Fractional and counting permissions in separation logic

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

Cristiano Calcagno
Department of Computing
Imperial College
University of London
ccris@doc.ic.ac.uk

Peter O'Hearn
Department of Computer Science
Queen Mary College
University of London
ohearn@dcs.qmul.ac.uk

Hongseok Yang
ROPAS
Korean Advanced Institute of
Science and Technology
hyang@ropas.kaist.ac.kr

Abstract

Boyland [2] has shown the utility of fractional permissions in reasoning about programs. They permit concurrent threads simultaneous read access to shared locations, and at the same time prevent disposal or any write access. Fractional permissions fit very well into separation logic. We can go further, and allow infinitesimal permissions to attest to the existence of a heap cell but not give access to its contents. Block permissions allow us to reason about a system which allocates and disposes entire buffers rather than single cells. By adopting an alternative model, we are able to deal with a well-known concurrent programming technique called permission counting.

1 Background

1.1 Basics

Separation logic is one particular model of BI. In the model a heap is a partial map from addresses to values. The simplest heaps are the empty heap **emp** and the singleton $x \mapsto E$; we write $x \mapsto _$ as a shorthand for $\exists v \cdot x \mapsto v$. Two heaps can be combined, using multiplicative conjunction ($\star$) iff their domains are disjoint. 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 P, then we can reason about C and its effects separately from P.

The language that separation logic treats includes new and dispose, abstractions of similar Pascal or C library primitives. In descriptions of separation logic so far new has been seen as a heap creator – it makes a singleton heap – and dispose a corresponding heap destroyer. In the simplest version of the language we don't care what value is in the heap that new creates:

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

There is absolutely no way to make a heap other than with new. 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 the program at the time. (Of course new is easy to implement, so it's really only stage magic.)

Addresses received from new are just integers, and can be manipulated as such, but all a program can do with with the heap via an address is access or modify the addressed value. Writing E for a 'pure' expression – one which doesn't involve heap access – and $[_]$ for heap access, we recognise three forms of assignment:

$$\begin{array}{lll} \{R^x_E\} & x{:=}E & \{R\} \\ \{E' \mapsto _\} & [E']{:=}E & \{E' \mapsto E\} \\ \{E' \mapsto E\} & x{:=}[E'] & \{E' \mapsto E \wedge x = E'\}\ (x \text{ not free in } E, E') \end{array} \tag{3}$$

The use of conventional Hoare logic in the first assignment axiom gives rise to the proviso in the frame rule. We'd love to get rid of that condition, but as we shall see that isn't so easy.

1.2 Ownership

O'Hearn, in [5, 6], gave an alternative reading of separation logic. In a multi-threaded program, $x \mapsto E$ expresses *ownership* of a heap location. He used conditional critical regions with invariants to transfer ownership between threads. This held out the promise that separation logic could build zero-execution-cost barriers between threads. Brookes, in [3], showed that the idea was sound. But Brookes's semantics went further: it required in a separation between writeable stack variables and heap locations of concurrent threads, but permitted threads to share read-only variables *and locations*. We already didn't know how to do that in separation logic. We called it the 'passivity problem', and it was a long-standing worry.

For separation logic users, O'Hearn's alternative reading of the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

$\mapsto$ relation was nevertheless a breakthrough, a liberation. But we needed help to take the next step.

1.3 Fractional permissions

In order to reason about non-interference of concurrent threads, Boyland [2] associates a number 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). $z = 1$ is total control; $0 < z < 1$ is shared access. This enables him to describe the allocation of memory access rights to threads and he points out, correctly, that separation logic can't match this. He suggests, however, that separation logic might be modified to include the equivalent of $P \vDash \varepsilon P \star (1-\varepsilon)P$ and thus be able to deal with shared heaps.

Boyland's suggestion turns out to be the solution to our passivity problem and very much else beside.

2 Fractional permissions in separation logic

Inspired by Boyland [2], we modify the model of separation logic. A heap is now a partial map from addresses to values with permissions. Our first treatment uses Boyland's 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.

$$x \underset{z}{\mapsto} E \implies 0 < z \leq 1 \tag{4}$$

Heaps can be combined iff they agree on values and their permissions combine arithmetically. Reading in the other direction, an existing permission can always be split in two.[1] (We require positive z and z' to avoid silly games like $1 \iff 2 \star -1$.)

$$x \underset{z}{\mapsto} E \star x \underset{z'}{\mapsto} E' \iff E = E' \wedge x \underset{z+z'}{\longmapsto} E \wedge z > 0 \wedge z' > 0 \tag{5}$$

new and dispose deal only in full permissions:

$$\begin{array}{c}\{\mathbf{emp}\}\ x := \mathrm{new}()\ \{x \underset{1}{\mapsto} _\} \\ \{E \underset{1}{\mapsto} _\}\ \ \mathrm{dispose}\, E\ \ \{\mathbf{emp}\}\end{array} \tag{6}$$

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

$$\begin{array}{rll}\{R^x_E\} & x{:=}E & \{R\} \\ \{x \underset{1}{\mapsto} _\} & [x]{:=}E & \{x \underset{1}{\mapsto} E\} \\ \{E' \underset{z}{\mapsto} E\} & x{:=}[E'] & \{E' \underset{z}{\mapsto} E \wedge x = E'\}\ (x \text{ not free in } E, E')\end{array} \tag{7}$$

It's then completely straightforward to check the correctness of the following program, in which parallel threads require simultaneous read access to location $[x]$:

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

2.1 The model of fractional permissions

Seems to be very easy. Seems to be easy to prove the frame property. Watch this space.

3 Fractional permissions for variables

The frame rule (1) takes a subset of the permissions provided to the whole program and supplies them to command C. It is clear that we might give permissions to variables – perhaps treating $(=)$ like $(\mapsto)$ – but it isn't clear how to make the Hoare assignment rule for variables work in that context. The problem is that assertions P, Q and R are read as algebraic formulae: names like x stand for values and can be replaced by formulae describing the same value. If we were to have a permission $x \underset{z}{=} E$ in an assertion, intending to describe a permission to access variable x, it's clear that we can't substitute something else for x – it's being used as a *name* not a description of a value.

At present this is a showstopper for us. If we knew how to do it, we could remove the side-condition from the frame rule, and from its descendants, to great benefit (aliasing conquered, in effect!).

4 Existence permissions

Like Brookes, Boyland equates dispose and write permission: you can dispose of a cell iff you can write it. But there are circumstances where you might be able to write a cell though you can't dispose it, and it is useful sometimes to be sure that a cell can't be disposed even though you or others are able to write it.

There is no need to stick to single-number permissions. We could have two sub-permissions: a dispose and a read/write permission. Either could be subdivided, and only $(0,0)$ would be meaningless. Then to dispose you would need $(1,1)$; to write you would need $(_,1)$ and to read $(_,_)$. Combining permissions would be piecewise addition.

What could you do with a permission of $(1,0)$? Nothing at all, not even read its content; you must wait to combine it with the missing $(0,1)$ fraction again if possible. But there is something *logical* you can do in a proof: if $x \underset{_,0}{\longmapsto}$ then after $y := \mathrm{new}()$ you know, because of the frame rule, that $x \neq y$. Even though you can never access $[x]$, your fraction of the permission constrains new not to overlap that cell.

We can actually do a neater trick which preserves single-number

[1] The numbers z and z' that we choose for a split are entirely arbitrary; only their sum matters as an *aide-memoire* for recombination. Reasoning about their magnitudes would seem to be like reasoning about the particular names we use for the parameters of a theorem. They should be treated as names.

$$
\begin{aligned}
\text{pdag nil } \mathsf{Empty}\ U\ U &\mathrel{\hat{=}} \mathbf{emp} \\
\text{pdag } d\ (\mathsf{Ptr}\ x)\ U\ U &\mathrel{\hat{=}} U\ x = d \wedge \mathbf{emp} \\
\text{pdag } d\ (x : \mathsf{Tip}\ \alpha)\ U\ V &\mathrel{\hat{=}} x \notin \operatorname{dom} U \wedge d, d+1 \notin \operatorname{ran} U \wedge V = U \oplus (x : d) \wedge d \underset{-}{\mapsto} 0, \alpha \\
\text{pdag } d\ (x : \mathsf{Node}\ \lambda\ \rho)\ U\ V &\mathrel{\hat{=}} \exists l, r, U', V' \cdot \begin{pmatrix} x \notin \operatorname{dom} U \wedge x \notin \operatorname{dom} V' \wedge d, d+1, d+2 \notin \operatorname{ran} U \wedge V = V' \oplus (x : d) \wedge \\ d \underset{-}{\mapsto} 1, l, r \star \text{pdag } l\ \lambda\ U\ U' \star \text{pdag } r\ \rho\ U'\ V' \end{pmatrix}
\end{aligned}
$$

Figure 1. Heap predicate for non-cyclic partial DAGs

permissions, and it is amusing to do so. Suppose ι is an infinitesimal: a number greater than zero but smaller than any fraction; then $1 - \iota$ is less than 1 but larger than any fraction. Multiple ιs never amount to as much as a fraction. ι-permissions give no access but prevent disposal of the $1 - \iota$ fragment alone, using the dispose axiom of (6).[2]

We do need to be careful with ι-permissions in the logic. You can't write $x \underset{\iota}{\longmapsto} E$, because that conflicts with the frame rule (you could then prove $\{x \underset{1-\iota}{\longmapsto} E \star x \underset{\iota}{\longmapsto} E\}[x] := E' \{x \underset{1-\iota}{\longmapsto} E' \star x \underset{\iota}{\longmapsto} E\}$). Instead we write $x \underset{\iota}{\longmapsto}$, without even a blank, to emphasise that it isn't shorthand for an existence formula; instead, it's witness for the existence of a permission. We must add axioms for combining ι-permissions (we can have multiple and part ιs – why not?) and for combining them with fractional permissions

$$
\begin{aligned}
x \underset{\iota}{\longmapsto} \star x \underset{\iota'}{\longmapsto} &\iff x \underset{\iota+\iota'}{\longmapsto} \\
x \underset{\iota}{\longmapsto} \star x \underset{z}{\longmapsto} E &\iff x \underset{z+\iota}{\longmapsto} E
\end{aligned} \tag{9}
$$

With ι-permissions we can usefully express the fact that dangling pointers point somewhere, and this seems to have something to say about the *copydag* specification problems discussed in [1].

4.1 The model of existence permissions

We haven't agreed one yet. If we can't, we will be content to forget existence permissions for the time being.

5 Block permissions

The single-cell new and dispose of (6) are usable, but they don't match the Pascal or C originals. It's necessary to be able to allocate records (buffers) of contiguous cells and to make sure that we dispose of them all in one piece. We also have to deal with the fact that pointers into the middle of allocated blocks don't have the same status as pointers to the beginning (the problem of 'skewed sharing'). All these problems can be solved, now we have permission technology!

One way might be to use ι-permissions when a block is split into segments. A neater way, we believe, is to annotate cells with a permission that describes the size and position of the block they came from (a 'ghostly outline') as well as access rights.

$$
x \overset{i,n}{\underset{z}{\longmapsto}} E \rightarrow 0 < z \leq 1 \wedge 0 \leq i < n \tag{10}
$$

In previous versions of the logic, with single-cell new and dispose, $x \mapsto E1, ..., En$ was simply a shorthand for an iterated conjunction $x \mapsto E1 \star x+1 \mapsto E2 \star ... \star x+n-1 \mapsto En$. Now $x \overset{i,n}{\underset{z}{\longmapsto}} E1, ..., Ek$ describes a k-element segment set inside a block whose outline is $x-i ... x-i+n-1$.

For ease of description we describe how to split a segment into single cells (note that the access right z is transmitted to the cells unchanged, because dispose permission is dealt with in (12) by requiring us to dispose an entire block):

$$
\begin{aligned}
&x \overset{i,n}{\underset{z}{\longmapsto}} E1, ..., Ej \iff \\
&\quad x \overset{i,n}{\underset{z}{\longmapsto}} E1 \star (x+1) \overset{i+1,n}{\underset{z}{\longmapsto}} E2 \star ... \star (x+j-1) \overset{i+j-1,n}{\underset{z}{\longmapsto}} Ej
\end{aligned} \tag{11}
$$

Like malloc and free, new and dispose deal with entire fully-populated blocks in a single action:

$$
\begin{array}{lcl}
\{\mathbf{emp}\} & x := \text{new}(E1, ..., En) & \{x \overset{0,n}{\underset{1}{\longmapsto}} E1, ..., En\} \\
\{E \overset{0,n}{\underset{1}{\longmapsto}} E1, ..., En\} & \text{dispose}\, E & \{\mathbf{emp}\}
\end{array} \tag{12}
$$

Two heaps can be combined iff they agree on values and permissions. Using block permissions, that means agreeing on outlines, which must overlap exactly or not at all.

$$
\begin{aligned}
&x \overset{i,n}{\underset{z}{\longmapsto}} E \star x' \overset{i',n'}{\underset{z'}{\longmapsto}} E' \rightarrow \\
&\quad (x = x' \rightarrow E = E' \wedge 0 < z + z' \leq 1) \wedge \\
&\quad \begin{pmatrix} (x - i = x' - i' \wedge n = n') \vee \\ \quad x - i + n \leq x' - i' \vee x' - i' + n' \leq x - i \end{pmatrix}
\end{aligned} \tag{13}
$$

5.1 The model of block permissions

We're still disputing this one. The models we've tried so far all permit situations that the logic can't deal with. We'd be reluctant to give up the idea of block permissions, but the present logical proposal may not survive.

6 Counting permissions and permission counting

The fractional idea suits certain programs and not others. It fits problems where the splitting is symmetrical – each split permission is like every other – and where the splitting is built into the structure of the program, as for example in parallel quicksort. That is, essentially, fork-join programs.

There are other problems which don't fit this paradigm. One is the readers-and-writers program; another is the problem of pipeline processing where a permission to access a buffer is passed from an originator thread to a number of assistants, all of which may pass it on further, and eventually dispose it without the originator's involvement.

The simplest example is the elegant readers and writers program of Courtois et al. [4], shown in figure 2. Two mutex semaphores *read* and *write* protect access; readers must pass through a read-entry

[2] It would be a mistake to regard an ι-permission as a dispose permission or even a fraction of a dispose permission. Disposal seems necessarily to require an *entire* permission, and ι is just large enough to deny that if necessary. In fact you can always subtract an ι from any permission, even from a fractional permission which already doesn't allow disposal, or you can halve an infinitesimal permission to make two infinitesimals.

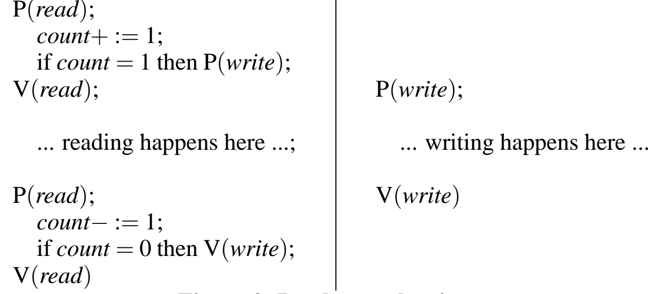

Figure 2. Readers and writers

prologue to gain access and through an epilogue to relinquish it. As initially presented, the program has several critical sections. Using O'Hearn's resource reading of semaphores (cite ??), we understand it using permissions and the atomic semaphore actions are the only units of mutual exclusion.

Suppose that the resource which is to be accessed is a cell pointed to by z. Then to begin with we have a single permission to read, write or dispose that cell. Readers don't need all that permission: they can make do with a part of it. Writers need it all, apart from the dispose permission. The program evidently counts: it should be clear that a permission mechanism which counts will be a better specification-fit than one which keeps track using fractions.

We propose a new logic, and a new model, which fits this problem and others like it. A read permission is now not like other permissions: in particular, you can't split it. We write $z \rightharpoonup E'$ for read access.[3] Other permissions include a count of the number of read permissions that have been chipped from them: we write $E \overset{n}{\longmapsto} E'$.

$$\begin{array}{l} E \overset{n}{\longmapsto} E' \to n \geq 0 \\ E \overset{n}{\longmapsto} E' \iff E \overset{n+1}{\longmapsto} E' \star E \rightharpoonup E' \end{array} \tag{14}$$

The assignment and new/dispose axioms are as you would expect. Only a total permission, $E \overset{0}{\longmapsto} E'$, gets write and dispose access.

$$\begin{array}{lll} \{R^x_E\} & x{:=}E & \{R\} \\ \left\{E' \overset{0}{\longmapsto} _\right\} & [E']{:=}E & \left\{E' \overset{0}{\longmapsto} E\right\} \\ \{E' \rightharpoonup E\} & x{:=}[E'] & \{E' \rightharpoonup E \wedge x = E\} \ (x \text{ not free in } E, E') \\ \{\mathbf{emp}\} & x{:=}\mathrm{new}(E) & \left\{x \overset{0}{\longmapsto} E\right\} \\ \left\{E' \overset{0}{\longmapsto} _\right\} & \mathrm{dispose}\, E' & \{\mathbf{emp}\} \end{array} \tag{15}$$

In the readers and writers program the two binary semaphores have invariants

$$\begin{array}{l} \mathit{write}\text{: if } \mathit{write} = 0 \text{ then } \mathbf{emp} \text{ else } z \overset{0}{\longmapsto} _ \text{ fi} \\ \mathit{read}\text{: if } \mathit{read} = 0 \vee \mathit{count} = 0 \text{ then } \mathbf{emp} \text{ else } z \overset{\mathit{count}}{\longmapsto} _ \text{ fi} \end{array} \tag{16}$$

Using O'Hearn's reading of semaphores, this says that, for example, passing P(*write*) releases a complete permission into the program, and in order to pass a V(*write*) the program must provide the same permission, which is locked up into the semaphore. The *read* invariant is more complicated, but has the same kind of effect. (We would like also to specify that the *read* semaphore owns the *count* variable, bit we don't yet have the logic to say so.) Then a proof that the readers prologue releases a read permission into the surrounding program goes as in figure 3; the epilogue reverses the action (proof not shown, but obviously similar to the prologue case).

This is all highly satisfactory. One oddity is that it isn't possible to write pre- and post-conditions for operations on the *read* semaphore without mentioning the *count* variable, which is notionally owned by the semaphore and locked up when $read = 1$. This problem will become clearer in the next example.

Suppose we have a network switch which has a read thread for every input port, and a write thread for every output port. Suppose the read thread is an infinite loop which acquires a buffer, fills it with packet contents from the network, then passes it to an address-resolution thread which somehow computes an output port, then passes the buffer to the write thread at that port which transmits the packet onwards and then disposes the buffer. There is no need for the read thread to account for its buffers; clearly each thread has **emp** as a loop invariant. This is the *single-casting* case, and it's covered by O'Hearn's notion of ownership transfer between threads.

Now suppose that a packet can contain several addresses. In this *multi-casting* case, the address-resolution thread may have to pass the same information to more than one output thread. Copying the buffer contents would be inefficient (which matters in the context of packet processing!) and so it's normal to send the same pointer to as many threads as necessary. In order to avoid space leaks, programmers use *permission counting* to record the number of permissions that have been issued. For keeping count accurately between concurrent threads we must use a counting semaphore. Each output thread can 'dispose' the buffer as before, but what really happens is that the semaphore counts down and, when it reaches zero, the buffer can be disposed.

The various threads are easily programmed:

```
while true do
   x := new();
   fill(x); enqueue(x, addressresolverQ)
od
```

```
while true do
  i := 0; x := dequeue(addressresolverQ);
  while i < addresses(x) do
     if i + 1 ≠ addresses(x) then split x fi;
     enqueue(x, addrQ(addr_i(x))); i+ := 1
  od
```

[3] Tentative notational proposal; suggestions welcome.

$\{\mathbf{emp}\}$
$\mathrm{P}(\mathit{read});$
$\quad\{\text{if } \mathit{count} = 0 \text{ then } \mathbf{emp} \text{ else } z \overset{\mathit{count}}{\longmapsto} _\}$
$\quad \mathit{count}{+} := 1;$
$\quad\{\text{if } \mathit{count} - 1 = 0 \text{ then } \mathbf{emp} \text{ else } z \overset{\mathit{count}-1}{\longmapsto} _\}$
$\quad \text{if } \mathit{count} = 1 \text{ then } \{\mathbf{emp}\}\ \mathrm{P}(\mathit{write})\ \{z \overset{0}{\longmapsto} _\}$
$\quad\quad\quad\quad \text{else} \quad\quad\quad\quad\quad\quad\quad\quad \{z \overset{\mathit{count}-1}{\longmapsto} _\};$
$\quad\{z \overset{\mathit{count}-1}{\longmapsto} _\} \therefore \{z \overset{\mathit{count}}{\longmapsto} _ \star z \rightharpoonup _\}$
$\mathrm{V}(\mathit{read});$
$\{z \rightharpoonup _\}$

Figure 3. Release of resource by the readers prologue

while true do
$\quad x := \mathit{dequeue}(\mathit{port}_j Q);$
$\quad \mathit{empty}(x);\ \mathrm{free}\, x$
od

The split$x$ operation can be implemented by $V(\mathit{count})$, since all it has to do is to increment a count of readers. The invariant of the semaphore is then if $\mathit{count} = 0$ then **emp** else $x \overset{\mathit{count}+1}{\longmapsto} _$ fi. When the semaphore is 0, the split user must provide a full-access permission which the semaphore will lock away, releasing in return two read permissions. When the semaphore is non-zero a read permission will apparently be split in two, though of course the trick is done via the permission locked inside the semaphore.

The free$x$ operation can't simply be a semaphore. It has to be a critical region, something like

$\mathit{count}{-} := 1;$
$\text{if } \mathit{count} = 0 \text{ then dispose}(x) \text{ fi}$

It's easy to see that a semaphore can be created each time we call new; less easy to see how permission to access that semaphore can be described; harder still to see how to de-allocate the semaphore when the buffer disappears. But it all looks as if it can be done.

Clearly, the precondition for $\mathrm{V}(\mathit{count})$ involves *count*: sometimes you must provide a $\overset{0}{\longmapsto}$ permission, sometimes a $\rightharpoonup$ permission. That's an interesting problem!

6.1 Model of counting permissions

Amazingly, all you need is to pair each value in the heap with a number. A positive number n corresponds to a $\overset{n}{\longmapsto}$ (or, ambiguously but harmlessly so, a $\overset{n+1}{\longmapsto}$ plus a $\rightharpoonup$ ... and so on); a negative number $-n$ corresponds to n $\rightharpoonup$ read permissions.

7 Conclusion

The notion of permissions has opened up a universe of possibilities in separation logic. There is much work still to do, though, most especially in dealing with stack variables as resources.

8 Acknowledgements

John Boyland put us on to the notion of reading $\mapsto$ as a permission, and pointed out laconically that $\frac{1}{2} + \frac{1}{2} = 1$. Matthew Parkinson has already noticed [www.cl.cam.ac.uk/ mjp41/shortnote.pdf] that our proposals don't consider recursive predicates, has fixed the problem and proposed a wonderful way to reconcile fractional and counting permissions. Watch this space, too!

9 Additional Authors

10 References

[1] R. Bornat, C. Calcagno, and P. O'Hearn. Local reasoning, separation and aliasing. Submitted to the SPACE Workshop, 2004.

[2] 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.

[3] S. Brookes. Semantics and Separation Logic for Parallel Programs with Shared Mutable Data. work in progress, July 2003.

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

[5] P. O'Hearn. Notes on conditional critical regions in spatial pointer logic. unpublished, Aug. 2001.

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