-1]$ are aliases, so that assignment to one affects the other and vice-versa. There is no possibility of outlawing this problem, and as a consequence program verifications turn, more often than not, on arithmetical arguments about array indices.

Array variables aren't the end of the difficulty. The program heap, in which dynamically-allocated data structures live, is really a very large array indexed by addresses – 'pointers' or 'references' – and the practical problems of aliasing in the heap have bedevilled Hoare-logic reasoners from the start. Following the success of structured programming in abolishing the goto, some people would have us abandon use of pointers. Hoare himself opined

> "References are like jumps, leading wildly from one part of a data structure to another. Their introduction into high-level languages has been a step backward from which we may never recover." [17]

Gotos have almost disappeared nowadays, but pointers/references have proved more durable. One of the most popular programming languages, Java, depends almost entirely on references. The heap, like concurrency, won't go away just because we can't reason very well about it. To recover we shall have to tame pointers rather than banish them.

1.4 A resource logic for the heap

One possible solution is to exploit separation. If, for example, there are two lists in the heap which don't share any cells, then changes to one don't affect the other, and we can reason about operations on one independently of operations on the other. This was first noted by Burstall [9]. I managed to prove some graph algorithms using a Hoare logic augmented with an operation which asserted separation of data structures [3], but the proofs were very clumsy. Separation logic [22, 30], which began by building on Burstall's insight, does it much more elegantly.

Notice, first, that each variable-assignment instruction in a program affects only one store location, and Hoare's variable-assignment axiom is beautiful just because it captures that simplicity in a purely formal mechanism. The same is true of heap-cell assignment: we can only assign one cell at a time, and if that cell is separate from the cells used by other parts of the program, then we can have a single-cell heap assignment axiom. In separation logic we write $E \mapsto F$ (E points to F) to assert exclusive ownership of the heap cell at address E with contents F. E is a positive integer expression and, for simplicity, F is an integer expression. To avoid getting tangled up in aliasing, E and F must be 'pure' expressions not depending on the heap – that is, involving only variables and constants. We write $E \mapsto \_$ to express ownership without knowledge of contents (it's equivalent to $\exists k \cdot (E \mapsto k)$), we write $[E]$ (pronounced 'contents of E') to describe the value stored at address E, and we have a heap assignment axiom

$$\{E \mapsto \_\}\,[E] := F\,\{E \mapsto F\} \tag{1.1}$$

– provided we exclusively own the cell at address E before the assignment, we can put a new value into it and be confident that we would find that value in the cell if we looked, and we still own that cell afterwards. We only need to own one cell to be able to assign it.

There's a similar heap-access assignment axiom

$$\{F = N \wedge E \mapsto F\}\,x := [E]\,\{x = N \wedge E \mapsto N\} \tag{1.2}$$

These two forms of assignment are the only way to deal with the heap in the logic, and it's easy to see how more complicated patterns of access and assignment can be compiled into sequences of variable assignment, heap assignment and heap access. There's a weakest-precondition version of these axioms, but I'm going to stick to forward reasoning in simple examples and won't need it.

In non-concurrent separation logic $\{Q\}\, C\, \{R\}$ is a no-fault total-correctness assertion: if you execute C in a situation whose resources are described by Q then it will terminate leaving resources described by R and its execution won't go outside the resource footprint described (that is, use a cell that it doesn't own). As we shall see, C can acquire or release resources during its operation, thus increasing or reducing its footprint, but the specification says that it will never exceed the footprint it has at any time.

We won't always be dealing with single heap cells, so we need a logical mechanism to handle combination of resources. Separation logic's 'star' does that: $A \star B$ says that we own resources described by A and at the same time but separately, resources described by B. It follows that

$$E \mapsto F \star E' \to F' \Rightarrow E \neq E' \tag{1.3}$$

– separation in the heap is separation of addresses. We allow address arithmetic: $E \mapsto F1, F2$ is shorthand for $E \mapsto F1 \star E + 1 \mapsto F2$. The normal conjunction $A \wedge B$ has a resource interpretation too: we own resources which are described by A and at the same time described by B. There's a resource implication $\mathrel{-\!\!\star}$ as well, but I'm not going to use it. And that's it so far as logical assertions go: separation logic uses classical logic with a separating conjunction and resource implication.

With no more than that logical machinery it's possible to describe how, for example, a list is represented in the heap. Lists are non-cyclic and end in nil (a special non-pointing pointer value); ever since the invention of LISP they have been a sequence of two-cell records the first cell of which contains a head value, the second a pointer to the rest of the list. The heap predicate $\mathrm{list}\, E\ \beta s$ says that E points to a list in the heap containing the sequence of head-values βs (it's true just exactly when the heap contains such a list and nothing else: combination of this heap with other heaps using $\star$ then expresses separation of the list from other data structures).

$$\begin{array}{ll} \mathrm{list}\, x\ \langle\,\rangle & \stackrel{\text{def}}{=} x = \mathrm{nil} \wedge \mathbf{emp} \\ \mathrm{list}\, x\ (\langle\alpha\rangle +\!\!+ \beta s) & \stackrel{\text{def}}{=} \exists x' \cdot (x \mapsto \alpha, x' \star \mathrm{list}\, x'\ \beta s) \end{array} \tag{1.4}$$

The first line says that an empty sequence is represented by nil and doesn't take any space: **emp** asserts ownership of nothing and is a zero, i.e. $A \star \mathbf{emp} \iff A$. The second line says that to represent a sequence which starts with α we have to start with a record that contains α and points to the rest of the list, which is contained in an entirely separate area of the heap. This captures precisely the separation of the head record from the rest of the list – and, recursively, separation of cells within the rest of the list. Notice also

that the assertion $x \mapsto \alpha, x'$ contains a 'dangling pointer' x': we may think we know it's a pointer but we don't know, within that assertion, what it points to. Separation logic embraces the dangling pointer, and that's the source of some of its power. In the conjunction $x \mapsto \alpha, x' \star \operatorname{list} x'\ \beta s$ we can see what x' points to, but in $x \mapsto \alpha, x'$ alone we don't know that it points to anything at all.

We can use a similar definition to describe how to represent a binary tree, with values at the nodes.

$$\begin{array}{ll} \operatorname{tree} x\ (\,) & \stackrel{\text{def}}{=} x = \operatorname{nil} \wedge \mathbf{emp} \\ \operatorname{tree} x\ (\alpha, \lambda, \rho) & \stackrel{\text{def}}{=} \exists l, r \cdot (x \mapsto l, \alpha, r \star \operatorname{tree} l\ \lambda \star \operatorname{tree} r\ \rho) \end{array} \tag{1.5}$$

An empty tree is nil; the root is separate from the left and right subtrees.

The most important inference rule of separation logic is the frame rule.

$$\frac{\{Q\}C\{R\}}{\{P \star Q\}C\{P \star R\}}\ (\operatorname{modifies} C\ \cap \operatorname{vars} P = \emptyset) \tag{1.6}$$

If we can prove $\{Q\}\,C\,\{R\}$, and if P is separate from Q, then execution of C can't affect it: so if we start C in a state described by $P \star Q$, it must terminate in $P \star R$. There's a side condition on the use of variables: C mustn't alter variables so as to alter the meaning of P. (A treatment of variables as resource – see section 1.7 – eliminates that side condition, but for the time being it's easiest to take it on board.)

Reading upwards, the frame rule enables us to focus on the minimum resources required by a program, and in particular it allows elegant axioms for new and dispose (similar to C's `malloc` and `free`).

$$\{\mathbf{emp}\}\,x := \operatorname{new}()\,\{x \mapsto \_\} \tag{1.7}$$

$$\{E \mapsto \_\}\ \operatorname{dispose} E\,\{\mathbf{emp}\} \tag{1.8}$$

In order to work with the frame rule, new must give us a pointer to a cell that isn't in use anywhere else in the program. That's easy to implement: new has a heap of cells, initially we have none, and each call to new merely transfers one from its heap to our own. Dispose pulls the reverse trick, and crucially, just as in real life, it leaves a dangling pointer – or maybe hundreds of dangling pointers – to the cell which we used to own. Crucially again, because of the frame rule, that cell can't be referred to in any assertion in the program. And, because new isn't constrained to give us brand-new cells, it may, just as in real life, give us a pointer to a cell which we used to own and to which there are dangling pointers. Heap programmers must learn to deal with that, just as they always have.

Now it's possible to show a first example. Disposetree disposes of a tree, guarantees to leave nothing behind, and doesn't put a foot outside the tree you give it – i.e. $\{\operatorname{tree} E\ \tau\}$ $\operatorname{disposetree} E\,\{\mathbf{emp}\}$. Figure 1.1 shows the procedure, and figure 1.2 its proof. The proof is a tedious calculation for such a small

result, but it is something that a compiler could follow, and indeed Berdine and Calcagno's Smallfoot tool [2] can guess this proof, given the definition of the tree predicate and the pre- and postconditions of disposetree.

```
disposetree x =
  if x = nil then skip else
    local l, r; l := [x]; r := [x + 2];
    disposetree l; disposetree r;
    dispose x; dispose(x + 1); dispose(x + 2)
  fi
```

Fig. 1.1. Tree disposal

```
disposetree x =
  {tree x τ}
  if x = nil then
    {x = nil ∧ emp}
    {emp}
    skip
    {emp}
  else
    {∃l, r · (x ↦ l, α, r ⋆ tree l λ ⋆ tree r ρ)}
    local l, r; l := [x]; r := [x + 2];
    {x ↦ l, α, r ⋆ tree l λ ⋆ tree r ρ}
    Framed: ⟨ {tree l λ} disposetree l {emp} ⟩;
    {x ↦ l, α, r ⋆ emp ⋆ tree r ρ}
    Framed: ⟨ {tree r ρ} disposetree r {emp} ⟩;
    {x ↦ l, α, r ⋆ emp ⋆ emp}
    Framed: ⟨ {x ↦ l} dispose x {emp} ⟩;
    {emp ⋆ x + 1 ↦ α, r ⋆ emp ⋆ emp}
    Framed: ⟨ {x + 1 ↦ α} dispose(x + 1) {emp} ⟩;
    {emp ⋆ emp ⋆ x + 2 ↦ r ⋆ emp ⋆ emp}
    Framed: ⟨ {x + 2 ↦ r} dispose(x + 2) {emp} ⟩;
    {emp ⋆ emp ⋆ emp ⋆ emp ⋆ emp}
    {emp}
  fi
```

Fig. 1.2. Tree disposal is safe

This approach to pointers gets us quite a long way, but as the examples in [4] show, it's not an immediate panacea. There is perhaps more to be wrung out of pointer problems, but the truth is that concurrency is so much more

exciting that the separation-logic research community has for the time being turned its attention there.[2]

1.5 A resource logic for concurrency

Since binary trees consist of three separate parts, there's no reason in principle why we shouldn't dispose of those three parts in parallel. I can show that the procedure in figure 1.3, which does just that, is safe. The disjoint-concurrency rule is

$$\frac{\{Q1\}\ C1\ \{R1\} \quad \dots \quad \{Qn\}\ Cn\ \{Rn\}}{\{Q1 \star \dots \star Qn\}\ (\ C1 \parallel \dots \parallel Cn\ )\ \{R1 \star \dots \star Rn\}}\ \left(\begin{pmatrix}\text{modifies } Cj \cap \\ (\text{vars } Qi \cup \text{vars } Ri)\end{pmatrix} = \emptyset\right) \tag{1.9}$$

– essentially, if you start concurrent threads with separated resources, they stay separated, and they produce separated results. The side condition on the use of variables says that threads can't modify variables free in each other's pre- and postconditions: for the time being it's simplest to imagine that $C1 \dots Cn$ each have their own variables.

```
disposetree x =
  if x = nil then skip else
    local l, r; l := [x]; r := [x + 2];
    ( disposetree l || disposetree r || dispose(x, x + 1, x + 2) )
  fi
```

Fig. 1.3. Tree disposal in parallel

The relevant part of the parallel disposetree proof is in figure 1.4. It isn't an industrial-strength example, but the point is in the simplicity of the reasoning. Parallel mergesort, and even parallel quicksort, can be proved safe in the same way (see [21]).

$$\begin{array}{l}\{x \mapsto l, \alpha, r \star \text{tree}\, l\ \lambda \star \text{tree}\, r\ \rho\} \\ \left(\begin{array}{lll} \{\text{tree}\, l\ \lambda\} & \parallel & \{\text{tree}\, r\ \rho\} \\ \text{disposetree}\, l & \parallel & \text{disposetree}\, r \\ \{\mathbf{emp}\} & \parallel & \{\mathbf{emp}\} \end{array} \right. \quad \left. \begin{array}{l} \parallel \\ \parallel \\ \parallel \end{array} \quad \begin{array}{l} \{x \mapsto l, \alpha, r\} \\ \text{dispose}(x, x+1, x+2) \\ \{\mathbf{emp}\} \end{array} \right) \\ \{\mathbf{emp} \star \mathbf{emp} \star \mathbf{emp}\}\end{array}$$

Fig. 1.4. Safety of the parallel section of parallel disposetree

[2] In fact, as I write, attention has turned back to heap-and-pointer problems. An automatic safety verification of a version of `malloc` has been completed, and the same techniques are being applied to finding bugs in Windows device drivers.

Concurrency without communication would be pretty boring. Separation logic uses a version of Hoare's conditional critical region (CCR) mechanism [16], which is easy to reason about but hard to implement, and then uses that to describe atomic actions that are easier to implement, like semaphores and `CAS` instructions. A program contains a number of resource names[3], each of which owns some heap. Each resource name r has an invariant I_r. The rule which sets up the resource is

$$\frac{\{Q\}\, C\, \{R\}}{\{I_r \star Q\}\, \mathsf{let\ resource}\ r : I_r\ \mathsf{in}\ C\ \mathsf{ni}\, \{R \star I_r\}} \tag{1.10}$$

(The formula I_r has to be 'precise' (i.e. pick out at most a single sub-heap in any heap: see [8]), but that's a minor technical point – program specifications are naturally precise for the most point.) Then the CCR command $\mathsf{with}\ r\ \mathsf{when}\ G\ \mathsf{do}\ C\ \mathsf{od}$ waits for exclusive access to resource name r: when it gets it, it evaluates the guard G; if the guard is false, it releases r and tries again; if the guard is true it executes C using the resources of the program plus the resources of I_r, finally re-establishing I_r and releasing r. The rule says just that, implicitly depending on the condition about exclusive access:

$$\frac{\{(Q \star I_b) \wedge G\}\, C\, \{R \star I_b\}}{\{Q\}\, \mathsf{with}\ r\ \mathsf{when}\ G\ \mathsf{do}\ C\ \mathsf{od}\, \{R\}} \tag{1.11}$$

The variable-use side-condition on this rule is so horrid that it's normally hidden, and I'll follow that tradition because I shall show in section 1.7 how to eliminate it. Much more important is to note that now we are into partial correctness: a proof in the antecedent that command C executes properly doesn't establish that the CCR command as a whole is ever successfully executed, because it might never be possible to get exclusive access to r when G holds. It only proves that *if* the CCR command is executed, *then* it transforms Q into R. So in concurrent separation logic including this rule, $\{Q\}\, C\, \{R\}$ is a no-fault partial-correctness assertion: C, given resources Q will not go wrong (exceed its resource footprint, use a cell it doesn't own) and *if* it terminates, it will deliver resources R. Partial correctness restricts the range of the logic, as we shall see, but it leaves plenty of interesting questions.

There's a nice example which shows how it all works. Suppose for the moment that we can implement a CCR command. Then the program in figure 1.5 can be shown to be safe. It allocates a heap cell in the left-hand thread, which it passes into the resource *buf*, from where the right-hand thread takes it out and disposes it. Of course the heap cell doesn't move: what moves is *ownership* of that cell. Once it has put it in the resource, the left-hand thread can't use that cell, because it doesn't own it anymore (see (1.2)), it can't modify it for just the same reason (see (1.1)) and it can't dispose it (see

[3] I think they should really be called resource *bundles*, because separation logic is about resource and resources in various forms, but history is hard to shake off.

(1.8)). We have, in effect, zero-cost execution barriers between threads so far as heap space is concerned – or we shall have, once we produce compilers that can carry out the necessary static checks.

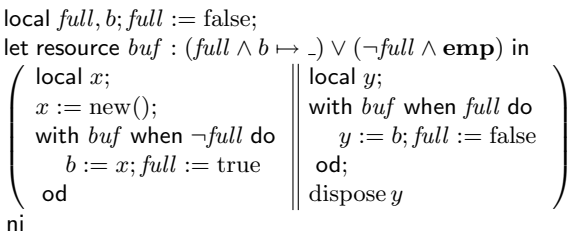

Fig. 1.5. A pointer-transferring buffer (adapted from [21])

The pointer-transferring program is unremarkable, no more than a single-place buffer transferring a pointer value. It's the separation-logic treatment that makes it special. Figure 1.6 gives the proof and shows how the left thread is left with nothing once ownership is transferred into the buffer, and conversely the right thread gets a cell ownership from the buffer. Each side starts with **emp** and finishes with **emp**; this proof doesn't show that the buffer resource finishes with **emp** too, but it does (see [21] for a proof). The trickery is all in the lines that deal with attachment and detachment of I_{buf} in the CCR commands. On the left we know on entry $\neg full$, so we know that the invariant reduces to $\neg full \wedge \mathbf{emp}$; on exit we know $full$, so the invariant reduces to $b \mapsto \_ \wedge full$, which we can provide – but there isn't any heap left over, so the thread is left with **emp**. Proof in the right thread is similar, but in the other direction.

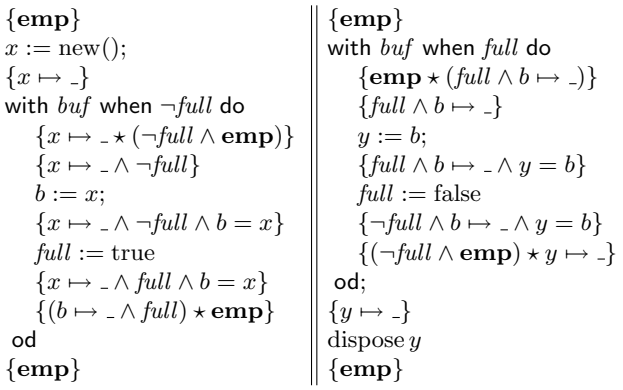

Fig. 1.6. Safety of pointer-transferring buffer

1.5.1 Dealing with semaphores

Semaphores are hard to reason with, so the more tractable CCRs and monitors [7] were invented. But separation logic has opened up the possibility of *modular* reasoning about semaphores. Crucially, we treat semaphores as a kind of CCR resource, reversing history, and in an inversion of the usual view treat them as come-in stores rather than keep-out locks.

Semaphores were invented to be guards of 'critical sections' of code, like railway signals at the ends of a block of track, with hardware mutual exclusion of execution guaranteed on P and V instructions. P(m), with a binary semaphore m, blocks – i.e. the executing thread is suspended – when $m = 0$; when $m \neq 0$ it executes $m := 0$ and proceeds. $V(m)$ simply executes $m := 1$. Then $P(m)$ can be used to guard the entrance to a critical section, and V(m) will release it on exit. The effect, with careful use of semaphores, is to produce software mutual exclusion of execution of separate critical sections each guarded by the same semaphore – i.e. bracketed with P(m) and V(m) for the same m. Consider, for example, the example in figure 1.7, which contains a parallel composition of lots of copies of a couple of critical sections, the first of which increments a heap cell, the second decreasing it. If all the uses of semaphore m are those shown, then there can be no interference – no 'races' – between the instructions which access the heap cell. But it is one thing to be sure that this is so, and quite another to prove it in a way that a formal tool such as a compiler can check your proof – or, better still, find a proof for itself.

$$
\begin{array}{l}
\mathsf{local}\ x, m, c; x := \mathrm{new}(); [x] := 0; m := 0; \\
\mathsf{let\ semaphore}\ m : (m = 0 \wedge \mathbf{emp}) \vee (m = 1 \wedge x \mapsto \_)\ \mathsf{in} \\
\left(\begin{array}{l|c|l}
\mathrm{P}(m); & & \mathrm{P}(m); \\
\quad c := [x]; c{+}{+}; [x] := c; & & \quad c := [x]; c{+}{+}; [x] := c; \\
\mathrm{V}(m); & & \mathrm{V}(m); \\
\ldots & \ldots & \ldots \\
\mathrm{P}(m); & & \mathrm{P}(m); \\
\quad c := [x]; c{-}{-}; [x] := c; & & \quad c := [x]; c{-}{-}; [x] := c; \\
\mathrm{V}(m) & & \mathrm{V}(m)
\end{array}\right) \\
\mathsf{ni}
\end{array}
$$

Fig. 1.7. Example critical sections

P(m) for a binary semaphore m is in effect $\mathsf{with}\ m\ \mathsf{when}\ m = 1\ \mathsf{do}\ m := 0\ \mathsf{od}$, and V($m$) is $\mathsf{with}\ m\ \mathsf{when}\ \mathrm{true}\ \mathsf{do}\ m := 1\ \mathsf{od}$ – note the pun in each case on the resource-name m and the semaphore-variable name m. Then in separation logic the semaphore must have an invariant: in figure 1.7 I've shown it as $(m = 0 \wedge \mathbf{emp}) \vee (m = 1 \wedge x \mapsto \_)$. It's possible then to prove, using the CCR rule, that P(m) retrieves ownership of the heap cell x from the semaphore (figure 1.8) and V(m) restores it (figure 1.9 – the simplification between the

first and second lines of the braced proof is possible since $x \mapsto \_ \star x \mapsto \_$ is false because of (1.3)).

$$
\begin{array}{l}
\{\mathbf{emp}\} \\
\mathrm{P}(m): \left\{ \begin{array}{l}
\{(\mathbf{emp} \star ((m = 0 \wedge \mathbf{emp}) \vee (m = 1 \wedge x \mapsto \_))) \wedge m = 1\} \\
\{\mathbf{emp} \star (m = 1 \wedge x \mapsto \_)\} \\
\{m = 1 \wedge x \mapsto \_\} \\
\quad m := 0 \\
\{m = 0 \wedge x \mapsto \_\} \\
\{(m = 0 \wedge \mathbf{emp}) \star x \mapsto \_\} \\
\{((m = 0 \wedge \mathbf{emp}) \vee (m = 1 \wedge x \mapsto \_)) \star x \mapsto \_\}
\end{array} \right\} \\
\{x \mapsto \_\}
\end{array}
$$

Fig. 1.8. P retrieves heap cell x from semaphore m in figure 1.7

$$
\begin{array}{l}
\{x \mapsto \_\} \\
\mathrm{V}(m): \left\{ \begin{array}{l}
\{(x \mapsto \_ \star ((m = 0 \wedge \mathbf{emp}) \vee (m = 1 \wedge x \mapsto \_))) \wedge \mathrm{true}\} \\
\{x \mapsto \_ \star (m = 0 \wedge \mathbf{emp})\} \\
\{m = 0 \wedge x \mapsto \_\} \\
\quad m := 1 \\
\{m = 1 \wedge x \mapsto \_\} \\
\{(m = 1 \wedge x \mapsto \_) \star \mathbf{emp}\} \\
\{((m = 0 \wedge \mathbf{emp}) \vee (m = 1 \wedge x \mapsto \_)) \star \mathbf{emp}\}
\end{array} \right\} \\
\{\mathbf{emp}\}
\end{array}
$$

Fig. 1.9. V replaces heap cell x in semaphore m in figure 1.7

Then the reasoning which lets us conclude that threads don't collide in critical sections is based on separation logic's *separation principle*: if threads start out with separated resources, if resource-names have separate invariants, and if all threads play by the rules, staying within their own separate resource footprints and communicating through CCR commands, then their resources are separate at every instant. After successfully executing $\mathrm{P}(m)$, a thread has $x \mapsto \_$, and it keeps hold of that cell until it executes $\mathrm{V}(m)$. No other thread can own or access that cell, by the separation principle. So, throughout the critical section, a thread has exclusive ownership of the cell pointed to by x, and there can't be a race. And that's *local* reasoning: we don't have to worry about what other threads are doing, merely that they are obeying the rules (that is, they are staying within their own local resource footprint). We have provable software mutual exclusion.

Before I show any more of the mechanisms of separation logic, I want to deal with an example of the sort of program that got semaphores a bad-reasoning name. O'Hearn's version of the unbounded buffer is shown in figure 1.10. A sample state of the buffer is shown in figure 1.11: producer and con-

sumer share a counting semaphore n; the buffer always holds a list of exactly n cells; the consumer has a (dangling) pointer to the front of the buffer list, and the back of the list always (danglingly) points to a cell owned by the producer. Because the algorithm works in terms of two-cell records I've used cons and uncons rather than new and dispose, and I've supposed there's a produce function and a consume procedure to deal with the values communicated.

```
⎛ CONSUMER                  ‖ BUFFER                  ‖ PRODUCER                  ⎞
⎜                           ‖                         ‖                           ⎟
⎜ while true do             ‖                         ‖ while true do             ⎟
⎜     tc := front;          ‖                         ‖     [back] := produce();  ⎟
⎜     P(n);                 ‖                         ‖     tp := cons();         ⎟
⎜     front := [front + 1]; ‖ semaphore n := 0        ‖     [back + 1] := tp;     ⎟
⎜     consume[tc];          ‖                         ‖     V(n);                 ⎟
⎜     uncons tc             ‖                         ‖     back := tp            ⎟
⎝ od                        ‖                         ‖ od                        ⎠
```

Fig. 1.10. An unbounded buffer (adapted from [21])

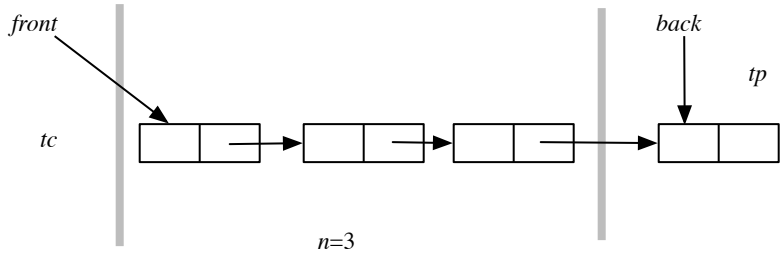

Fig. 1.11. A sample state of the unbounded buffer

The most startling thing about this algorithm is that there are no critical sections. Only the consumer Ps, and only the producer Vs. Nevertheless we shall see that because producer and consumer work at opposite ends of the buffer list, there is effective separation – even though, when the buffer list is empty, the consumer's dangling pointer points to the producer's private cell.

The details of the proof require some more information. First, because n is a counting semaphore, P(n) is with n when $n \neq 0$ do $n := n-1$ od and V(n) is with n when true do $n := n + 1$ od – only a little harder to implement in mutual exclusion than a binary semaphore. Second, in order to describe the resource held by the semaphore, I have to define a list segment – a straight-

line section of a list that doesn't necessarily end in nil and doesn't loop back on itself:

$$\begin{array}{l}\text{listseg}\, n\ x\ y \stackrel{\text{def}}{=} \\ \quad (n = 0 \wedge x = y \wedge \mathbf{emp}) \vee \\ \quad\quad (n > 0 \wedge x \neq y \wedge \exists x' \cdot (x \mapsto \_, x' \star \text{listseg}\ (n-1)\ x'\ y)))\end{array} \tag{1.12}$$

(note that in listseg n E F, E always points to the front of the segment, unless it's empty, but F is the pointer at the end of the segment, not the address of its last cell). Finally, I have to use two 'auxiliary variables' – variables put in for the purposes of the proof, whose values don't affect the operation of the program. I need a variable f that always points to the front of the buffer-list-segment, and a variable b that always tracks the pointer at the end of that segment. From now on I'll highlight auxiliary variables like this – $\boxed{f}$, $\boxed{b}$ – as a reminder that they don't really exist.

Now the consumer's loop-invariant is $\mathit{front} = \boxed{f} \wedge \mathbf{emp}$; the producer's is $\mathit{back} = \boxed{b} \wedge \mathit{back} \mapsto \_, \_$, and the counting-semaphore invariant is listseg n $\boxed{f}$ $\boxed{b}$. P(n) has to move $\boxed{f}$ forward, to allow the consumer to take possession of the first cell in the list – it can do so instantaneously, because $\boxed{f}$ doesn't really exist – and becomes with n when $n \neq 0$ do $n := n-1$; $\boxed{f} := [\boxed{f}+1]$ od; similarly V(n) becomes with n when true do $n := n+1$; $\boxed{b} := [\boxed{b}+1]$ od.

With those preparations the proof is straightforward, shown in figure 1.12 and figure 1.13. By reducing n and moving $\boxed{f}$ one place forward, the P(n) operation leaves an orphaned cell, which is pointed to by *front* in the producer. By increasing n and moving $\boxed{b}$ one place forward, the V(n) operation absorbs the producer's private cell into the buffer list (it takes an induction, which I've omitted, to show that a list can be extended at its tail in that way).

If you've been paying really close attention you will have noticed something fishy. In the last example the auxiliary variables $\boxed{f}$ and $\boxed{b}$ were shared between the producer/consumer threads and the buffer, and the buffer operations altered them even though they appear also in the loop invariants of the threads. It's more than a little unclear that the variable-use side conditions on the CCR rule are being obeyed. Actually they are, but it takes a little more logical machinery to show it.

1.6 Permissions

Dijkstra's methodology for concurrency was to make processes/threads loosely coupled, apart from moments of synchronisation. To achieve that we are supposed to divide the variables used by a thread into three groups:

- those used exclusively by the thread, which can be read or written;
- those shared with other threads, which can only be read;
- those read and written in mutual exclusion (e.g. in critical sections, or in CCR commands).

$\{front = \boxed{f} \wedge \mathbf{emp}\}$
$\quad tc := front$
$\{tc = front \wedge front = \boxed{f} \wedge \mathbf{emp}\}$

$$\mathrm{P}(n) : \left\{ \begin{array}{l} \{((tc = front \wedge front = \boxed{f} \wedge \mathbf{emp}) \star \mathrm{listseg}\, n\ \boxed{f}\ \boxed{b}) \wedge n \neq 0\} \\ \{tc = front \wedge front = \boxed{f} \wedge (\exists x \cdot (\boxed{f} \mapsto \_, x \star \mathrm{listseg}\ (n-1)\ x\ \boxed{b}))\} \\ \quad n := n - 1 \\ \{tc = front \wedge front = \boxed{f} \wedge (\exists x \cdot (\boxed{f} \mapsto \_, x \star \mathrm{listseg}\, n\ x\ \boxed{b}))\} \\ \quad \boxed{f} := [\boxed{f} + 1] \\ \{tc = front \wedge (front \mapsto \_, \boxed{f} \star \mathrm{listseg}\, n\ \boxed{f}\ \boxed{b})\} \\ \{(tc = front \wedge front \mapsto \_, \boxed{f}) \star \mathrm{listseg}\, n\ \boxed{f}\ \boxed{b})\} \end{array} \right\}$$

$\{tc = front \wedge front \mapsto \_, \boxed{f}\}$
$\quad front := [front + 1]$
$\{tc \mapsto \_, \boxed{f} \wedge front = \boxed{f}\}$
$\quad \mathrm{consume}[tc]$
$\{tc \mapsto \_, \boxed{f} \wedge front = \boxed{f}\}$
$\quad \mathrm{uncons}\, tc$
$\{front = \boxed{f} \wedge \mathbf{emp}\}$

Fig. 1.12. Consumer loop invariant is safely preserved

$\{back = \boxed{b} \wedge back \mapsto \_, \_\}$
$\quad [back] := \mathrm{produce}()$
$\{back = \boxed{b} \wedge back \mapsto \_, \_\}$
$\quad tp := \mathrm{cons}()$
$\{back = \boxed{b} \wedge (back \mapsto \_, \_ \star tp \mapsto \_, \_)\}$
$\quad [back + 1] := tp$
$\{back = \boxed{b} \wedge (back \mapsto \_, tp \star tp \mapsto \_, \_)\}$

$$\mathrm{V}(n) : \left\{ \begin{array}{l} \{(back = \boxed{b} \wedge (back \mapsto \_, tp \star tp \mapsto \_, \_)) \star \mathrm{listseg}\, n\ \boxed{f}\ \boxed{b}\} \\ \quad n := n + 1 \\ \{(back = \boxed{b} \wedge (back \mapsto \_, tp \star tp \mapsto \_, \_)) \star \mathrm{listseg}\ (n-1)\ \boxed{f}\ \boxed{b}\} \\ \quad \boxed{b} := [\boxed{b} + 1] \\ \{(\mathrm{listseg}\ (n-1)\ \boxed{f}\ back \star back \mapsto \_, tp \star tp \mapsto \_, \_) \wedge \boxed{b} = tp\} \\ \{(\mathrm{listseg}\ n\ \boxed{f}\ tp \star tp \mapsto \_, \_) \wedge \boxed{b} = tp\} \\ \{\mathrm{listseg}\ (n-1)\ \boxed{f}\ \boxed{b} \star (tp \mapsto \_, \_ \wedge \boxed{b} = tp)\} \end{array} \right\}$$

$\{tp \mapsto \_, \_ \wedge \boxed{b} = tp\}$
$\quad back := tp$
$\{back = \boxed{b} \wedge back \mapsto \_, \_\}$

Fig. 1.13. Producer loop invariant is safely preserved

I've been describing a logic which deals with concurrent use of heap cells rather than variables, but the principle is the same. $E \mapsto F$ (exclusive ownership) corresponds to the first case, and the use of resource-name invariants allows me to deal with the third. But so far there is nothing in heap cells that corresponds to read-only shared variables.

The solution, as Boyland pointed out in [6], is to realise that ownership is licence to do anything we like with a cell (read, write, dispose), but that this can be cut up into a number of fractional *permissions*, each of which allows

only read access. Some of the consequences of this idea are explored in [4]: since that paper we've gone beyond the initial idea of fractional permissions, and some of our newer ideas will emerge in later examples.

With simple ownership $E \mapsto \_ \star E \mapsto \_$ is simply false: you can't split a heap so that both halves contain the same cell, so you can't give exclusive ownership of a cell in the disjoint-concurrency rule (1.9) to two threads. But you can give each thread a fractional permission, via which it can read but not write or dispose. We write $E \underset{z}{\mapsto} F$ for a fractional permission, and we have

$$E \underset{1}{\mapsto} F \iff E \mapsto F \tag{1.13}$$

$$E \underset{z}{\mapsto} F \Rightarrow 0 < z \leq 1 \tag{1.14}$$

$$E \underset{z}{\mapsto} F \star E' \underset{z'}{\mapsto} F' \Rightarrow E = E' \Rightarrow (F = F' \wedge z + z' \leq 1) \tag{1.15}$$

$$E \underset{z}{\mapsto} F \wedge 0 < z' < z \iff z \leq 1 \wedge (E \underset{z-z'}{\longmapsto} F \star E \underset{z'}{\mapsto} F) \tag{1.16}$$

– $\underset{1}{\mapsto}$ means total ownership; we can only use non-zero fractions; we can combine permissions provided we don't exceed a $\underset{1}{\mapsto}$ permission; we can split permissions indefinitely. Only total ownership allows writing – i.e. we still use the axiom of (1.1) – whereas reading is allowed with fractional ownership of any size (including, of course, total ownership):

$$\{F = N \wedge E \underset{z}{\mapsto} F\}\, x := [E]\, \{x = N \wedge E \underset{z}{\mapsto} N\} \tag{1.17}$$

Permissions are no more 'real' than ownership: they are artefacts of a proof, ignored by the machine but clear to the prover. In practice, apart from the restrictions applied by inter-process hardware address barriers, any instruction in a program can access any part of the heap. Like types, permissions are something we can check to be sure that our instructions only access the parts of the heap that we want them to.[4] Permissions aren't types, though: they alter dynamically as a program executes and they can be connected to the values of program variables.

All this means that we can deal with shared heap cells. But that's by no means all, because we can apply the same treatment to variables.

1.7 A resource logic for variables

Concurrent separation logic, as I've described it so far, deals with heap cells as resource, and accounts for their ownership and transfer during the operation of a program. But in some programs ownership of variables has to be accounted for as well. At a deeper level, the frame rule, the disjoint-concurrency rule

[4] There's an analogy in the real world: in canon law an unfrocked priest is capable of celebrating marriage but forbidden to do so (Marek Sergot, talk on logics of permission and belief; see also *Daily Telegraph*, 21/vi/93).

and the CCR rule have variable-use side conditions. If we treat variables as resource, those side conditions go away, and the entire load is carried by the separating conjunction ($\star$).

In conventional non-concurrent programs use of variables is controlled by scoping rules, and these work well except when you get aliasing between reference parameters and global variables in procedure calls. (Separation logic can deal with that problem too, though I'm not going to address it here.) But in concurrent programs ownership of variables gets transferred into and out of resource-holders – semaphores or CCRs or whatever – and that complicates things. Consider, for example, the unbounded buffer of figure 1.10. The auxiliary variable f is referred to in the invariants of the semaphore n and of the consumer. If each could have a half permission for that variable, that sharing could be formally legitimated – each partner can read f, neither has a total permission so neither can write. But f is written by the consumer when it executes P(n), which looks like a contradiction. There is in fact no contradiction: when the CCR command is executed, the permissions of the consumer and the invariant of the semaphore are combined ($Q \star I_n$ in the CCR rule), two halves make a whole, and it's possible both to read and write that variable for the duration of that instruction.

In order to preserve the simplicity of Hoare logic's treatment of variables as far as possible, it's necessary to split ownership assertions from assertions about the values of variables, rather like lvalues and rvalues but unlike the $\mapsto$ connective used to talk about the heap. Ownership of a variable can be described by a predicate Own, which can be subscripted like $\mapsto$ to describe fractional ownership. To simplify the notation we gather all the ownership predicates together, separate them from the rest of the assertion with a turnstile ($\Vdash$), and then don't bother to write Own at all. An assertion is then 'well-scoped' if on the left of the turnstile it claims ownership of all the variables mentioned to the right of the turnstile. So, for example, the consumer's loop invariant in the program of figure 1.10 is $\mathit{front}, \mathit{tc}, f_{0.5} \Vdash \mathit{front} = f \wedge \mathbf{emp}$: it owns *front* and *tc* exclusively and it owns half of f, so it can state the relationship between *front* and f legitimately. The semaphore invariant is $f_{0.5}, b_{0.5} \Vdash \mathrm{listseg}\, n\ f\ b$: it owns half each of f and b, so it can use their values to delimit the list. The producer's loop invariant is $\mathit{back}, \mathit{tp}, b_{0.5} \Vdash \mathit{back} = b \wedge \mathit{back} \mapsto \_, \_$, claiming ownership of *back* and *tp* and half (i.e. read) permission for b.

To assign to a variable you need total permission:

$$\{\Gamma, x \Vdash R[E/x]\}\, x := E\, \{\Gamma, x \Vdash R\} \tag{1.18}$$

This axiom needs the side-condition that both assertions are well-scoped. An initial version of the logic is in [5]. A later version, which eliminates even that side condition, is in [27].

It is possible to deal with access to semaphores using permissions – the semaphore keeps a permission to read and write its own value, clients get permission to P and V it – but in this paper I'll spare you the details, and ignore the fact that semaphores are really variables.

Treating variables as resource is very fiddly, but it is essential to explain the operation of some very famous algorithms like readers and writers.

1.7.1 Readers and writers

The readers-and-writers algorithm of figure 1.14 shows how a shared variable (or any shared data structure) may be given either many simultaneous readers or a single writer (we don't need to avoid races between readers, but we mustn't race readers and writers, or writers and writers). It's a very succinct algorithm, but it has quite a tricky proof. It has three visible critical sections, but the 'reading happens' part isn't one of them, because there can be many readers in it at the same time. Nevertheless, that part is clearly executed in software mutual exclusion with the 'writing happens' critical section. The problem is to explain how this is so.

The easiest part of the solution is to recognise that the m semaphore owns the c variable, which it uses to count the readers. It's then impossible to read or write c without first going through $\mathrm{P}(m)$ – not at all a scoping restriction. Then clearly c counts the number of read permissions that have been handed out, and that's where the fun begins. It might be possible to track fractional permissions, but not without unreasonable arithmetic agility (I certainly can't work out how to do it!) so instead I use a 'counting permission'. The idea is that you have a permission which is like a flint block, from which you chip away read permissions rather like flake arrowheads. Each flake can, in principle, be glued back into the block, and each time you chip away a flake you still have a block in your hand.

$E \overset{N}{\longmapsto} F$, with N above the arrow to distinguish it from a fractional permission, is a block from which you have chipped away exactly N counted read permissions; $E \rightarrowtail F$ is a counted read permission. For simplicity I suppose that the counted read permissions can't be fractionally subdivided.

$$E \overset{0}{\longmapsto} F \iff E \mapsto F \tag{1.19}$$

$$E \overset{N}{\longmapsto} F \quad \Rightarrow \quad N \geq 0 \tag{1.20}$$

$$E \overset{N}{\longmapsto} F \iff E \overset{N+1}{\longmapsto} F \star E \rightarrowtail \_ \tag{1.21}$$

– $\overset{0}{\longmapsto}$ is a total permission; you can't have a negative counting permission; you can always chip away a new counted permission or recombine one with its source.

We can do counting permissions for variables too, and we superscript them by analogy with heap counting permissions: x^N corresponds to $\overset{N}{\longmapsto}$ and $x^>$ to $\rightarrowtail$. For simplicity I suppose that the readers and writers are competing for access to a single shared variable y. In the algorithm we start with a total permission for y in the w semaphore, which has the invariant

$$(\Vdash w = 0) \vee (y \Vdash w = 1) \tag{1.22}$$

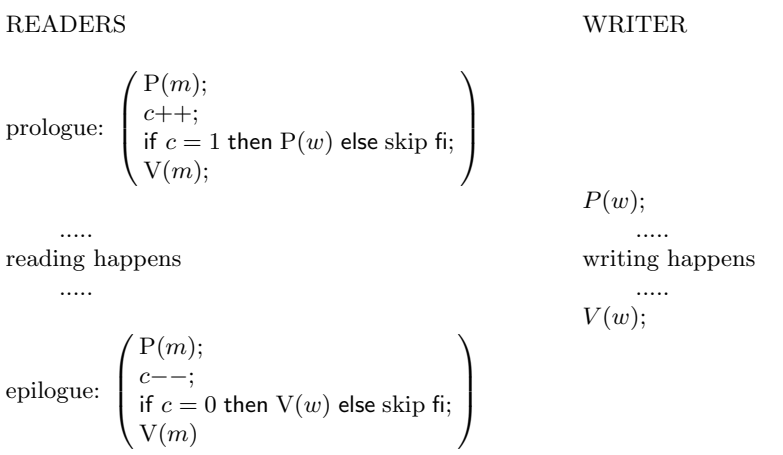

Fig. 1.14. Readers and writers (adapted from [11])

– when $w = 0$ the y-permission has been lent out; when $w = 1$ it's safely locked away. That's enough to explain how any thread can $\mathrm{P}(w)$ to gain exclusive access to the 'writing happens' critical section, and then do what it likes with y until it executes $V(w)$.

A first guess at the invariant for semaphore m might be

$$(\Vdash m = 0) \vee (c \Vdash m = 1 \wedge c = 0) \vee (c, y^c \Vdash m = 1 \wedge c > 0)$$

– when $m = 0$ everything has been lent out; when $m = 1$ either there are no readers, in which case the y permission is either with a writer or in the w semaphore, or there are c readers and it has permission for y from which it has chipped off exactly c read permissions. That's plausible, but it isn't quite right. It allows me to prove that the prologue gives a read permission for variable y – $\{\Vdash \mathrm{true}\}$ *prologue* $\{y^> \Vdash \mathrm{true}\}$ – but it doesn't let me prove that the epilogue takes it back – $\{y^> \Vdash \mathrm{true}\}$ *epilogue* $\{\Vdash \mathrm{true}\}$. The reason for the failure is quite subtle: we can't deduce from ownership of a read permission $y^>$ that $c > 0$. In practical terms, a rogue writer could check out total y ownership from the w semaphore, chip away a read permission, and then present that to the readers' epilogue: nothing in the algorithm stops it from doing so! The net effect would be to produce $c = -1$, and the m semaphore's invariant would be destroyed.

The solution is ingenious. We use an auxiliary variable t to make tickets that are handed out by the reader prologue, and which have to be handed in at the epilogue. The variable doesn't appear in the program at all, so it's the most splendidly auxiliary addition, just like f and b in the unbounded buffer. The invariant is then

$$(\Vdash m = 0) \vee (c, t \Vdash m = 1 \wedge c = 0) \vee (c, t^c, y^c \Vdash m = 1 \wedge c > 0) \qquad (1.23)$$

Now it is possible – see figure 1.15 – to prove $\{y^{>}, \boxed{t}^{>} \Vdash \text{true}\}$ *epilogue* $\{\Vdash \text{true}\}$. With the ticket, the epilogue works perfectly, and without the ticket, a rogue writer can't bust in (or rather, if it does, we can't prove that the epilogue will work properly). In combining the semaphore invariant with the precondition, we can eliminate the first alternative because $m = 1$, we can eliminate the second because $\boxed{t} \star \boxed{t}^{>}$ implies $\boxed{t}^{-1}$, which is false, and we are left with the last alternative, which tells us that $c > 0$. Reasoning is then straightforward until the conclusion of the if-then-else, which generates a disjunction: either we have y^c or we don't. That disjunction is just what is needed when splitting off the semaphore invariant at the end of $\mathrm{V}(m)$. Proof of the prologue is similar but more straightforward, since the ticket is generated but plays no significant rôle.

It's possible to do more complicated examples than figure 1.15 – in [5], for example, there's a proof of a version of readers-and-writers which avoids starvation of either group – but they don't give any extra illumination. What's happening is very close *resource accounting*, showing that separate threads can't possibly interfere with each other in their use of variables or the heap. This is useful, and to complain that it's tediously intricate is to miss the point. It's conceivable that this sort of approach may lead to programming tools that can do this kind of resource accounting, either automatically or with minimal hints from the programmer. Tools are good at tedious intricacy. Who would not say, for example, that the derivation of the type of *map* in the Hindley-Milner type system is not tediously intricate? It is, but we don't care because the compiler does the calculation for us. Maybe, one day, a compiler will do our resource accounting too. 'Resourcing' may, perhaps, be the new typing.

1.8 Non-blocking concurrency

I'd be dishonest if I pretended that everything in the separation logic pond is plain sailing. We have reached a point where we can make a good fist of the proofs of some race-free concurrent algorithms, but there are always more challenges. People nowadays, for example, are getting very excited about so-called 'non-blocking' concurrency [20], in which concurrent threads attempt to make updates to a shared data structure at exactly the same time, backing off and trying again if their attempt fails, rather in the way that database transactions have been dealt with for some time now. This is claimed to have efficiency advantages – modern processors hate to block, and if you have more than one processor concurrently acting, it makes sense to let it try to make progress – and programming advantages too, in the shape of the 'atomic' construct of the software-transactional-memory approach [14]. It's beyond the scope of this paper to consider whether transactional memory and/or non-blocking concurrency is the wave of the future. It's enough to say that it's a hot research topic, and one that separation logic might hope to pick up.

$$
\begin{array}{l}
\{y^{>}, \boxed{t}^{>} \Vdash \text{true}\} \\
\quad \mathrm{P}(m) : \left(\begin{array}{l}
\left\{ \begin{array}{l} (y^{>}, \boxed{t}^{>} \Vdash \text{true} \wedge m = 0 \wedge m = 1) \vee \\ (y^{>}, \boxed{t}^{>}, c, t \Vdash \text{true} \wedge m = 1 \wedge c = 0 \wedge m = 1) \vee \\ (y^{>}, \boxed{t}^{>}, c, \boxed{t}^{c}, y^{c} \Vdash \text{true} \wedge m = 1 \wedge c > 0 \wedge m = 1) \end{array} \right\} \\
\{y^{>}, \boxed{t}^{>}, c, \boxed{t}^{c}, y^{c} \Vdash m = 1 \wedge c > 0\} \\
\{c, \boxed{t}^{c-1}, y^{c-1} \Vdash m = 1 \wedge c > 0\} \\
\quad m := 0 \\
\{c, \boxed{t}^{c-1}, y^{c-1} \Vdash m = 0 \wedge c > 0\} \\
\{(c, \boxed{t}^{c-1}, y^{c-1} \Vdash c > 0) \star (\Vdash m = 0)\}
\end{array} \right) \\
\{c, \boxed{t}^{c-1}, y^{c-1} \Vdash c > 0\} \\
\quad c{-}{-} \\
\{c, \boxed{t}^{c}, y^{c} \Vdash c \geq 0\} \\
\quad \mathsf{if}\ c = 0\ \mathsf{then} \\
\qquad \{c, \boxed{t}^{c}, y^{c} \Vdash c \geq 0 \wedge c = 0\} \\
\qquad \{c, \boxed{t}, y \Vdash c = 0\} \\
\qquad \mathrm{V}(w) : \left(\begin{array}{l}
\left\{ \begin{array}{l} (c, \boxed{t}, y, y \Vdash c = 0 \wedge w = 1 \wedge \text{true}) \vee \\ (c, \boxed{t}, y \Vdash c = 0 \wedge w = 0 \wedge \text{true}) \end{array} \right\} \\
\{c, \boxed{t}, y \Vdash c = 0 \wedge w = 0\} \\
\quad w := 1 \\
\{c, \boxed{t}, y \Vdash c = 0 \wedge w = 1\} \\
\{(c, \boxed{t} \Vdash c = 0) \star (y \Vdash w = 1)\}
\end{array} \right) \\
\qquad \{c, \boxed{t} \Vdash c = 0\} \\
\qquad \{c, \boxed{t}^{c} \Vdash c = 0\} \\
\quad \mathsf{else} \\
\qquad \{c, \boxed{t}^{c}, y^{c} \Vdash c \geq 0 \wedge c \neq 0\} \\
\qquad \{c, \boxed{t}^{c}, y^{c} \Vdash c > 0\} \\
\qquad\quad \text{skip} \\
\qquad \{c, \boxed{t}^{c}, y^{c} \Vdash c > 0\} \\
\quad \mathsf{fi} \\
\{(c, \boxed{t} \Vdash c = 0) \vee (c, \boxed{t}^{c}, y^{c} \Vdash c > 0)\} \\
\quad \mathrm{V}(m) : \left(\begin{array}{l}
\left\{ \begin{array}{l} (c, \boxed{t} \Vdash c = 0 \wedge m = 0 \wedge \text{true}) \vee \\ (c, \boxed{t}, c, t \Vdash c = 0 \wedge m = 1 \wedge c = 0 \wedge \text{true}) \vee \\ (c, \boxed{t}, c, \boxed{t}^{c}, y^{c} \Vdash c = 0 \wedge m = 1 \wedge c > 0 \wedge \text{true}) \vee \\ (c, \boxed{t}^{c}, y^{c} \Vdash c > 0 \wedge m = 0 \wedge \text{true}) \vee \\ (c, \boxed{t}^{c}, y^{c}, c, t \Vdash c > 0 \wedge m = 1 \wedge c = 0 \wedge \text{true}) \vee \\ (c, \boxed{t}^{c}, y^{c}, c, \boxed{t}^{c}, y^{c} \Vdash c > 0 \wedge m = 1 \wedge c > 0 \wedge \text{true}) \end{array} \right\} \\
\left\{ \begin{array}{l} (c, \boxed{t} \Vdash c = 0 \wedge m = 0 \wedge \text{true}) \vee \\ (c, \boxed{t}^{c}, y^{c} \Vdash c > 0 \wedge m = 0 \wedge \text{true}) \end{array} \right\} \\
\quad m := 1 \\
\left\{ \begin{array}{l} (c, \boxed{t} \Vdash c = 0 \wedge m = 1 \wedge \text{true}) \vee \\ (c, \boxed{t}^{c}, y^{c} \Vdash c > 0 \wedge m = 1 \wedge \text{true}) \end{array} \right\} \\
\left\{ (\Vdash \text{true}) \star \left(\begin{array}{l} (c, \boxed{t} \Vdash c = 0 \wedge m = 1) \vee \\ (c, \boxed{t}^{c}, y^{c} \Vdash c > 0 \wedge m = 1) \end{array} \right) \right\}
\end{array} \right) \\
\{\Vdash \text{true}\}
\end{array}
$$

Fig. 1.15. Proof of safety of the reader epilogue

In joint work with Matthew Parkinson, I have helped to find a proof for a particularly small and simple non-blocking algorithm (Parkinson has a neater invariant, but in the interests of intellectual honesty, I give my own here). We have also been able to show that proofs like this can be made *modular*, in the sense that the racy interference necessary to make inter-thread communication can entirely be confined to the internals of the operations which implement that communication. That is, communications can be treated as black-box operations with conventional pre- and post-conditions, and the rest of the code of a thread can be handled independently, as Dijkstra recommended [28]. This is in contrast to other approaches to racy concurrency, such as rely-guarantee [19], in which the whole of a thread's proof is affected by interference. But on the other hand, again in the interests of intellectual honesty, I have to say that our proofs don't go far enough, and we hope to learn from rely-guarantee and temporal logic so as to be able to extend our logic and make modular proofs which are more readable and understandable and so more convincing.

Despite those caveats, we do have some proofs, and it isn't really necessary to apologise if work at the frontier of science is at first a little scrappy. New approaches, new avenues, new thoughts: all are likely to be more than a little messy at first. I press on.

Simpson's 4-slot algorithm [31] is shown in figure 1.16. It's a remarkable mechanism: it has *absolutely no* synchronisation instructions at all and, unlike most non-blocking algorithms, it doesn't have a loop-and-try-again structure. It implements a kind of shared variable: a single writer thread puts values into one of the slots of the shared array *data*, and a single reader thread later retrieves it. If they act at exactly the same time, then the reader gets a value that the writer put there earlier. The reader always sees a value which is no earlier than the latest value written at the instant it starts the *read* procedure, but it can be overtaken by the writer as often as maybe, so the value it reads can be somewhat out-of-date.

Simpson's algorithm relies for its correctness, as do all non-blocking algorithms, on serialisation of store operations. A shared memory will be able to read or to write but not both at the same time, nor even to overlap two operations of the same kind in the same area. 'Word'-sized operations are serialised, and synchronisation problems arise because we want to communicate using more than a single word. Simpson makes certain that most of his variables will have serialised access by making them single-bit; only the elements of the *data* array can be larger and cannot therefore be assumed to be hardware-serialised. In figure 1.16 the operations that we need to worry about are boxed: every other assignment involves only a single-bit access or a single-bit update to shared variables. In effect, the hardware makes those accesses and updates mutually exclusive, and we can treat them as if they were CCR commands for a single resource-name which owns all the shared store, both variables and array.

```
var   readcol := 0, latest := 0 : bit
      slot :  array bit of bit := {0, 0}
      data :  array bit of  array bit of datatype

procedure  write   (item : datatype);
           var     col, row : bit;
           begin
                   col := not(readcol);
                   row := not(slot[col]);
                   [data[col, row] := item];
                   slot[col] := row;
                   latest := col
           end;

procedure  read :  datatype;
           var     col, row : bit;
           begin
                   col := latest;
                   readcol := col;
                   row := slot[col];
                   [read := data[col, row]]
           end;
```

Fig. 1.16. Simpson's 4-slot algorithm (adapted from [31])

Essentially that's the way the proof works. Writing 'atomic{C}' as shorthand for 'with *shareddata* when true do C od', and adding auxiliary variables *writecol* to indicate the column the writer is working in and *readrow* for the row the reader is looking at, the code of the algorithm is shown in figure 1.17. It's easy to see that each atomic instruction reads or writes exactly one one-bit global variable (note that each procedure has its own local variables *col* and *row*) or one element from the bit-array *slot*. (In separation logic *slot* is a pointer to two heap cells and *slot*[*col*] is really [*slot* + *col*], and similarly *data*[*col*, *row*] can be translated into a pointer operation, but I've left the array notation in the program and the proof, for simplicity's sake.)

Just as in the unbounded buffer, ownership of a variable or a heap cell can be part in a thread, part in the shared resource. Some variables never need to be written – the *slot* and *data* pointers, for example – but all three components need to refer to them, so ownership can be split three ways. Ownership of those which are written in an atomic operation – *latest*, for example – can be split between a thread and the shared data invariant: then the other thread can read that variable in an atomic instruction, but not write it.

The proof depends on the fact that the writer uses a slot in the *writecol* column, and the reader uses the *readrow* row of the *readcol* column. These slots, we shall see, don't overlap. The pre- and post-condition of the *write* procedure is

```
var       readcol := 0, latest := 0 : bit
          slot : array bit of bit := {0, 0}
          data : array bit of array bit of datatype
auxvar    writecol := −1, latest := −1 : twobit

procedure write (item : datatype);
          var col, row : bit;
          begin
                atomic{col := not(readcol); writecol := col};
                row := not(slot[col]);
                [data[col, row] := item];
                atomic{slot[col] := row; writecol := −1};
                atomic{latest := col}
          end;

procedure read : datatype;
          var col, row : bit;
          begin
                atomic{col := latest};
                atomic{readcol := col};
                atomic{row := slot[col]; readrow := row};
                [read := data[col, row]];
                atomic{readrow := −1}
          end;
```

Fig. 1.17. Simpson's 4-slot algorithm with 'atomic' annotations and auxiliary variables

$$latest_{0.5}, slot_{0.5}, data_{0.33}, writecol_{0.5}, col, row \Vdash writecol = -1 \wedge slot \xmapsto[0.5]{} \_, \_ \tag{1.24}$$

– it shares *latest*, *slot* and *writecol* with the shared invariant, and *data* with the shared invariant and the reader; and it shares the content of the *slot* array with the shared invariant. When *writecol* is 0 or 1 inside the procedure, the writer also owns $data[writecol, \mathrm{not}(slot[writecol])]$.

The reader is slightly simpler because it doesn't refer to the *slot* array:

$$readcol_{0.5}, readrow_{0.5}, data_{0.33}, col, row \Vdash readrow = -1 \tag{1.25}$$

– it shares *readcol* and *readrow* with the shared invariant, and *data* with the shared invariant and the reader. When *readrow* is 0 or 1, the reader also owns $data[readcol, readrow]$. By the separation principle, this can't be the same as the $data[writecol, \mathrm{not}(slot[writecol])]$ slot which the writer owns. To show that everything works, it's only necessary to show that the atomic operations which allocate slots to the writer and the reader can always do their job and allocate separate slots.

The invariant of the shared data resource is shown in figure 1.18. It's possible to express this invariant more succinctly, as Parkinson does, but I've

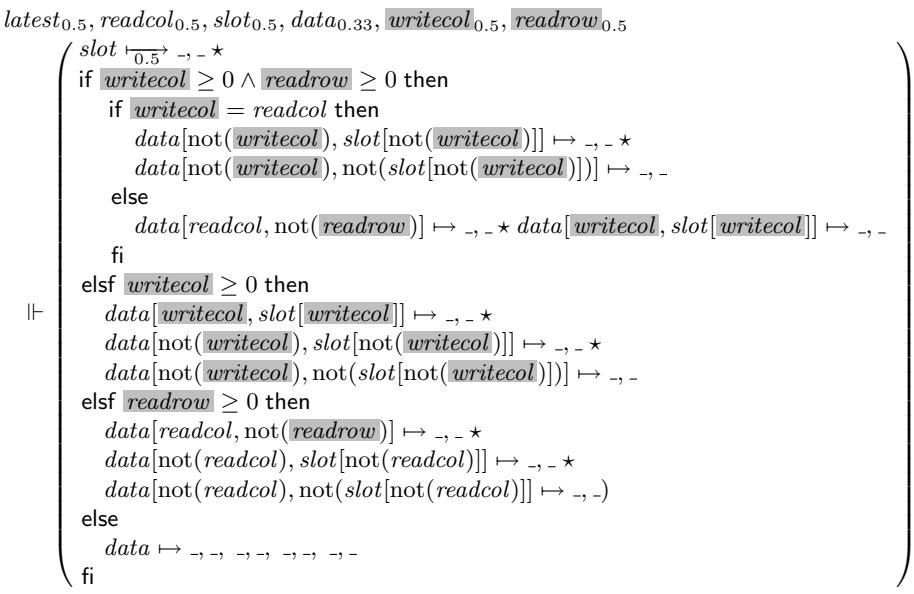

Fig. 1.18. The shared data invariant

chosen to do it as an enumeration of states because that illuminates the way that the algorithm works. The resource owns matching permissions for the variables that the threads part own, and shares the *slot* array with the writer. When the writer is communicating (lines 2 and 3 of the *write* procedure) *writecol* is 0 or 1, and similarly *readrow* is 0 or 1 when the reader is communicating (lines 3 and 4 of the *read* procedure). Either both threads are communicating – in which case the shared invariant owns the slots the threads don't own – or one or the other is communicating – in which case it owns the slots the communicating thread doesn't use – or neither is communicating – in which case it owns the whole array. In the case that both threads are communicating there are two possibilities: either they are working in separate rows of the same column (this can happen if the reader is overtaken by the writer), or they are working in separate columns. To emphasise that the slots of the *data* array can't be written atomically, I've made them two heap cells each.

Proof of separation for the writer is summarised in figure 1.19. In its first step it extracts ownership of the slot it is going to assign, and this step is shown in detail in figure 1.20. Entering with *writecol* $= -1$, the *data* array splits into those parts which the reader might own (the *readcol* column), and those it does not (the not(*readcol*) column). The writer picks one of the ones that the reader doesn't at the moment seem to require, splits the invariant off, and exits with the slot it wants. The *writecol* := *col* action can be crammed

into the same atomic action since it only affects an auxiliary variable, and therefore doesn't really happen.

$$
\begin{array}{l}
\left\{ \begin{array}{l} latest_{0.5}, slot_{0.5}, data_{0.33}, \boxed{writecol}_{0.5}, col, row \\ \quad \Vdash \boxed{writecol} = -1 \wedge (slot[0] \overset{0.5}{\longmapsto} \_ \star slot[1] \overset{0.5}{\longmapsto} \_) \end{array} \right\} \\
\mathsf{atomic}\{col := \mathrm{not}(readcol); \boxed{writecol} := col\}; \\
\left\{ \begin{array}{l} latest_{0.5}, slot_{0.5}, data_{0.33}, \boxed{writecol}_{0.5}, col, row \\ \quad \Vdash \boxed{writecol} = col \wedge \left(\begin{array}{l} slot[0] \overset{0.5}{\longmapsto} \_ \star slot[1] \overset{0.5}{\longmapsto} \_ \star \\ data[\boxed{writecol}, \mathrm{not}(slot[\boxed{writecol}])] \mapsto \_, \_ \end{array} \right) \end{array} \right\} \\
row := \mathrm{not}(slot[col]); \\
\left\{ \begin{array}{l} latest_{0.5}, slot_{0.5}, data_{0.33}, \boxed{writecol}_{0.5}, col, row \\ \quad (\boxed{writecol} = col \wedge row = \mathrm{not}(slot[col])) \star \\ \quad \Vdash \left(\begin{array}{l} slot[0] \overset{0.5}{\longmapsto} \_ \star slot[1] \overset{0.5}{\longmapsto} \_ \star \\ data[\boxed{writecol}, \mathrm{not}(slot[\boxed{writecol}])] \mapsto \_, \_ \end{array} \right) \end{array} \right\} \\
data[col, row] := item; \\
\left\{ \begin{array}{l} latest_{0.5}, slot_{0.5}, data_{0.33}, \boxed{writecol}_{0.5}, col, row \\ \quad \Vdash \boxed{writecol} = col \wedge \left(\begin{array}{l} slot[0] \overset{0.5}{\longmapsto} \_ \star slot[1] \overset{0.5}{\longmapsto} \_ \star \\ data[\boxed{writecol}, \mathrm{not}(slot[\boxed{writecol}])] \mapsto item \end{array} \right) \end{array} \right\} \\
\mathsf{atomic}\{slot[col] := row; \boxed{writecol} := -1\}; \\
\left\{ \begin{array}{l} latest_{0.5}, slot_{0.5}, data_{0.33}, \boxed{writecol}_{0.5}, col, row \\ \quad \Vdash \boxed{writecol} = -1 \wedge (slot[0] \overset{0.5}{\longmapsto} \_ \star slot[1] \overset{0.5}{\longmapsto} \_) \end{array} \right\} \\
\mathsf{atomic}\{latest := col\} \\
\left\{ \begin{array}{l} latest_{0.5}, slot_{0.5}, data_{0.33}, \boxed{writecol}_{0.5}, col, row \\ \quad \Vdash \boxed{writecol} = -1 \wedge (slot[0] \overset{0.5}{\longmapsto} \_ \star slot[1] \overset{0.5}{\longmapsto} \_) \end{array} \right\}
\end{array}
$$

Fig. 1.19. The writer proof summarised

Nothing else the writer does in figure 1.19 is so complicated. The second step is not even atomic (the writer already has half permission for the contents of the *slot* array, and *row* and *col* are local); the third step writes into the slot of the *data* array owned since the first step; the fourth step alters the *slot* array and returns the borrowed *data* slot; the final step, as we shall see, is irrelevant to this proof.

Proving separation in the reader is a little more intricate, but hardly more complicated. Figure 1.21 shows the summary. The first two steps pick a value for *readcol*, and the third step claims a slot. Figure 1.22 shows that step in detail.

There is an uncertainty at the beginning of the reader's third step, because nothing provides that *readcol* and $\boxed{writecol}$ are different at that instant. But whichever is true, no problem: if the variables are the same then the reader claims the slot in $\boxed{writecol}$ that the writer isn't using, and if they are different then there's no possibility of a clash. In both cases, separation is preserved.

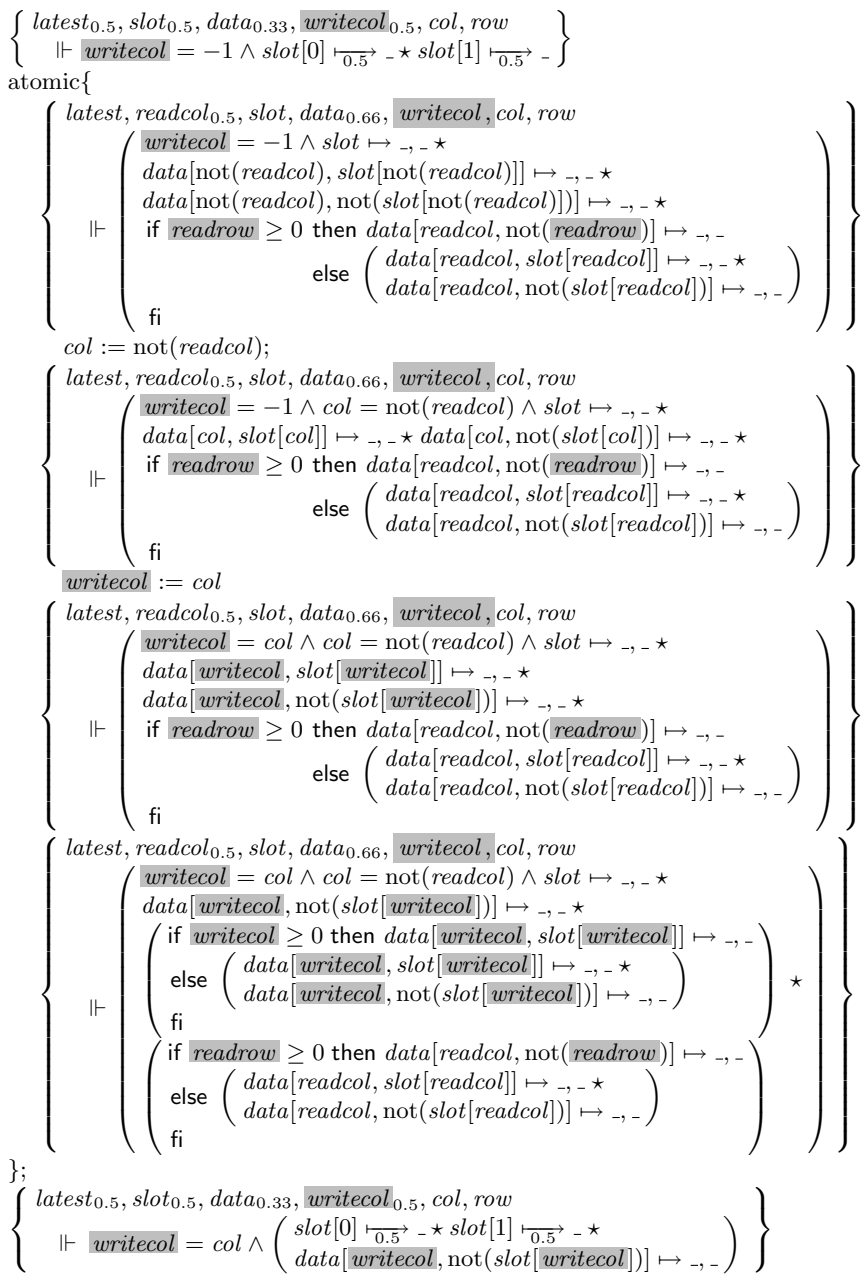

Fig. 1.20. The writer's first step in detail

$\{\ readcol_{0.5}, \boxed{readrow}_{0.5}, data_{0.33}, col, row \Vdash \boxed{readrow} = -1 \wedge \mathbf{emp}\ \}$
$\quad \mathsf{atomic}\{col := latest\};$
$\{\ readcol_{0.5}, \boxed{readrow}_{0.5}, data_{0.33}, col, row \Vdash \boxed{readrow} = -1 \wedge \mathbf{emp}\ \}$
$\quad \mathsf{atomic}\{readcol := col\};$
$\{\ readcol_{0.5}, \boxed{readrow}_{0.5}, data_{0.33}, col, row \Vdash \boxed{readrow} = -1 \wedge readcol = col \wedge \mathbf{emp}\ \}$
$\quad \mathsf{atomic}\{row := slot[col]; \boxed{readrow} := row\};$
$\left\{ \begin{array}{l} readcol_{0.5}, \boxed{readrow}_{0.5}, data_{0.33}, col, row \\ \quad \Vdash \boxed{readrow} = row \wedge readcol = col \wedge data[col, row] \mapsto \_, \_ \end{array} \right\}$
$\quad read := data[col, row];$
$\left\{ \begin{array}{l} readcol_{0.5}, \boxed{readrow}_{0.5}, data_{0.33}, col, row \\ \quad \Vdash \boxed{readrow} = row \wedge readcol = col \wedge \exists i \cdot (data[col, row] \mapsto i \wedge read = i) \end{array} \right\}$
$\quad \mathsf{atomic}\{\boxed{readrow} := -1\}$
$\{\ readcol_{0.5}, \boxed{readrow}_{0.5}, data_{0.33}, col, row \Vdash \boxed{readrow} = -1 \wedge \mathbf{emp}\ \}$

Fig. 1.21. The reader proof summarised

1.8.1 What has been proved?

The four-slot algorithm does not really need a formal proof, because it can be completely model-checked – Simpson himself did so in [32]. But not all difficult concurrent algorithms have such a small state space, and if we are ever to have a 'verifying compiler' for concurrent programs, we shall have to deal with problems like this one. The techniques employed here – shared-data invariant, atomic operations containing a single serialised shared-store operation – have already been employed to deal with a shared-stack problem too large to model-check [28].

This proof is also rather small compared with the hundred-page refinement development in [10], which uses rely-guarantee techniques. One of the reasons for that is that the separation logic proof need only deal with interference inside the *read* and *write* procedures, using logical techniques that I don't have space to go into here but which are dealt with in [28]. The aim of separation logic developments is to increase modularity in proofs, and in this case we've managed that rather well.

All the ingenuity in this proof is in the choice of the auxiliary variables $\boxed{readrow}$ and $\boxed{writecol}$ and in the shared-data invariant. After that it really is a matter of tedious calculation, not all of which I have detailed. I think that is a strength: if 'resourcing' is to be as successful as typing, we must make algorithmic the steps which connect user-supplied assertions to the program, so that a compiler or the like can carry them out. I don't pretend that we have reached that stage, nor do I pretend that the hints that a user would have to give to a formal programming tool can yet be simple enough to be popularly acceptable.

But there's a more important sense in which this proof might be considered unsatisfactory: it doesn't show that the reader and writer actually communicate. The *slot* array and the *latest* variable are used by the writer to point the reader to the most recent value written. Indeed it's possible to see

Fig. 1.22. The reader's third step

that the reader will not read a value written earlier than the last one written before it begins to execute a *read* operation. All we have been able to show so far is that reader and writer do not collide, not that effective communication occurs. It ought to be possible, in this particular example which only uses true in the guards of its CCR commands, to do a little better.

1.9 Conclusion

Separation logic is a step forward in the search for modular proofs of safety assertions. As a partial-correctness logic, it cannot deal with questions of liveness and termination. But since it can deal easily with pointers, safety properties of a wider range of programs than before are now practically specifiable and verifiable. The examples in this paper show that the frontier is now much farther forward than was envisaged when work on the logic began. Active work is going on to incorporate this advance into practical programming tools, and to extend the logic's range even further by dealing with some properties that at present need global proof, by incorporating some of the insights of rely-guarantee and/or temporal logic. Watch this space!

Acknowledgements

Some of the proofs in this paper are my own, and I've contributed some of the logic. But almost all the logic is due to O'Hearn and Reynolds, and the early examples are theirs too. In addition I owe an enormous debt to all the other members – Josh Berdine, Cristiano Calcagno, Dino DiStefano, Ivana Mijhailovic, Matthew Parkinson – of the group we call the East London Massive, in whose meetings these ideas were kicked around and collectively refined. The mistakes, I need hardly add, are my own.

References

1. Howard Barringer, Ruurd Kuiper, and Amir Pnueli. Now you may compose temporal logic specifications. In *STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 51–63, New York, NY, USA, 1984. ACM Press.
2. J. Berdine, C. Calcagno, and P.W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Proceedings of FMCO'05*, volume 4111 of *LNCS*, 2006.
3. Richard Bornat. Proving pointer programs in Hoare logic. In R. C. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction, 5th International Conference*, LNCS, pages 102–126. Springer, 2000.
4. Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 259–270, New York, NY, USA, January 2005. ACM Press.
5. Richard Bornat, Cristiano Calcagno, and Hongseok Yang. Variables as resource in separation logic. In *Proceedings of MFPS XXI*. Elsevier ENTCS, May 2005.
6. John 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.

7. P. Brinch Hansen, editor. *The Origin of Concurrent Programming*. Springer-Verlag, 2002.
8. S. D. Brookes. A semantics for concurrent separation logic. In *CONCUR'04: 15th International Conference on Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 16–34, London, August 2004. Springer. Extended version to appear in *Theoretical Computer Science*.
9. R.M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.
10. Jonathan Burton. *The Theory and Practice of Refinement-After-Hiding*. Ph.D. thesis, University of Newcastle upon Tyne, Newcastle upon Tyne, UK, 2005. Available as technical report CS-TR-904.
11. P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14(10):667–668, 1971.
12. E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, 1968. Reprinted in [7].
13. David Gries. An exercise in proving parallel programs correct. *Commun. ACM*, 20(12):921–930, 1977.
14. Tim Harris, Simon Marlowe, Simon Peyton-Jones, and Maurice P. Herlihy. Composable memory transactions. In *Principles and Practice of Parallel Programming (PPOPP)*, 2005. To appear.
15. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
16. C. A. R. Hoare. Towards a theory of parallel programming. In Hoare and Perrott, editors, *Operating System Techniques*, pages 61–71. Academic Press, 1972. Reprinted in [7].
17. C. A. R. Hoare. Hints on programming language design. Technical Report CS-TR-73-403, Stanford University, Computer Science Dept., 1973. Keynote address to ACM SIGPLAN conference. Reprinted in [18], pp 193–216.
18. C. B. Jones, editor. *Essays in Computing Science*. Prentice/Hall International, 1989.
19. Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.
20. Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275, New York, NY, USA, 1996. ACM Press.
21. P. W. O'Hearn. Resources, concurrency and local reasoning. In *CONCUR'04: 15th International Conference on Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 49–67, London, August 2004. Springer. Extended version to appear in *Theoretical Computer Science*.
22. Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *CSL 2001*, pages 1–19. Springer-Verlag, 2001. LNCS 2142.
23. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 19:319–340, 1976.
24. S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.
25. Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3):455–495, 1982.

26. Susan Speer Owicki. *Axiomatic proof techniques for parallel programs.* PhD thesis, Cornell, 1975. Technical report TR75-251.
27. Matthew Parkinson, Richard Bornat, and Cristiano Calcagno. Variables as resource in Hoare logics. In *Proceedings of LICS*, pages 137–146. IEEE, 2006.
28. Matthew Parkinson, Richard Bornat, and Peter O'Hearn. A modular verification of a non-blocking algorithm. To be presented at POPL, 2007.
29. Amir Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13(1):45–60, 1981.
30. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.
31. H. Simpson. Four-slot fully asynchronous communication mechanism. *IEE Proceedings*, 137(1):17–30, January 1990.
32. H.R. Simpson. Role model analysis of an asynchronous communication mechanism. *IEE Proceedings of Computer Digital Technology*, 144:232–240, July 1997.