Permissions and concurrency: a breakthrough and a Grand Challenge

Richard Bornat (Middlesex, UK)

22nd Oct 2004

Typing

- one of the things that formalism has given us;

Typing

- one of the things that formalism has given us;
- a relatively recent success (only took C in the early 90s);

Typing

- one of the things that formalism has given us;
- a relatively recent success (only took C in the early 90s);
- never a Grand Challenge;

Typing

- one of the things that formalism has given us;
- a relatively recent success (only took C in the early 90s);
- never a Grand Challenge;
- who could live without it? (well ok, low life).

Typing

- one of the things that formalism has given us;
- a relatively recent success (only took C in the early 90s);
- never a Grand Challenge;
- who could live without it? (well ok, low life).
- (why do the low life do without types? what can we do for them?)

A Rough History

Types began with Russell, but we got them from FORTRAN, as hints to a compiler.

A Rough History

Types began with Russell, but we got them from FORTRAN, as hints to a compiler.

COBOL had structural descriptions, not seen as types.

A Rough History

Types began with Russell, but we got them from FORTRAN, as hints to a compiler.

COBOL had structural descriptions, not seen as types.

We’ve tried typeless languages – e.g. BCPL – and run-time typed languages – e.g. Lisp.

A Rough History

Types began with Russell, but we got them from FORTRAN, as hints to a compiler.

COBOL had structural descriptions, not seen as types.

We’ve tried typeless languages – e.g. BCPL – and run-time typed languages – e.g. Lisp.

Types became popular as protection against wild use of pointers (instead of just a pointer, a pointer to an *X* structure) and procedure addresses (argument and result types).

A Rough History

Types began with Russell, but we got them from FORTRAN, as hints to a compiler.

COBOL had structural descriptions, not seen as types.

We've tried typeless languages – e.g. BCPL – and run-time typed languages – e.g. Lisp.

Types became popular as protection against wild use of pointers (instead of just a pointer, a pointer to an *X* structure) and procedure addresses (argument and result types).

Typing is about static, value/state independent properties of names.

A Rough History

Types began with Russell, but we got them from FORTRAN, as hints to a compiler.

COBOL had structural descriptions, not seen as types.

We've tried typeless languages – e.g. BCPL – and run-time typed languages – e.g. Lisp.

Types became popular as protection against wild use of pointers (instead of just a pointer, a pointer to an *X* structure) and procedure addresses (argument and result types).

Typing is about static, value/state independent properties of names.

The world doesn't yet believe in type inference, and the inventor of C++ has never been put on trial.

Types are not enough

Types are not enough

Types do not protect us from:

- dereferencing null pointers;

Types are not enough

Types do not protect us from:

- dereferencing null pointers;
- double free;

Types are not enough

Types do not protect us from:

- dereferencing null pointers;
- double free;
- rogue librarians (what does `System.out.print(s)` do?);

Types are not enough

Types do not protect us from:

- dereferencing null pointers;
- double free;
- rogue librarians (what does `System.out.print(s)` do?);
- the ‘variant record error’;

Types are not enough

Types do not protect us from:

- dereferencing null pointers;
- double free;
- rogue librarians (what does `System.out.print(s)` do?);
- the ‘variant record error’;
- ...

Types are not enough

Types do not protect us from:

- dereferencing null pointers;
- double free;
- rogue librarians (what does `System.out.print(s)` do?);
- the 'variant record error';
- ...

"Well-typed programs do not go wrong": we don't apply a function to the wrong type of value. But the wrong value of the type ...

Types are not enough

Types do not protect us from:

- dereferencing null pointers;
- double free;
- rogue librarians (what does `System.out.print(s)` do?);
- the ‘variant record error’;
- ...

“Well-typed programs do not go wrong”: we don’t apply a function to the wrong type of value. But the wrong value of the type ...

$hd([\,])$, anybody?

A classic resource problem

`malloc`/`new` gives you a pointer to a new-ish buffer (one you don't own at the time of asking). You can do what you like with the pointer.

A classic resource problem

`malloc`/`new` gives you a pointer to a new-ish buffer (one you don't own at the time of asking). You can do what you like with the pointer.

(ISO C says you mustn't do anything naughty, but nobody can tell if you do.)

A classic resource problem

`malloc`/`new` gives you a pointer to a new-ish buffer (one you don’t own at the time of asking). You can do what you like with the pointer.

(ISO C says you mustn’t do anything naughty, but nobody can tell if you do.)

`free`/`dispose`, given a pointer to a buffer you do own, takes the buffer away.

A classic resource problem

`malloc`/`new` gives you a pointer to a new-ish buffer (one you don't own at the time of asking). You can do what you like with the pointer.

(ISO C says you mustn't do anything naughty, but nobody can tell if you do.)

`free`/`dispose`, given a pointer to a buffer you do own, takes the buffer away.

You are left with a pointer (maybe several copies of a pointer) which you are not allowed to use.

A classic resource problem

`malloc`/`new` gives you a pointer to a new-ish buffer (one you don't own at the time of asking). You can do what you like with the pointer.

(ISO C says you mustn't do anything naughty, but nobody can tell if you do.)

`free`/`dispose`, given a pointer to a buffer you do own, takes the buffer away.

You are left with a pointer (maybe several copies of a pointer) which you are not allowed to use.

Vivid analogy: *"An unfrocked priest is capable of celebrating marriage but forbidden to do so"*
(Marek Sergot, talk on logic of permission and belief; see also *Daily Telegraph*, 21/vi/93).

A classic resource problem

`malloc`/`new` gives you a pointer to a new-ish buffer (one you don't own at the time of asking). You can do what you like with the pointer.

(ISO C says you mustn't do anything naughty, but nobody can tell if you do.)

`free`/`dispose`, given a pointer to a buffer you do own, takes the buffer away.

You are left with a pointer (maybe several copies of a pointer) which you are not allowed to use.

Vivid analogy: *"An unfrocked priest is capable of celebrating marriage but forbidden to do so"*
(Marek Sergot, talk on logic of permission and belief; see also *Daily Telegraph*, 21/vi/93).

(BTW, "Ownership types" and/or nulling disposed pointers don't touch this problem.)

The hardest kind of programming?

Concurrency – multi-threading, multi-processing – is notoriously difficult.

The hardest kind of programming?

Concurrency – multi-threading, multi-processing – is notoriously difficult.

40 years ago, Dijkstra invented semaphores, mutual exclusion, critical sections (*Cooperating Sequential Processes*, 1965).

The hardest kind of programming?

Concurrency – multi-threading, multi-processing – is notoriously difficult.

40 years ago, Dijkstra invented semaphores, mutual exclusion, critical sections (*Cooperating Sequential Processes*, 1965).

– also fairness, deadlock, starvation, the Banker’s algorithm, bounded and unbounded buffer, ...

The hardest kind of programming?

Concurrency – multi-threading, multi-processing – is notoriously difficult.

40 years ago, Dijkstra invented semaphores, mutual exclusion, critical sections (*Cooperating Sequential Processes*, 1965).

– also fairness, deadlock, starvation, the Banker's algorithm, bounded and unbounded buffer, ...

"*We have stipulated that processes should be connected loosely; by this we mean that apart from the (rare) moments of explicit intercommunication, the individual processes themselves are to be regarded as completely independent of each other.*"

Separation logic

Separation logic

- Just a bastard child of BI (Pym, O'Hearn).

Separation logic

- Just a bastard child of BI (Pym, O'Hearn).
- $E \mapsto E'$ (points to) is permission to read/write/dispose cell at heap address E with contents E'.

Separation logic

- Just a bastard child of BI (Pym, O'Hearn).
- $E \mapsto E'$ (points to) is permission to read/write/dispose cell at heap address E with contents E'.
- $\mapsto$ can also be read as ownership and/or a heap predicate.

Separation logic

- Just a bastard child of BI (Pym, O'Hearn).
- $E \mapsto E'$ (points to) is permission to read/write/dispose cell at heap address E with contents E'.
- $\mapsto$ can also be read as ownership and/or a heap predicate.
- $\mathbf{emp}$ is no permission.

Separation logic

- Just a bastard child of BI (Pym, O'Hearn).
- $E \mapsto E'$ (points to) is permission to read/write/dispose cell at heap address E with contents E'.
- $\mapsto$ can also be read as ownership and/or a heap predicate.
- $\mathbf{emp}$ is no permission.
- $A \star B$ (star) is separation of resource.

Separation logic

- Just a bastard child of BI (Pym, O'Hearn).
- $E \mapsto E'$ (points to) is permission to read/write/dispose cell at heap address E with contents E'.
- $\mapsto$ can also be read as ownership and/or a heap predicate.
- $\mathbf{emp}$ is no permission.
- $A \star B$ (star) is separation of resource.
- $A \wedge B$ (and) is identity of resource.

Separation logic

- Just a bastard child of BI (Pym, O'Hearn).
- $E \mapsto E'$ (points to) is permission to read/write/dispose cell at heap address E with contents E'.
- $\mapsto$ can also be read as ownership and/or a heap predicate.
- $\mathbf{emp}$ is no permission.
- $A \star B$ (star) is separation of resource.
- $A \wedge B$ (and) is identity of resource.
- $A \wedge (B \star \mathrm{true})$ is all A, partly B.

Framing, hence small axioms

Framing, hence small axioms

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

Framing, hence small axioms

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

$$\begin{array}{rll}
\{R^x_E\} & x{:=}E & \{R\} \\
\{x \mapsto _\} & [x]{:=}E & \{x \mapsto E\} \\
\{E' \mapsto E\} & x{:=}[E'] & \{E' \mapsto E \wedge x = E\} \ (x \text{ not free in } E, E') \\
\{\mathbf{emp}\} & x{:=}\mathrm{new}(E) & \{x \mapsto E\} \\
\{E \mapsto _\} & \mathrm{dispose}\, E & \{\mathbf{emp}\}
\end{array}$$

Concurrency rules

Concurrency rules

$$\frac{\{Q_1\}\ C_1\ \{R_1\} \cdots \{Q_n\}\ C_n\ \{R_n\}}{\{Q_1 \star \cdots \star Q_n\} \quad (C_1 \parallel \cdots \parallel C_n)\{R_1 \star \cdots \star R_n\}} \text{ (non-interference-of-variables)}$$

Concurrency rules

$$\frac{\{Q_1\}\ C_1\ \{R_1\} \cdots \{Q_n\}\ C_n\ \{R_n\}}{\{Q_1 \star \cdots \star Q_n\} \quad (C_1 \parallel \cdots \parallel C_n)\{R_1 \star \cdots \star R_n\}} \ \text{(non-interference-of-variables)}$$

$$\frac{\{(Q \star I_r) \wedge B\}C\{R \star I_r\}}{\{Q\}\text{with } r \text{ when } B \text{ do } C \text{ od}\{R\}}$$

Concurrency rules

$$\frac{\{Q_1\}\,C_1\,\{R_1\}\cdots\{Q_n\}\,C_n\,\{R_n\}}{\{Q_1\star\cdots\star Q_n\}\quad(C_1\parallel\cdots\parallel C_n)\{R_1\star\cdots\star R_n\}}\ \text{(non-interference-of-variables)}$$

$$\frac{\{(Q\star I_r)\wedge B\}C\{R\star I_r\}}{\{Q\}\text{with } r \text{ when } B \text{ do } C \text{ od}\{R\}}$$

- Both proved sound by Brookes.

Concurrency rules

$$\frac{\{Q_1\}\; C_1\; \{R_1\} \cdots \{Q_n\}\; C_n\; \{R_n\}}{\{Q_1 \star \cdots \star Q_n\} \quad (C_1 \parallel \cdots \parallel C_n)\{R_1 \star \cdots \star R_n\}} \text{ (non-interference-of-variables)}$$

$$\frac{\{(Q \star I_r) \wedge B\}C\{R \star I_r\}}{\{Q\}\mathrm{with}\ r\ \mathrm{when}\ B\ \mathrm{do}\ C\ \mathrm{od}\{R\}}$$

- Both proved sound by Brookes.
- A version of the CCR rule covers semaphores, in which C is either $m := m + 1$ or $m := m - 1$.

The ownership trick (O'Hearn 2002)

$$\text{resource-bundle } r : \text{Vars } \mathit{full}, b;\ \mathit{full} := \text{false};$$

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

The ownership trick (O'Hearn 2002)

resource-bundle r : Vars full, b; $\mathit{full} :=$ false;

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

The ownership trick (O'Hearn 2002)

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

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

The ownership trick (O'Hearn 2002)

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

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

The ownership trick (O'Hearn 2002)

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

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

The ownership trick (O'Hearn 2002)

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

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

The ownership trick (O'Hearn 2002)

$$\begin{array}{ll}\text{resource-bundle } r : & \text{Vars } \mathit{full}, b;\ \ \mathit{full} := \text{false};\\ & \text{Invariant } (\mathit{full} \wedge b \mapsto \_) \vee (\neg\mathit{full} \wedge \mathbf{emp})\end{array}$$

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

The ownership trick (O'Hearn 2002)

$$\begin{array}{ll}\text{resource-bundle } r : & \text{Vars } \mathit{full}, b;\ \ \mathit{full} := \text{false};\\ & \text{Invariant } (\mathit{full} \wedge b \mapsto \_) \vee (\neg \mathit{full} \wedge \mathbf{emp})\end{array}$$

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

The ownership trick (O'Hearn 2002)

$$\begin{array}{ll}\text{resource-bundle } r : & \text{Vars } \mathit{full}, b;\ \ \mathit{full} := \text{false}; \\ & \text{Invariant } (\mathit{full} \wedge b \mapsto \_) \vee (\neg \mathit{full} \wedge \mathbf{emp})\end{array}$$

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

The ownership trick (O'Hearn 2002)

$$\text{resource-bundle } r : \text{Vars } \mathit{full}, b;\ \ \mathit{full} := \text{false};$$
$$\text{Invariant } (\mathit{full} \wedge b \mapsto \_) \vee (\neg \mathit{full} \wedge \mathbf{emp})$$

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

The ownership trick (O'Hearn 2002)

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

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

Passivity

Passivity

- Passivity is a property of a program and a resource: the program doesn't change the contents of the resource.

Passivity

- Passivity is a property of a program and a resource: the program doesn't change the contents of the resource.
- We want to specify passivity by specifying a read-only resource.

Passivity

- Passivity is a property of a program and a resource: the program doesn't change the contents of the resource.
- We want to specify passivity by specifying a read-only resource.
- We require that a program, given a read-only resource, cannot change its contents.

Splitting and sharing

Splitting and sharing

- Since Dijkstra, we have known that we can safely share read-only resources.

Splitting and sharing

- Since Dijkstra, we have known that we can safely share read-only resources.
- Total permission $E \mapsto E'$, given by new, allows read/write/dispose.

Splitting and sharing

- Since Dijkstra, we have known that we can safely share read-only resources.
- Total permission $E \mapsto E'$, given by new, allows read/write/dispose.
- Concurrent read permissions must be $(\star)$ separable, because of the concurrency rule.

Accounting

Accounting

- Splitting into multiple read permissions is easy.

Accounting

- Splitting into multiple read permissions is easy.
- To write or dispose we have to know when we have all the read permissions back.

Accounting

- Splitting into multiple read permissions is easy.
- To write or dispose we have to know when we have all the read permissions back.
- A program which doesn't keep account leaks resource.

Boyland's suggestion: $\frac{1}{2} + \frac{1}{2} = 1$

Boyland's suggestion: $\frac{1}{2} + \frac{1}{2} = 1$

- Boyland (Wisconsin) developed a means of permission accounting in disjoint concurrency, dealing with variables and heap locations.

Boyland's suggestion: $\frac{1}{2} + \frac{1}{2} = 1$

- Boyland (Wisconsin) developed a means of permission accounting in disjoint concurrency, dealing with variables and heap locations.
- He associates a number z with each permission: $z = 1$ total; $0 < z < 1$ read-only.

Boyland's suggestion: $\frac{1}{2} + \frac{1}{2} = 1$

- Boyland (Wisconsin) developed a means of permission accounting in disjoint concurrency, dealing with variables and heap locations.
- He associates a number z with each permission: $z = 1$ total; $0 < z < 1$ read-only.
- Fractional permissions are specification-only (cf. types).

Boyland's suggestion: $\frac{1}{2} + \frac{1}{2} = 1$

- Boyland (Wisconsin) developed a means of permission accounting in disjoint concurrency, dealing with variables and heap locations.
- He associates a number z with each permission: $z = 1$ total; $0 < z < 1$ read-only.
- Fractional permissions are specification-only (cf. types).
- In practice the arithmetic is very easy: fractions are simpler to use than (e.g.) sets of binary trees.

Boyland's suggestion: $\frac{1}{2} + \frac{1}{2} = 1$

- Boyland (Wisconsin) developed a means of permission accounting in disjoint concurrency, dealing with variables and heap locations.
- He associates a number z with each permission: $z = 1$ total; $0 < z < 1$ read-only.
- Fractional permissions are specification-only (cf. types).
- In practice the arithmetic is very easy: fractions are simpler to use than (e.g.) sets of binary trees.
- The magnitude of non-integral fractions doesn't matter, except as a matter of accounting.

A fractional model (Calcagno, O'Hearn)

A fractional model (Calcagno, O'Hearn)

- Heaps are partial maps from Nat to (int, fraction). (Previously Nat to int.)

A fractional model (Calcagno, O'Hearn)

- Heaps are partial maps from Nat to (int, fraction). (Previously Nat to int.)
- A simpler model – just read / total permissions – fails to account and doesn't have the frame property.

Proof theory

Proof theory

$$
\begin{aligned}
E \xmapsto[z]{} E' \quad &\Rightarrow \quad 0 < z \leq 1 \\
E \xmapsto[z+z']{} E' \wedge z > 0 \wedge z' > 0 \;&\Longleftrightarrow\; E \xmapsto[z]{} E' \star E \xmapsto[z']{} E'
\end{aligned}
$$

Proof theory

$$E \xmapsto[z]{} E' \quad \Rightarrow \quad 0 < z \leq 1$$

$$E \xmapsto[z+z']{} E' \wedge z > 0 \wedge z' > 0 \iff E \xmapsto[z]{} E' \star E \xmapsto[z']{} E'$$

$$\begin{array}{rll}
\{R^x_E\} & x{:=}E & \{R\} \\
\{E' \xmapsto[1]{} _\} & [E']{:=}E & \{E' \xmapsto[1]{} E\} \\
\{E' \xmapsto[z]{} E\} & x{:=}[E'] & \{E' \xmapsto[z]{} E \wedge x = E\} \text{ (x not free in E, E')} \\
\{\mathbf{emp}\} & x{:=}\mathrm{new}(E) & \{x \xmapsto[1]{} E\} \\
\{E \xmapsto[1]{} _\} & \text{dispose}\, E & \{\mathbf{emp}\}
\end{array}$$

Proof theory

$$E \underset{z}{\mapsto} E' \quad \Rightarrow \quad 0 < z \leq 1$$

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

$$\begin{array}{lll}
\{R^x_E\} & x{:=}E & \{R\} \\
\{E' \underset{1}{\mapsto} _\} & [E']{:=}E & \{E' \underset{1}{\mapsto} E\} \\
\{E' \underset{z}{\mapsto} E\} & x{:=}[E'] & \{E' \underset{z}{\mapsto} E \wedge x = E\} \text{ (x not free in E, E')} \\
\{\mathbf{emp}\} & x{:=}\mathrm{new}(E) & \{x \underset{1}{\mapsto} E\} \\
\{E \underset{1}{\mapsto} _\} & \mathrm{dispose}\, E & \{\mathbf{emp}\}
\end{array}$$

- Not (yet) proved sound by Brookes. (But surely ...)

Proof

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

$[x] := 1;$

$$\left(y := [x] \qquad\qquad \middle\| \; z := [x] + 1 \qquad\qquad \right);$$

$\mathrm{dispose}\, x$
$\{\mathbf{emp} \wedge y = 1 \wedge z = 2\}$

Proof

$\{\mathbf{emp}\}$
$x := \mathrm{new}();$
$\{x \underset{1}{\longmapsto} \_\}$
$[x] := 1;$

$$\left(y := [x] \quad \Bigg\| \quad z := [x] + 1 \right);$$

$\mathrm{dispose}\, x$
$\{\mathbf{emp} \wedge y = 1 \wedge z = 2\}$

Proof

$$\{\mathbf{emp}\}$$
$$x := \mathrm{new}();$$
$$\{x \underset{1}{\longmapsto} \_\}$$
$$[x] := 1;$$
$$\{x \underset{1}{\longmapsto} 1\}$$
$$\left(y := [x] \qquad\qquad \Big\| \; z := [x] + 1 \qquad\qquad \right);$$
$$\mathrm{dispose}\, x$$
$$\{\mathbf{emp} \wedge y = 1 \wedge z = 2\}$$

Proof

$$\{\mathbf{emp}\}$$
$$x := \mathrm{new}();$$
$$\{x \underset{1}{\longmapsto} \_\}$$
$$[x] := 1;$$
$$\{x \underset{1}{\longmapsto} 1\} \therefore \{x \underset{0.5}{\longmapsto} 1 \star x \underset{0.5}{\longmapsto} 1\}$$
$$\left(y := [x] \quad \middle\| \quad z := [x] + 1 \right);$$

$$\mathrm{dispose}\, x$$
$$\{\mathbf{emp} \wedge y = 1 \wedge z = 2\}$$

Proof

$$\{\mathbf{emp}\}$$
$$x := \mathrm{new}();$$
$$\{x \xmapsto[1]{} \_\}$$
$$[x] := 1;$$
$$\{x \xmapsto[1]{} 1\} .\because. \{x \xmapsto[0.5]{} 1 \star x \xmapsto[0.5]{} 1\}$$
$$\left(\begin{array}{l} \{x \xmapsto[0.5]{} 1\} \\ y := [x] \\ \end{array} \middle\| \begin{array}{l} \{x \xmapsto[0.5]{} 1\} \\ z := [x] + 1 \\ \end{array}\right);$$

$$\mathrm{dispose}\, x$$
$$\{\mathbf{emp} \wedge y = 1 \wedge z = 2\}$$

Proof

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

Proof

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

Proof

$$\{\mathbf{emp}\}$$
$$x := \mathrm{new}();$$
$$\{x \xmapsto[1]{} \_\}$$
$$[x] := 1;$$
$$\{x \xmapsto[1]{} 1\} .\because. \{x \xmapsto[0.5]{} 1 \star x \xmapsto[0.5]{} 1\}$$
$$\left(\begin{array}{l} \{x \xmapsto[0.5]{} 1\} \\ y := [x] \\ \{x \xmapsto[0.5]{} 1 \wedge y = 1\} \end{array} \middle\| \begin{array}{l} \{x \xmapsto[0.5]{} 1\} \\ z := [x] + 1 \\ \{x \xmapsto[0.5]{} 1 \wedge z = 2\} \end{array}\right);$$
$$\{(x \xmapsto[0.5]{} 1 \wedge y = 1) \star (x \xmapsto[0.5]{} 1 \wedge z = 2)\}$$
$$\mathrm{dispose}\, x$$
$$\{\mathbf{emp} \wedge y = 1 \wedge z = 2\}$$

Proof

$$
\begin{array}{l}
\{\mathbf{emp}\} \\
x := \mathrm{new}(); \\
\{x \underset{1}{\longmapsto} \_\} \\
[x] := 1; \\
\{x \underset{1}{\longmapsto} 1\} .\!\cdot\!. \{x \underset{0.5}{\longmapsto} 1 \star x \underset{0.5}{\longmapsto} 1\} \\
\left(
\begin{array}{l}
\{x \underset{0.5}{\longmapsto} 1\} \\
y := [x] \\
\{x \underset{0.5}{\longmapsto} 1 \wedge y = 1\}
\end{array}
\middle\|
\begin{array}{l}
\{x \underset{0.5}{\longmapsto} 1\} \\
z := [x] + 1 \\
\{x \underset{0.5}{\longmapsto} 1 \wedge z = 2\}
\end{array}
\right); \\
\{(x \underset{0.5}{\longmapsto} 1 \wedge y = 1) \star (x \underset{0.5}{\longmapsto} 1 \wedge z = 2)\} .\!\cdot\!. \{x \underset{1}{\longmapsto} 1 \wedge y = 1 \wedge z = 2\} \\
\mathrm{dispose}\, x \\
\{\mathbf{emp} \wedge y = 1 \wedge z = 2\}
\end{array}
$$

Proof

$$
\begin{array}{l}
\{\mathbf{emp}\} \\
x := \mathrm{new}(); \\
\{x \xmapsto[1]{} \_\} \\
[x] := 1; \\
\{x \xmapsto[1]{} 1\} .\!\cdot\!. \{x \xmapsto[0.5]{} 1 \star x \xmapsto[0.5]{} 1\} \\
\left(\begin{array}{l||l}
\{x \xmapsto[0.5]{} 1\} & \{x \xmapsto[0.5]{} 1\} \\
y := [x] & z := [x] + 1 \\
\{x \xmapsto[0.5]{} 1 \wedge y = 1\} & \{x \xmapsto[0.5]{} 1 \wedge z = 2\}
\end{array}\right); \\
\{(x \xmapsto[0.5]{} 1 \wedge y = 1) \star (x \xmapsto[0.5]{} 1 \wedge z = 2)\} .\!\cdot\!. \{x \xmapsto[1]{} 1 \wedge y = 1 \wedge z = 2\} \\
\mathrm{dispose}\; x \\
\{\mathbf{emp} \wedge y = 1 \wedge z = 2\}
\end{array}
$$

- That is exactly how hard it is to use fractional permissions.

UnProof

$$
\begin{array}{l}
\{\mathbf{emp}\} \\
x := \mathrm{new}(); \\
\{x \underset{1}{\longmapsto} \_\} \\
[x] := 1; \\
\{x \underset{1}{\longmapsto} 1\} .\!\cdot\!. \{x \underset{0.5}{\longmapsto} 1 \star x \underset{0.5}{\longmapsto} 1\} \\
\left(\begin{array}{l|l}
\{x \underset{0.5}{\longmapsto} 1\} & \{x \underset{0.5}{\longmapsto} 1\} \\
y := [x]; & [x] := 2; \\
 & \\
\mathrm{dispose}\, x & z := [x] + 1 \\
 &
\end{array}\right) \\
 \\
[x] := y + z
\end{array}
$$

UnProof

$$
\begin{array}{l}
\{\mathbf{emp}\} \\
x := \mathrm{new}(); \\
\{x \xmapsto[1]{} \_\} \\
[x] := 1; \\
\{x \xmapsto[1]{} 1\} .\!\!\cdot\!\!. \{x \xmapsto[0.5]{} 1 \star x \xmapsto[0.5]{} 1\} \\
\left(\begin{array}{l|l}
\{x \xmapsto[0.5]{} 1\} & \{x \xmapsto[0.5]{} 1\} \\
y := [x]; & [x] := 2; \\
\{x \xmapsto[0.5]{} 1 \wedge y = 1\} & \\
\mathrm{dispose}\, x & z := [x] + 1 \\
 &
\end{array}\right) \\
\\
[x] := y + z
\end{array}
$$

UnProof

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

UnProof

$$\{\mathbf{emp}\}$$

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

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

$$[x] := 1;$$

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

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

$$[x] := y + z$$

UnProof

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

Passivity and fractions

Termination Monotonicity: if C must terminate normally in h and $h \star h'$ is defined, then C must terminate normally in $h \star h'$.

Passivity and fractions

Termination Monotonicity: if C must terminate normally in h and $h \star h'$ is defined, then C must terminate normally in $h \star h'$.

- We can prove termination monotonicity for all commands in our language.

Passivity and fractions

Termination Monotonicity: if C must terminate normally in h and $h \star h'$ is defined, then C must terminate normally in $h \star h'$.

- We can prove termination monotonicity for all commands in our language.
- Suppose $\{10 \xmapsto[0.5]{} N\} C \{10 \xmapsto[0.5]{} N + 1\}$, and it terminates.

Passivity and fractions

Termination Monotonicity: if C must terminate normally in h and $h \star h'$ is defined, then C must terminate normally in $h \star h'$.

- We can prove termination monotonicity for all commands in our language.
- Suppose $\{10 \xmapsto[0.5]{} N\}C\{10 \xmapsto[0.5]{} N+1\}$, and it terminates.
- Then (frame rule)

$$\dfrac{\begin{array}{c}\vdots\\ \{10 \xmapsto[0.5]{} N\}C\{10 \xmapsto[0.5]{} N+1\}\end{array}}{\{10 \xmapsto[0.5]{} N \star 10 \xmapsto[0.5]{} N\}C\{10 \xmapsto[0.5]{} N \star 10 \xmapsto[0.5]{} N+1\}}$$

Passivity and fractions

Termination Monotonicity: if C must terminate normally in h and $h \star h'$ is defined, then C must terminate normally in $h \star h'$.

- We can prove termination monotonicity for all commands in our language.
- Suppose $\{10 \xmapsto[0.5]{} N\}C\{10 \xmapsto[0.5]{} N+1\}$, and it terminates.
- Then (frame rule)

$$\dfrac{\begin{array}{c}\vdots \\ \{10 \xmapsto[0.5]{} N\}C\{10 \xmapsto[0.5]{} N+1\}\end{array}}{\{10 \xmapsto[0.5]{} N \star 10 \xmapsto[0.5]{} N\}C\{10 \xmapsto[0.5]{} N \star 10 \xmapsto[0.5]{} N+1\}}$$

- – i.e. it won't terminate in $10 \xmapsto[1.0]{} N$.

Passivity and fractions

Termination Monotonicity: if C must terminate normally in h and $h \star h'$ is defined, then C must terminate normally in $h \star h'$.

- We can prove termination monotonicity for all commands in our language.
- Suppose $\{10 \xmapsto[0.5]{} N\}C\{10 \xmapsto[0.5]{} N+1\}$, and it terminates.
- Then (frame rule)

$$\dfrac{\begin{array}{c}\vdots\\ \{10 \xmapsto[0.5]{} N\}C\{10 \xmapsto[0.5]{} N+1\}\end{array}}{\{10 \xmapsto[0.5]{} N \star 10 \xmapsto[0.5]{} N\}C\{10 \xmapsto[0.5]{} N \star 10 \xmapsto[0.5]{} N+1\}}$$

- – i.e. it won't terminate in $10 \xmapsto[1.0]{} N$.
- Therefore C isn't in our language.

Passivity and fractions

Termination Monotonicity: if C must terminate normally in h and $h \star h'$ is defined, then C must terminate normally in $h \star h'$.

- We can prove termination monotonicity for all commands in our language.
- Suppose $\{10 \xmapsto[0.5]{} N\}C\{10 \xmapsto[0.5]{} N+1\}$, and it terminates.
- Then (frame rule)

$$\frac{\begin{array}{c}\vdots\\ \{10 \xmapsto[0.5]{} N\}C\{10 \xmapsto[0.5]{} N+1\}\end{array}}{\{10 \xmapsto[0.5]{} N \star 10 \xmapsto[0.5]{} N\}C\{10 \xmapsto[0.5]{} N \star 10 \xmapsto[0.5]{} N+1\}}$$

- – i.e. it won't terminate in $10 \xmapsto[1.0]{} N$.
- Therefore C isn't in our language.
- Thus we have passivity!

Permission counting

Permission counting

- Some programs naturally weigh out permissions to their child threads: e.g. parallel tree-copy, parallel tree-rewriting (see proceedings).

Permission counting

- Some programs naturally weigh out permissions to their child threads: e.g. parallel tree-copy, parallel tree-rewriting (see proceedings).
- Some programs count permissions: e.g. pipeline multicasting, readers-and-writers.

Permission counting

- Some programs naturally weigh out permissions to their child threads: e.g. parallel tree-copy, parallel tree-rewriting (see proceedings).
- Some programs count permissions: e.g. pipeline multicasting, readers-and-writers.
- Permission counting is not specification-only.

Readers and Writers (Courtois et.al. 1972)

$\mathrm{P}(\mathit{read});$

$\quad \text{if } \mathit{count} = 0 \text{ then } \mathrm{P}(\mathit{write})$

$\qquad\qquad \text{else skip fi};$

$\quad \mathit{count}{+} := 1;$

$\mathrm{V}(\mathit{read});$

... reading happens here ...

$\mathrm{P}(\mathit{read});$

$\quad \mathit{count}{-} := 1;$

$\quad \text{if } \mathit{count} = 0 \text{ then } \mathrm{V}(\mathit{write})$

$\qquad\qquad \text{else skip fi};$

$\mathrm{V}(\mathit{read})$

$\mathrm{P}(\mathit{write});$

... writing happens here ...

$\mathrm{V}(\mathit{write})$

Readers and writers (CCR version)

Readers and writers (CCR version)

with *read* when true do
  if $count = 0$ then $\mathrm{P}(write)$
    else skip fi;
  $count{+}{:=}\,1$
od;

... reading happens here ...

with *read* when $count > 0$ do
  $count{-}{:=}\,1$;
  if $count = 0$ then $\mathrm{V}(write)$
    else skip fi
od

$\mathrm{P}(write)$;

... writing happens here ...

$\mathrm{V}(write)$

Readers and writers (CCR version)

$\{\mathbf{emp}\}$

with read when true do

  if $\mathit{count} = 0$ then $\mathrm{P}(\mathit{write})$

    else skip fi;

  $\mathit{count}{+}{:=}\ 1$

od;

$\{z \rightarrowtail N\}$

  ... reading happens here ...

with read when $\mathit{count} > 0$ do

  $\mathit{count}{-}{:=}\ 1;$

  if $\mathit{count} = 0$ then $\mathrm{V}(\mathit{write})$

    else skip fi

od

$\mathrm{P}(\mathit{write});$

  ... writing happens here ...

$\mathrm{V}(\mathit{write})$

Readers and writers (CCR version)

$\{\mathbf{emp}\}$

with read when true do

  if $\mathit{count} = 0$ then $\mathrm{P}(\mathit{write})$

    else skip fi;

  $\mathit{count}{+}{:=}\ 1$

od;

$\{z \rightarrowtail N\}$

  ... reading happens here ...

$\{z \rightarrowtail N\}$

with read when $\mathit{count} > 0$ do

  $\mathit{count}{-}{:=}\ 1;$

  if $\mathit{count} = 0$ then $\mathrm{V}(\mathit{write})$

    else skip fi

od

$\{\mathbf{emp}\}$

$\mathrm{P}(\mathit{write});$

  ... writing happens here ...

$\mathrm{V}(\mathit{write})$

Readers and writers (CCR version)

$\{\mathbf{emp}\}$
with *read* when true do
  if $count = 0$ then $\mathrm{P}(write)$
    else skip fi;
  $count{+}{:=}\ 1$
od;
$\{z \rightarrowtail N\}$
  ... reading happens here ...
$\{z \rightarrowtail N\}$
with *read* when $count > 0$ do
  $count{-}{:=}\ 1$;
  if $count = 0$ then $\mathrm{V}(write)$
    else skip fi
od
$\{\mathbf{emp}\}$

$\{\mathbf{emp}\}$
$\mathrm{P}(write)$;
$\{z \overset{0}{\longmapsto} M\}$
  ... writing happens here ...

$\mathrm{V}(write)$

Readers and writers (CCR version)

$\{\mathbf{emp}\}$
with *read* when true do
  if $count = 0$ then $\mathrm{P}(write)$
    else skip fi;
  $count{+}{:=}\,1$
od;
$\{z \rightarrowtail N\}$
  ... reading happens here ...
$\{z \rightarrowtail N\}$
with *read* when $count > 0$ do
  $count{-}{:=}\,1$;
  if $count = 0$ then $\mathrm{V}(write)$
    else skip fi
od
$\{\mathbf{emp}\}$

$\{\mathbf{emp}\}$
$\mathrm{P}(write)$;
$\{z \overset{0}{\longmapsto} M\}$
  ... writing happens here ...
$\{z \overset{0}{\longmapsto} M'\}$
$\mathrm{V}(write)$
$\{\mathbf{emp}\}$

A counting model (Calcagno, Parkinson)

A counting model (Calcagno, Parkinson)

- Heaps are partial maps from Nat to (int, permission).

A counting model (Calcagno, Parkinson)

- Heaps are partial maps from Nat to (int, permission).
- Permissions are $-n$ (n read permissions), or $+n$ (a “block” from which n read permissions have been “flaked”).

A counting model (Calcagno, Parkinson)

- Heaps are partial maps from Nat to (int, permission).
- Permissions are $-n$ (n read permissions), or $+n$ (a “block” from which n read permissions have been “flaked”).
- 0 is total permission.

A counting model (Calcagno, Parkinson)

- Heaps are partial maps from Nat to (int, permission).
- Permissions are $-n$ (n read permissions), or $+n$ (a “block” from which n read permissions have been “flaked”).
- 0 is total permission.
- $E \overset{i}{\mapsto} E' \star E \overset{j}{\mapsto} E' = \begin{cases} \text{undefined} & i \geq 0 \wedge j \geq 0 \\ \text{undefined} & (i \geq 0 \vee j \geq 0) \wedge i + j < 0 \\ E \overset{i+j}{\longmapsto} E' & \text{otherwise} \end{cases}$

A counting model (Calcagno, Parkinson)

- Heaps are partial maps from Nat to (int, permission).
- Permissions are $-n$ (n read permissions), or $+n$ (a “block” from which n read permissions have been “flaked”).
- 0 is total permission.
- $E \overset{i}{\mapsto} E' \star E \overset{j}{\mapsto} E' = \begin{cases} \text{undefined} & i \geq 0 \land j \geq 0 \\ \text{undefined} & (i \geq 0 \lor j \geq 0) \land i + j < 0 \\ E \overset{i+j}{\longmapsto} E' & \text{otherwise} \end{cases}$
- $E \rightarrowtail E'$ is a notational convenience for $E \overset{-1}{\longmapsto} E'$.

A counting model (Calcagno, Parkinson)

- Heaps are partial maps from Nat to (int, permission).
- Permissions are $-n$ (n read permissions), or $+n$ (a “block” from which n read permissions have been “flaked”).
- 0 is total permission.
- $E \overset{i}{\mapsto} E' \star E \overset{j}{\mapsto} E' = \begin{cases} \text{undefined} & i \geq 0 \wedge j \geq 0 \\ \text{undefined} & (i \geq 0 \vee j \geq 0) \wedge i + j < 0 \\ E \xmapsto{i+j} E' & \text{otherwise} \end{cases}$
- $E \rightarrowtail E'$ is a notational convenience for $E \xmapsto{-1} E'$.
- We have passivity (same proof as before).

Proof theory

Proof theory

$$E \xmapsto{n} E' \quad \Rightarrow \quad n \geq 0$$

$$E \xmapsto{n} E' \iff E \xmapsto{n+1} E' \star E \rightarrowtail E'$$

Proof theory

$$E \overset{n}{\longmapsto} E' \;\Rightarrow\; n \geq 0$$

$$E \overset{n}{\longmapsto} E' \iff E \overset{n+1}{\longmapsto} E' \star E \rightarrowtail E'$$

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

Resource safety proof

$write$: if $write = 0$ then **emp** else $z \stackrel{0}{\longmapsto} N$ fi

$read$: if $count = 0$ then **emp** else $z \xmapsto{count} N$ fi

$\{\mathbf{emp}\}$

with $read$ when true do

if $count = 0$ then $\quad$ P($write$)

else $\quad$ skip

fi;

$count{+}{:=}\,1$

od

$\{z \rightarrowtail N\}$

Resource safety proof

$$write : \text{if } write = 0 \text{ then } \mathbf{emp} \text{ else } z \xmapsto{0} N \text{ fi}$$

$$read : \text{if } count = 0 \text{ then } \mathbf{emp} \text{ else } z \xmapsto{count} N \text{ fi}$$

$\{\mathbf{emp}\}$

with *read* when true do

  $\{\text{if } count = 0 \text{ then } \mathbf{emp} \text{ else } z \xmapsto{count} N \text{ fi} \star \mathbf{emp}\}$

  if $count = 0$ then $\mathrm{P}(write)$

   else skip

  fi;

  $count {+}{:=} 1$

od

$\{z \rightarrowtail N\}$

Resource safety proof

$$write : \text{if } write = 0 \text{ then } \mathbf{emp} \text{ else } z \overset{0}{\longmapsto} N \text{ fi}$$
$$read : \text{if } count = 0 \text{ then } \mathbf{emp} \text{ else } z \overset{count}{\longmapsto} N \text{ fi}$$

$\{\mathbf{emp}\}$

with $read$ when true do

$\quad \{\text{if } count = 0 \text{ then } \mathbf{emp} \text{ else } z \overset{count}{\longmapsto} N \text{ fi} \star \mathbf{emp}\}$

$\quad$ if $count = 0$ then $\quad \{\mathbf{emp}\}$ P$(write)$

$\qquad\qquad\qquad$ else $\quad \{z \overset{count}{\longmapsto} N\}$ $\quad$ skip

$\quad$ fi;

$\quad count{+}{:=}\ 1$

od

$\{z \rightarrowtail N\}$

Resource safety proof

$$\mathit{write} : \text{if } \mathit{write} = 0 \text{ then } \mathbf{emp} \text{ else } z \xmapsto{0} N \text{ fi}$$

$$\mathit{read} : \text{if } \mathit{count} = 0 \text{ then } \mathbf{emp} \text{ else } z \xmapsto{\mathit{count}} N \text{ fi}$$

$\{\mathbf{emp}\}$

with *read* when true do

$\{\text{if } \mathit{count} = 0 \text{ then } \mathbf{emp} \text{ else } z \xmapsto{\mathit{count}} N \text{ fi} \star \mathbf{emp}\}$

if $\mathit{count} = 0$ then $\{\mathbf{emp}\}$ $\mathrm{P}(\mathit{write})$ $\{z \xmapsto{0} N\}$

else $\{z \xmapsto{\mathit{count}} N\}$ skip

fi;

$\mathit{count} {+}{:=} 1$

od

$\{z \rightarrowtail N\}$

Resource safety proof

$write : \text{if } write = 0 \text{ then } \mathbf{emp} \text{ else } z \xmapsto{0} N \text{ fi}$
$read : \text{if } count = 0 \text{ then } \mathbf{emp} \text{ else } z \xmapsto{count} N \text{ fi}$

$\{\mathbf{emp}\}$
with *read* when true do
$\{\text{if } count = 0 \text{ then } \mathbf{emp} \text{ else } z \xmapsto{count} N \text{ fi} \star \mathbf{emp}\}$
if $count = 0$ then $\{\mathbf{emp}\}$ P($write$) $\{z \xmapsto{0} N\}$
else $\{z \xmapsto{count} N\}$ skip $\{z \xmapsto{count} N\}$
fi;

$count {+}{:=}\, 1$

od
$\{z \rightarrowtail N\}$

Resource safety proof

$$\mathit{write} : \text{if } \mathit{write} = 0 \text{ then } \mathbf{emp} \text{ else } z \overset{0}{\longmapsto} N \text{ fi}$$
$$\mathit{read} : \text{if } \mathit{count} = 0 \text{ then } \mathbf{emp} \text{ else } z \xmapsto{\mathit{count}} N \text{ fi}$$

$\{\mathbf{emp}\}$

with *read* when true do

$\{\text{if } \mathit{count} = 0 \text{ then } \mathbf{emp} \text{ else } z \xmapsto{\mathit{count}} N \text{ fi} \star \mathbf{emp}\}$

if $\mathit{count} = 0$ then $\{\mathbf{emp}\}$ $\mathrm{P}(\mathit{write})$ $\{z \overset{0}{\longmapsto} N\}$

else $\{z \xmapsto{\mathit{count}} N\}$ skip $\{z \xmapsto{\mathit{count}} N\}$

fi;

$\{z \xmapsto{\mathit{count}} N\}$

$\mathit{count} {+}{:=} 1$

od

$\{z \rightarrowtail N\}$

Resource safety proof

$$\mathit{write} : \text{if } \mathit{write} = 0 \text{ then } \mathbf{emp} \text{ else } z \xmapsto{0} N \text{ fi}$$

$$\mathit{read} : \text{if } \mathit{count} = 0 \text{ then } \mathbf{emp} \text{ else } z \xmapsto{\mathit{count}} N \text{ fi}$$

$\{\mathbf{emp}\}$

with *read* when true do

$\quad \{\text{if } \mathit{count} = 0 \text{ then } \mathbf{emp} \text{ else } z \xmapsto{\mathit{count}} N \text{ fi} \star \mathbf{emp}\}$

$\quad$ if $\mathit{count} = 0$ then $\quad \{\mathbf{emp}\}\ \mathrm{P}(\mathit{write})\ \{z \xmapsto{0} N\}$

$\qquad\qquad$ else $\quad \{z \xmapsto{\mathit{count}} N\}\ \text{skip}\ \{z \xmapsto{\mathit{count}} N\}$

$\quad$ fi;

$\quad \{z \xmapsto{\mathit{count}} N\}$

$\quad \mathit{count} {+}{:=} 1$

$\quad \{z \xmapsto{\mathit{count}-1} N\}$

od

$\{z \rightarrowtail N\}$

Resource safety proof

$$\mathit{write} : \text{if } \mathit{write} = 0 \text{ then } \mathbf{emp} \text{ else } z \overset{0}{\longmapsto} N \text{ fi}$$

$$\mathit{read} : \text{if } \mathit{count} = 0 \text{ then } \mathbf{emp} \text{ else } z \xmapsto{\mathit{count}} N \text{ fi}$$

$\{\mathbf{emp}\}$

with read when true do

$\quad \{\text{if } \mathit{count} = 0 \text{ then } \mathbf{emp} \text{ else } z \xmapsto{\mathit{count}} N \text{ fi} \star \mathbf{emp}\}$

$\quad \text{if } \mathit{count} = 0 \text{ then} \qquad \{\mathbf{emp}\}\ \mathrm{P}(\mathit{write})\ \{z \overset{0}{\longmapsto} N\}$

$\qquad\qquad\qquad \text{else} \quad \{z \xmapsto{\mathit{count}} N\} \quad \text{skip} \quad \{z \xmapsto{\mathit{count}} N\}$

$\quad \text{fi};$

$\quad \{z \xmapsto{\mathit{count}} N\}$

$\quad \mathit{count} +:= 1$

$\quad \{z \xmapsto{\mathit{count}-1} N\} \therefore \{z \xmapsto{\mathit{count}} N \star z \rightarrowtail N\}$

od

$\{z \rightarrowtail N\}$

Do we need two models?

Do we need two models?

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

Do we need two models?

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

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

Do we need two models?

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

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

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

Parallel tree rewriting

$\mathit{subst}\ x\ y\ v =$

$\quad \text{if } [x] = 0 \text{ then}$ // Lam

$\qquad \text{if } [x+1] \neq v \text{ then } [x+2] := \mathit{subst}\ [x+2]\ y\ v \text{ else } \mathit{skip} \text{ fi};$

$\qquad x$

$\quad \text{elsf } [x] = 1 \text{ then}$ // App – do it in parallel

$\qquad \big([x+1] := \mathit{subst}\ [x+1]\ y\ v \,\|\, [x+2] := \mathit{subst}\ [x+2]\ y\ v)\big)\ ;$

$\qquad x$

$\quad \text{elsf } [x+1] = v \text{ then}$ // Var, same v

$\qquad \text{dispose } x; \text{dispose}(x+1); \text{new}(2, \mathit{copy}\ y)$

$\quad \text{else}$ // Var, different v

$\qquad x$

$\quad \text{fi}$

Parallel tree rewriting

$\mathit{subst}\ x\ y\ v =$

  if $[x] = 0$ then // Lam

    if $[x+1] \neq v$ then $[x+2] := \mathit{subst}\ [x+2]\ y\ v$ else *skip* fi;

    x

  elsf $[x] = 1$ then // App – do it in parallel

    $\big([x+1] := \mathit{subst}\ [x+1]\ y\ v \,\big\|\, [x+2] := \mathit{subst}\ [x+2]\ y\ v)\big)$;

    x

  elsf $[x+1] = v$ then // Var, same v

    dispose x; dispose$(x+1)$; new$(2, \mathit{copy}\ y)$

  else // Var, different v

    x

  fi

– proof easy with fractions, ridiculous with counting permissions; readers and writers swings the other way.

Parallel tree rewriting

$\mathit{subst}\ x\ y\ v =$

  if $[x] = 0$ then // Lam

    if $[x+1] \neq v$ then $[x+2] := \mathit{subst}\ [x+2]\ y\ v$ else *skip* fi;

    x

  elsf $[x] = 1$ then // App – do it in parallel

    $\big([x+1] := \mathit{subst}\ [x+1]\ y\ v \,\|\, [x+2] := \mathit{subst}\ [x+2]\ y\ v)\big)$;

    x

  elsf $[x+1] = v$ then // Var, same v

    dispose x; dispose$(x+1)$; new$(2, \mathit{copy}\ y)$

  else // Var, different v

    x

  fi

– proof easy with fractions, ridiculous with counting permissions; readers and writers swings the other way.

We need more than one model!

Passivity and concurrency

- If I have $x \xmapsto[0.5]{} \_$, I can be sure that you can't write to it.

Passivity and concurrency

- If I have $x \xmapsto[0.5]{} \_$, I can be sure that you can't write to it.
- If I give you $x \xmapsto[0.5]{} \_$ in the static case, I can be sure you can't write to it.

Passivity and concurrency

- If I have $x \xmapsto[0.5]{} \_$, I can be sure that you can't write to it.
- If I give you $x \xmapsto[0.5]{} \_$ in the static case, I can be sure you can't write to it.
- In the concurrent/modular case, you might have the other half, or get it temporarily from elsewhere.

Passivity and concurrency

- If I have $x \underset{0.5}{\longmapsto} \_$, I can be sure that you can't write to it.
- If I give you $x \underset{0.5}{\longmapsto} \_$ in the static case, I can be sure you can't write to it.
- In the concurrent/modular case, you might have the other half, or get it temporarily from elsewhere.
- Moral: keep your hand on your ha'penny; don't give them everything you've got.

Passivity and concurrency

- If I have $x \xmapsto[0.5]{} \_$, I can be sure that you can't write to it.
- If I give you $x \xmapsto[0.5]{} \_$ in the static case, I can be sure you can't write to it.
- In the concurrent/modular case, you might have the other half, or get it temporarily from elsewhere.
- Moral: keep your hand on your ha'penny; don't give them everything you've got.
- (Same applies to counting permissions.)

We don't understand recursive definitions any more

We don't understand recursive definitions any more

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

We don't understand recursive definitions any more

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

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

We don't understand recursive definitions any more

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

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

$x \underset{0.5}{\longmapsto} 1, l, l \star l \underset{1.0}{\longmapsto} 0, 3$ satisfies $\mathrm{ztree}\ 0.5\ x\ (\mathsf{Node}\ (\mathsf{Tip}\ 3)\ (\mathsf{Tip}\ 3))$
(and we can write to it)!!

We don't understand recursive definitions any more

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

$$
\begin{aligned}
\mathrm{ztree}\ z\ \mathrm{nil}\ \mathsf{Empty} &\mathrel{\hat{=}} \mathbf{emp} \\
\mathrm{ztree}\ z\ t\ (\mathsf{Tip}\ \alpha) &\mathrel{\hat{=}} t \xmapsto[z]{} 0, \alpha \\
\mathrm{ztree}\ z\ t\ (\mathsf{Node}\ \lambda\ \rho) &\mathrel{\hat{=}} \exists l, r \cdot \Big(t \xmapsto[z]{} 1, l, r\ \star \mathrm{ztree}\ z\ l\ \lambda \star \mathrm{ztree}\ z\ r\ \rho\Big)
\end{aligned}
$$

$x \xmapsto[0.5]{} 1, l, l \star l \xmapsto[1.0]{} 0, 3$ satisfies $\mathrm{ztree}\ 0.5\ x\ (\mathsf{Node}\ (\mathsf{Tip}\ 3)\ (\mathsf{Tip}\ 3))$ (and we can write to it)!!

We have $\mathrm{ztree}\ (z + z')\ t\ \tau \iff \mathrm{ztree}\ z\ t\ \tau \star \mathrm{ztree}\ z'\ t\ \tau$, but sometimes only vacuously.

We don't understand recursive definitions any more

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

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

$x \underset{0.5}{\longmapsto} 1, l, l \star l \underset{1.0}{\longmapsto} 0, 3$ satisfies $\mathrm{ztree}\ 0.5\ x\ (\mathsf{Node}\ (\mathsf{Tip}\ 3)\ (\mathsf{Tip}\ 3))$ (and we can write to it)!!

We have $\mathrm{ztree}\ (z + z')\ t\ \tau \iff \mathrm{ztree}\ z\ t\ \tau \star \mathrm{ztree}\ z'\ t\ \tau$, but sometimes only vacuously.

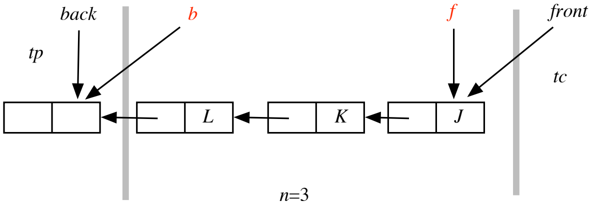
We can write programs which work with $\mathrm{ztree}\ 0.5$, but crash with $\mathrm{ztree}\ 0.499$.

The unbounded buffer

```
begin integer numberOfQueuingPortions,
              bufferManipulation;
      numberOfQueuingPortions := 0;
      bufferManipulation := 1;
      parbegin
      producer:  begin
             again1:  produce next portion;
                      add portion to buffer;
                      V(numberOfQueuingPortions);
                      goto again1
                 end;
      consumer:  begin
             again2:  P(numberOfQueuingPortions);
                      take portion from buffer;
                      process portion taken;
                      goto again2
                 end
      parend
end
```

Proposed and withdrawn in 1965; proved safe, Habermann 1972.

A proof with variables-as-resource



A proof with variables-as-resource

// Producer.

$$\left\{ \begin{array}{l} back, tp, b_{\frac{1}{2}} \vdash \\ \quad back \mapsto \_, \_ \wedge back = b \end{array} \right\}$$

$back.0 := \text{produce}();$
$tp := \text{new}();$
$back.1 := tp;$
$\mathrm{V}(n);$
$back := tp$

$$\left\{ \begin{array}{l} back, tp, b_{\frac{1}{2}} \vdash \\ \quad back \mapsto \_, \_ \wedge back = b \end{array} \right\}$$

$\|$

// Semaphore n

$$\{n, f_{\frac{1}{2}}, b_{\frac{1}{2}} \vdash \text{listseg } n\ f\ b\}$$

$\mathrm{P} : \text{dec}\, n; f := f.2$

$\mathrm{V} : \text{inc}\, n; b := b.2$

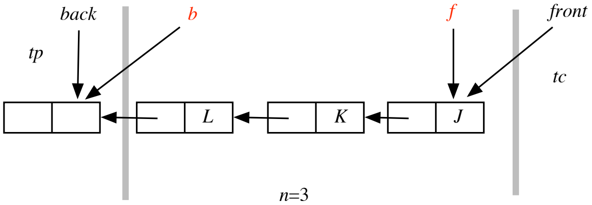
$\|$

// Consumer.

$$\{front, tc, f_{\frac{1}{2}} \vdash front = f\}$$

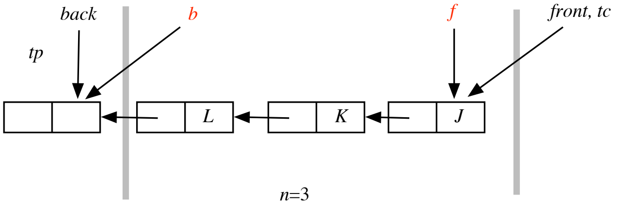
$tc := front;$
$\mathrm{P}(n);$
$front := front.2;$
$\text{consume } tc.0;$
$\text{dispose } tc$

$$\{front, tc, f_{\frac{1}{2}} \vdash front = f\}$$



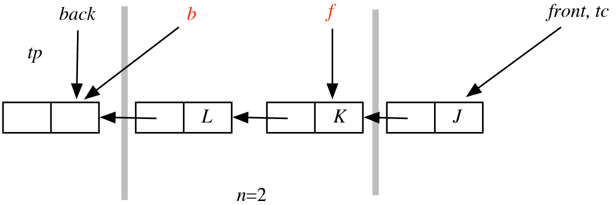
A proof with variables-as-resource

$$
\left(
\begin{array}{l|l|l}
\text{// Producer.} & \text{// Semaphore } n & \text{// Consumer.} \\
\left\{ \begin{array}{l} back, tp, b_{\frac{1}{2}} \vdash \\ \quad back \mapsto \_, \_ \wedge back = b \end{array} \right\} & \{n, f_{\frac{1}{2}}, b_{\frac{1}{2}} \vdash \mathrm{listseg}\ n\ f\ b\} & \{front, tc, f_{\frac{1}{2}} \vdash front = f\} \\
back.0 := \mathrm{produce}(); & & tc := front; \\
tp := \mathrm{new}(); & \mathrm{P} : \mathrm{dec}\, n; f := f.2 & \mathrm{P}(n); \\
back.1 := tp; & & front := front.2; \\
\mathrm{V}(n); & \mathrm{V} : \mathrm{inc}\, n; b := b.2 & \mathrm{consume}\ tc.0; \\
back := tp & & \mathrm{dispose}\ tc \\
\left\{ \begin{array}{l} back, tp, b_{\frac{1}{2}} \vdash \\ \quad back \mapsto \_, \_ \wedge back = b \end{array} \right\} & & \{front, tc, f_{\frac{1}{2}} \vdash front = f\}
\end{array}
\right)
$$



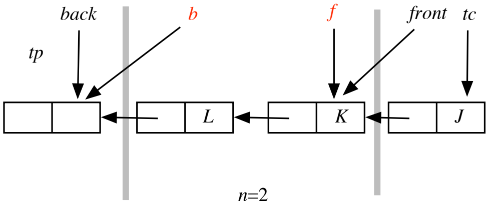
A proof with variables-as-resource

// Producer.	// Semaphore n	// Consumer.
$\left\{\begin{array}{l} back, tp, b_{\frac{1}{2}} \vdash \\ \quad back \mapsto \_, \_ \wedge back = b \end{array}\right\}$	$\{n, f_{\frac{1}{2}}, b_{\frac{1}{2}} \vdash \text{listseg}\ n\ f\ b\}$	$\{front, tc, f_{\frac{1}{2}} \vdash front = f\}$
$back.0 := \text{produce}();$		$tc := front;$
$tp := \text{new}();$	$\text{P} : \text{dec}\, n; f := f.2$	$\text{P}(n);$
$back.1 := tp;$		$front := front.2;$
$\text{V}(n);$	$\text{V} : \text{inc}\, n; b := b.2$	consume $tc.0;$
$back := tp$		dispose tc
$\left\{\begin{array}{l} back, tp, b_{\frac{1}{2}} \vdash \\ \quad back \mapsto \_, \_ \wedge back = b \end{array}\right\}$		$\{front, tc, f_{\frac{1}{2}} \vdash front = f\}$



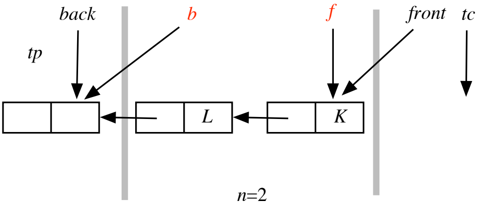
A proof with variables-as-resource

// Producer.	// Semaphore n	// Consumer.
$\left\{ \begin{array}{l} back, tp, b_{\frac{1}{2}} \vdash \\ \quad back \mapsto \_, \_ \wedge back = b \end{array} \right\}$	$\{n, f_{\frac{1}{2}}, b_{\frac{1}{2}} \vdash \text{listseg } n\ f\ b\}$	$\{front, tc, f_{\frac{1}{2}} \vdash front = f\}$
$back.0 := \text{produce}();$		$tc := front;$
$tp := \text{new}();$	$\mathrm{P} : \text{dec}\, n; f := f.2$	$\mathrm{P}(n);$
$back.1 := tp;$		$front := front.2;$
$\mathrm{V}(n);$	$\mathrm{V} : \text{inc}\, n; b := b.2$	$\text{consume } tc.0;$
$back := tp$		$\text{dispose } tc$
$\left\{ \begin{array}{l} back, tp, b_{\frac{1}{2}} \vdash \\ \quad back \mapsto \_, \_ \wedge back = b \end{array} \right\}$		$\{front, tc, f_{\frac{1}{2}} \vdash front = f\}$



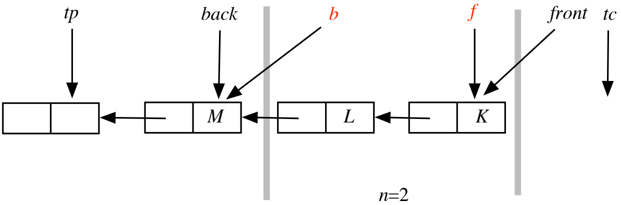
A proof with variables-as-resource

$$\left(\begin{array}{l} \text{// Producer.} \\ \left\{ \begin{array}{l} back, tp, b_{\frac{1}{2}} \vdash \\ \quad back \mapsto \_, \_ \wedge back = b \end{array} \right\} \\ back.0 := \text{produce}(); \\ tp := \text{new}(); \\ back.1 := tp; \\ \mathrm{V}(n); \\ back := tp \\ \left\{ \begin{array}{l} back, tp, b_{\frac{1}{2}} \vdash \\ \quad back \mapsto \_, \_ \wedge back = b \end{array} \right\} \end{array} \middle\| \begin{array}{l} \text{// Semaphore } n \\ \{n, f_{\frac{1}{2}}, b_{\frac{1}{2}} \vdash \text{listseg } n\ f\ b\} \\ \\ \mathrm{P} : \text{dec}\, n; f := f.2 \\ \\ \mathrm{V} : \text{inc}\, n; b := b.2 \\ \\ \\ \end{array} \middle\| \begin{array}{l} \text{// Consumer.} \\ \{front, tc, f_{\frac{1}{2}} \vdash front = f\} \\ tc := front; \\ \mathrm{P}(n); \\ front := front.2; \\ \text{consume } tc.0; \\ \text{dispose } tc \\ \{front, tc, f_{\frac{1}{2}} \vdash front = f\} \end{array} \right)$$



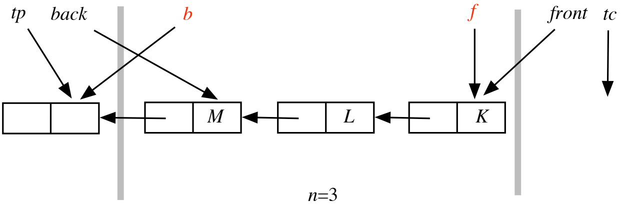
A proof with variables-as-resource

$$
\left(
\begin{array}{l}
\text{// Producer.} \\
\left\{ \begin{array}{l} back, tp, b_{\frac{1}{2}} \vdash \\ \quad back \mapsto \_, \_ \wedge back = b \end{array} \right\} \\
back.0 := \mathrm{produce}(); \\
tp := \mathrm{new}(); \\
back.1 := tp; \\
\mathrm{V}(n); \\
back := tp \\
\left\{ \begin{array}{l} back, tp, b_{\frac{1}{2}} \vdash \\ \quad back \mapsto \_, \_ \wedge back = b \end{array} \right\}
\end{array}
\left\|
\begin{array}{l}
\text{// Semaphore } n \\
\{ n, f_{\frac{1}{2}}, b_{\frac{1}{2}} \vdash \mathrm{listseg}\ n\ f\ b \} \\
\\
\mathrm{P} : \mathrm{dec}\, n; f := f.2 \\
\\
\mathrm{V} : \mathrm{inc}\, n; b := b.2 \\
\\
\\
\\
\end{array}
\right\|
\begin{array}{l}
\text{// Consumer.} \\
\{ front, tc, f_{\frac{1}{2}} \vdash front = f \} \\
tc := front; \\
\mathrm{P}(n); \\
front := front.2; \\
\mathrm{consume}\ tc.0; \\
\mathrm{dispose}\ tc \\
\{ front, tc, f_{\frac{1}{2}} \vdash front = f \}
\end{array}
\right)
$$



A proof with variables-as-resource

// Producer.	// Semaphore n	// Consumer.
$\left\{ \begin{array}{l} back, tp, b_{\frac{1}{2}} \vdash \\ \quad back \mapsto \_, \_ \wedge back = b \end{array} \right\}$	$\{n, f_{\frac{1}{2}}, b_{\frac{1}{2}} \vdash \text{listseg } n\ f\ b\}$	$\{front, tc, f_{\frac{1}{2}} \vdash front = f\}$
$back.0 := \text{produce}();$		$tc := front;$
$tp := \text{new}();$	$\text{P} : \text{dec } n; f := f.2$	$\text{P}(n);$
$back.1 := tp;$		$front := front.2;$
$\text{V}(n);$	$\text{V} : \text{inc } n; b := b.2$	$\text{consume } tc.0;$
$back := tp$		$\text{dispose } tc$
$\left\{ \begin{array}{l} back, tp, b_{\frac{1}{2}} \vdash \\ \quad back \mapsto \_, \_ \wedge back = b \end{array} \right\}$		$\{front, tc, f_{\frac{1}{2}} \vdash front = f\}$



A proof with variables-as-resource

// Producer.

$\left\{ \begin{array}{l} back, tp, b_{\frac{1}{2}} \vdash \\ \quad back \mapsto \_, \_ \land back = b \end{array} \right\}$

$back.0 := \mathrm{produce}();$

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

$back.1 := tp;$

$\mathrm{V}(n);$

$back := tp$

$\left\{ \begin{array}{l} back, tp, b_{\frac{1}{2}} \vdash \\ \quad back \mapsto \_, \_ \land back = b \end{array} \right\}$

// Semaphore n

$\{n, f_{\frac{1}{2}}, b_{\frac{1}{2}} \vdash \mathrm{listseg}\ n\ f\ b\}$

$\mathrm{P} : \mathrm{dec}\, n; f := f.2$

$\mathrm{V} : \mathrm{inc}\, n; b := b.2$

// Consumer.

$\{front, tc, f_{\frac{1}{2}} \vdash front = f\}$

$tc := front;$

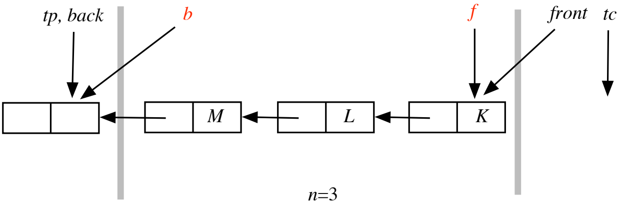
$\mathrm{P}(n);$

$front := front.2;$

$\mathrm{consume}\ tc.0;$

$\mathrm{dispose}\ tc$

$\{front, tc, f_{\frac{1}{2}} \vdash front = f\}$



We can prove it!

$$
\left(
\begin{array}{l||l||l}
\text{// Producer.} & \text{// Semaphore } n & \text{// Consumer.} \\
\left\{ \begin{array}{l} back, tp, b_{\frac{1}{2}} \vdash \\ \quad back \mapsto \_, \_ \wedge back = b \end{array} \right\} & \{n, f_{\frac{1}{2}}, b_{\frac{1}{2}} \vdash \mathrm{listseg}\ n\ f\ b\} & \{front, tc, f_{\frac{1}{2}} \vdash front = f\} \\
back.0 := \mathrm{produce}(); & & tc := front; \\
tp := \mathrm{new}(); & \mathrm{P} : \mathrm{dec}\, n; f := f.2 & \mathrm{P}(n); \\
back.1 := tp; & & front := front.2; \\
\mathrm{V}(n); & \mathrm{V} : \mathrm{inc}\, n; b := b.2 & \mathrm{consume}\ tc.0; \\
back := tp & & \mathrm{dispose}\ tc \\
\left\{ \begin{array}{l} back, tp, b_{\frac{1}{2}} \vdash \\ \quad back \mapsto \_, \_ \wedge back = b \end{array} \right\} & & \{front, tc, f_{\frac{1}{2}} \vdash front = f\}
\end{array}
\right)
$$

We can prove it!

$$
\left(\begin{array}{l|l|l}
\text{// Producer.} & \text{// Semaphore } n & \text{// Consumer.} \\
\left\{\begin{array}{l} back, tp, b_{\frac{1}{2}} \vdash \\ \quad back \mapsto \_, \_ \wedge back = b \end{array}\right\} & \{n, f_{\frac{1}{2}}, b_{\frac{1}{2}} \vdash \text{listseg } n\ f\ b\} & \{front, tc, f_{\frac{1}{2}} \vdash front = f\} \\
back.0 := \text{produce}(); & & tc := front; \\
tp := \text{new}(); & \mathrm{P} : \text{dec } n; f := f.2 & \mathrm{P}(n); \\
back.1 := tp; & & front := front.2; \\
\mathrm{V}(n); & \mathrm{V} : \text{inc } n; b := b.2 & \text{consume } tc.0; \\
back := tp & & \text{dispose } tc \\
\left\{\begin{array}{l} back, tp, b_{\frac{1}{2}} \vdash \\ \quad back \mapsto \_, \_ \wedge back = b \end{array}\right\} & & \{front, tc, f_{\frac{1}{2}} \vdash front = f\}
\end{array}\right)
$$

Assertion $vs \vdash P$ says "owning variables vs, P holds". P can only mention variables in vs. You can't write to fractionally-owned variables.

We can prove it!

$$
\left(\begin{array}{l||l||l}
\text{// Producer.} & \text{// Semaphore } n & \text{// Consumer.} \\
\left\{\begin{array}{l} back, tp, b_{\frac{1}{2}} \vdash \\ \quad back \mapsto \_, \_ \wedge back = b \end{array}\right\} & \{n, f_{\frac{1}{2}}, b_{\frac{1}{2}} \vdash \text{listseg } n\ f\ b\} & \{front, tc, f_{\frac{1}{2}} \vdash front = f\} \\
back.0 := \text{produce}(); & & tc := front; \\
tp := \text{new}(); & \text{P} : \text{dec}\, n; f := f.2 & \text{P}(n); \\
back.1 := tp; & & front := front.2; \\
\text{V}(n); & \text{V} : \text{inc}\, n; b := b.2 & \text{consume } tc.0; \\
back := tp & & \text{dispose } tc \\
\left\{\begin{array}{l} back, tp, b_{\frac{1}{2}} \vdash \\ \quad back \mapsto \_, \_ \wedge back = b \end{array}\right\} & & \{front, tc, f_{\frac{1}{2}} \vdash front = f\}
\end{array}\right)
$$

Assertion $vs \vdash P$ says "owning variables vs, P holds". P can only mention variables in vs. You can't write to fractionally-owned variables.

P can describe separation of the heap: $back \mapsto \_, \_$ describes ownership of a two-word record; $back \mapsto \_, \_ \star front \mapsto \_, \_$ describes ownership of two cons-cells separately.

We have more ideas than we can deal with

We have more ideas than we can deal with

- existence (no read, no write) permissions: e.g. P+V+read/write for semaphores;

We have more ideas than we can deal with

- existence (no read, no write) permissions: e.g. P+V+read/write for semaphores;
- mobile channels: e.g. read one end, write the other in occam;

We have more ideas than we can deal with

- existence (no read, no write) permissions: e.g. P+V+read/write for semaphores;
- mobile channels: e.g. read one end, write the other in occam;
- semaphores in the heap (for shared buffers which reclaim themselves);

We have more ideas than we can deal with

- existence (no read, no write) permissions: e.g. P+V+read/write for semaphores;
- mobile channels: e.g. read one end, write the other in occam;
- semaphores in the heap (for shared buffers which reclaim themselves);
- mobile code, maybe (if David May will tell us how it works);

We have more ideas than we can deal with

- existence (no read, no write) permissions: e.g. P+V+read/write for semaphores;
- mobile channels: e.g. read one end, write the other in occam;
- semaphores in the heap (for shared buffers which reclaim themselves);
- mobile code, maybe (if David May will tell us how it works);
- ...

The Grand Challenge

The Grand Challenge

Resourcing problems are everywhere. The problem is to make a resourcing solution:

The Grand Challenge

Resourcing problems are everywhere. The problem is to make a resourcing solution:

- as lightweight as typing;

The Grand Challenge

Resourcing problems are everywhere. The problem is to make a resourcing solution:

- as lightweight as typing;
- built into language designs;

The Grand Challenge

Resourcing problems are everywhere. The problem is to make a resourcing solution:

- as lightweight as typing;
- built into language designs;
- built into compilers.

The Grand Challenge

Resourcing problems are everywhere. The problem is to make a resourcing solution:

- as lightweight as typing;
- built into language designs;
- built into compilers.

If we build it, they will come (as they came for types).