Ownership and permissions in Separation logic

Richard Bornat

School of Computing Science, Middlesex University

5th December 2003

Outline

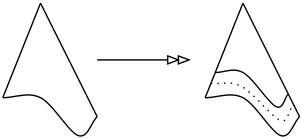
Some Problems
- Trees, DAGs and graphs
- Concurrency and Ownership
- Pipeline processing
- Summary

Possible solutions
- Fractional permissions
- Infinitesimal permissions
- Block permissions
- Permission counting

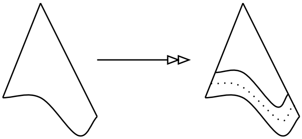
Confessions

Summary

Non-empty binary trees (Bird trees)

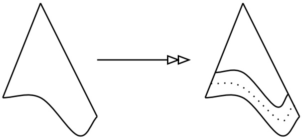


Non-empty binary trees (Bird trees)



$$B ::= \mathsf{Node}\ B\ B \mid \mathsf{Tip}\ \mathrm{val}$$

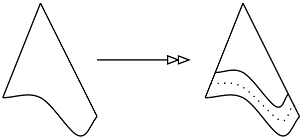
Non-empty binary trees (Bird trees)



$$B ::= \mathsf{Node}\ B\ B \mid \mathsf{Tip}\ \mathrm{val}$$

$$\begin{aligned} \mathit{fringe}\ (\mathsf{Tip}\ \ v) &\mathrel{\hat{=}} \langle v \rangle \\ \mathit{fringe}\ (\mathsf{Node}\ \lambda\ \rho) &\mathrel{\hat{=}} \mathit{fringe}\ \lambda +\!\!+\ \mathit{fringe}\ \rho \end{aligned}$$

Non-empty binary trees (Bird trees)



$$B ::= \mathsf{Node}\ B\ B \mid \mathsf{Tip}\ \mathrm{val}$$

$$\begin{aligned} \mathit{fringe}\ (\mathsf{Tip}\ \ v) &\mathrel{\hat{=}} \langle v \rangle \\ \mathit{fringe}\ (\mathsf{Node}\ \lambda\ \rho) &\mathrel{\hat{=}} \mathit{fringe}\ \lambda \mathbin{+\!\!+} \mathit{fringe}\ \rho \end{aligned}$$

$$\begin{aligned} \mathrm{btree}\ t\ (\mathsf{Tip}\ \ v) &\mathrel{\hat{=}} t \mapsto \mathrm{nil}, v, \_ \\ \mathrm{btree}\ t\ (\mathsf{Node}\ \lambda\ \rho) &\mathrel{\hat{=}} \exists l, r \cdot (t \mapsto l, \_, r \star \mathrm{btree}\ l\ \lambda \star \mathrm{btree}\ r\ \rho) \end{aligned}$$

Fringe-linking a tree – 1

$$
\begin{aligned}
\mathrm{btree}\; t\; (\mathsf{Tip}\;\; v) &\mathrel{\hat{=}} t \mapsto \mathrm{nil}, v, \_ \\
\mathrm{btree}\; t\; (\mathsf{Node}\; \lambda\; \rho) &\mathrel{\hat{=}} \exists l, r \cdot (t \mapsto l, \_, r \star \mathrm{btree}\; l\; \lambda \star \mathrm{btree}\; r\; \rho)
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{lseg}\; y\; y\; \langle\,\rangle &\mathrel{\hat{=}} \mathbf{emp} \\
\mathrm{lseg}\; x\; y\; (\langle v \rangle +\!\!+\; vs) &\mathrel{\hat{=}} \exists x' \cdot (x \mapsto v, x' \star \mathrm{lseg}\; x'\; y\; vs)
\end{aligned}
$$

Fringe-linking a tree – 1

$$\begin{aligned} \mathrm{btree}\ t\ (\mathsf{Tip}\ \ v) &\mathrel{\hat{=}} t \mapsto \mathrm{nil}, v, \_ \\ \mathrm{btree}\ t\ (\mathsf{Node}\ \lambda\ \rho) &\mathrel{\hat{=}} \exists l, r \cdot (t \mapsto l, \_, r \star \mathrm{btree}\ l\ \lambda \star \mathrm{btree}\ r\ \rho) \end{aligned}$$

$$\begin{aligned} \mathrm{lseg}\ y\ y\ \langle\,\rangle &\mathrel{\hat{=}} \mathbf{emp} \\ \mathrm{lseg}\ x\ y\ (\langle v \rangle +\!\!+\ vs) &\mathrel{\hat{=}} \exists x' \cdot (x \mapsto v, x' \star \mathrm{lseg}\ x'\ y\ vs) \end{aligned}$$

$$\begin{aligned} \mathit{fringelink}\ t\ c \mathrel{\hat{=}}\ & \mathrm{if}\ [t] = \mathrm{nil}\ \mathrm{then}\ [t+2] := c;\ t+1 \\ & \mathrm{else}\ \mathit{fringelink}\ [t]\ (\mathit{fringelink}\ [t+2]\ c) \\ & \mathrm{fi} \end{aligned}$$

Fringe-linking a tree – 1

$$\begin{aligned}
\text{btree } t \; (\mathsf{Tip} \;\; v) &\mathrel{\hat{=}} t \mapsto \text{nil}, v, _ \\
\text{btree } t \; (\mathsf{Node} \; \lambda \; \rho) &\mathrel{\hat{=}} \exists l, r \cdot (t \mapsto l, _, r \star \text{btree } l \; \lambda \star \text{btree } r \; \rho)
\end{aligned}$$

$$\begin{aligned}
\text{lseg } y \, y \; \langle\,\rangle &\mathrel{\hat{=}} \mathbf{emp} \\
\text{lseg } x \, y \; (\langle v \rangle +\!\!+ \; vs) &\mathrel{\hat{=}} \exists x' \cdot (x \mapsto v, x' \star \text{lseg } x' \; y \; vs)
\end{aligned}$$

$$\begin{aligned}
\mathit{fringelink} \; t \; c \mathrel{\hat{=}} \; & \text{if } [t] = \text{nil then } [t+2] := c; \; t+1 \\
& \text{else } \mathit{fringelink} \; [t] \; (\mathit{fringelink} \; [t+2] \; c) \\
& \text{fi}
\end{aligned}$$

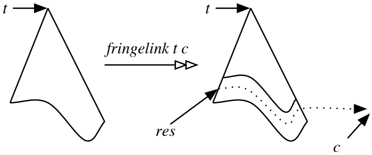
$$\begin{aligned}
&\{\text{btree } t \; \tau\} \\
&\quad \mathit{res} := \mathit{fringelink} \; t \; c \\
&\{(\text{lseg } \mathit{res} \; c \; (\mathit{fringe} \; \tau) \star \text{True}) \wedge \text{btree } t \; \tau\}
\end{aligned}$$

Fringe-linking a tree – 2

$$
\begin{array}{l}
\mathit{fringelink}\ t\ c \mathrel{\hat{=}} \text{if } [t] = \text{nil then } [t+2] := c;\ t+1 \\
\qquad\qquad\qquad\quad \text{else } \mathit{fringelink}\ [t]\ (\mathit{fringelink}\ [t+2]\ c) \\
\qquad\qquad\qquad\quad \text{fi}
\end{array}
$$

$$
\begin{array}{l}
\{\text{btree } t\ \tau\} \\
\quad \mathit{res} := \mathit{fringelink}\ t\ c \\
\{(\text{lseg } \mathit{res}\ c\ (\mathit{fringe}\ \tau) \star \text{True}) \wedge \text{btree } t\ \tau\}
\end{array}
$$

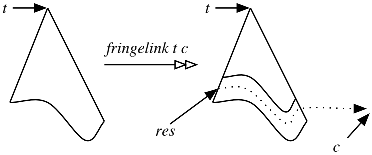
Fringe-linking a tree – 2



$$\begin{array}{l} \mathit{fringelink}\ t\ c \mathrel{\hat{=}} \text{if } [t] = \text{nil then } [t+2] := c;\ t+1 \\ \qquad\qquad\qquad \text{else } \mathit{fringelink}\ [t]\ (\mathit{fringelink}\ [t+2]\ c) \\ \qquad\qquad\qquad \text{fi} \end{array}$$

$$\begin{array}{l} \{\text{btree } t\ \tau\} \\ \quad \mathit{res} := \mathit{fringelink}\ t\ c \\ \{(\text{lseg } \mathit{res}\ c\ (\mathit{fringe}\ \tau) \star \text{True}) \wedge \text{btree } t\ \tau\} \end{array}$$

Fringe-linking a tree – 2

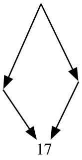


$$\begin{aligned} \mathit{fringelink}\ t\ c \mathrel{\hat{=}} {} & \text{if } [t] = \text{nil then } [t+2] := c;\ t+1 \\ & \text{else } \mathit{fringelink}\ [t]\ (\mathit{fringelink}\ [t+2]\ c) \\ & \text{fi} \end{aligned}$$

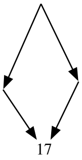
$$\begin{array}{l} \{\text{btree } t\ \tau\} \\ \quad \mathit{res} := \mathit{fringelink}\ t\ c \\ \{(\text{lseg } \mathit{res}\ c\ (\mathit{fringe}\ \tau) \star \text{True}) \wedge \text{btree } t\ \tau\} \end{array}$$

$$\begin{aligned} \text{lseg } y\ y\ \langle\,\rangle &\mathrel{\hat{=}} \mathbf{emp} \\ \text{lseg } x\ y\ (\langle v \rangle +\!\!+\ vs) &\mathrel{\hat{=}} \exists x' \cdot (x \mapsto v, x' \star \text{lseg } x'\ y\ vs) \end{aligned}$$

Directed Acyclic Graphs (DAGs) – 1

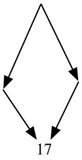


Directed Acyclic Graphs (DAGs) – 1



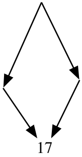
- We'd like to describe a DAG-heap in the same sort of way as we describe a tree-heap (root, left subDAG, right subDAG).

Directed Acyclic Graphs (DAGs) – 1



- We'd like to describe a DAG-heap in the same sort of way as we describe a tree-heap (root, left subDAG, right subDAG).
- But DAGs have sharing, so subDAGs have dangling pointers.

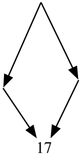
Directed Acyclic Graphs (DAGs) – 1



- We'd like to describe a DAG-heap in the same sort of way as we describe a tree-heap (root, left subDAG, right subDAG).
- But DAGs have sharing, so subDAGs have dangling pointers.

$$D ::= \ \mathsf{Empty} \mid \mathsf{Tip}\ \mathrm{int} \mid \mathsf{Node}\ D\ D \mid \mathsf{Ptr}\ \mathrm{var} \mid \mathrm{let}\ \mathrm{var} = D\ \mathrm{in}\ D$$

Directed Acyclic Graphs (DAGs) – 1

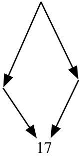


- We'd like to describe a DAG-heap in the same sort of way as we describe a tree-heap (root, left subDAG, right subDAG).
- But DAGs have sharing, so subDAGs have dangling pointers.

$$D ::= \; \mathsf{Empty} \mid \mathsf{Tip}\ \mathrm{int} \mid \mathsf{Node}\ D\ D \mid \mathsf{Ptr}\ \mathrm{var} \mid \mathrm{let}\ \mathrm{var} = D\ \mathrm{in}\ D$$

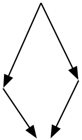
$$\begin{array}{l} \mathrm{let}\ c = \mathsf{Tip}\ 17\ \mathrm{in}\ \mathsf{Node}\ (\mathsf{Node}\ \mathsf{Empty}\ (\mathsf{Ptr}\ c)) \\ \qquad\qquad\qquad\qquad\qquad (\mathsf{Node}\ (\mathsf{Ptr}\ c)\ \mathsf{Empty}) \end{array}$$

Directed Acyclic Graphs (DAGs) – 2



$$\text{let } c = \mathsf{Tip}\ 17 \text{ in } \mathsf{Node}\ (\mathsf{Node}\ \mathsf{Empty}\ (\mathsf{Ptr}\ c))\ (\mathsf{Node}\ (\mathsf{Ptr}\ c)\ \mathsf{Empty})$$

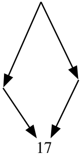
Directed Acyclic Graphs (DAGs) – 2



$$\text{let } c = \mathsf{Tip}\ 17 \text{ in } \mathsf{Node}\ (\mathsf{Node}\ \mathsf{Empty}\ (\mathsf{Ptr}\ c))\ (\mathsf{Node}\ (\mathsf{Ptr}\ c)\ \mathsf{Empty})$$

$$\begin{aligned}
\text{lidag nil } \mathsf{Empty}\ U &\mathrel{\hat{=}} \mathbf{emp} \\
\text{lidag } d\ (\mathsf{Tip}\ \alpha)\ U &\mathrel{\hat{=}} d \mapsto 0, \alpha \\
\text{lidag } d\ (\mathsf{Node}\ \lambda\ \rho)\ U &\mathrel{\hat{=}} \exists l, r \cdot \begin{pmatrix} d \mapsto 1, l, r \star \text{lidag } l\ \lambda\ U \star \\ \text{lidag } r\ \rho\ U \end{pmatrix} \\
\text{lidag } d\ (\mathsf{Ptr}\ x)\ U &\mathrel{\hat{=}} U\ x = d \wedge \mathbf{emp} \\
\text{lidag } d\ (\text{let } x = \delta \text{ in } \delta')\ U &\mathrel{\hat{=}} \exists d' \cdot \begin{pmatrix} \text{lidag } d'\ \delta\ U \star \\ \text{lidag } d\ \delta'\ (U \oplus (x : d')) \end{pmatrix}
\end{aligned}$$

Directed Acyclic Graphs (DAGs) – 2



$$\text{let } c = \mathsf{Tip}\ 17 \text{ in } \mathsf{Node}\ (\mathsf{Node}\ \mathsf{Empty}\ (\mathsf{Ptr}\ c))\ (\mathsf{Node}\ (\mathsf{Ptr}\ c)\ \mathsf{Empty})$$

$$\begin{aligned}
\text{lidag nil } \mathsf{Empty}\ U &\mathrel{\hat{=}} \mathbf{emp} \\
\text{lidag } d\ (\mathsf{Tip}\ \alpha)\ U &\mathrel{\hat{=}} d \mapsto 0, \alpha \\
\text{lidag } d\ (\mathsf{Node}\ \lambda\ \rho)\ U &\mathrel{\hat{=}} \exists l, r \cdot \begin{pmatrix} d \mapsto 1, l, r \star \text{lidag } l\ \lambda\ U \star \\ \text{lidag } r\ \rho\ U \end{pmatrix} \\
\text{lidag } d\ (\mathsf{Ptr}\ x)\ U &\mathrel{\hat{=}} U\ x = d \wedge \mathbf{emp} \\
\text{lidag } d\ (\text{let } x = \delta \text{ in } \delta')\ U &\mathrel{\hat{=}} \exists d' \cdot \begin{pmatrix} \text{lidag } d'\ \delta\ U \star \\ \text{lidag } d\ \delta'\ (U \oplus (x : d')) \end{pmatrix}
\end{aligned}$$

... provided that x occurs free in δ' ...

Directed Acyclic Graphs (DAGs) – 3

But the algorithm doesn't find the sharing *and then* do the copying! Instead it uses a 'forwarding function'.

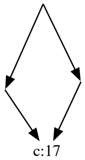
Directed Acyclic Graphs (DAGs) – 3

But the algorithm doesn't find the sharing *and then* do the copying! Instead it uses a 'forwarding function'.

$$
\begin{array}{ll}
\mathit{copydag}\ d\ f \ \hat{=} & \mathrm{if}\ d = \mathrm{nil\ then\ nil}, f \\
& \mathrm{elsf}\ d \in \mathrm{dom} f\ \mathrm{then}\ f\ d, f \\
& \mathrm{elsf}\ d.\mathit{tag} = 0\ \mathrm{then} \\
& \quad d' := \mathit{new}(0, d.\mathit{val});\ d', f \oplus (d : d') \\
& \mathrm{else} \\
& \quad l, f' := \mathit{copydag}\ d.\mathit{left}\ f; \\
& \quad r, f'' := \mathit{copydag}\ d.\mathit{right}\ f'; \\
& \quad d' := \mathit{new}(1, l, r); \\
& \quad d', f'' \oplus (d : d') \\
& \mathrm{fi}
\end{array}
$$

Directed Acyclic Graphs (DAGs) – 4

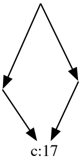
A description readable left-to-right:



$$\mathsf{Node}\ (\mathsf{Node\ Empty}\ (c : \mathsf{Tip}\ 17))$$
$$(\mathsf{Node}\ (\mathsf{Ptr}\ c)\ \mathsf{Empty})$$

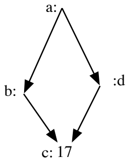
Directed Acyclic Graphs (DAGs) – 4

A description readable left-to-right:



$$\begin{array}{l}\mathsf{Node}\ (\mathsf{Node\ Empty}\ (c : \mathsf{Tip}\ 17))\\ \qquad\quad (\mathsf{Node}\ (\mathsf{Ptr}\ c)\ \mathsf{Empty})\end{array}$$

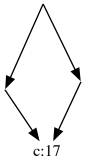
A description in which *every* element is labelled:



$$\begin{array}{l}a : \mathsf{Node}\ (b : \mathsf{Node\ Empty}\ (c : \mathsf{Tip}\ 17))\\ \qquad\qquad\quad (d : \mathsf{Node}\ (\mathsf{Ptr}\ c)\ \mathsf{Empty})\end{array}$$

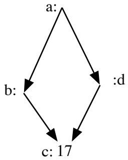
Directed Acyclic Graphs (DAGs) – 4

A description readable left-to-right:



$$\mathsf{Node}\ (\mathsf{Node\ Empty}\ (c : \mathsf{Tip}\ 17))\ (\mathsf{Node}\ (\mathsf{Ptr}\ c)\ \mathsf{Empty})$$

A description in which *every* element is labelled:



$$a : \mathsf{Node}\ (b : \mathsf{Node\ Empty}\ (c : \mathsf{Tip}\ 17))\ (d : \mathsf{Node}\ (\mathsf{Ptr}\ c)\ \mathsf{Empty})$$

$$D ::= \mathsf{Empty} \mid \mathsf{Ptr}\ \mathrm{lab} \mid \mathrm{lab} : \mathsf{Tip}\ \mathrm{int} \mid \mathrm{lab} : \mathsf{Node}\ D\ D$$

Directed Acyclic Graphs (DAGs) – 5

$$D ::= \mathsf{Empty} \mid \mathsf{Ptr}\ \mathrm{lab} \mid \mathrm{lab} : \mathsf{Tip}\ \mathrm{int} \mid \mathrm{lab} : \mathsf{Node}\ D\ D$$

Directed Acyclic Graphs (DAGs) – 5

$$D ::= \mathsf{Empty} \mid \mathsf{Ptr}\ \mathrm{lab} \mid \mathrm{lab} : \mathsf{Tip}\ \mathrm{int} \mid \mathrm{lab} : \mathsf{Node}\ D\ D$$

We need *input* environment U and *output* environment V ($= U \oplus$ internals of δ):

$$\begin{aligned}
\mathrm{pdag}\ \mathrm{nil}\ \mathsf{Empty}\ U\ U &\hat{=} \mathbf{emp} \\
\mathrm{pdag}\ d\ (\mathsf{Ptr}\ x)\ U\ U &\hat{=} U\ x = d \wedge \mathbf{emp} \\
\mathrm{pdag}\ d\ (x : \mathsf{Tip}\ \alpha)\ U\ V &\hat{=} d \mapsto 0, \alpha \wedge V = U \oplus (x : d) \\
\mathrm{pdag}\ d\ (x : \mathsf{Node}\ \lambda\ \rho)\ U\ V &\hat{=} \exists l, r, U', V' \cdot \begin{pmatrix} d \mapsto 1, l, r\ \star \\ \mathrm{pdag}\ l\ \lambda\ U\ U'\ \star \\ \mathrm{pdag}\ r\ \rho\ U'\ V' \wedge \\ V = V' \oplus (x : d) \end{pmatrix}
\end{aligned}$$

Directed Acyclic Graphs (DAGs) – 5

$$D ::= \mathsf{Empty} \mid \mathsf{Ptr}\ \mathrm{lab} \mid \mathrm{lab} : \mathsf{Tip}\ \mathrm{int} \mid \mathrm{lab} : \mathsf{Node}\ D\ D$$

We need *input* environment U and *output* environment V ($= U \oplus$ internals of δ):

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

- This is fine for *closed* examples (U empty).

Directed Acyclic Graphs (DAGs) – 5

$$D ::= \mathsf{Empty} \mid \mathsf{Ptr}\ \mathrm{lab} \mid \mathrm{lab} : \mathsf{Tip}\ \mathrm{int} \mid \mathrm{lab} : \mathsf{Node}\ D\ D$$

We need *input* environment U and *output* environment V ($= U \oplus$ internals of δ):

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

- This is fine for *closed* examples (U empty).
- And examples without errors like multiple declarations.

Directed Acyclic Graphs (DAGs) – 6

We would like to prove

$$\{\mathrm{pdag}\ d\ \delta\ U\ V \wedge \mathrm{ran}\, U = \mathrm{dom}\, f\}$$
$$d', f' := \mathit{copydag}\ d\ f$$
$$\{\mathrm{pdag}\ d\ \delta\ U\ V \star \mathrm{pdag}\ d'\ \delta\ (f \bullet U)\ (f' \bullet V) \wedge \mathrm{ran}\, V = \mathrm{dom}\, f'\}$$

Directed Acyclic Graphs (DAGs) – 6

We would like to prove

$$\{\mathrm{pdag}\ d\ \delta\ U\ V \land \mathrm{ran}\, U = \mathrm{dom}\, f\}$$
$$d', f' := \mathit{copydag}\ d\ f$$
$$\{\mathrm{pdag}\ d\ \delta\ U\ V \star \mathrm{pdag}\ d'\ \delta\ (f \bullet U)\ (f' \bullet V) \land \mathrm{ran}\, V = \mathrm{dom}\, f'\}$$

- – but the inductive step fails! We need to know that $\mathrm{dom}\, f$ points at originally-existing structures *elsewhere* in the heap and $\mathrm{ran}\, f$ points at their copies (even more elsewhere).

Directed Acyclic Graphs (DAGs) – 6

We would like to prove

$$\{\mathrm{pdag}\ d\ \delta\ U\ V \wedge \mathrm{ran}\, U = \mathrm{dom}\, f\}$$
$$d', f' := \mathit{copydag}\ \mathsf{d}\ f$$
$$\{\mathrm{pdag}\ d\ \delta\ U\ V \star \mathrm{pdag}\ d'\ \delta\ (f \bullet U)\ (f' \bullet V) \wedge \mathrm{ran}\, V = \mathrm{dom}\, f'\}$$

- – but the inductive step fails! We need to know that dom f points at originally-existing structures *elsewhere* in the heap and ran f points at their copies (even more elsewhere).
- We don't want dom f or ran f to be part of the footprint; we don't even want read access to those locations.

Directed Acyclic Graphs (DAGs) – 6

We would like to prove

$$\{\mathrm{pdag}\ d\ \delta\ U\ V \wedge \mathrm{ran}\, U = \mathrm{dom}\, f\}$$
$$d', f' := \mathit{copydag}\ d\ f$$
$$\{\mathrm{pdag}\ d\ \delta\ U\ V \star \mathrm{pdag}\ d'\ \delta\ (f \bullet U)\ (f' \bullet V) \wedge \mathrm{ran}\, V = \mathrm{dom}\, f'\}$$

- – but the inductive step fails! We need to know that $\mathrm{dom}\, f$ points at originally-existing structures *elsewhere* in the heap and $\mathrm{ran}\, f$ points at their copies (even more elsewhere).
- We don't want $\mathrm{dom}\, f$ or $\mathrm{ran}\, f$ to be part of the footprint; we don't even want read access to those locations.
- Must we fudge this example?

Concurrency and Ownership

Concurrency and Ownership

- Separation logic deals with pointer safety (no dereferencing nil or a disposed pointer) and space leaks.

Concurrency and Ownership

- Separation logic deals with pointer safety (no dereferencing nil or a disposed pointer) and space leaks.
- In concurrent programs we are also worried about *race conditions*: one thread writing a shared variable, others reading or writing as well.

Concurrency and Ownership

- Separation logic deals with pointer safety (no dereferencing nil or a disposed pointer) and space leaks.
- In concurrent programs we are also worried about *race conditions*: one thread writing a shared variable, others reading or writing as well.
- Since Dijkstra, we know that race conditions are avoided by read/write *private* variables, read-only *shared* variables, and communication via shared read/write variables in mutually-exclusive code sections.

Concurrency and Ownership

- Separation logic deals with pointer safety (no dereferencing nil or a disposed pointer) and space leaks.
- In concurrent programs we are also worried about *race conditions*: one thread writing a shared variable, others reading or writing as well.
- Since Dijkstra, we know that race conditions are avoided by read/write *private* variables, read-only *shared* variables, and communication via shared read/write variables in mutually-exclusive code sections.
- Can we share *locations* as well as variables?

Ownership transfer (O'Hearn)

Ownership transfer (O'Hearn)

$$\text{Resource } r : \text{Vars } \mathit{full}, b;$$

$$\left(\begin{array}{l|l} x := \text{new}(); & \text{with } r \text{ when } \mathit{full} \text{ do} \\ \text{with } r \text{ when } \neg\mathit{full} \text{ do} & \quad y := b; \\ \quad b := x; & \quad \mathit{full} := \text{false} \\ \quad \mathit{full} := \text{true} & \text{od}; \\ \text{od} & \text{dispose } y \end{array}\right)$$

Ownership transfer (O'Hearn)

$$\begin{array}{l} \text{Resource } r : \text{Vars } \mathit{full}, b; \\ \qquad \text{Invariant } (\mathit{full} \wedge b \mapsto \_) \vee (\neg\mathit{full} \wedge \mathbf{emp}) \end{array}$$

$$\left(\begin{array}{l|l} x := \text{new}(); & \text{with } r \text{ when } \mathit{full} \text{ do} \\ \text{with } r \text{ when } \neg\mathit{full} \text{ do} & \quad y := b; \\ \quad b := x; & \quad \mathit{full} := \text{false} \\ \quad \mathit{full} := \text{true} & \text{od}; \\ \text{od} & \text{dispose } y \end{array}\right)$$

Ownership transfer (O'Hearn)

$$\text{Resource } r : \text{Vars } \mathit{full}, b; \quad \text{Invariant } (\mathit{full} \land b \mapsto \_) \lor (\neg \mathit{full} \land \mathbf{emp})$$

$$\left(\begin{array}{l|l}
\{\mathbf{emp}\} & \\
x := \text{new}(); & \text{with } r \text{ when } \mathit{full} \text{ do} \\
\{x \mapsto _\} & \\
\text{with } r \text{ when } \neg\mathit{full} \text{ do} & \quad y := b; \\
\quad \{\neg\mathit{full} \land \mathbf{emp} \star x \mapsto _\} & \\
\quad b := x; & \quad \mathit{full} := \text{false} \\
\quad \{\neg\mathit{full} \land \mathbf{emp} \star x \mapsto _ \land b = x\} & \\
\quad \mathit{full} := \text{true} & \text{od}; \\
\quad \{\mathit{full} \land b \mapsto _ \star \mathbf{emp}\} & \\
\text{od} & \text{dispose } y \\
\{\mathbf{emp}\} &
\end{array}\right)$$

Ownership transfer (O'Hearn)

$$\text{Resource } r : \text{Vars } \mathit{full}, b; \quad \text{Invariant } (\mathit{full} \wedge b \mapsto \_) \vee (\neg \mathit{full} \wedge \mathbf{emp})$$

$$\left(\begin{array}{l} \{\mathbf{emp}\} \\ x := \text{new}(); \\ \{x \mapsto \_\} \\ \text{with } r \text{ when } \neg \mathit{full} \text{ do} \\ \quad \{\neg \mathit{full} \wedge \mathbf{emp} \star x \mapsto \_\} \\ \quad b := x; \\ \quad \{\neg \mathit{full} \wedge \mathbf{emp} \star x \mapsto \_ \wedge b = x\} \\ \quad \mathit{full} := \text{true} \\ \quad \{\mathit{full} \wedge b \mapsto \_ \star \mathbf{emp}\} \\ \text{od} \\ \{\mathbf{emp}\} \end{array} \middle\| \begin{array}{l} \{\mathbf{emp}\} \\ \text{with } r \text{ when } \mathit{full} \text{ do} \\ \quad \{\mathit{full} \wedge b \mapsto \_ \star \mathbf{emp}\} \\ \quad y := b; \\ \quad \{\mathit{full} \wedge b \mapsto \_ \star \mathbf{emp} \wedge y = b\} \\ \quad \mathit{full} := \text{false} \\ \quad \{\neg \mathit{full} \wedge \mathbf{emp} \star y \mapsto \_\} \\ \text{od}; \\ \{y \mapsto \_\} \\ \text{dispose } y \\ \{\mathbf{emp}\} \end{array} \right)$$

Ownership transfer (O'Hearn) – 2

- So: can we share locations between threads?

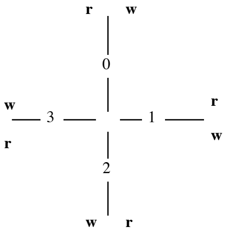
Ownership transfer (O'Hearn) – 2

- So: can we share locations between threads?
- Brookes's semantics of O'Hearn's proposal suggests we should be able to.

Ownership transfer (O'Hearn) – 2

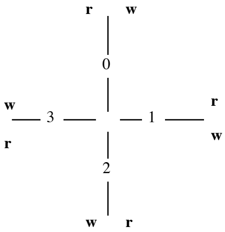
- So: can we share locations between threads?
- Brookes's semantics of O'Hearn's proposal suggests we should be able to.
- But the logic doesn't deal with read-only locations, so far.

Packet switching (singlecast)



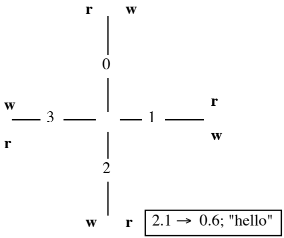
- Imagine a multi-port ethernet switch which has a read and write thread at each port.

Packet switching (singlecast)



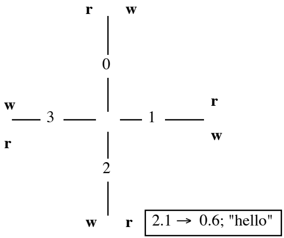
- Imagine a multi-port ethernet switch which has a read and write thread at each port.
- A packet arriving at a port is stored in a buffer created by the read thread.

Packet switching (singlecast)



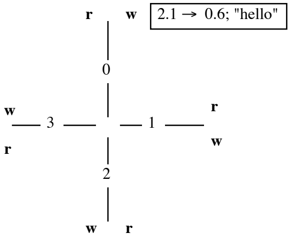
- Imagine a multi-port ethernet switch which has a read and write thread at each port.
- A packet arriving at a port is stored in a buffer created by the read thread.

Packet switching (singlecast)



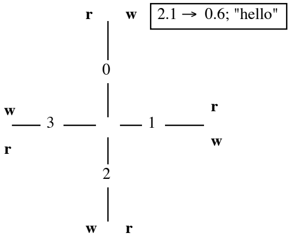
- Imagine a multi-port ethernet switch which has a read and write thread at each port.
- A packet arriving at a port is stored in a buffer created by the read thread.
- Ownership is transferred to the relevant write thread ...

Packet switching (singlecast)



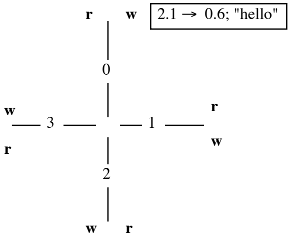
- Imagine a multi-port ethernet switch which has a read and write thread at each port.
- A packet arriving at a port is stored in a buffer created by the read thread.
- Ownership is transferred to the relevant write thread ...

Packet switching (singlecast)



- Imagine a multi-port ethernet switch which has a read and write thread at each port.
- A packet arriving at a port is stored in a buffer created by the read thread.
- Ownership is transferred to the relevant write thread ...
- the data is transmitted ...

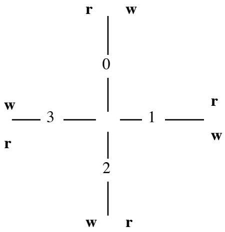
Packet switching (singlecast)



- Imagine a multi-port ethernet switch which has a read and write thread at each port.
- A packet arriving at a port is stored in a buffer created by the read thread.

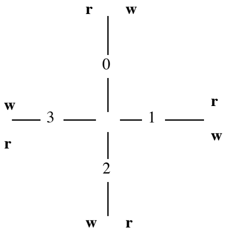
- Ownership is transferred to the relevant write thread ...
- the data is transmitted ...
- and the buffer is disposed by the write thread.

Packet switching (singlecast)



- Imagine a multi-port ethernet switch which has a read and write thread at each port.
- A packet arriving at a port is stored in a buffer created by the read thread.
- Ownership is transferred to the relevant write thread ...
- the data is transmitted ...
- and the buffer is disposed by the write thread.

Packet switching (singlecast)

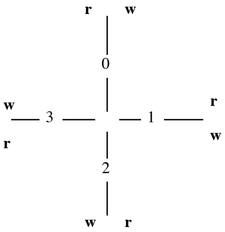


- Imagine a multi-port ethernet switch which has a read and write thread at each port.
- A packet arriving at a port is stored in a buffer created by the read thread.

- Ownership is transferred to the relevant write thread ...
- the data is transmitted ...
- and the buffer is disposed by the write thread.
- Perfect!

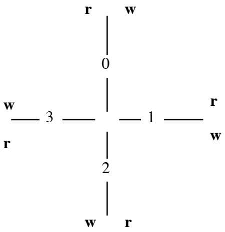
Packet switching (multicast)

- Suppose we have solved the problem of sharing ...



Packet switching (multicast)

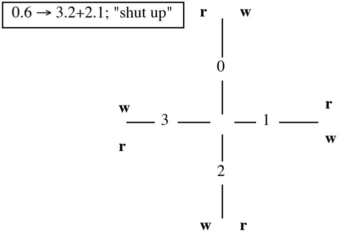
- Suppose we have solved the problem of sharing ...



- A packet arrives with two addresses ...

Packet switching (multicast)

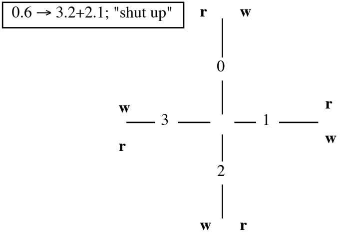
- Suppose we have solved the problem of sharing ...



- A packet arrives with two addresses ...

Packet switching (multicast)

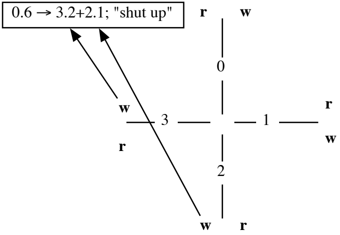
- Suppose we have solved the problem of sharing ...



- A packet arrives with two addresses ...
- and is shared by two write threads.

Packet switching (multicast)

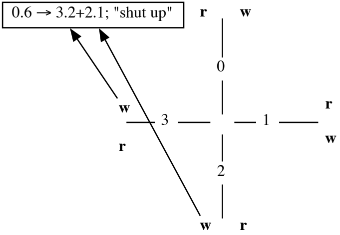
- Suppose we have solved the problem of sharing ...



- A packet arrives with two addresses ...
- and is shared by two write threads.

Packet switching (multicast)

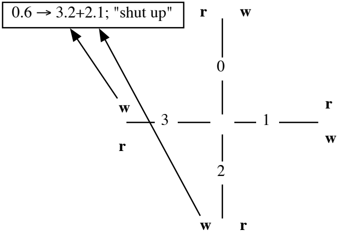
- Suppose we have solved the problem of sharing ...



- A packet arrives with two addresses ...
- and is shared by two write threads.
- But how and when is it disposed?

Packet switching (multicast)

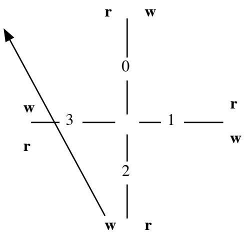
- Suppose we have solved the problem of sharing ...



- A packet arrives with two addresses ...
- and is shared by two write threads.
- But how and when is it disposed?
- Certainly not by the first write thread to finish ...

Packet switching (multicast)

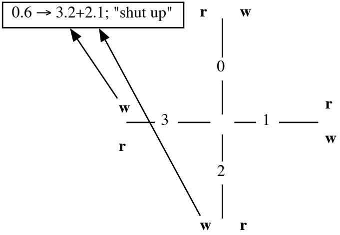
- Suppose we have solved the problem of sharing ...



- A packet arrives with two addresses ...
- and is shared by two write threads.
- But how and when is it disposed?
- Certainly not by the first write thread to finish ...

Packet switching (multicast)

- Suppose we have solved the problem of sharing ...

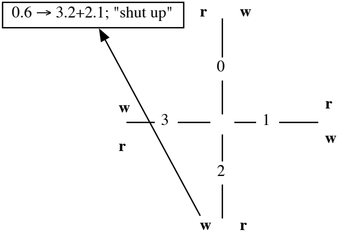


- A packet arrives with two addresses ...
- and is shared by two write threads.
- But how and when is it disposed?
- Certainly not by the first write thread to finish ...

- the read thread might wait (expensively) for them both to signal ...

Packet switching (multicast)

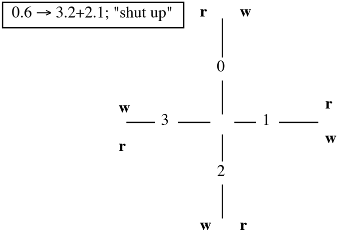
- Suppose we have solved the problem of sharing ...



- A packet arrives with two addresses ...
- and is shared by two write threads.
- But how and when is it disposed?
- Certainly not by the first write thread to finish ...
- the read thread might wait (expensively) for them both to signal ...

Packet switching (multicast)

- Suppose we have solved the problem of sharing ...

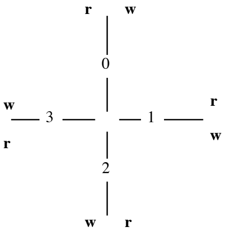


- A packet arrives with two addresses ...
- and is shared by two write threads.
- But how and when is it disposed?
- Certainly not by the first write thread to finish ...

- the read thread might wait (expensively) for them both to signal ...

Packet switching (multicast)

- Suppose we have solved the problem of sharing ...

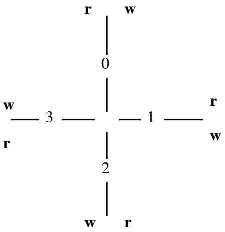


- A packet arrives with two addresses ...
- and is shared by two write threads.
- But how and when is it disposed?
- Certainly not by the first write thread to finish ...

- the read thread might wait (expensively) for them both to signal ...

Packet switching (multicast)

- Suppose we have solved the problem of sharing ...



- A packet arrives with two addresses ...
- and is shared by two write threads.
- But how and when is it disposed?
- Certainly not by the first write thread to finish ...

- the read thread might wait (expensively) for them both to signal ...
- In practice, programs use *permission counting* to deal with this problem.

Problems summarised

Problems summarised

- existence outside footprint;

Problems summarised

- existence outside footprint;
- shared locations;

Problems summarised

- existence outside footprint;
- shared locations;
- permission counting;

Problems summarised

- existence outside footprint;
- shared locations;
- permission counting;
- and some realism in new and dispose (malloc and free deal in buffers, not cells).

A concurrent example

Here is a simple example of a program without a race condition:

$$
\begin{array}{l}
x := \mathrm{new}(); [x] := 1; \\
\left(i := [x] + 1 \,\|\, j := [x] + 2\right); \\
k := i + j; \mathrm{dispose}\, x
\end{array}
$$

A concurrent example

Here is a simple example of a program without a race condition:

$$\begin{array}{l} x := \mathrm{new}(); [x] := 1; \\ \left(i := [x] + 1 \,\|\, j := [x] + 2\right); \\ k := i + j; \mathrm{dispose}\, x \end{array}$$

And here is one with two races (one for i, several for $[x]$):

$$\begin{array}{l} x := \mathrm{new}(); [x] := 1; \\ \left(i := [x] + 1; \mathrm{dispose}\, x \,\|\, [x] := 2; i := [x] + 2\right) \end{array}$$

Fractional permissions

$$x := \mathrm{new}(); [x] := 1; \\ \left(i := [x] + 1 \,\middle\|\, j := [x] + 2\right); \\ k := i + j; \mathrm{dispose}\, x$$

$$x := \mathrm{new}(); [x] := 1; \\ \left(\begin{array}{l} i := [x] + 1; \\ \mathrm{dispose}\, x \end{array} \middle\| \begin{array}{l} [x] := 2; \\ i := [x] + 2 \end{array}\right)$$

- John Boyland explained these programs using the notion of *fractional permissions*.

Fractional permissions

$$\begin{array}{l} x := \mathrm{new}(); [x] := 1; \\ \left(i := [x] + 1 \,\|\, j := [x] + 2\right); \\ k := i + j; \mathrm{dispose}\, x \end{array} \qquad \begin{array}{l} x := \mathrm{new}(); [x] := 1; \\ \left(\begin{array}{l} i := [x] + 1; \\ \mathrm{dispose}\, x \end{array} \middle\| \begin{array}{l} [x] := 2; \\ i := [x] + 2 \end{array}\right) \end{array}$$

- John Boyland explained these programs using the notion of *fractional permissions*.
- An *entire* permission (equivalent to separation logic's $\mapsto$) permits dispose, write and read actions.

Fractional permissions

$$x := \mathrm{new}(); [x] := 1; \\ \big(i := [x] + 1 \,\|\, j := [x] + 2\big)\,; \\ k := i + j; \mathrm{dispose}\, x$$

$$x := \mathrm{new}(); [x] := 1; \\ \left(\begin{array}{l} i := [x] + 1; \\ \mathrm{dispose}\, x \end{array} \middle\| \begin{array}{l} [x] := 2; \\ i := [x] + 2 \end{array}\right)$$

- John Boyland explained these programs using the notion of *fractional permissions*.
- An *entire* permission (equivalent to separation logic's $\mapsto$) permits dispose, write and read actions.
- A *fractional* permission (new to separation logic) permits read access only.

Fractional permissions

$$\begin{array}{l} x := \mathrm{new}(); [x] := 1; \\ \left(i := [x] + 1 \,\|\, j := [x] + 2\right); \\ k := i + j; \mathrm{dispose}\, x \end{array} \qquad \begin{array}{l} x := \mathrm{new}(); [x] := 1; \\ \left(\begin{array}{l} i := [x] + 1; \\ \mathrm{dispose}\, x \end{array} \middle\| \begin{array}{l} [x] := 2; \\ i := [x] + 2 \end{array} \right) \end{array}$$

- John Boyland explained these programs using the notion of *fractional permissions*.
- An *entire* permission (equivalent to separation logic's $\mapsto$) permits dispose, write and read actions.
- A *fractional* permission (new to separation logic) permits read access only.
- Entire permissions can be split into fractions; fractions into smaller fractions;

Fractional permissions

$$x := \mathrm{new}(); [x] := 1; \\ \left(i := [x] + 1 \,\|\, j := [x] + 2\right); \\ k := i + j; \mathrm{dispose}\, x$$

$$x := \mathrm{new}(); [x] := 1; \\ \left(\begin{array}{l} i := [x] + 1; \\ \mathrm{dispose}\, x \end{array} \middle\| \begin{array}{l} [x] := 2; \\ i := [x] + 2 \end{array}\right)$$

- John Boyland explained these programs using the notion of *fractional permissions*.
- An *entire* permission (equivalent to separation logic's $\mapsto$) permits dispose, write and read actions.
- A *fractional* permission (new to separation logic) permits read access only.
- Entire permissions can be split into fractions; fractions into smaller fractions;
- ... and the parts can be *reassembled* arithmetically.

Fractional permissions in separation logic

We propose, following Boyland, some axioms for separation logic:

Fractional permissions in separation logic

We propose, following Boyland, some axioms for separation logic:

$$E \underset{z}{\longmapsto} E' \;\rightarrow\; 0 < z \leq 1$$
$$E \underset{z}{\longmapsto} E' \star E \underset{z'}{\longmapsto} E'' \Longleftrightarrow E' = E'' \wedge E \underset{z+z'}{\longmapsto} E'$$

Fractional permissions in separation logic

We propose, following Boyland, some axioms for separation logic:

$$E \underset{z}{\mapsto} E' \;\rightarrow\; 0 < z \leq 1$$
$$E \underset{z}{\mapsto} E' \star E \underset{z'}{\mapsto} E'' \iff E' = E'' \wedge E \underset{z+z'}{\longmapsto} E'$$

$$\{\mathbf{emp}\}\; x := \mathrm{new}()\; \{x \underset{1}{\mapsto} \_\}$$
$$\{E \underset{1}{\mapsto} \_\}\;\; \mathrm{dispose}\, E \;\; \{\mathbf{emp}\}$$

Fractional permissions in separation logic

We propose, following Boyland, some axioms for separation logic:

$$E \xmapsto[z]{} E' \quad\rightarrow\quad 0 < z \leq 1$$
$$E \xmapsto[z]{} E' \star E \xmapsto[z']{} E'' \iff E' = E'' \wedge E \xmapsto[z+z']{} E'$$

$$\{\mathbf{emp}\}\ x := \mathrm{new}()\ \{x \xmapsto[1]{} \_\}$$
$$\{E \xmapsto[1]{} \_\}\ \ \mathrm{dispose}\, E\ \ \{\mathbf{emp}\}$$

$$\{R^x_E\}\quad x{:=}E \quad \{R\}$$
$$\{x \xmapsto[1]{} \_\}\ [x]{:=}E \quad \{x \xmapsto[1]{} E\}$$
$$\{E' \xmapsto[z]{} E\}\quad x{:=}[E']\ \{E' \xmapsto[z]{} E \wedge x = E'\}\ (x \text{ not free in } E, E')$$

Fractional permissions rule! – 1

$$x := \mathrm{new}();$$

$$[x] := 1;$$

$$\left(i := [x] + 1 \quad \middle\| \quad j := [x] + 2 \right);$$

$$k := i + j;$$

$$\mathrm{dispose}\, x$$

Fractional permissions rule! – 1

$$\{\mathbf{emp}\}$$
$$x := \mathrm{new}();$$
$$\{x \underset{1}{\longmapsto} \_\}$$
$$[x] := 1;$$
$$\{x \underset{1}{\longmapsto} 1\}$$
$$\left(i := [x] + 1 \quad \Big\| \quad j := [x] + 2 \right);$$
$$k := i + j;$$
$$\mathrm{dispose}\, x$$

Fractional permissions rule! – 1

$\{\mathbf{emp}\}$

$x := \mathrm{new}();$

$\{x \xmapsto[1]{} \_\}$

$[x] := 1;$

$\{x \xmapsto[1]{} 1\} \therefore \{x \xmapsto[0.5]{} 1 \star x \xmapsto[0.5]{} 1\}$

$$\left(i := [x] + 1 \quad \Big\| \quad j := [x] + 2 \right);$$

$k := i + j;$

$\mathrm{dispose}\, x$

Fractional permissions rule! – 1

$\{\mathbf{emp}\}$

$x := \text{new}();$

$\{x \underset{1}{\longmapsto} \_\}$

$[x] := 1;$

$\{x \underset{1}{\longmapsto} 1\} \therefore \{x \underset{0.5}{\longmapsto} 1 \star x \underset{0.5}{\longmapsto} 1\}$

$$\left(\begin{array}{l} \{x \underset{0.5}{\longmapsto} 1\} \\ i := [x] + 1 \\ \{x \underset{0.5}{\longmapsto} 1 \wedge i = 2\} \end{array} \middle\| \begin{array}{l} \{x \underset{0.5}{\longmapsto} 1\} \\ j := [x] + 2 \\ \{x \underset{0.5}{\longmapsto} 1 \wedge j = 3\} \end{array} \right);$$

$k := i + j;$

$\text{dispose}\, x$

Fractional permissions rule! – 1

$\{\mathbf{emp}\}$

$x := \text{new}();$

$\{x \underset{1}{\longmapsto} \_\}$

$[x] := 1;$

$\{x \underset{1}{\longmapsto} 1\} \therefore \{x \underset{0.5}{\longmapsto} 1 \star x \underset{0.5}{\longmapsto} 1\}$

$$\left(\begin{array}{l|l} \{x \underset{0.5}{\longmapsto} 1\} & \{x \underset{0.5}{\longmapsto} 1\} \\ i := [x] + 1 & j := [x] + 2 \\ \{x \underset{0.5}{\longmapsto} 1 \wedge i = 2\} & \{x \underset{0.5}{\longmapsto} 1 \wedge j = 3\} \end{array}\right);$$

$\{(x \underset{0.5}{\longmapsto} 1 \wedge i = 2) \star (x \underset{0.5}{\longmapsto} 1 \wedge j = 3)\}$

$k := i + j;$

dispose x

Fractional permissions rule! – 1

$\{\mathbf{emp}\}$

$x := \mathrm{new}();$

$\{x \underset{1}{\longmapsto} \_\}$

$[x] := 1;$

$\{x \underset{1}{\longmapsto} 1\} \therefore \{x \underset{0.5}{\longmapsto} 1 \star x \underset{0.5}{\longmapsto} 1\}$

$$\left(\begin{array}{l} \{x \underset{0.5}{\longmapsto} 1\} \\ i := [x] + 1 \\ \{x \underset{0.5}{\longmapsto} 1 \wedge i = 2\} \end{array} \middle\| \begin{array}{l} \{x \underset{0.5}{\longmapsto} 1\} \\ j := [x] + 2 \\ \{x \underset{0.5}{\longmapsto} 1 \wedge j = 3\} \end{array}\right);$$

$\{(x \underset{0.5}{\longmapsto} 1 \wedge i = 2) \star (x \underset{0.5}{\longmapsto} 1 \wedge j = 3)\} \therefore \{x \underset{1}{\longmapsto} 1 \wedge i = 2 \wedge j = 3\}$

$k := i + j;$

$\mathrm{dispose}\, x$

Fractional permissions rule! – 1

$\{\mathbf{emp}\}$

$x := \mathrm{new}();$

$\{x \underset{1}{\longmapsto} \_\}$

$[x] := 1;$

$\{x \underset{1}{\longmapsto} 1\} \therefore \{x \underset{0.5}{\longmapsto} 1 \star x \underset{0.5}{\longmapsto} 1\}$

$$\left(\begin{array}{l} \{x \underset{0.5}{\longmapsto} 1\} \\ i := [x] + 1 \\ \{x \underset{0.5}{\longmapsto} 1 \wedge i = 2\} \end{array} \middle\| \begin{array}{l} \{x \underset{0.5}{\longmapsto} 1\} \\ j := [x] + 2 \\ \{x \underset{0.5}{\longmapsto} 1 \wedge j = 3\} \end{array}\right);$$

$\{(x \underset{0.5}{\longmapsto} 1 \wedge i = 2) \star (x \underset{0.5}{\longmapsto} 1 \wedge j = 3)\} \therefore \{x \underset{1}{\longmapsto} 1 \wedge i = 2 \wedge j = 3\}$

$k := i + j;$

$\{x \underset{1}{\longmapsto} 1 \wedge i = 2 \wedge j = 3 \wedge k = 5\}$

$\mathrm{dispose}\, x$

$\{\mathbf{emp} \wedge i = 2 \wedge j = 3 \wedge k = 5\}$

Fractional permissions rule! – 2

$$x := \mathrm{new}();$$

$$[x] := 1;$$

$$\left(\begin{array}{l} i := [x] + 1; \\ \\ \mathrm{dispose}\, x \end{array} \quad \middle\| \quad \begin{array}{l} [x] := 2; \\ \\ j := [x] + 2 \end{array} \right)$$

Fractional permissions rule! – 2

$$\{\mathbf{emp}\}$$
$$x := \mathrm{new}();$$
$$\{x \xmapsto[1]{} \_\}$$
$$[x] := 1;$$
$$\{x \xmapsto[1]{} 1\} \therefore \{x \xmapsto[0.5]{} 1 \star x \xmapsto[0.5]{} 1\}$$
$$\left(\begin{array}{l\|l} i := [x] + 1; & [x] := 2; \\ \\ \mathrm{dispose}\, x & j := [x] + 2 \end{array}\right)$$

Fractional permissions rule! – 2

$$\{\mathbf{emp}\}$$
$$x := \mathrm{new}();$$
$$\{x \xmapsto[1]{} \_\}$$
$$[x] := 1;$$
$$\{x \xmapsto[1]{} 1\} \therefore \{x \xmapsto[0.5]{} 1 \star x \xmapsto[0.5]{} 1\}$$
$$\left(\begin{array}{l|l} \{x \xmapsto[0.5]{} 1\} & \\ i := [x] + 1; & [x] := 2; \\ \{x \xmapsto[0.5]{} 1 \wedge i = 2\} & \\ \mathrm{dispose}\, x & j := [x] + 2 \\ \{??\} & \end{array}\right)$$

Fractional permissions rule! – 2

$\{\mathbf{emp}\}$
$x := \mathrm{new}();$
$\{x \xmapsto[1]{} \_\}$
$[x] := 1;$
$\{x \xmapsto[1]{} 1\} \therefore \{x \xmapsto[0.5]{} 1 \star x \xmapsto[0.5]{} 1\}$

$$\left(\begin{array}{l|l} \{x \xmapsto[0.5]{} 1\} & \{x \xmapsto[0.5]{} 1\} \\ i := [x] + 1; & [x] := 2; \\ \{x \xmapsto[0.5]{} 1 \wedge i = 2\} & \{??\} \\ \mathrm{dispose}\, x & j := [x] + 2 \\ \{??\} & \{??\} \end{array}\right)$$

$\{??\}$

Existence indicators – 1

- The Boyland-permission axioms *completely solve* the problem of sharing ('passivity').

Existence indicators – 1

- The Boyland-permission axioms *completely solve* the problem of sharing (‘passivity’).
- But they equate dispose and write permission,

Existence indicators – 1

- The Boyland-permission axioms *completely solve* the problem of sharing (‘passivity’).
- But they equate dispose and write permission,
- and they don’t solve the problem of existence outside the footprint (see $copydag$ and pdag).

Existence indicators – 1

- The Boyland-permission axioms *completely solve* the problem of sharing ('passivity').
- But they equate dispose and write permission,
- and they don't solve the problem of existence outside the footprint (see $copydag$ and pdag).
- Suppose we split an entire permission into one which is large enough to do read and write, and another which is too small to do *anything* ...

Existence indicators – 2

- ι – iota – is an infinitesimal, smaller than any fraction.

Existence indicators – 2

- ι – iota – is an infinitesimal, smaller than any fraction.
- $E \underset{\iota}{\longmapsto}$ – note no E' – says 'E points somewhere, but we don't know what it points to'.

Existence indicators – 2

- ι – iota – is an infinitesimal, smaller than any fraction.
- $E \underset{\iota}{\longmapsto}$ – note no E' – says ‘E points somewhere, but we don’t know what it points to’.

$$x \underset{\iota}{\longmapsto} \;\star\; x \underset{\iota'}{\longmapsto} \iff x \underset{\iota+\iota'}{\longmapsto}$$
$$x \underset{\iota}{\longmapsto} \;\star\; x \underset{z}{\longmapsto} E \iff x \underset{z+\iota}{\longmapsto} E$$

Existence indicators – 2

- ι – iota – is an infinitesimal, smaller than any fraction.
- $E \underset{\iota}{\longmapsto}$ – note no E' – says 'E points somewhere, but we don't know what it points to'.

$$x \underset{\iota}{\longmapsto} \star\, x \underset{\iota'}{\longmapsto} \iff x \underset{\iota+\iota'}{\longmapsto}$$
$$x \underset{\iota}{\longmapsto} \star\, x \underset{z}{\longmapsto} E \iff x \underset{z+\iota}{\longmapsto} E$$

$$\{x \underset{1-\iota}{\longmapsto} \_\}\ [x] := E\ \{x \underset{1-\iota}{\longmapsto} E\}$$

Existence indicators – 2

- ι – iota – is an infinitesimal, smaller than any fraction.
- $E \underset{\iota}{\mapsto}$ – note no E' – says 'E points somewhere, but we don't know what it points to'.

$$x \underset{\iota}{\mapsto} \star x \underset{\iota'}{\mapsto} \iff x \underset{\iota+\iota'}{\longmapsto}$$
$$x \underset{\iota}{\mapsto} \star x \underset{z}{\mapsto} E \iff x \underset{z+\iota}{\longmapsto} E$$

$$\{x \underset{1-\iota}{\longmapsto} \_\} \; [x] := E \; \{x \underset{1-\iota}{\longmapsto} E\}$$

- dispose still needs an entire permission, so if you have an ι permission, your partners can't dispose what they have.

Existence indicators – 3

Now we can have a non-cyclic pdag :

Existence indicators – 3

Now we can have a non-cyclic pdag :

$$
\begin{aligned}
\mathrm{pdag}\ \mathrm{nil}\ \mathsf{Empty}\ U\ U &\mathrel{\hat{=}} \mathbf{emp} \\
\mathrm{pdag}\ d\ (\mathsf{Ptr}\ x)\ U\ U &\mathrel{\hat{=}} U\ x = d \wedge \mathbf{emp} \\
\mathrm{pdag}\ d\ (x : \mathsf{Tip}\ \alpha)\ U\ V &\mathrel{\hat{=}} \left(\begin{array}{l} \{d, d+1\} \cap \mathrm{ran}\, U = \emptyset \wedge \\ d \mapsto 0, \alpha \wedge V = U \oplus (x : d) \end{array}\right) \\
\mathrm{pdag}\ d\ (x : \mathsf{Node}\ \lambda\ \rho)\ U\ V &\mathrel{\hat{=}} \exists l, r, U', V' \cdot \\
&\qquad \left(\begin{array}{l} \{d, d+1, d+2\} \cap \mathrm{ran}\, U = \emptyset \wedge \\ d \mapsto 1, l, r \star \\ \mathrm{pdag}\ l\ \lambda\ U\ U' \star \\ \mathrm{pdag}\ r\ \rho\ U'\ V' \wedge \\ V = V' \oplus (x : d) \end{array}\right)
\end{aligned}
$$

Existence indicators – 4

We can say the right thing about DAGs at last:

Existence indicators – 4

We can say the right thing about DAGs at last:

$$\{\mathrm{pdag}\ d\ (x : \delta)\ U\ V \qquad\qquad\qquad\qquad \}$$
$$d' := \mathrm{new}(1, d, d)$$
$$\left\{ \begin{array}{l} \mathrm{pdag}\ d'\ (y : \mathsf{Node}\ (x : \delta)\ (\mathsf{Ptr}\ x))\ U\ (V \oplus (y : d'))\ \star \\ \\ \end{array} \right\}$$

Existence indicators – 4

We can say the right thing about DAGs at last:

$$\{\mathrm{pdag}\; d\; (x : \delta)\; U\; V \star \forall_\star z \in \mathrm{ran}\, U \cdot z \underset{\iota}{\longmapsto}\}$$
$$d' := \mathrm{new}(1, d, d)$$
$$\left\{\begin{array}{l}\mathrm{pdag}\; d'\; (y : \mathsf{Node}\; (x : \delta)\; (\mathsf{Ptr}\; x))\; U\; (V \oplus (y : d')) \star \\ \quad \forall_\star z \in \mathrm{ran}\, U \cdot z \underset{\iota}{\longmapsto}\end{array}\right\}$$

Block permissions – 1

- C's malloc and free (like Pascal's new and dispose) allocate/de-allocate buffers all at once.

Block permissions – 1

- C's malloc and free (like Pascal's new and dispose) allocate/de-allocate buffers all at once.
- In particular, C's free disposes of an entire buffer when given only a pointer to its first cell.

Block permissions – 1

- C's malloc and free (like Pascal's new and dispose) allocate/de-allocate buffers all at once.
- In particular, C's free disposes of an entire buffer when given only a pointer to its first cell.
- Suppose that every cell permission carries a 'ghostly outline' of the buffer it came from.

Block permissions – 1

- C's malloc and free (like Pascal's new and dispose) allocate/de-allocate buffers all at once.
- In particular, C's free disposes of an entire buffer when given only a pointer to its first cell.
- Suppose that every cell permission carries a 'ghostly outline' of the buffer it came from.
- We write $E \xmapsto[z]{i,n} E'$ to say that E points to the ith cell of an n-element buffer (block) with (fractional) access permission z and value E'.

Block permissions – 1

- C's malloc and free (like Pascal's new and dispose) allocate/de-allocate buffers all at once.
- In particular, C's free disposes of an entire buffer when given only a pointer to its first cell.
- Suppose that every cell permission carries a 'ghostly outline' of the buffer it came from.
- We write $E \xmapsto[z]{i,n} E'$ to say that E points to the ith cell of an n-element buffer (block) with (fractional) access permission z and value E'.
- Clearly, new gives out 1-permission for a block,

Block permissions – 1

- C's malloc and free (like Pascal's new and dispose) allocate/de-allocate buffers all at once.
- In particular, C's free disposes of an entire buffer when given only a pointer to its first cell.
- Suppose that every cell permission carries a 'ghostly outline' of the buffer it came from.
- We write $E \overset{i,n}{\underset{z}{\longmapsto}} E'$ to say that E points to the ith cell of an n-element buffer (block) with (fractional) access permission z and value E'.
- Clearly, new gives out 1-permission for a block,
- and you can't dispose unless you have 1-permission for the entire block.

Block permissions – 2

Block permissions – 2

$$E \xmapsto[z]{i,n} E' \rightarrow 0 < z \leq 1 \wedge 0 \leq i < n$$

Block permissions – 2

$$E \xmapsto[z]{i,n} E' \rightarrow 0 < z \leq 1 \wedge 0 \leq i < n$$

$$E \xmapsto[z]{i,n} E1, ..., Ej \iff$$
$$E \xmapsto[z]{i,n} E1 \star (E+1) \xmapsto[z]{i+1,n} E2 \star ... \star (E+j-1) \xmapsto[z]{i+j-1,n} Ej$$

Block permissions – 2

$$E \xmapsto[z]{i,n} E' \rightarrow 0 < z \leq 1 \land 0 \leq i < n$$

$$\begin{aligned} E \xmapsto[z]{i,n} E1, \ldots, Ej \iff \\ E \xmapsto[z]{i,n} E1 \star (E+1) \xmapsto[z]{i+1,n} E2 \star \ldots \star (E+j-1) \xmapsto[z]{i+j-1,n} Ej \end{aligned}$$

$$x \xmapsto[z]{i,n} E \star x \xmapsto[z']{i',n'} E' \rightarrow i = i' \land n = n' \land E = E' \land x \xmapsto[z+z']{i,n} E$$

Block permissions – 2

$$E \xmapsto[z]{i,n} E' \rightarrow 0 < z \leq 1 \wedge 0 \leq i < n$$

$$\begin{aligned}
&E \xmapsto[z]{i,n} E1, \ldots, Ej \iff \\
&\quad E \xmapsto[z]{i,n} E1 \star (E+1) \xmapsto[z]{i+1,n} E2 \star \ldots \star (E+j-1) \xmapsto[z]{i+j-1,n} Ej
\end{aligned}$$

$$x \xmapsto[z]{i,n} E \star x \xmapsto[z']{i',n'} E' \rightarrow i = i' \wedge n = n' \wedge E = E' \wedge x \xmapsto[z+z']{i,n} E$$

$$x \xmapsto[z]{i,n} E \star x' \xmapsto[z']{i',n'} E' \wedge x \neq x' \rightarrow \begin{pmatrix} (x - i = x' - i' \wedge n = n') \vee \\ x - i + n \leq x' - i' \vee \\ x' - i' + n' \leq x - i \end{pmatrix}$$

Block permissions – 2

$$E \xmapsto[z]{i,n} E' \rightarrow 0 < z \leq 1 \wedge 0 \leq i < n$$

$$E \xmapsto[z]{i,n} E1, \ldots, Ej \iff$$
$$E \xmapsto[z]{i,n} E1 \star (E+1) \xmapsto[z]{i+1,n} E2 \star \ldots \star (E+j-1) \xmapsto[z]{i+j-1,n} Ej$$

$$x \xmapsto[z]{i,n} E \star x \xmapsto[z']{i',n'} E' \rightarrow i = i' \wedge n = n' \wedge E = E' \wedge x \xmapsto[z+z']{i,n} E$$

$$x \xmapsto[z]{i,n} E \star x' \xmapsto[z']{i',n'} E' \wedge x \neq x' \rightarrow \begin{pmatrix} (x - i = x' - i' \wedge n = n') \vee \\ x - i + n \leq x' - i' \vee \\ x' - i' + n' \leq x - i \end{pmatrix}$$

$$\{\mathbf{emp}\}\ x := \mathrm{new}(E1, \ldots, En)\ \{x \xmapsto[1]{0,n} E1, \ldots, En\}$$
$$\{E \xmapsto[1]{0,n} E1, \ldots, En\}\quad \mathrm{dispose}\, E \quad \{\mathbf{emp}\}$$

The magic of new

- The frame rule – $\{Q\}C\{R\} \implies \{P \star Q\}C\{P \star R\}$ – is the centre of separation logic.

The magic of new

- The frame rule – $\{Q\}C\{R\} \implies \{P \star Q\}C\{P \star R\}$ – is the centre of separation logic.
- (And it has an interesting side-condition, which we shall return to).

The magic of new

- The frame rule – $\{Q\}C\{R\} \Longrightarrow \{P \star Q\}C\{P \star R\}$ – is the centre of separation logic.
- (And it has an interesting side-condition, which we shall return to).
- The axiom for new – $\{\mathbf{emp}\}\ x := new()\ \{x \mapsto \_\}$ – requires new to be magic: it must never assign a value to x which will break the frame rule.

The magic of new

- The frame rule – $\{Q\}C\{R\} \implies \{P \star Q\}C\{P \star R\}$ – is the centre of separation logic.
- (And it has an interesting side-condition, which we shall return to).
- The axiom for new – $\{\mathbf{emp}\}\ x := new()\ \{x \mapsto \_\}$ – requires new to be magic: it must never assign a value to x which will break the frame rule.
- It's only stage magic: new has a pile of stuff; you have a separate pile; it gives you one from its pile on request; dispose takes one from your pile and gives it back to new.

Permission counting – 1

- Suppose that new keeps a hidden count for every cell/block it gives you.

Permission counting – 1

- Suppose that new keeps a hidden count for every cell/block it gives you.
- You aren't allowed to split permissions 'silently' as before, but you can ask to have it done (it increases the permission count).

Permission counting – 1

- Suppose that new keeps a hidden count for every cell/block it gives you.
- You aren't allowed to split permissions 'silently' as before, but you can ask to have it done (it increases the permission count).
- Suppose that dispose will accept a fractional permission. Silently, it decreases the permission count, and reclaims the space iff the count is now zero.

Permission counting – 1

- Suppose that new keeps a hidden count for every cell/block it gives you.
- You aren't allowed to split permissions 'silently' as before, but you can ask to have it done (it increases the permission count).
- Suppose that dispose will accept a fractional permission. Silently, it decreases the permission count, and reclaims the space iff the count is now zero.
- There's a possibility that the fractional permission you are holding is the last fraction left on earth (because other people have disposed their fractions). You should surely be able to ask if this is so!

Permission counting – 1

- Suppose that new keeps a hidden count for every cell/block it gives you.
- You aren't allowed to split permissions 'silently' as before, but you can ask to have it done (it increases the permission count).
- Suppose that dispose will accept a fractional permission. Silently, it decreases the permission count, and reclaims the space iff the count is now zero.
- There's a possibility that the fractional permission you are holding is the last fraction left on earth (because other people have disposed their fractions). You should surely be able to ask if this is so!
- Can we make a logic for this language?

Permission counting – 1

- Suppose that new keeps a hidden count for every cell/block it gives you.
- You aren't allowed to split permissions 'silently' as before, but you can ask to have it done (it increases the permission count).
- Suppose that dispose will accept a fractional permission. Silently, it decreases the permission count, and reclaims the space iff the count is now zero.
- There's a possibility that the fractional permission you are holding is the last fraction left on earth (because other people have disposed their fractions). You should surely be able to ask if this is so!
- Can we make a logic for this language?
- Of course! (We may have to wait for the logicians to agree.)

Permission counting – 2

Permission counting – 2

$$x \underset{z1}{\longmapsto} E1 \star x \underset{z2}{\longmapsto} E2 \star \ldots \star x \underset{zn}{\longmapsto} En \rightarrow$$
$$E1 = E2 = \ldots = En \wedge z1 + z2 + \ldots + zn \leq 1$$

Permission counting – 2

$$x \underset{z1}{\longmapsto} E1 \star x \underset{z2}{\longmapsto} E2 \star \ldots \star x \underset{zn}{\longmapsto} En \rightarrow$$
$$E1 = E2 = \ldots = En \wedge z1 + z2 + \ldots + zn \leq 1$$

$$\left(\begin{array}{l} x \underset{z1}{\longmapsto} E \star x \underset{z2}{\longmapsto} E \star \ldots \star x \underset{zn}{\longmapsto} E \wedge \\ (z1 + z2 + \ldots + zn) \geq (z1' + z2' + \ldots + zn') \end{array} \right) \rightarrow$$
$$x \underset{z1'}{\longmapsto} E \star x \underset{z2'}{\longmapsto} E \star \ldots \star x \underset{zn'}{\longmapsto} E$$

Permission counting – 2

$$x \underset{z1}{\longmapsto} E1 \star x \underset{z2}{\longmapsto} E2 \star \ldots \star x \underset{zn}{\longmapsto} En \rightarrow$$
$$E1 = E2 = \ldots = En \wedge z1 + z2 + \ldots + zn \leq 1$$

$$\left(\begin{array}{l} x \underset{z1}{\longmapsto} E \star x \underset{z2}{\longmapsto} E \star \ldots \star x \underset{zn}{\longmapsto} E \wedge \\ (z1 + z2 + \ldots + zn) \geq (z1' + z2' + \ldots + zn') \end{array}\right) \rightarrow$$
$$x \underset{z1'}{\longmapsto} E \star x \underset{z2'}{\longmapsto} E \star \ldots \star x \underset{zn'}{\longmapsto} E$$

$$\{\mathbf{emp}\}\ x := \mathrm{new}()\ \{x \underset{1}{\mapsto} \_\}$$
$$\{E \underset{z+z'}{\longmapsto} E'\}\quad \mathrm{split}\, E \quad \{E \underset{z}{\mapsto} E' \star E \underset{z'}{\mapsto} E'\}$$
$$\{E \underset{z}{\mapsto} E' \star E \underset{z'}{\mapsto} E'\}\ \mathrm{dispose}\, E\ \{E \underset{z+z'}{\longmapsto} E'\}$$
$$\{E \underset{z}{\mapsto} \_\}\ \mathrm{dispose}\, E\ \{\mathbf{emp}\}$$
$$\{E \underset{z}{\mapsto} E'\}\ b := \mathrm{neo}\, E\ \{(b \wedge E \underset{1}{\mapsto} E') \vee (\neg b \wedge E \underset{z}{\mapsto} E')\}$$

Permission counting – 3

$$x := \mathrm{new}();$$

$$[x] := 1;$$

$$\mathrm{split}\, x;$$

$$\left(\begin{array}{l} i := [x] + 1; \\ \\ \mathrm{dispose}\, x \end{array} \quad \middle\| \quad \begin{array}{l} j := [x] + 2; \\ \\ \mathrm{dispose}\, x \end{array} \right)$$

Permission counting – 3

$$\{\mathbf{emp}\}$$
$$x := \mathrm{new}();$$
$$\{x \underset{1}{\longmapsto} \_\}$$
$$[x] := 1;$$
$$\{x \underset{1}{\longmapsto} 1\}$$
$$\mathrm{split}\, x;$$
$$\{x \underset{0.5}{\longmapsto} 1 \star x \underset{0.5}{\longmapsto} 1\}$$
$$\left(\begin{array}{l} i := [x] + 1; \\ \\ \mathrm{dispose}\, x \end{array} \;\middle\|\; \begin{array}{l} j := [x] + 2; \\ \\ \mathrm{dispose}\, x \end{array}\right)$$

Permission counting – 3

$$\{\mathbf{emp}\}$$
$$x := \mathrm{new}();$$
$$\{x \underset{1}{\longmapsto} \_\}$$
$$[x] := 1;$$
$$\{x \underset{1}{\longmapsto} 1\}$$
$$\mathrm{split}\, x;$$
$$\{x \underset{0.5}{\longmapsto} 1 \star x \underset{0.5}{\longmapsto} 1\}$$
$$\left(\begin{array}{l||l}
\{x \underset{0.5}{\longmapsto} 1\} & \\
i := [x] + 1; & j := [x] + 2; \\
\{x \underset{0.5}{\longmapsto} 1 \wedge i = 2\} & \\
\mathrm{dispose}\, x & \mathrm{dispose}\, x \\
\{\mathbf{emp} \wedge i = 2\} &
\end{array}\right)$$

Permission counting – 3

$$
\begin{array}{l}
\{\mathbf{emp}\} \\
x := \mathrm{new}(); \\
\{x \underset{1}{\longmapsto} \_\} \\
[x] := 1; \\
\{x \underset{1}{\longmapsto} 1\} \\
\mathrm{split}\, x; \\
\{x \underset{0.5}{\longmapsto} 1 \star x \underset{0.5}{\longmapsto} 1\} \\
\left(\begin{array}{l}
\{x \underset{0.5}{\longmapsto} 1\} \\
i := [x] + 1; \\
\{x \underset{0.5}{\longmapsto} 1 \wedge i = 2\} \\
\mathrm{dispose}\, x \\
\{\mathbf{emp} \wedge i = 2\}
\end{array}
\middle\|
\begin{array}{l}
\{x \underset{0.5}{\longmapsto} 1\} \\
j := [x] + 2; \\
\{x \underset{0.5}{\longmapsto} 1 \wedge j = 3\} \\
\mathrm{dispose}\, x \\
\{\mathbf{emp} \wedge j = 3\}
\end{array}\right)
\end{array}
$$

Permission counting – 3

$$\{\mathbf{emp}\}$$
$$x := \text{new}();$$
$$\{x \underset{1}{\longmapsto} \_\}$$
$$[x] := 1;$$
$$\{x \underset{1}{\longmapsto} 1\}$$
$$\text{split}\, x;$$
$$\{x \underset{0.5}{\longmapsto} 1 \star x \underset{0.5}{\longmapsto} 1\}$$
$$\left(\begin{array}{l} \{x \underset{0.5}{\longmapsto} 1\} \\ i := [x] + 1; \\ \{x \underset{0.5}{\longmapsto} 1 \wedge i = 2\} \\ \text{dispose}\, x \\ \{\mathbf{emp} \wedge i = 2\} \end{array} \middle\| \begin{array}{l} \{x \underset{0.5}{\longmapsto} 1\} \\ j := [x] + 2; \\ \{x \underset{0.5}{\longmapsto} 1 \wedge j = 3\} \\ \text{dispose}\, x \\ \{\mathbf{emp} \wedge j = 3\} \end{array}\right)$$
$$\{(\mathbf{emp} \wedge i = 2 \star (\mathbf{emp} \wedge j = 3)\}$$

Permission counting – 3

$$\{\mathbf{emp}\}$$
$$x := \mathrm{new}();$$
$$\{x \underset{1}{\longmapsto} \_\}$$
$$[x] := 1;$$
$$\{x \underset{1}{\longmapsto} 1\}$$
$$\mathrm{split}\, x;$$
$$\{x \underset{0.5}{\longmapsto} 1 \star x \underset{0.5}{\longmapsto} 1\}$$
$$\left(\begin{array}{l|l} \{x \underset{0.5}{\longmapsto} 1\} & \{x \underset{0.5}{\longmapsto} 1\} \\ i := [x] + 1; & j := [x] + 2; \\ \{x \underset{0.5}{\longmapsto} 1 \wedge i = 2\} & \{x \underset{0.5}{\longmapsto} 1 \wedge j = 3\} \\ \mathrm{dispose}\, x & \mathrm{dispose}\, x \\ \{\mathbf{emp} \wedge i = 2\} & \{\mathbf{emp} \wedge j = 3\} \end{array}\right)$$
$$\{(\mathbf{emp} \wedge i = 2 \star (\mathbf{emp} \wedge j = 3)\} \therefore \{\mathbf{emp} \wedge i = 2 \wedge j = 3\}$$

Permission counting – 4

$x := \mathrm{new}();$

$\mathrm{split}\, x;$

$\mathrm{dispose}\, x;$

$[x] = 0;$

$\mathrm{dispose}\, x$

Permission counting – 4

$\{\mathbf{emp}\}$
$x := \mathrm{new}();$
$\{x \xrightarrow[1]{} \_\}$
$\mathrm{split}\, x;$
$\{x \xrightarrow[0.5]{} \_ \star x \xrightarrow[0.5]{} \_\}$
$\mathrm{dispose}\, x;$
$\{x \xrightarrow[1]{} \_\}$
$[x] = 0;$
$\{x \xrightarrow[1]{} 0\}$
$\mathrm{dispose}\, x$
$\{\mathbf{emp}\}$

Permission counting – 5

neo needs global reasoning!!

$x := \text{new}();$

$\text{split}\, x;$

$$\left(\text{dispose}\, x \;\middle\|\; \text{skip} \right);$$

$\text{if } \text{neo}\, x \text{ then } [x] := 0 \text{ else fault fi}$

Permission counting – 5

neo needs global reasoning!!

$$
\begin{array}{l}
\{\mathbf{emp}\} \\
x := \mathrm{new}(); \\
\{x \xmapsto[1]{} \_\} \\
\mathrm{split}\, x; \\
\{x \xmapsto[0.5]{} \_ \star x \xmapsto[0.5]{} \_\} \\
\left(\begin{array}{l} \{x \xmapsto[0.5]{} \_\} \\ \mathrm{dispose}\, x \\ \{\mathbf{emp}\} \end{array} \middle\| \begin{array}{l} \{x \xmapsto[0.5]{} \_\} \\ \mathrm{skip} \\ \{x \xmapsto[0.5]{} \_\} \end{array}\right); \\
\{\mathbf{emp} \star x \xmapsto[0.5]{} \_\} \therefore \{x \xmapsto[0.5]{} \_\} \\
\mathrm{if\ \ neo}\, x \mathrm{\ then\ } [x] := 0 \mathrm{\ else\ fault\ fi} \\
\{??\}
\end{array}
$$

Permission counting – a confession

Permission counting – a confession

- You probably think that neo is a big departure – but it's no more magic than new.

Permission counting – a confession

- You probably think that neo is a big departure – but it's no more magic than new.
- But there is something wrong somewhere. Either 'writing down' of permissions and/or multiple dispose axioms causes an apparent paradox (Hongseok Yang).

Permission counting – a confession

- You probably think that neo is a big departure – but it's no more magic than new.
- But there is something wrong somewhere. Either 'writing down' of permissions and/or multiple dispose axioms causes an apparent paradox (Hongseok Yang).
- Write z instead of $x \underset{z}{\mapsto} 17$, write 0 instead of $\mathbf{emp}$:

$$\dfrac{\{0.5 \star 0.5\}\ \mathrm{dispose}\, x\ \{1\} \qquad \dfrac{\{0.5\}\ \mathrm{dispose}\, x\ \{0\}}{\{0.5 \star \neg 1\}\ \mathrm{dispose}\, x\ \{\neg 1\}}}{\{(0.5 \star 0.5) \wedge (0.5 \star \neg 1)\}\ \mathrm{dispose}\, x\ \{1 \wedge \neg 1\}}$$

- Oh dear!

Permissions for variables – a confession

Permissions for variables – a confession

- The frame rule has a side-condition:

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

Permissions for variables – a confession

- The frame rule has a side-condition:

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

- Boyland deals with permission to access variables as well as locations.

Permissions for variables – a confession

- The frame rule has a side-condition:

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

- Boyland deals with permission to access variables as well as locations.
- Brookes' semantics for ownership transfer needs a logical treatment of permissions for variables, too.

Permissions for variables – a confession

- The frame rule has a side-condition:

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

- Boyland deals with permission to access variables as well as locations.
- Brookes' semantics for ownership transfer needs a logical treatment of permissions for variables, too.
- We don't know how to do it!

Permissions for variables – a confession

- The frame rule has a side-condition:

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

- Boyland deals with permission to access variables as well as locations.
- Brookes' semantics for ownership transfer needs a logical treatment of permissions for variables, too.
- We don't know how to do it!
- - without losing Hoare logic

Permissions for variables – a confession

- The frame rule has a side-condition:

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

- Boyland deals with permission to access variables as well as locations.
- Brookes' semantics for ownership transfer needs a logical treatment of permissions for variables, too.
- We don't know how to do it!
- - without losing Hoare logic
- - and/or needing a garbage-collected 'stack'

Permissions for variables – a confession

- The frame rule has a side-condition:

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

- Boyland deals with permission to access variables as well as locations.
- Brookes' semantics for ownership transfer needs a logical treatment of permissions for variables, too.
- We don't know how to do it!
- - without losing Hoare logic
- - and/or needing a garbage-collected 'stack'
- Oh dear, oh dear!

Summary

Summary

- Fractional permissions are wonderful.

Summary

- Fractional permissions are wonderful.
- Infinitesimal permissions are interesting, and may be wonderful one day.

Summary

- Fractional permissions are wonderful.
- Infinitesimal permissions are interesting, and may be wonderful one day.
- Block permissions are a bit complicated, and need some work.

Summary

- Fractional permissions are wonderful.
- Infinitesimal permissions are interesting, and may be wonderful one day.
- Block permissions are a bit complicated, and need some work.
- I *think* the permission counting idea might be made to work.

Summary

- Fractional permissions are wonderful.
- Infinitesimal permissions are interesting, and may be wonderful one day.
- Block permissions are a bit complicated, and need some work.
- I *think* the permission counting idea might be made to work.
- Local reasoning is still hard.

Summary

- Fractional permissions are wonderful.
- Infinitesimal permissions are interesting, and may be wonderful one day.
- Block permissions are a bit complicated, and need some work.
- I *think* the permission counting idea might be made to work.
- Local reasoning is still hard.
- We must do variable-permissions.

Summary

- Fractional permissions are wonderful.
- Infinitesimal permissions are interesting, and may be wonderful one day.
- Block permissions are a bit complicated, and need some work.
- I *think* the permission counting idea might be made to work.
- Local reasoning is still hard.
- We must do variable-permissions.
- We are nowhere near the edge of this field yet.