Permission Accounting in Separation Logic

Richard Bornat
School of Computing Science
Middlesex University
LONDON N17 8HR, UK
R.Bornat@mdx.ac.uk

Cristiano Calcagno
Department of Computing
Imperial College, University of London
LONDON SW7 2AZ, UK
ccris@doc.ic.ac.uk

Peter O'Hearn
Department of Computer Science
Queen Mary, University of London
LONDON E1 4NS, UK
ohearn@dcs.qmul.ac.uk

Matthew Parkinson
Computer Laboratory
University of Cambridge
CAMBRIDGE CB3 0FD, UK
mjp41@cl.cam.ac.uk

ABSTRACT

A lightweight logical approach to race-free sharing of heap storage between concurrent threads is described, based on the notion of permission to access. Transfer of permission between threads, subdivision and combination of permission is discussed. The roots of the approach are in Boyland's [3] demonstration of the utility of fractional permissions in specifying non-interference between concurrent threads. We add the notion of counting permission, which mirrors the programming technique called permission counting. Both fractional and counting permissions permit passivity, the specification that a program can be permitted to access a heap cell yet prevented from altering it. Models of both mechanisms are described. The use of two different mechanisms is defended. Some interesting problems are acknowledged and some intriguing possibilities for future development, including the notion of resourcing as a step beyond typing, are paraded.

Categories and Subject Descriptors

D.2.4 [**Software/Program verification**]: Correctness proofs, Formal methods, Validation; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Logics of programs

General Terms

Languages, theory, verification

Keywords

separation, logic, concurrency, permissions

POPL'05, January 12–14, 2005, Long Beach, California, USA.

1. BACKGROUND

Separation logic has its roots in the observation by Burstall in 1972 [7] that separate program texts which work on separate sections of the store can be reasoned about independently. Reynolds, O'Hearn and Yang [20, 19, 14] and others developed the logic to describe mutation of the heap based on a notion of separate locations. In logical terms it's a particular model of BI [17, 18], but in programming terms it's a really cool hack with Hoare logic, making earlier attempts to prove pointer-mutating programs (see [1] for references) look ridiculously complicated and ad-hoc. Small but intricate graph-manipulating programs can be specified and proved with relatively little fuss [2].

The ambitions of the separation logic community extend far beyond the description of graph-mutating programs. The aim all along was to understand, specify and prove properties of fundamental programs, for example operating systems, written in low-level languages and running without support on naked hardware. That requires an attack, first of all, on the problems of concurrency (and, of course, there will be many more problems to come: it's too early to storm the walls yet).

O'Hearn has shown [15] that separation logic can describe *ownership transfer*, where concurrent program threads move ownership of heap cells into and out of shared resources which can be semaphores, conditional critical regions or monitors. Breakthrough though it is, this isn't enough by itself. Separation logic deals with separation: exclusive ownership by one side or another of each heap cell. In practice heap cells, like variables in Dijkstra's original descriptions [9], can safely be shared between concurrent threads provided they all promise only to read, never to write. This has echoes of the notion of *passivity* which appears to be necessary in a separation-logic treatment of sequential programs: we have to be able to say that a program has read access to a heap cell but doesn't have the right to change it.

The invention of ownership transfer began a change in the way that separation logic assertions are read. The basic assertion $N \mapsto E$, pronounced N '*points to*' E, is a predicate of a heap asserting that it consists of a single cell with integer address N and integer contents E. It can equally be read as a *permission* asserting the right to read, write or dispose

READERS

P(m);
  $count := count + 1$;
  if $count = 1$ then P($write$);
V(m);

  ... reading happens here ...;

P(m);
  $count := count - 1$;
  if $count = 0$ then V($write$);
V(m)

WRITER

P($write$);

  ... writing happens here ...

V($write$)

Figure 1: Readers and writers (from [8], with shortened names)

that particular cell. It's the permission rather than the cell that is transferred between threads in an ownership transfer. The basic assertion **emp** is a predicate that the heap has no cells; as a permission it means 'without permission to access any cell'. Nothing changes semantically, but for many programmers the permission reading of separation logic is easier to grasp than the predicate version.

Viewing ownership as total read/write/dispose permission makes it possible to begin to see how heap cells might be shared. A total permission can be split into as many read-only permissions as needed and shared around as necessary. Giving a read-only permission surely ought to guarantee passivity (as it does, as we shall see). The only problem is in gathering them back in. How many read permissions make a total permission? Clearly, you need them all – but how many is all? (Answer: it depends how many you made when you started.) What if some of the read permissions we handed out were split by their recipients? (Answer: if they do that, then either you have to know about it or they have to put them back together before handing them in.) How can you keep account?

You surely need to keep account. Suppose there is a program which has total ownership of cell N, and which temporarily splits into two threads which concurrently read N but don't write to it or transfer it elsewhere. After the threads recombine, you can harvest the permissions you gave them. Now the program must have total access once more: if not, where has the permission gone? To lose permissions is to leak resource; accurate accounting is essential.

It's the need for *permission accounting* which constrains the treatment of permissions in separation logic. You have to measure them out, and you have to measure them all back in. The design choices are all about simplicity and convenience of different kinds of measurement.

There are several oddities about permission accounting as we currently understand it. One is immediately obvious: there are at least two alternative accounting mechanisms. Another is that it is proving difficult to extend the treatment of heap cells to recursively-defined data structures. Finally, exploration of permissions has clearly exposed a deeper problem in the treatment of variables as resource.

Nevertheless we have come a considerable distance in the eight months since we first heard Boyland's laconic hint "$\frac{1}{2}+\frac{1}{2}=1$". I hope that we are blazing a trail towards the goal of *resourcing*, a step beyond program typing in which the quantity as well as the kind of resource that is supplied to a program or command must be described and verified.

My intention is to discover the principles of resourcing by exploring resource properties of programs that can be specified and verified. Proof-theoretic exploration is dangerous: you can go far out on very thin ice, and if it's unsound you get very cold and wet. On the other hand, sound but unexplored logics don't necessarily make useful reasoning tools. Surely there's room in our subject for those who experiment proof-theoretically as well as those who deal in certainty. I'm interested in proof; I believe that it's worth trying to find logics that look so obviously useful that they deserve a soundness proof.

Soundness matters, though, even to me. In this work I haven't gone very far from land: soundness, I hope and expect, will be a matter of building small bridges to previous work, dotting is and crossing ts. In the meantime I'm searching for 'nice proofs': proofs that can be understood, concise proofs, proofs you can read. I'm hoping to prove programs that people already think they understand but which don't yet have satisfying proofs. I'm aiming, in the end, for proofs that a compiler could follow and, if I'm allowed to dream, proofs that a compiler could guess.

2. A PROGRAM IN NEED OF RESOURCING

The readers and writers algorithm of Courtois et al. [8], shown in figure 1, allows multiple readers concurrent access to a shared variable, but restricts writers to exclusive access. It would be possible to read the algorithm as a description of two parallel processes, but it is far more concurrent than that. There are various components which can be executed concurrently, with varying degrees of mutual exclusion:

- the four uses of the binary mutex m;
- the reader prologue $count := count + 1$...;
- the reader action section;
- the reader epilogue $count := count - 1$...;
- the two uses of the binary mutex *write*;
- the writer action section.

A resourcing of this program must explain how that concurrency is controlled. Further, it must explain how use of variable *count* is restricted to reader prologue and epilogue.

All of these resourcing questions are addressed below. I have answers to all but the dynamic restriction of the scope of *count* applied by use of the mutex m (and in that case we can provide an explanation of an alternative version of the program).

3. BASICS

I give a brief description of separation logic. A more careful treatment, particularly of critical regions and resource bundles, is in [15] and [6], where there is also a discussion of the relation to earlier work on concurrency.

In the original model of separation logic a heap is a partial map from addresses to values. The simplest heaps are the empty heap **emp** and the singleton heap with address E and content E', written as $E \mapsto E'$. We write $E \mapsto \_$ as a shorthand for $\exists v \cdot E \mapsto v$. Two heaps can be combined, using multiplicative conjunction ($\star$) iff their (address) domains are disjoint.

The *frame property* (section 11.3) of separation logic requires that if a program doesn't go wrong in a particular stack/heap configuration s, h, then it will not go wrong in a larger configuration $s, (h \star h')$; its effect will still be to change h, leaving the added heap h' completely unaffected. As a result, separation is policed and exploited by the frame rule

$$\frac{\{Q\}C\{R\}}{\{P \star Q\}C\{P \star R\}} \ (\mathit{modifies}\ C\ \cap \mathit{vars}\ P = \emptyset) \tag{1}$$

– if C can't modify the variables of P, and if the heap it manipulates is disjoint from that of P, then we can reason about C and its effects separately from P. The side-condition is required because separation logic deals only with separation of heap cells, not (stack) variables.

The language that separation logic treats includes new and dispose, abstractions of similar Pascal or C library primitives. new is a heap creator – it makes a singleton heap – and dispose a matching heap destroyer. In the simplest version of the language we don't care what value is in the heap that new creates or dispose destroys. Writing E for a 'pure' expression – one which doesn't involve heap access – we have:

$$\begin{array}{c} \{\mathbf{emp}\}\ x := \mathrm{new}()\ \{x \mapsto \_\} \\ \{E \mapsto \_\}\ \ \mathrm{dispose}\, E\ \ \{\mathbf{emp}\} \end{array} \tag{2}$$

There is absolutely no way to make a heap other than with new, or to destroy one other than with dispose.[1] To make the frame rule work, we know that new has to be magic, in the sense of program refinement: it must always return an address which is disjoint from the domain of any heap in use at the time. Of course this is easy to implement using a list of locations not yet handed out to the program, so it's really only stage magic.

Addresses received from new are integers, but all a program can do with with the heap via an address is to access or modify the addressed value. Writing $[\_]$ for heap access, we recognise three forms of assignment:

$$\begin{array}{lll} \{R^x_E\} & x{:=}E & \{R\} \\ \{E' \mapsto \_\} & [E']{:=}E & \{E' \mapsto E\} \\ \{E' \mapsto E\} & x{:=}[E'] & \{E' \mapsto E \wedge x = E\} \end{array} \tag{3}$$

The use of conventional Hoare logic in the first assignment axiom gives rise to the proviso in the frame rule. I'd love to get rid of that condition, but as you shall see (section 13.2) that isn't easy.

The other two axioms are presented as forward reasoning steps (backward versions, using BI's 'magic wand' operator $\mathrel{-\!\!\star}$ are possible, but I don't give them here). The last rule, as a forward step, requires a side-condition that x does not occur free in E or E'.

3.1 Concurrency

Since Dijkstra [9] the description of safe concurrency has been based on a separation of variables into distinct groups:

- read/write variables unique to each thread;
- read-only variables shared between threads;
- read/write shared variables accessible only in mutually-exclusive critical sections of program code.

Other treatments – conditional critical regions, monitors – have provided alternative linguistic expression of the same fundamental notion. Our approach is in the same tradition, but treating heap locations rather than variables.

The concurrency rule

$$\frac{\{Q_1\}\, C_1\, \{R_1\} \cdots \{Q_n\}\, C_n\, \{R_n\}}{\{Q_1 \star \cdots \star Q_n\}(C_1 \parallel \cdots \parallel C_n)\{R_1 \star \cdots \star R_n\}} \tag{4}$$

describes how concurrent threads with $\star$-separable heap resources can be treated separately. The side-condition, which guarantees non-interference of variables, is that no variable free in Q_i or R_i is modified in C_j when $j \neq i$. It is remarkable that such a simple rule is possible in our logic. It holds out the promise that we might specify and verify zero-execution-cost barriers between threads.

As a concurrent program executes, heap resources must remain separated but the separation need not be fixed: ownership can be transferred between threads. Following [13] our treatment is based on conditional critical regions. A conditional critical region (CCR) [11] is a command

$$\text{with } b \text{ when } G \text{ do } C \text{ od}$$

where b is a *resource-bundle* name.[2] A bundle, like a thread, may possess its own private read/write variables; the boolean G and the command C in a CCR command may refer to these variables. Execution of a CCR is in mutual exclusion with all other CCRs for the same bundle. It proceeds, rather like a monitor procedure execution, as follows:

[1] The axioms in (2) allocate and dispose a single cell, for simplicity. Axioms which deal with any particular record size are possible. A treatment which deals with computable record sizes, unrecorded by the program but tracked by the specification – that is, a treatment of C's `malloc` and `free` – is one aim of the work reported here.

[2] Hoare called resource bundles simply resources, but I want that word to apply to the items – heap locations and variables at least, perhaps time and stack space and whatever else we can manage – and the permissions that are owned by, shared between or transferred between the threads of a concurrent program. A resource bundle contains a bundle of resources described by an invariant formula – hence the nomenclature.

Bundle b : Vars *full*, *buf*; *buf* := false;
Invariant if *full* then *buf* $\mapsto$ _ else **emp** fi

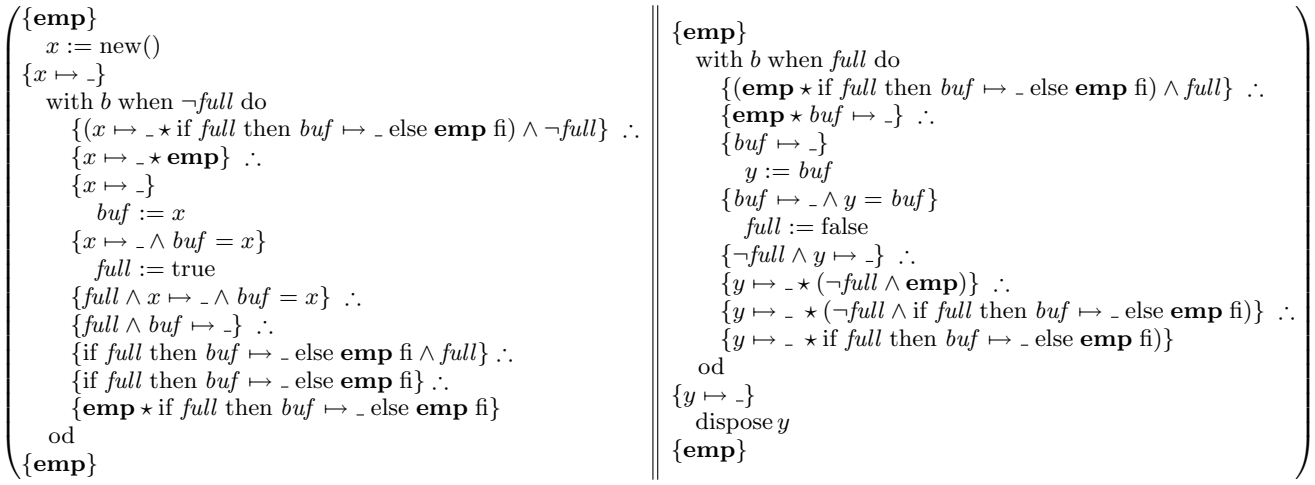

Figure 2: Ownership transfer between concurrent threads using CCR commands

1. acquire bundle b;

2. evaluate the boolean guard G;

3. if G is true, execute the command C and release b;

4. if G is false, release b and try again.

Following [15], a mutex semaphore m is a bundle whose CCRs are either

P**:** with m when $m \neq 0$ do $m := 0$ od, or

V**:** with m when true do $m := 1$ od.

A counting semaphore c is similar, but with commands $c := c - 1$ and $c := c + 1$. This is much more than a convenient equivalence: it inverts the normal treatment of semaphores, converting them from (negative) locks keeping you out to (positive) stores of resource that you can use.

In our treatment each bundle must have an invariant formula describing its resources in terms of its private variables, ($\star$)-separated from each other and from the resource of any thread in a version of the concurrency rule. If the resource of bundle b is described by invariant I_b, the conditional critical region rule is

$$\frac{\{(Q \star I_b) \wedge G\} C \{R \star I_b\}}{\{Q\}\text{with } b \text{ when } G \text{ do } C \text{ od}\{R\}} \quad (5)$$

The non-interference side condition is that processes cannot refer to variables of the bundle outside a CCR command.

This approach has been proved sound by Brookes in [6]: the result is that given invariant formulae for each bundle and ($\star$)-separation of bundle resources from each other and from thread resources, we can reason sequentially about each thread and the CCRs it employs. Brookes's semantics allows threads to share read-only variables *and locations*: I shall return to that point.

4. OWNERSHIP

O'Hearn, in [13], gave an alternative reading of $E \mapsto E'$ as expressing *ownership* of a heap location. He used the conditional critical region rule to transfer ownership between threads. Figure 2 shows his example with assertions of ownership.

For separation logic users, O'Hearn's alternative reading of the $\mapsto$ relation was a breakthrough, a liberation. But we needed help to take the next step towards permission accounting and shareable resource.

5. FRACTIONAL PERMISSIONS

In order to reason about non-interference of concurrent threads, Boyland [3] associates a rational z with each stack variable and heap location. Like Brookes, he distinguishes total control (dispose, read and write permission) from shared access (read only: no thread can write or dispose). $z = 1$ gives exclusive ownership and total control; $0 < z < 1$ allows shared access. This enabled him to describe the allocation of memory access rights to threads. He proved the determinacy of disjoint concurrency with shared read access. He pointed out, correctly, that separation logic couldn't match this: the concurrency rule only deals with exclusive access. He suggested, however, that separation logic might be modified to include the equivalent of $P \vDash \epsilon P \star (1 - \epsilon)P$ and thus be able to deal with shared heaps.

Boyland's suggestion turns out to deal very nicely with fork-join programs where permission splitting and combining is part of the program structure. Fractional permission accounting, like program typing, is a compile-time discipline. The program does nothing to support the accounting: everything happens in the specifications and the proof. The magnitude of non-integral fractions don't seem to matter: a program can do exactly as well with 0.1 as with 0.9 (but see section 13.1).

Those who, like me, would hesitate before mixing rational arithmetic and logic need not be scared of its use in permission accounting. The complexity of the arithmetic deductions is only that required by a particular specification:

READERS

```
with read when true do
  if count = 0 then P(write) else skip fi;
  count +:= 1
od;

  ... reading happens here ...;

with read when count > 0 do
  count −:= 1;
  if count = 0 then V(write) else skip fi
od
```

WRITER

```
P(write);

  ... writing happens here ...

V(write)
```

Figure 3: Readers and writers: CCR version

typically, no more than observing that $z+z' = z'+z$ or that two halves make a one. Fractions seem to be more convenient to use than history-based mechanisms like sets of binary trees.

6. COUNTING PERMISSIONS

Not every program is suitable for fractional permission accounting. Programs which keep a semaphore-protected count of the number of permissions handed out need an alternative treatment. A famous example is the readers-and-writers problem; another example is pipeline processing where permission to access a buffer is passed from an originator thread to a number of assistants, any of which may pass it on further, and eventually dispose the permission without the originator's involvement.

To deal with permission counting we have counting permissions. A central "permissions authority" holds a source permission, annotated with the number of read permissions that have been split off from it; the split-off read permissions can't be split further; only a source with no split-off children gives total read/write/dispose ownership. An analogy is Neolithic flint knapping: arrowheads were split from a stone that remained capable of providing more of the same. In principle the arrowheads could be re-attached to re-create the original stone.

Permission counting is not reference counting: it has nothing to do with reachability. The number of permissions can be many fewer than the number of reachable pointers. (Separation logic embraces the dangling pointer, yet again!)

7. FRACTIONAL PERMISSIONS IN DETAIL

We modify the model of separation logic (see section 10 for more detail). A heap is now a partial map from addresses to values with permissions. We use Boyland's [3] numerical scheme: a permission is z, where $0 < z \leq 1$; $z = 1$ allows dispose, write and read; any other value is read access only. We annotate the $\mapsto$ relation to show the level of permission it carries:

$$x \underset{z}{\longmapsto} E \implies 0 < z \leq 1 \tag{6}$$

Heaps can be combined with ($\star$) iff, where their addresses coincide, they agree on values and their permissions combine arithmetically. Reading in the other direction, an existing permission can always be split in two.

$$x \underset{z}{\longmapsto} E \star x \underset{z'}{\longmapsto} E \iff x \underset{z+z'}{\longmapsto} E \wedge z > 0 \wedge z' > 0 \tag{7}$$

We require positive z and z' to avoid silly nonsense like $2 \star -1 \iff 1$: otherwise, the fractions we choose are arbitrary, an aide-memoire for future recombination. Reasoning about their magnitudes would seem to be like reasoning about the identity of the names we use for the parameters of a theorem.

new and dispose deal only in full permissions:

$$\begin{array}{c} \{\mathbf{emp}\}\ x := \text{new}()\ \{x \underset{1}{\longmapsto} \_\} \\ \{E \underset{1}{\longmapsto} \_\}\ \ \text{dispose}\, E\ \ \{\mathbf{emp}\} \end{array} \tag{8}$$

Assignment needs full access for writing, any access at all for reading:

$$\begin{array}{ccc} \{R^x_E\} & x{:=}E & \{R\} \\ \{x \underset{1}{\longmapsto} \_\} & [x]{:=}E & \{x \underset{1}{\longmapsto} E\} \\ \{E' \underset{z}{\longmapsto} E\} & x{:=}[E'] & \{E' \underset{z}{\longmapsto} E \wedge x = E'\} \end{array} \tag{9}$$

(the side-condition on the last rule is once again x not free in E or E'). It's then completely straightforward to check the correctness of the program in figure 4, in which parallel threads require simultaneous read access to location $[x]$.

Most fractional problems are as simple as this. It really is that easy. Section 9 discusses a larger example.

$$\begin{array}{l} \{\mathbf{emp}\} \\ x := \text{new}(); \\ \{x \underset{1}{\longmapsto} \_\} \\ [x] := 7; \\ \{x \underset{1}{\longmapsto} 7\} \therefore \{x \underset{0.5}{\longmapsto} 7 \star x \underset{0.5}{\longmapsto} 7\} \\ \left(\begin{array}{l|l} \{x \underset{0.5}{\longmapsto} 7\} & \{x \underset{0.5}{\longmapsto} 7\} \\ y := [x] - 1 & z := [x] + 1 \\ \{x \underset{0.5}{\longmapsto} 7 \wedge y = 6\} & \{x \underset{0.5}{\longmapsto} 7 \wedge z = 8\} \end{array}\right); \\ \{x \underset{0.5}{\longmapsto} 7 \star x \underset{0.5}{\longmapsto} 7 \wedge y = 6 \wedge z = 8\} \therefore \\ \{x \underset{1}{\longmapsto} 7 \wedge y = 6 \wedge z = 8\} \\ \text{dispose}\, x; \\ \{\mathbf{emp} \wedge y = 6 \wedge z = 8\} \end{array}$$

Figure 4: Fractions are easy

7.1 Passivity

Passivity is a property of a command which has access to a heap cell but leaves it unchanged. Any fractional permission less than 1 prescribes passivity, by the following argument.

$$
\begin{array}{l}
\{\mathbf{emp}\} \\
\quad \mathrm{P}(\mathit{write}) : \left(\begin{array}{l}
\{(\mathbf{emp} \star \text{if } \mathit{write} = 0 \text{ then } \mathbf{emp} \text{ else } y \overset{0}{\mapsto} \_ \text{ fi}) \wedge \mathit{write} = 1\} \ \therefore \\
\{(\mathbf{emp} \star y \overset{0}{\mapsto} \_) \wedge \mathit{write} = 1\} \\
\quad \mathit{write} := 0 \\
\{y \overset{0}{\mapsto} \_ \star (\mathbf{emp} \wedge \mathit{write} = 0)\} \ \therefore \\
\{y \overset{0}{\mapsto} \_ \star (\text{if } \mathit{write} = 0 \text{ then } \mathbf{emp} \text{ else } y \overset{0}{\mapsto} \_ \text{ fi} \wedge \mathit{write} = 0)\}
\end{array}\right) \\
\{y \overset{0}{\mapsto} \_\} \\
\\
\{y \overset{0}{\mapsto} \_\} \\
\quad \mathrm{V}(\mathit{write}) : \left(\begin{array}{l}
\{y \overset{0}{\mapsto} \_ \star \text{if } \mathit{write} = 0 \text{ then } \mathbf{emp} \text{ else } y \overset{0}{\mapsto} \_ \text{ fi}\} \ \therefore \\
\{y \overset{0}{\mapsto} \_ \star (\mathbf{emp} \wedge \mathit{write} = 0)\} \\
\quad \mathit{write} := 1 \\
\{\mathbf{emp} \star (y \overset{0}{\mapsto} \_ \wedge \mathit{write} = 1)\} \ \therefore \\
\{\mathbf{emp} \star (\text{if } \mathit{write} = 0 \text{ then } \mathbf{emp} \text{ else } y \overset{0}{\mapsto} \_ \text{ fi} \wedge \mathit{write} = 1)\}
\end{array}\right) \\
\{\mathbf{emp}\}
\end{array}
$$

Figure 5: Proof of pre- and post-condition of $\mathrm{P}(\mathit{write})$ **and** $\mathrm{V}(\mathit{write})$

Commands in our language obey the frame property. In the sequential sub-language they also display *termination monotonicity* (section 11.3): if a command terminates in a particular heap, then it terminates in any larger heap. Suppose that C is a command which is given fractional permission to access cell 10, and which manages to change that cell somehow – say to increase its value. That is, it obeys

$$\{10 \underset{0.5}{\longmapsto} N\} C \{10 \underset{0.5}{\longmapsto} N+1\}$$

and it terminates. It must therefore terminate in any larger heap. Using the frame rule you can show

$$\{10 \underset{0.5}{\longmapsto} N \star 10 \underset{0.5}{\longmapsto} N\} C \{10 \underset{0.5}{\longmapsto} N \star 10 \underset{0.5}{\longmapsto} N+1\}$$

– but the postcondition is false, so C can't terminate in the larger heap, so it can't be a command of the sequential sub-language since it doesn't exhibit termination monotonicity.

That proof, and its conclusion, must be treated with care in the non-sequential case, because a command can apply to a bundle for additional resource. Suppose

$$\begin{array}{l} I_b \equiv 10 \underset{0.5}{\longmapsto} \_ \\ C \equiv \text{with } b \text{ when true do } [10] := 3 \text{ od} \end{array}$$

then you can show with the CCR rule that

$$\{10 \underset{0.5}{\longmapsto} 2\} C \{10 \underset{0.5}{\longmapsto} 3\}$$

Using the frame rule you can prove

$$\{10 \underset{0.5}{\longmapsto} 2 \star 10 \underset{0.5}{\longmapsto} 2\} C \{10 \underset{0.5}{\longmapsto} 2 \star 10 \underset{0.5}{\longmapsto} 3\}$$

But the proof is useless, because to use this triple in parallel with the resource bundle b the conclusion of the concurrency rule must be

$$\{I_b \star 10 \underset{0.5}{\longmapsto} 2 \star 10 \underset{0.5}{\longmapsto} 2\} C \{I_b \star 10 \underset{0.5}{\longmapsto} 2 \star 10 \underset{0.5}{\longmapsto} 3\}$$

The precondition is false; there is no such heap; the conclusion is vacuous.

In practice you can constrain a command to passivity by passing it only a proportion of the permission you hold. Then it cannot possibly acquire a total permission from anywhere, and you can be sure of its passivity.

8. COUNTING PERMISSIONS IN DETAIL

To model permission counting we have to distinguish between the "source permission", from which read permissions are taken, and the read permissions themselves. We also have to distinguish a total permission from one which lacks some split-off parts.

A total permission is written $E \overset{0}{\mapsto} E'$. A source from which n read permissions have been split is written $E \overset{n}{\longmapsto} E'$. A read permission is written $E \rightarrowtail E'$.[3]

$$\begin{array}{l} E \overset{n}{\longmapsto} E' \rightarrow n \geq 0 \\ E \overset{n}{\longmapsto} E' \wedge n \geq 0 \iff E \overset{n+1}{\longmapsto} E' \star E \rightarrowtail E' \end{array} \tag{10}$$

The assignment and new/dispose axioms are very like (8). Only a total permission, $E \overset{0}{\mapsto} E'$, allows write and dispose.

$$\begin{array}{lll}
\{\mathbf{emp}\} & x{:=}\text{new}(E) & \{x \overset{0}{\mapsto} E\} \\
\{E' \overset{0}{\mapsto} _\} & \text{dispose } E' & \{\mathbf{emp}\} \\
\{R^x_E\} & x{:=}E & \{R\} \\
\{E' \overset{0}{\mapsto} _\} & [E']{:=}E & \{E' \overset{0}{\mapsto} E\} \\
\{E' \rightarrowtail E\} & x{:=}[E'] & \{E' \rightarrowtail E \wedge x = E\}
\end{array} \tag{11}$$

Read permissions ($\rightarrowtail$) guarantee passivity in just the same way as non-integral fractional permissions.

8.1 A counting permission example

I can't yet treat the original version of the readers-and-writers algorithm because I can't yet deal formally with permission to access stack variables (see section 13.2). I can deal with it,though, if I transform the readers prologue and epilogue, both mutex-protected critical sections, into CCRs, as shown in figure 3. I've added a guard ($\mathit{count} > 0$) on the reader epilogue, and made some insignificant changes which make the proof presentation easier.

Suppose the shared resource is a cell pointed to by y and the two bundles have invariants

$$\begin{array}{l} \mathit{write}\text{: if } \mathit{write} = 0 \text{ then } \mathbf{emp} \text{ else } y \overset{0}{\mapsto} \_ \text{ fi} \\ \mathit{read}\text{: if } \mathit{count} = 0 \text{ then } \mathbf{emp} \text{ else } y \overset{\mathit{count}}{\longmapsto} \_ \text{ fi} \end{array} \tag{12}$$

[3]In terms of the model (section 10.2), it should be written $E \overset{-1}{\longmapsto} E'$, but it simplifies the proof theory if I use a special arrow and reserve the annotation of permissions for positive integers.

$\{\mathbf{emp}\}$
  with *read* when true do
    $\{$if $count = 0$ then $\mathbf{emp}$ else $y \xmapsto{count} \_$ fi $\star\, \mathbf{emp}\}$
      if $count = 0$ then $\{\mathbf{emp}\}$ P(*write*) $\{y \xmapsto{0} \_\}$
                     else $\{y \xmapsto{count} \_\}$ skip $\{y \xmapsto{count} \_\}$
      fi
    $\{y \xmapsto{count} \_\}$
      $count +:= 1$
    $\{y \xmapsto{count-1} \_\} \therefore \{y \xmapsto{count} \_ \star z \rightarrowtail \_\}$
  od
$\{z \rightarrowtail N\}$

$\{z \rightarrowtail N\}$
  with *read* when $count > 0$ do
    $\{$if $count = 0$ then $\mathbf{emp}$ else $y \xmapsto{count} \_$ fi $\star\, z \rightarrowtail N \wedge count > 0\}$
      $count -:= 1$
    $\{$if $count + 1 = 0$ then $\mathbf{emp}$ else $y \xmapsto{count+1} \_$ fi $\star\, z \rightarrowtail N \wedge count + 1 > 0\}$ $\therefore$
    $\{y \xmapsto{count+1} \_ \star z \rightarrowtail N \wedge count \geq 0\} \therefore \{y \xmapsto{count} \_ \wedge count \geq 0\}$
      if $count = 0$ then $\{y \xmapsto{0} \_\}$ V(*write*) $\{\mathbf{emp}\}$
                     else $\{y \xmapsto{count} \_\}$ skip $\{y \xmapsto{count} \_\}$
      fi
    $\{$if $count = 0$ then $\mathbf{emp}$ else $y \xmapsto{count} \_$ fi $\star\, \mathbf{emp}\}$
  od
$\{\mathbf{emp}\}$

Figure 6: Resource release in readers prologue and reclamation in epilogue

The *write* semaphore-bundle owns a total permission which it releases on P and claims on V. It's easy to prove that using the CCR rule, as shown in figure 5. From the proofs you can see that it would be impossible to P the semaphore if you already own the permission, and wrong to V it if you don't.

Then a proof that the readers prologue releases a read permission into the surrounding program goes as in figure 6. The epilogue reverses the action, with the additional requirement that *count* must be non-zero on entry to ensure that the resource-bundle invariant is preserved. (Investigations are underway to eliminate this infelicity in our treatment: if the readers and/or writers don't do anything silly, of course $count > 0$ on entry to the epilogue.)

8.2 No more critical sections?

When Dijkstra [9] introduced semaphores, the name referred to those mechanical railway signals which let only one train at a time onto a critical (signal-controlled) section of track. This *block signalling* technique provides mutual exclusion in the critical section. Hardware provides mutual exclusion only between executions of the test-and-set / increment instructions which implement the semaphore and we must rely on proof techniques to show mutual exclusion in critical sections. Sometimes the critical sections of a program are hard to identify or non-existent. Brinch Hansen, arguing for the use of monitors instead of semaphores, stated the problem:

> Since a semaphore can be used to solve arbitrary synchronizing problems, a compiler cannot conclude that a pair of *wait* and *signal* operations on a given semaphore initialized to 1 delimits a critical region, nor that a missing member of such a pair is an error. [4]

Our treatment (following [15]) inverts Dijkstra's view by focussing on permission rather than prohibition. A thread in possession of a permission can use it at any time. Separation guarantees absence of races even while permitting sharing. Semaphores are resource-holders which can be unlocked, not guardians of critical sections.

In figure 3 there is mutual exclusion between the readers prologue and epilogue and between the four uses of the *write* semaphore, but otherwise it is unnecessary to invoke the notion of critical section. I can write a silly but perfectly verifiable pattern use of read permissions:

$$\begin{array}{l} prologue; prologue; prologue; \\ \left(\begin{array}{c} reader_1; \\ epilogue \end{array} \middle\| reader_2 \middle\| reader_3\right); \\ epilogue; reader_4; epilogue \end{array}$$

and an even sillier use of total permission:

$$\mathrm{P}(write); writer_1; \left(reader_5 \,\|\, reader_6\right); writer_2; \mathrm{V}(write)$$

If the *count* variable of figure 1 were in the heap, I could apply resourcing to a version of the algorithm which uses a mutex m instead of the CCRs of figure 3, and produce a proof entirely free of the notion of critical section (but see also section 13.2).

9. WHY TWO MECHANISMS?

The most striking feature of our presentation is that there are two distinct models and two distinct logics. That's because proof requires two distinct and somewhat incompat-

ible properties: unbounded divisibility suits some problems; unbounded counting suits others.

Problems which can exploit fractional permissions exhibit symmetrical splitting, indefinite subdivision, and simple and predictable split/combine behaviour. Those which need counting permissions have asymmetrical splitting with an authority and a user, counting in the program, and split / combine as actions of the program rather than properties of its structure.

It should be clear already that some problems don't suit the notion of fractional permissions. It would be extremely difficult, perhaps impossible, to specify and prove the readers-and-writers program using that technique. The read permissions are all given out from the same point and are all identical: they can be given back in any order, and anything other than counting-accounting would be absurdly over-complicated.

Since counting is so clearly sometimes necessary, I have to make a similar case for fractions. I do so by example.

9.1 Lambda-term substitution

Our example is substitution on a lambda term, performed in parallel for the sub-terms of a function application.

The syntax of lambda terms is

$$T ::= \mathsf{Lam}\ v\ T \mid \mathsf{App}\ T\ T \mid \mathsf{Var}\ v \tag{13}$$

I define substitution (for simplicity, allowing variable capture) in the obvious way

$$\begin{aligned}
(\mathsf{Lam}\ v'\ \beta)[\tau/v] &= \begin{cases} \mathsf{Lam}\ v'\ (\beta[\tau/v]) & v \neq v' \\ \mathsf{Lam}\ v'\ \beta & v' = v \end{cases} \\
(\mathsf{App}\ \phi\ \alpha)[\tau/v] &= \mathsf{App}\ (\phi[\tau/v])\ (\alpha[\tau/v]) \\
(\mathsf{Var}\ v')[\tau/v] &= \begin{cases} \mathsf{Var}\ v' & v \neq v' \\ \tau & v = v' \end{cases}
\end{aligned}$$

A possible heap representation predicate for a lambda term pointed to by x with access permission z is

$$\begin{aligned}
&\mathrm{AST}\ x\ (\mathsf{Lam}\ v\ \beta)\ z \mathrel{\hat{=}} \exists b.(x \overset{z}{\mapsto} 0, v, b \star \mathrm{AST}\ b\ \beta\ z \\
&\mathrm{AST}\ x\ (\mathsf{App}\ \phi\ \alpha)\ z \mathrel{\hat{=}} \exists f, a. \begin{pmatrix} x \overset{z}{\mapsto} 1, f, a \star \mathrm{AST}\ f\ \phi\ z \star \\ \mathrm{AST}\ a\ \alpha\ z \end{pmatrix} \\
&\mathrm{AST}\ x\ (\mathsf{Var}\ v)\ z \mathrel{\hat{=}} x \overset{z}{\mapsto} 2, v
\end{aligned}$$

For simplicity, variables are represented by integers; the 0/1/2 tags which distinguish different kinds of nodes in the heap are arbitrarily chosen.

The substitution function is given in Figure 7 (the program is abbreviated: some of the calculations and assignments in the figure represent sequences of correct separation-logic assignments). The algorithm reads the node type from the heap: for a lambda abstraction it checks if the bound variable is the same variable as the substitution and if not substitutes on the body; for an application it performs the substitution on each sub-term concurrently; and for a variable if it is the variable being replaced it calls a copy function and returns a pointer to that copy.

The copy function has the specification

$\{\mathrm{AST}\ y\ \tau\ z\}$ `x := copy y` $\{\mathrm{AST}\ y\ \tau\ z \star \mathrm{AST}\ x\ \tau\ 1\}$

The substitution function is specified as

$\{\mathrm{AST}\ x\ \tau\ 1 \star \mathrm{AST}\ y\ \tau'\ z\}$
`z := subst x y v`
$\{\mathrm{AST}\ z\ (\tau[\tau'/v])\ 1 \star \mathrm{AST}\ y\ \tau'\ z\}$

The interesting part of the proof is the application case ($[x] = 1$).

$\{\mathrm{AST}\ x\ (\mathsf{App}\ \phi\ \alpha)\ 1 \star \mathrm{AST}\ y\ \tau'\ z\}$
`[x+1] := subst [x+1] y v ||`
`[x+2] := subst [x+2] y v`
$\{\mathrm{AST}\ x\ (\mathsf{App}\ (\phi[\tau'/v])\ (\alpha[\tau'/v]))\ 1 \star \mathrm{AST}\ y\ \tau'\ z\}$

The proof requires the substituted lambda term to be split into two pieces, and needs the equivalence

$$\mathrm{AST}\ y\ \tau\ (z + z') \Leftrightarrow \mathrm{AST}\ y\ \tau\ z \star \mathrm{AST}\ y\ \tau\ z'$$

This equivalence is proved by induction on the structure of τ.[4] Using the Hoare-logic rule of consequence with this equivalence and the definition of AST, followed by an application of the frame rule, I can derive the following proof obligation

$$\left\{ \begin{matrix} x \mapsto 1, f, a \star \mathrm{AST}\ f\ \phi\ 1 \star \mathrm{AST}\ y\ \tau'\ (z/2) \star \\ \mathrm{AST}\ a\ \alpha\ 1 \star \mathrm{AST}\ y\ \tau'\ (z/2) \end{matrix} \right\}$$

`[x+1] := subst [x+1] y v ||`
`[x+2] := subst [x+2] y v`

$$\left\{ \begin{matrix} x \mapsto 1, f', a' \star \mathrm{AST}\ f'\ (\phi[\tau'/v])\ 1 \star \mathrm{AST}\ y\ \tau'\ (z/2) \star \\ \mathrm{AST}\ a'\ (\alpha[\tau'/v])\ 1 \star \mathrm{AST}\ y\ \tau'\ (z/2) \end{matrix} \right\}$$

The proof is straightforward from the specification of subst. But – and this is the point which justifies fractional rather than counting permissions – because the proof uses fractions I don't need to know how many times the permission $\mathrm{AST}\ y\ \tau'\ (z/2)$ will have to be split to complete either of the parallel threads (i.e. how many application nodes there are altogether in ϕ and α). The split is genuinely symmetrical; both sides may need to split further; there isn't any machinery in the program which corresponds to a splitting authority.

This example illustrates a situation in which fractional permissions lead to simpler and more usable proofs than counting. Counting fits problems where a thread or a library module is used as an authority to give out ownership. Either approach can conceivably be used in the other's domain, but at an unnecessary cost.

10. MODELS

Although there are two logical mechanisms, their models are very similar.

10.1 General structure of models

We will consider models where heaps are partial functions

$$\mathit{Heaps} = L \rightharpoonup (V \times M)$$

where L and V are the sets of locations and values respectively, and M is equipped with a partial commutative semigroup structure, where the binary operator is denoted $\star$. The idea is that $\star$ adds permissions together, and the order in which permissions are combined does not matter. We extend $\star$ to the set $V \times M$ as follows:

$$(v, m) \star (v', m') = \begin{cases} (v, m \star m') & \text{if } v = v' \text{ and} \\ & m \star m' \ \mathit{defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

[4] But see section 13.1.

```
subst x y v =
  if [x] = 0 then
    if [x+1] != v then
      [x+2] := subst [x+2] y v
    else skip fi;
    x
  elsf [x] = 1 then
    ([x+1] := subst [x+1] y v ||
      [x+2] := subst [x+2] y v);
    x
  elsf [x+1] = v then
     dispose x; dispose (x+1);
     new(2, copy y)
  else
     x
  fi
```

Figure 7: Substitution Source

and correspondingly to the set *Heaps*:

- $h \star h'$ defined iff $h(l) \star h'(l)$ defined for each $l \in dom(h) \cap dom(h')$
- $(h \star h')(l) = \begin{cases} h(l) & \text{if } h'(l) \text{ undefined} \\ h'(l) & \text{if } h(l) \text{ undefined} \\ h(l) \star h'(l) & \text{otherwise} \end{cases}$

Given a choice of M, the syntax and semantics of the $(\mapsto)$ predicate is

$$s, h \vDash E \xmapsto{m} E' \quad \text{iff} \quad \begin{pmatrix} dom(h) = [\![E]\!]s \text{ and} \\ h([\![E]\!]s) = ([\![E']\!]s, m) \end{pmatrix}$$

A model (M, m_W) is given by a concrete M, together with a distinguished element $m_W \in M$, the write permission, such that:

$$m_W \star m' \text{ undefined for any } m' \in M \tag{14}$$

$$\begin{array}{c} \text{for all } m' \in M \text{ there exists } m'' \in M \\ \text{such that } m' \star m'' = m_W \end{array} \tag{15}$$

Intuitively, the two conditions say that m_W is the maximal permission, and any permission can be extended to obtain the maximal one.

10.2 Model of counting permissions

We distinguish read permissions from others. We count the number of read permissions that have been flaked off a source permission. You can't combine two source permissions. You can't combine a source permission with more read permissions than it's generated. Given that, you can record permission to access a heap cell is represented by an integer: 0 for a total permission, -1 for a read permission, $+k$ for a source permission from which k read permissions have been taken.

Formally, the model is $(Z, \star_1)$, where Z is the set of integers and $\star_1$ is defined as follows:

$$i \star_1 j = \begin{cases} \text{undefined} & \text{if } i \geq 0 \text{ and } j \geq 0 \\ \text{undefined} & \text{if } (i \geq 0 \text{ or } j \geq 0) \text{ and } i + j < 0 \\ i + j & \text{otherwise} \end{cases}$$

The write permission is 0. The following properties hold:

$$\begin{array}{l} E \xmapsto{n} E' \iff E \xmapsto{n+m} E' \star E \xmapsto{-m} E' \\ \qquad \text{when } n \geq 0 \text{ and } m > 0 \\ E \xmapsto{-(n+m)} E' \iff E \xmapsto{-n} E' \star E \xmapsto{-m} E' \\ \qquad \text{when } n, m > 0 \end{array} \tag{16}$$

10.3 Model of fractional permissions

Fractions are easy: just add them up, make sure you don't go zero, negative or greater than 1.

The model is $(\{q \in Q \mid 0 < q \leq 1\}, \star_2)$, where Q is the set of rational numbers and $\star_2$ is defined as follows:

$$q \star_2 q' = \begin{cases} \text{undefined} & \text{if } q + q' > 1 \\ q + q' & \text{otherwise} \end{cases}$$

The write permission is 1. The following property holds:

$$E \xmapsto{q+q'} E' \iff (E \xmapsto{q} E' \star E \xmapsto{q'} E') \wedge q + q' \leq 1 \tag{17}$$

10.4 Combined Model

By making read permissions divisible, it's possible to combine the properties of fractional and counting permissions. You finish up with an asymmetrical fractional model. Despite the fact that there is only one model, there are still two ideas – proliferation and divisibility – each of which seems to be necessary, neither of which is subservient to the other. The proofs sketched above are all supportable in the combined model. The only significant difference is that it is impossible in the combined model to set up a logic in which read permissions *cannot* be split once issued, and control is entirely with the splitting authority – a programming discipline which may prove to be useful in certain situations.

The model $(Q, \star_3)$ combines counting and fractional permissions, where Q is the set of rational numbers and $\star_3$ is defined as follows:

$$q \star_3 q' = \begin{cases} \text{undefined} & \text{if } q \geq 0 \text{ and } q' \geq 0 \\ \text{undefined} & \text{if } (q \geq 0 \text{ or } q' \geq 0) \text{ and } q + q' < 0 \\ q + q' & \text{otherwise} \end{cases}$$

The write permission is 0. The following properties hold:

$$\begin{array}{l} E \xmapsto{q} E' \iff E \xmapsto{q+q'} E' \star E \xmapsto{-q'} E' \\ \qquad \text{when } q \geq 0 \text{ and } q' > 0 \\ E \xmapsto{-(q+q')} E' \iff E \xmapsto{-q} E' \star E \xmapsto{-q'} E' \\ \qquad \text{when } q, q' > 0 \end{array} \tag{18}$$

11. SEQUENTIAL SEMANTICS

If we restrict attention to the sequential case, the semantics of commands in the permissions model is a minor modification of the usual semantics. It is then possible to show all the usual results about locality, weakest preconditions etc.

11.1 Semantics of commands

Given a model we define the semantics of atomic commands as follows

$$\begin{array}{c} \dfrac{[\![E]\!]s = v}{x := E, s, h \rightsquigarrow (s \mid x \mapsto v), h} \\ \dfrac{[\![E']\!]s = l \quad [\![E]\!]s = v \quad h(l) = (\_, m_W)}{[E'] := E, s, h \rightsquigarrow s, (h \mid l \mapsto (v, m_W))} \\ \dfrac{[\![E']\!]s = l \quad h(l) = (v, m)}{x := [E'], s, h \rightsquigarrow (s \mid x \mapsto v), h} \\ \dfrac{l \in L - dom(h) \quad [\![E]\!]s = v}{x := \mathrm{new}(E), s, h \rightsquigarrow (s \mid x \mapsto l), (h \mid l \mapsto (v, m_W))} \\ \dfrac{[\![E']\!]s = l \quad h(l) = (\_, m_W)}{\mathrm{dispose}(E'), s, h \rightsquigarrow s, (h - l)} \end{array} \tag{19}$$

We observe that this is the usual standard semantics of these commands, plus runtime checks on permissions.

11.2 Small Axioms

We give small axioms for the atomic commands, in the style of [14]; the frame rule can be used to infer complex specifications from these simple ones.

The assignment and new/dispose axioms are as you would expect. Only the total permission, m_W, gets write and dispose access. In contrast, any permission m grants read access.

$$\begin{array}{lll} \{R^x_E\} & x{:=}E & \{R\} \\ \{E' \xmapsto{m_W} \_\} & [E']{:=}E & \{E' \xmapsto{m_W} E\} \\ \{E' \xmapsto{m} E\} & x{:=}[E'] & \{E' \xmapsto{m} E \wedge x = E\} \\ \{\mathbf{emp}\} & x{:=}\mathrm{new}(E) & \{x \xmapsto{m_W} E\} \\ \{E' \xmapsto{m_W} \_\} & \mathrm{dispose}\, E' & \{\mathbf{emp}\} \end{array} \tag{20}$$

The side condition on the third axiom is that x does not occur free in E or E'.

11.3 Frame Property, termination and safety monotonicity

Soundness of the frame rule depends on the local behaviour of commands. The locality of commands was formalized in [21] with three properties:

- Safety Monotonicity: if C, s, h is safe and $h \star h'$ is defined, then $C, s, h \star h'$ is safe.
- Termination Monotonicity: if C, s, h must terminate normally and $h \star h'$ is defined, then $C, s, h \star h'$ must terminate normally.
- Frame Property: if C, s, h_0 is safe, and $C, s, h_0 \star h_1 \rightsquigarrow^\star s', h'$ then there is h'_0 such that $C, s, h_0 \rightsquigarrow^\star s', h'_0$ and $h' = h'_0 \star h_1$.

The same properties hold when heaps are built using permission models. In particular, condition (14) ensures that Safety Monotonicity and Frame Property hold for the commands in (19). A simple proof of soundness of the Frame Rule follows.

11.4 Weakest preconditions

Weakest preconditions are obtained as a variation of the usual definitions by decorating the $\mapsto$ assertions. Weakest preconditions are derivable, as usual, from the small axioms.

12. SOUNDNESS

The soundness of our logics would appear to be shown by adapting the proof presented in [6] to each of our versions of resource permission and separation. Adaptation is not a daunting task because of the framework of that proof. We intend, however, to take an alternative route: work is already in progress on the soundness of a general model which can be instantiated with a range of different definitions.

13. FUTURE WORK

The notion of permission is a strong fertiliser for novel ideas about interesting problems. We already have more than we can deal with. Some of those closest to a solution are variables as resource, existence permissions and semaphores in the heap.

13.1 Oddities of inductive definitions

A separation-logic heap predicate for a tree (e.g. in [2]: versions differ according to whether they have explicit Tips or store values at Nodes) is

$$\begin{array}{r} \mathrm{tree\ nil\ Empty} \mathrel{\hat{=}} \mathbf{emp} \\ \mathrm{tree}\ t\ (\mathsf{Tip}\ \alpha) \mathrel{\hat{=}} t \mapsto 0, \alpha \\ \mathrm{tree}\ t\ (\mathsf{Node}\ \lambda\ \rho) \mathrel{\hat{=}} \exists l, r \cdot \begin{pmatrix} t \mapsto 1, l, r \star \\ \mathrm{tree}\ l\ \lambda \star \mathrm{tree}\ r\ \rho \end{pmatrix} \end{array} \tag{21}$$

It's tempting to define a ztree as a tree whose pointers are all decorated with a fractional permission:

$$\begin{array}{r} \mathrm{ztree}\ z\ \mathrm{nil\ Empty} \mathrel{\hat{=}} \mathbf{emp} \\ \mathrm{ztree}\ z\ t\ (\mathsf{Tip}\ \alpha) \mathrel{\hat{=}} t \underset{z}{\mapsto} 0, \alpha \\ \mathrm{ztree}\ z\ t\ (\mathsf{Node}\ \lambda\ \rho) \mathrel{\hat{=}} \exists l, r \cdot \begin{pmatrix} t \underset{z}{\mapsto} 1, l, r \star \\ \mathrm{ztree}\ z\ l\ \lambda \star \mathrm{ztree}\ z\ r\ \rho \end{pmatrix} \end{array} \tag{22}$$

(cf. the AST predicate in the term-rewriting example above).

We do now have $\mathrm{ztree}\ (z + z')\ t\ \tau \iff \mathrm{ztree}\ z\ t\ \tau \star \mathrm{ztree}\ z'\ t\ \tau$, but sometimes only vacuously! ($\star$) no longer guarantees disjointness of domains, because of (7), so I can demonstrate some peculiarities. Consider the following example (heavily abbreviated, in particular using $\wedge\!\!\wedge$ for conditional conjunction, like C's `&&`):

$$\begin{array}{l} \text{if } t \neq nil \wedge\!\!\wedge\ [t] = 1 \wedge\!\!\wedge\ [t+1] = [t+2] \wedge\!\!\wedge \\ \quad [t+1] \neq \mathrm{nil} \wedge\!\!\wedge\ [[t+1]] = 0 \\ \text{then } [[t+1]+1] := [[t+1]+1] + 1 \text{ else skip fi} \end{array} \tag{23}$$

This program checks if it has been given a heap consisting of a Node in which left and right pointers are equal and point to a Tip; it then attempts to increment the value in that tip. Such a heap contains a DAG, not a tree: I would have hoped that the ztree predicate enforced tree structure just as tree does. Sharing can occur in ztrees when $z \leq 0.5$, because nothing in the definition provides against the possibility that part or all of the l heap isn't then shared with the r heap.

That's not all. The heap

$$x \xmapsto{0.5} 1, l, l \star l \xmapsto{0.5} 0, 3 \star l \xmapsto{0.5} 0, 3$$

satisfies

$$\mathrm{ztree}\ 0.5\ x\ (\mathsf{Node}\ (\mathsf{Tip}\ 3)\ (\mathsf{Tip}\ 3))$$

Program (23)) will change it so that

$$\mathrm{ztree}\ 0.5\ x\ (\mathsf{Node}\ (\mathsf{Tip}\ 4)\ (\mathsf{Tip}\ 4))$$

Given

$$x \xmapsto[0.25]{} 1, l, l \star l \xmapsto[0.25]{} 0, 3 \star l \xmapsto[0.25]{} 0, 3$$

– the same locations with a different fractional permission – the same program will abort. It's impossible to have

$$x \xmapsto[0.75]{} 1, l, l \star l \xmapsto[0.75]{} 0, 3 \star l \xmapsto[0.75]{} 0, 3$$

– there's no sharing in a ztree when $z > 0.5$.

This is all very peculiar. We don't have passivity in ztrees as we did with single cells and the values of fractions seem to matter: has everything gone horribly wrong? Well no, it hasn't: not quite. You can use the technique suggested in section 7.1: pass subprograms only a part of the permission you hold. The term-rewriting example above isn't scuppered if the AST you pass in is a DAG, because the copy rule makes a new copy with 1.0 permission, and it's a sequential program so parallel subprograms can't conspire to accumulate total permission. In effect we can rely on passivity and there's no paradox, after all.

Inductive definitions can be similarly confusing using counting read-only permissions rather than fractions: there's no possibility of modification by coincidence of subtrees, but once again DAGs are allowed where we'd like to have only trees. Separation logic isn't broken by this discovery, but we don't yet know how to write inductive definitions which combine obvious separation with obvious reduction of permission.

13.2 Variables as resources

Separation logic's success with the heap is partly good luck. Hoare logic's variable-assignment rule finesses the distinction between program variables and logical variables and assumes an absence of program-variable aliasing. The price for that sleight of hand is paid in the array-element-assignment rule, which has to deal with aliasing of integer indices using arithmetic in the proof.

In programming languages a little more developed than that treated by Hoare logic, Strachey's distinction of rvalue (variable address) and lvalue (variable contents) is made explicit and can be exploited. Because heap rvalues and lvalues alike are integers, separation logic can ignore the distinction and use the conventional Hoare logic variable assignment rule. The use of 'pure' expressions (constants and variable names) not referring to the heap, and the restriction to particular forms of assignment that essentially constrain us to consider single transfers between the 'stack' (registers) and the heap, make it all work. Descriptions of the heap are essentially pictures of separation; issues of aliasing then rarely arise, and we can regard separation as *the* problem.

Separation logic treats heap locations and variables quite differently. Heap locations are localised resources whose allocation can be reasoned about, for example in the frame rule. But stacks are global: that fact shows up in the frame rule's proviso, which requires extra-logical syntactic separation between resource formula P and the set of variables assigned to in C. I'd much prefer to be able to integrate descriptions of variables as resources into the frame rule, make ($\star$) do all the work, and eliminate the proviso.

The most obvious solution puts the stack in the heap. This naive approach doesn't work – or rather, it doesn't work conveniently, because it destroys the main advantage of Hoare logic, which is the elegant simplicity of the variable-assignment rule. Drawing pictures of separation in the stack necessarily exposes the rvalue/lvalue distinction and the pun between logical and program variables which lies behind Hoare logic's use of straightforward substitution no longer makes sense.

The problems of reasoning about concurrent programs make treatment of variables-as-resources more than a matter of aesthetics, more than a desire to eliminate ugly provisos. I would like to be able to describe transfer of ownership of *variables* into and out of resource bundles. I can explain the original readers and writers algorithm (figure 1) if the *count* variable is locked away in the m mutex, released by P and reclaimed by V. Semantically the notion isn't very difficult, but integrating it into a useful proof theory is proving difficult. It's crucial that this step is made so that we can have an effective logic of storage-resource in concurrent programs (and, by the by, eliminate any logical dependence on critical sections and split binary semaphores, and maybe even provide a Hoare logic that deals with variable aliasing).

13.3 Existence permissions

The treatments above separate total permission from read permission. This is not the only distinction it is useful to draw. A semaphore, for example, has permission to read and write its own variable. A concurrent thread has no access to that variable but can P or V it. *Existence permissions* provide evidence of a resource's existence, but no access to its contents. They allow us to separate total from read/write permissions. A user knows that a semaphore exists, but cannot read it. The semaphore can't dispose itself (see below) unless its permission is total – that is, unless there are no users with existence permissions.

The proof theory of existence permissions seems to be a variation on fractional permissions. We don't yet have a satisfying and elegant model.

13.4 Semaphores in the heap

I first encountered permission counting in the context of pipeline processing in the Intel IXP network processor chip [12]. A read thread waits for packet data to arrive on a particular network port, and assembles packet fragments into a newly-allocated packet buffer. It then immediately passes the buffer, through an inter-thread queue, to the first processing thread, and turns to wait for the next packet. The processing thread does some work on the buffer and passes it on to the next processing thread, and so on until eventually it arrives at a write thread which disassembles the processed packet, transmits the pieces of data through its network port, and disposes the buffer.

This is *single-casting*, in which every packet has a single destination address, and it's a beautiful example of the power of ownership transfer. Each thread owns the buffer until it transfers it into an inter-thread queue, an example of a shared resource bundle. Each thread has a loop invariant of **emp**, so if there are any space leaks it can only be that a queue is overlooked and never emptied. The most important feature of the technique is its simplicity – the read thread, which allocates the buffer, has nothing to do with its disposal – and efficiency – no need for accounting in the program, only in the proof.

In *multicasting* a single packet can be distributed to several destinations at once. An obvious technique would be to copy the incoming packet into several buffers, but the desire for efficiency and maximum packet throughput com-

pels sharing. The solution adopted is to use a semaphore-protected count of access permissions to determine when everybody has finished and the buffer can be disposed. In principle it's not much more difficult to program, but there's many a slip, so it would be good to be able to formalise it [10].

The obstacles to a proof don't seem unsurpassable but I cannot claim that they are conquered already. The program must dynamically allocate semaphores as well as buffers, and the idea of semaphores in the heap makes theoreticians wince. The semaphore has to be available to a shared resource bundle: that means a bundle will contain a bundle which contains resource, a notion which makes everybody's eyes water. None of it seems impossible, but it's a significant problem, and solving it will be a small triumph.

Acknowledgements

Doug Lea first suggested that we should read $\mapsto$ as a permission. John Boyland put us on to a means of accounting. Josh Berdine, a conspirator in the East London Massive, helped us argue through early ideas and refine later ones. John Reynolds questioned some wild early versions. Our honorary guru Hongseok Yang beat us back from the wilder shores of speculation. Those excesses that remain are all my own.

Calcagno, O'Hearn and Parkinson were supported by EPSRC.

14. REFERENCES

[1] R. 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.

[2] R. Bornat, C. Calcagno, and P. O'Hearn. Local reasoning, separation and aliasing. SPACE Workshop, Venice, 2004.

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

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

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

[6] 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*.

[7] R. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.

[8] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14(10):667–668, 1971.

[9] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, 1968. Reprinted in [5].

[10] R. Ennals, R. Sharp, and A. Mycroft. Linear types for packet processing. In *To appear. Proceedings of the 2004 European Symposium on Programming (ESOP)*, LNCS. Springer-Verlag, 2004.

[11] C. A. R. Hoare. Towards a theory of parallel programming. In Hoare and Perrott, editors, *, in , ed. Operating System Techniques,, 1972.*, pages 61–71. Academic Press, 1972.

[12] E. J. Johnson and A. Kunze. *IXP2400/2800 Programming: The Complete Microengine Coding Guide*. Intel Press, 2003.

[13] P. O'Hearn. Notes on separation logic for shared-variable concurrency. unpublished, Jan. 2002.

[14] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *CSL 2001*, pages 1–19. Springer-Verlag, 2001. LNCS 2142.

[15] P. W. O'Hearn. Resources, concurrency and local reasoning. to appear in *Theoretical Computer Science*; preliminary version published as [16].

[16] 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 is [15].

[17] P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, June 1999.

[18] D. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*, volume 26 of *Applied Logic Series*. Kluwer Academic Publishers, 2002.

[19] J. Reynolds. Separation logic: a logic for shared mutable data structures. Invited Paper, LICS'02, 2002.

[20] J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In J. Davies, B. Roscoe, and J. Woodcock, editors, *Millennial Perspectives in Computer Science*, pages 303–321. Palgrave, 2000.

[21] H. Yang and P. O'Hearn. A semantic basis for local reasoning. In *5th FOSSACS*, pages 402–416. Springer-Verlag, 2002.