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

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:

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

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

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

(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.

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

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.

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

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

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

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

– 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

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

then you can show with the CCR rule that

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

Using the frame rule you can prove

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

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

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

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 $E \overset{0}{\longmapsto} E'$. 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]

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

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

$$\begin{array}{rll} \{\mathbf{emp}\} & x{:=}\mathrm{new}(E) & \{x \overset{0}{\longmapsto} E\} \\ \{E' \overset{0}{\longmapsto} \_\} & \quad\mathrm{dispose}\, E' & \{\mathbf{emp}\} \\ \{R^x_E\} & x{:=}E & \{R\} \\ \{E' \overset{0}{\longmapsto} \_\} & [E']{:=}E & \{E' \overset{0}{\longmapsto} E\} \\ \{E' \rightarrowtail E\} & x{:=}[E'] & \{E' \rightarrowtail E \wedge x = E\} \end{array} \tag{11}$$

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

```
READERS                                    WRITER

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

  ... reading happens here ...;              ... writing happens here ...

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

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

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 ($count > 0$) on the reader epilogue, and made some insignificant changes which make the proof presentation easier.

```
READERS                                              WRITER

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

  ... reading happens here ...; ??                     ... writing happens here ...

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

Figure 3: Readers and writers: CCR version

Additional Properties of Permissions

$$E \xmapsto{n_1} E_1 * E \xmapsto{n_2} E_2 \Rightarrow E_1 = E_2$$

$$E \xmapsto{n_1} E_1 * E \xmapsto{n_2} E_2 \Leftrightarrow (E \xmapsto{n_1} E_1 * E \xmapsto{n_2} E_2) \wedge E_1 = E_2$$

$$E \xmapsto{n_1} - * E \xmapsto{n_2} -$$

$$\Leftrightarrow (\exists x_1\ E \xmapsto{n_1} x_1) * (\exists x_2 .\ E \xmapsto{n_2} x_2) \qquad x_1, x_2 \text{ fresh}$$

$$\Leftrightarrow \exists x_1, x_2 .\ E \xmapsto{n_1} x_1 * E \xmapsto{n_2} x_2$$

$$\Leftrightarrow \exists x_1, x_2 (E \xmapsto{n_1} x_1 * E \xmapsto{n_2} x_2) \wedge x_1 = x_2$$

$$\Leftrightarrow \exists x_1 .\ (E \xmapsto{n_1} x_1 * E \xmapsto{n_2} x_1)$$

$$E \xmapsto{n+1} - * E \mapsto -$$

$$\Leftrightarrow \exists x_1 .\ (E \xmapsto{n+1} x_1 * E \mapsto x_1)$$

$$\Leftrightarrow \exists x_1\ E \xmapsto{n} x_1$$

$$\Leftrightarrow E \xmapsto{n} -$$

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

$$\begin{array}{r}\textit{write}\text{: if } \textit{write} = 0 \text{ then } \mathbf{emp} \text{ else } y \xmapsto{0} \_ \text{ fi}\\ \textit{read}\text{: if } \textit{count} = 0 \text{ then } \mathbf{emp} \text{ else } y \xmapsto{\textit{count}} \_ \text{ fi}\end{array} \qquad (12)$$

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

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

Figure 5: Proof of pre- and post-condition of P(*write*) and V(*write*)

P(write) ≡ with write when write = 1 do write := 0

V (write) ≡ with write when true do write := 1

$$\begin{array}{r}\textit{write}\text{: if } \textit{write} = 0 \text{ then } \mathbf{emp} \text{ else } y \overset{0}{\mapsto} \_ \text{ fi} \\ \textit{read}\text{: if } \textit{count} = 0 \text{ then } \mathbf{emp} \text{ else } y \xmapsto{\textit{count}} \_ \text{ fi}\end{array} \qquad (12)$$

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

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

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

$$\begin{array}{c} \mathit{prologue}; \mathit{prologue}; \mathit{prologue}; \\ \left(\begin{array}{c} \mathit{reader}_1; \\ \mathit{epilogue} \end{array} \middle\| \; \mathit{reader}_2 \; \middle\| \; \mathit{reader}_3 \right); \\ \mathit{epilogue}; \mathit{reader}_4; \mathit{epilogue} \end{array}$$

and an even sillier use of total permission:

$$\mathrm{P}(\mathit{write}); \mathit{writer}_1; \left(\mathit{reader}_5 \,\|\, \mathit{reader}_6\right); \mathit{writer}_2; \mathrm{V}(\mathit{write})$$

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).

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

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

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

```
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)~~ x := copy y
  else
     x
  fi
```

Figure 7: Substitution Source

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

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

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

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

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 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.

From specification of subst:

$\{\mathrm{AST}\ f\ \phi\ 1 * \mathrm{AST}\ y\ \tau'\ z/2\}$

$f' := \mathrm{subst}\ f\ y\ v$

$\{\mathrm{AST}\ f'\ (\phi[\tau'/v])\ 1 * \mathrm{AST}\ y\ \tau'\ z/2\}$

Then, using the frame rule:

$\{\exists f.\ x+1 \overset{1}{\mapsto} f * \mathrm{AST}\ f\ \phi\ 1 * \mathrm{AST}\ y\ \tau'\ z/2\}$

$f := [x+1];$

$\{x+1 \overset{1}{\mapsto} f * \mathrm{AST}\ f\ \phi\ 1 * \mathrm{AST}\ y\ \tau'\ z/2\}$

$f' := \mathrm{subst}\ f\ y\ v;$

$\{x+1 \overset{1}{\mapsto} f * \mathrm{AST}\ f'\ (\phi[\tau'/v])\ 1 * \mathrm{AST}\ y\ \tau'\ z/2\}$

$[x+1] := f'$

$\{x+1 \overset{1}{\mapsto} f' * \mathrm{AST}\ f'\ (\phi[\tau'/v])\ 1 * \mathrm{AST}\ y\ \tau'\ z/2\}$

$\{\exists f'.\ x+1 \overset{1}{\mapsto} f' * \mathrm{AST}\ f'\ (\phi[\tau'/v])\ 1 * \mathrm{AST}\ y\ \tau'\ z/2\}$

Similarly,

$\{\exists a.\ x+2 \overset{1}{\mapsto} a * \mathrm{AST}\ a\ \alpha\ 1 * \mathrm{AST}\ y\ \tau'\ z/2\}$

$a := [x+2];$

$a' := \mathrm{subst}\ a\ y\ v;$

$[x+2] := a'$

$\{\exists a'.\ x+2 \overset{1}{\mapsto} a' * \mathrm{AST}\ a'\ (\alpha[\tau'/v])\ 1 * \mathrm{AST}\ y\ \tau'\ z/2\}$

Then by parallel composition

$$\{ \exists f, a.\ x+1 \overset{1}{\mapsto} f, a * \mathsf{AST}\ f\ \phi\ 1 * \mathsf{AST}\ a\ \alpha\ 1 * \mathsf{AST}\ y\ \tau'\ z/2 * \mathsf{AST}\ y\ \tau'\ z/2 \}$$

$$\{ \exists f.\ x+1 \overset{1}{\mapsto} f * \mathsf{AST}\ f\ \phi\ 1 * \mathsf{AST}\ y\ \tau'\ z/2 \}$$

```
f := [x+1];
f' := subst f y v;
[x+1] := f'
```

$$\{ \exists f'.\ x+1 \overset{1}{\mapsto} f' * \mathsf{AST}\ f'\ (\phi[\tau'/v])\ 1 * \mathsf{AST}\ y\ \tau'\ z/2 \}$$

||

$$\{ \exists a.\ x+2 \overset{1}{\mapsto} a * \mathsf{AST}\ a\ \alpha\ 1 * \mathsf{AST}\ y\ \tau'\ z/2 \}$$

```
a := [x+2];
a' := subst a y v;
[x+2] := a'
```

$$\{ \exists a'.\ x+2 \overset{1}{\to} a' * \mathsf{AST}\ a'\ (\alpha[\tau'/v])\ 1 * \mathsf{AST}\ y\ \tau'\ z/2 \}$$

$$\{ \exists f', a'.\ x+1 \overset{1}{\mapsto} f', a' * \mathsf{AST}\ f'\ (\phi[\tau'/v])\ 1 * \mathsf{AST}\ a'\ (\alpha[\tau'/v])\ 1 * \mathsf{AST}\ y\ \tau'\ z/2 * \mathsf{AST}\ y\ \tau'\ z/2 \}$$

and by framing with $x \overset{1}{\mapsto} 1$,

$\{AST\ x\ (APP\ \phi\ \alpha)\ 1 * AST\ y\ \tau'\ z\}$

$\{\exists f, a.\ x \overset{1}{\mapsto} 1, f, a * AST\ f\ \phi\ 1 * AST\ a\ \alpha\ 1$

$* AST\ y\ \tau'\ z/2 * AST\ y\ \tau'\ z/2\}$

$f := [x+1];$	$\|\|$	$a := [x+2];$
$f' := subst\ f\ y\ v;$		$a' := subst\ a\ y\ v;$
$[x+1] := f'$		$[x+2] := a'$

$\{\exists f', a'.\ x \overset{1}{\mapsto} 1, f', a' * AST\ f'\ (\phi[\tau'/v])\ 1 * AST\ a'\ (\alpha[\tau'/v])\ 1$

$* AST\ y\ \tau'\ z/2 * AST\ y\ \tau'\ z/2\}$

$\{AST\ x\ (App\ (\phi[\tau'/v])(\alpha[\tau'/v]))\ 1 * AST\ y\ \tau'\ z\}$

$\{AST\ x\ ((App\ \phi\ \alpha)[\tau'/v])\ 1 * AST\ y\ \tau'\ z\}$

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:

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

and correspondingly to the set *Heaps*:

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

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

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

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:

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

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

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:

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

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:

What about $q = 2, q' = -1$?

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

Compare (7)

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:

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

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

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

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.

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

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 $C, s, h_0 \star h_1 \leadsto^* s', h'$ then there is h'_0 such that $C, s, h_0 \leadsto^* s', h'_0$ 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.

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

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

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

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

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

We do now have ztree $(z + z')\ t\ \tau \iff$ ztree $z\ t\ \tau \star$ 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 &&):

$$\begin{aligned} &\text{if } t \neq \mathit{nil} \mathrel{\wedge\!\!\wedge} [t] = 1 \mathrel{\wedge\!\!\wedge} [t+1] = [t+2] \mathrel{\wedge\!\!\wedge} \\ &\qquad [t+1] \neq \text{nil} \mathrel{\wedge\!\!\wedge} [[t+1]] = 0 \\ &\ \text{then } [[t+1]+1] := [[t+1]+1]+1 \text{ else skip fi} \end{aligned} \tag{23}$$

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

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

satisfies

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

Program (23)) will change it so that

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

Given

$$x \underset{0.25}{\longmapsto} 1, l, l \star l \underset{0.25}{\longmapsto} 0, 3 \star l \underset{0.25}{\longmapsto} 0, 3$$

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

$$x \underset{0.75}{\longmapsto} 1, l, l \star l \underset{0.75}{\longmapsto} 0, 3 \star l \underset{0.75}{\longmapsto} 0, 3$$

– 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 compels 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.