Proof methods for concurrent programs

2. concurrent separation logic

Francesco Zappa Nardelli

INRIA Paris-Rocquencourt, MOSCOVA project-team

francesco.zappa_nardelli@inria.fr

http://moscova.inria.fr/~zappa/teaching/mpri/2010/

Warm-up

Hoare logic:

- Commands operate on the state: `C / s` $\longrightarrow$ `C' / s'`;
- statements P are assertions on the state: $s \vDash P$;
- a triple {P} `C` {Q} states that whenever `C` is executed in a state satisfying P and the execution of `C` terminates, the state in which `C`'s execution terminates satisfies Q;
- a logic system allows us to prove $\vdash$ {P} `C` {Q}. The logic system is sound.

Separation logic. All of the above plus:

- Special assertions, P * Q, $E_1 \longmapsto E_2$, empty, to describe the heap part of the state.

Warm-up

Special assertions, P * Q, $E_1 \longmapsto E_2$, empty, to describe the heap part of the state.

Three axioms to reason about separation:

- *write:* { E ↦ _ } [E] = E' { E ↦ E' }
- *dispose:* { E ↦ _ } dispose(E) { empty }
- *alloc:* { empty } x = cons(E1,…,En) { x ↦ E1 * x+1 ↦ E2 * … * x+(n-1) ↦ En }

where E ↦ _ is a shorthand for ∃ x. E ↦ x .

Exercise: prove that { i ↦ v } x := [i] { i ↦ v ∧ x = v } .

Pure assertions

Remark: some assertions are independent of the heap, e.g. $x = v$.

Definition: an assertion P is *pure*, iff for all stores s and heaps h_1 and h_2, it holds

$$(s, h_1) \vdash P \quad \text{iff} \quad (s, h_2) \vdash P\,.$$

Some key properties of pure assertions:

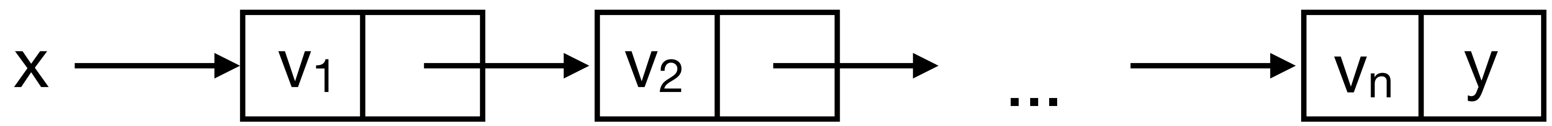
$$P \wedge Q \Rightarrow P * Q \qquad \text{when P or Q is pure;}$$

$$P * Q \Rightarrow P \wedge Q \qquad \text{when P and Q are pure;}$$

$$(P \wedge Q) * R \Rightarrow (P * R) \wedge Q \qquad \text{when Q is pure.}$$

Warm-up: list segments

The lseg predicate denotes *list segments*:



$$\mathrm{lseg}\ [\,]\ (x,y) \equiv \mathrm{empty} \wedge x = y$$

$$\mathrm{lseg}\ v{::}\alpha\ (x,y) \equiv \exists\, j.\ x \mapsto v * (x{+}1 \mapsto j) * \mathrm{lseg}\ \alpha\ (j,y)$$

Exercise: prove that the triple below holds.

$$\{\ \mathrm{lseg}\ a \cdot \alpha\ (i,k)\ \}\ \texttt{r := [i+1]; dispose i; dispose i+1; i := r}\ \{\ \mathrm{lseg}\ \alpha\ (i,k)\ \}$$

Remark: it is important to be able to reason on the assertion. Prove, by structural induction on α, that:

$$\mathrm{lseg}\ \alpha \cdot \beta\ (x,y) \Leftrightarrow \exists\, j.\ \mathrm{lseg}\ \alpha\ (x,j) \wedge \mathrm{lseg}\ \beta\ (j,y)$$

Warm-up: a cyclic buffer

We implement a *cyclic buffer* using:

- an active list segment lseg α ($\mathtt{i},\mathtt{j}$) (where α is the content of the buffer);
- an inactive list segment lseg β ($\mathtt{j},\mathtt{i}$) (where β is arbitrary);
- an unchanged variable `n` records the combined length of the two lists.

When `i=j` the buffer is empty or full:

- a variable `m` records the length of the active list segment.

Inserting and deleting elements on the buffer must preserve the invariant:

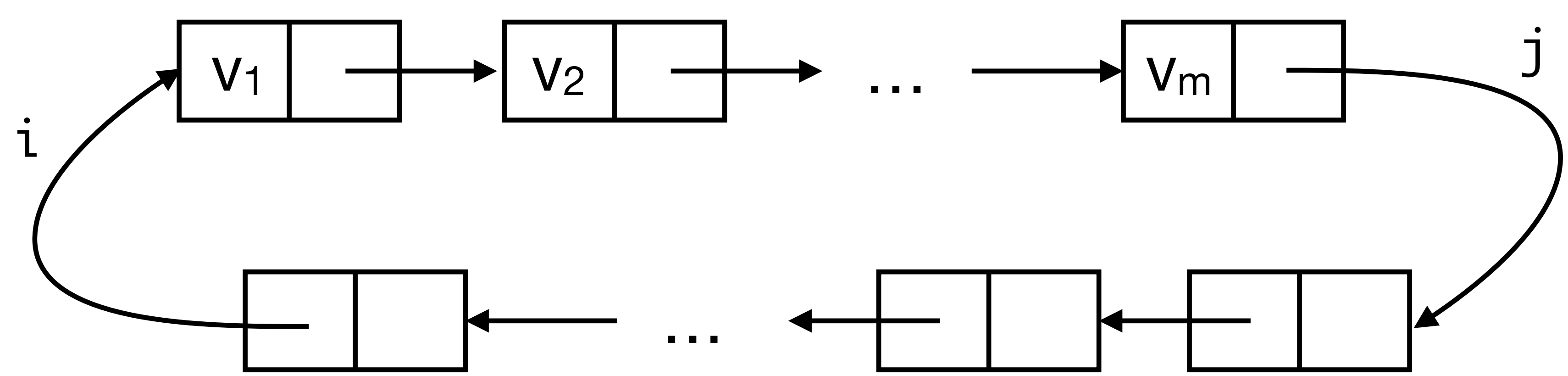
$$\exists\, \beta.\ (\mathrm{lseg}\ \alpha\ (\mathtt{i},\mathtt{j}) * \mathrm{lseg}\ \beta\ (\mathtt{j},\mathtt{i})) \wedge \mathtt{m} = \#\alpha \ \wedge\ \mathtt{n} = \#\alpha + \#\beta$$

(where # computes the length of a sequence).

Warm-up: a cyclic buffer

Adding `x` to the buffer can be done by the code below (under the hypothesis that `n-m > 0`):

```
[j] := x;
j := [j+1];
m := m+1;
```

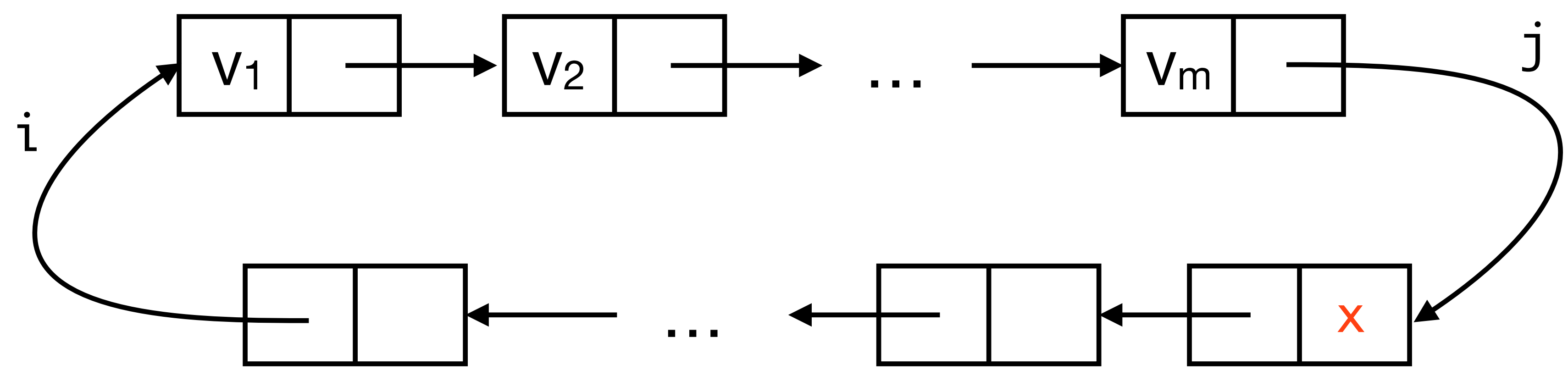


For reference: $\exists\, \beta.\ (\mathrm{lseg}\ \alpha\ (i,j) * \mathrm{lseg}\ \beta\ (j,i)) \wedge m = \#\alpha \wedge n = \#\alpha + \#\beta$

Warm-up: a cyclic buffer

Adding `x` to the buffer can be done by the code below (under the hypothesis that `n-m > 0`):

```
[j] := x;
j := [j+1];
m := m+1;
```

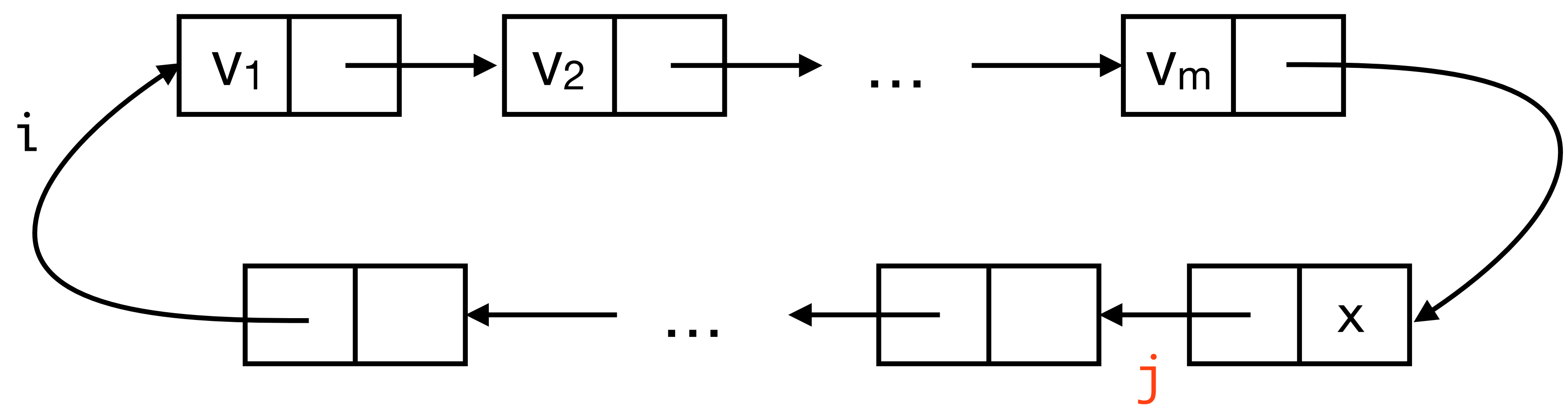


For reference: $\exists\, \beta.\ (\mathrm{lseg}\ \alpha\ (i,j) * \mathrm{lseg}\ \beta\ (j,i)) \wedge m = \#\alpha \wedge n = \#\alpha + \#\beta$

Warm-up: a cyclic buffer

Adding `x` to the buffer can be done by the code below (under the hypothesis that `n-m > 0`):

```
[j] := x;
j := [j+1];
m := m+1;
```

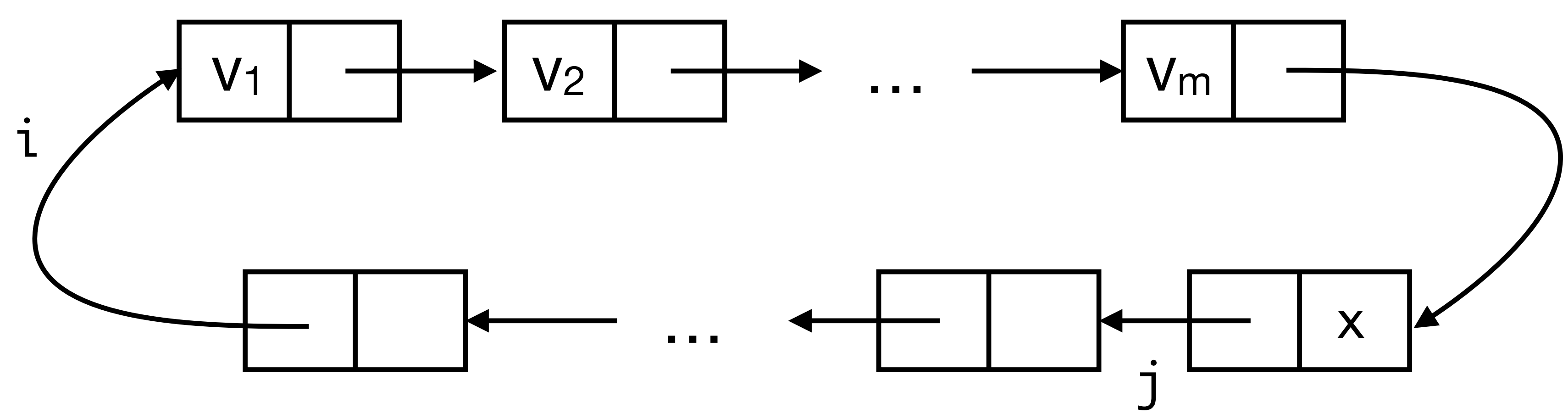


For reference: $\exists\, \beta.\ (\mathrm{lseg}\ \alpha\ (i,j) * \mathrm{lseg}\ \beta\ (j,i)) \wedge m = \#\alpha \wedge n = \#\alpha + \#\beta$

Warm-up: a cyclic buffer

Adding `x` to the buffer can be done by the code below (under the hypothesis that `n-m > 0`):

```
[j] := x;
j := [j+1];
m := m+1;
```



For reference: $\exists\, \beta.\ (\mathrm{lseg}\ \alpha\ (i,j) * \mathrm{lseg}\ \beta\ (j,i)) \wedge m = \#\alpha \wedge n = \#\alpha + \#\beta$

Warm-up: a cyclic buffer

Exercise: we prove that the code below inserts `x` in the buffer.

$\{ \exists\, \beta.\ (\mathrm{lseg}\ \alpha\ (\mathtt{i},\mathtt{j}) * \mathrm{lseg}\ \beta\ (\mathtt{j},\mathtt{i})) \wedge \mathtt{m} = \#\alpha \wedge \mathtt{n} = \#\alpha + \#\beta \wedge \mathtt{n\text{-}m} > 0 \}$

```
[j] := x;

j := [j+1];

m := m+1;
```

$\{ \exists\, \mathtt{l},\beta.\ (\mathrm{lseg}\ \alpha \cdot \mathtt{x}\ (\mathtt{i},\mathtt{j}) * \mathrm{lseg}\ \beta\ (\mathtt{j},\mathtt{i})) \wedge \mathtt{m} = \#\alpha \cdot \mathtt{x} \wedge \mathtt{n} = \#\alpha \cdot \mathtt{x} + \#\beta \}$

Warm-up: a cyclic buffer

Exercise: we prove that the code below inserts $\mathtt{x}$ in the buffer.

$\{\ \exists\, \beta.\ (\mathrm{lseg}\ \alpha\ (\mathtt{i},\mathtt{j}) * \mathrm{lseg}\ \beta\ (\mathtt{j},\mathtt{i})) \land \mathtt{m} = \#\alpha\ \land\ \mathtt{n} = \#\alpha + \#\beta \land \mathtt{n{-}m} > \mathtt{0}\ \}$

$\{\ \exists\, \mathtt{b},\beta.\ (\mathrm{lseg}\ \alpha\ (\mathtt{i},\mathtt{j}) * \mathrm{lseg}\ \mathtt{b}\cdot\beta\ (\mathtt{j},\mathtt{i})) \land \mathtt{m} = \#\alpha\ \land\ \mathtt{n} = \#\alpha + \#\mathtt{b}\cdot\beta\ \}$

$\{\ \exists\, \mathtt{k},\beta.\ (\mathrm{lseg}\ \alpha\ (\mathtt{i},\mathtt{j}) * \mathtt{j} \mapsto \_,\mathtt{k} * \mathrm{lseg}\ \beta\ (\mathtt{k},\mathtt{i})) \land \mathtt{m} = \#\alpha\ \land\ \mathtt{n{-}1} = \#\alpha + \#\beta\ \}$

```
[j] := x;
```

$\{\ \exists\, \mathtt{k},\beta.\ (\mathrm{lseg}\ \alpha\ (\mathtt{i},\mathtt{j}) * \mathtt{j} \mapsto \mathtt{x},\mathtt{k} * \mathrm{lseg}\ \beta\ (\mathtt{k},\mathtt{i})) \land \mathtt{m} = \#\alpha\ \land\ \mathtt{n{-}1} = \#\alpha + \#\beta\ \}$

$\{\ \exists\, \mathtt{k},\beta.\ \mathtt{j{+}1} \mapsto \mathtt{k} * (\mathrm{lseg}\ \alpha\ (\mathtt{i},\mathtt{j}) * \mathtt{j} \mapsto \mathtt{x} * \mathrm{lseg}\ \beta\ (\mathtt{k},\mathtt{i})) \land \mathtt{m} = \#\alpha\ \land\ \mathtt{n{-}1} = \#\alpha + \#\beta\ \}$

```
j := [j+1];
```

$\{\ \exists\, \mathtt{l},\beta.\ \mathtt{l{+}1} \mapsto \mathtt{j} * (\mathrm{lseg}\ \alpha\ (\mathtt{i},\mathtt{l}) * \mathtt{l} \mapsto \mathtt{x} * \mathrm{lseg}\ \beta\ (\mathtt{j},\mathtt{i})) \land \mathtt{m} = \#\alpha\ \land\ \mathtt{n{-}1} = \#\alpha + \#\beta\ \}$

$\{\ \exists\, \mathtt{l},\beta.\ (\mathrm{lseg}\ \alpha\ (\mathtt{i},\mathtt{l}) * \mathtt{l} \mapsto \mathtt{x},\mathtt{j} * \mathrm{lseg}\ \beta\ (\mathtt{j},\mathtt{i})) \land \mathtt{m} = \#\alpha\ \land\ \mathtt{n{-}1} = \#\alpha + \#\beta\ \}$

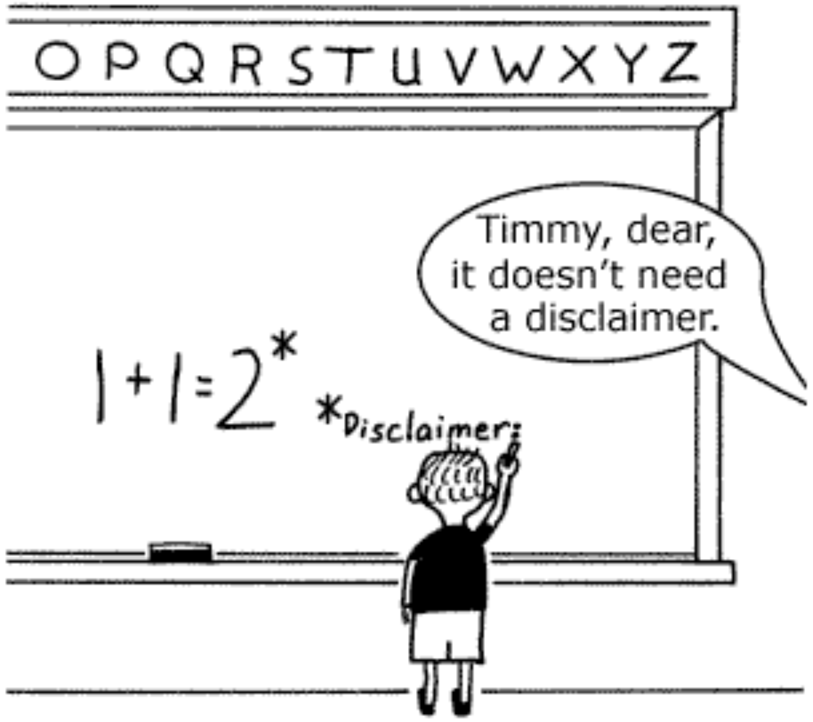
$\{\ \exists\, \mathtt{l},\beta.\ (\mathrm{lseg}\ \alpha\cdot\mathtt{x}\ (\mathtt{i},\mathtt{j}) * \mathrm{lseg}\ \beta\ (\mathtt{j},\mathtt{i})) \land \mathtt{m{+}1} = \#\alpha\cdot\mathtt{x}\ \land\ \mathtt{n} = \#\alpha\cdot\mathtt{x} + \#\beta\ \}$

```
m := m+1;
```

$\{\ \exists\, \mathtt{l},\beta.\ (\mathrm{lseg}\ \alpha\cdot\mathtt{x}\ (\mathtt{i},\mathtt{j}) * \mathrm{lseg}\ \beta\ (\mathtt{j},\mathtt{i})) \land \mathtt{m} = \#\alpha\cdot\mathtt{x}\ \land\ \mathtt{n} = \#\alpha\cdot\mathtt{x} + \#\beta\ \}$

Be careful

Despite the appearances...



…mastering separation logics takes time…

(after all, we are reasoning about the heap!)

Concurrent separation logic

1. threads that mind their own bussiness

Threads that mind their own bussiness

Imagine a program composed by two threads, one updates `[x]`, the other `[y]`:

```
[x] := 4      ||      [y] := 5
```

What can we prove about it?

Threads that mind their own bussiness

Imagine a program composed by two threads, one updates [x], the other [y]:

$$
\begin{array}{ccc}
\{ x \mapsto \_ \} & & \{ y \mapsto \_ \} \\
\texttt{[x] := 4} & \| & \texttt{[y] := 5} \\
\{ x \mapsto 4 \} & & \{ y \mapsto 5 \}
\end{array}
$$

1) We can give a (sequential) specification to each thread.

Threads that mind their own bussiness

Imagine a program composed by two threads, one updates [x], the other [y]:

$$\{\, x \mapsto \_ * y \mapsto \_ \,\}$$

$$\begin{array}{ccc} \{\, x \mapsto \_ \,\} & & \{\, y \mapsto \_ \,\} \\ \texttt{[x] := 4} & \parallel & \texttt{[y] := 5} \\ \{\, x \mapsto 4 \,\} & & \{\, y \mapsto 5 \,\} \end{array}$$

$$\{\, x \mapsto 4 * y \mapsto 5 \,\}$$

1) We can give a (sequential) specification to each thread.

2) If x and y do not point to the same location, then we can guarantee that the final state satisfies ($x \mapsto 4 * y \mapsto 5$) .

Parallel composition of non-interfering threads

$$\frac{\{\ P_1\ \}\ \ c_1\ \ \{\ Q_1\}\qquad\qquad \{\ P_2\ \}\ \ c_2\ \ \{\ Q_2\ \}}{\{\ P_1 * P_2\ \}\ \ c_1 \parallel c_2\ \ \{\ Q_1 * Q_2\ \}}$$

if $\mathrm{modifies}(c_1) \cap \mathrm{fv}(P_2) = \mathrm{modifies}(c_2) \cap \mathrm{fv}(P_1) = \varnothing$, and
$\mathrm{modifies}(c_1) \cap \mathrm{fv}(c_2) = \mathrm{modifies}(c_2) \cap \mathrm{fv}(c_1) = \varnothing$.

Parallel composition of non-interfering threads

$$\frac{\{ P_1 \}\ c_1\ \{ Q_1 \} \qquad \{ P_2 \}\ c_2\ \{ Q_2 \}}{\{ P_1 * P_2 \}\ c_1 \parallel c_2\ \{ Q_1 * Q_2 \}}$$

if $\mathrm{modifies}(c_1) \cap \mathrm{fv}(P_2) = \mathrm{modifies}(c_2) \cap \mathrm{fv}(P_1) = \varnothing$, and
$\mathrm{modifies}(c_1) \cap \mathrm{fv}(c_2) = \mathrm{modifies}(c_2) \cap \mathrm{fv}(c_1) = \varnothing$.

... apart from the (rare) moments of explicit communication, *processes are to be regarded as completely independent of each other...*

Parallel composition of non-interfering threads

Remark: the "proof figure" below

$$\{ x \mapsto \_ * y \mapsto \_ \}$$

$$\begin{array}{ccc} \{ x \mapsto \_ \} & & \{ y \mapsto \_ \} \\ [x] := 4 & \| & [y] := 5 \\ \{ x \mapsto 4 \} & & \{ y \mapsto 5 \} \end{array}$$

$$\{ x \mapsto 4 * y \mapsto 5 \}$$

is an annotation form for

$$\frac{\{ x \mapsto \_ \}\ [x] := 4\ \{ x \mapsto 4 \} \qquad \{ y \mapsto \_ \}\ [y] := 5\ \{ y \mapsto 5 \}}{\{ x \mapsto \_ * y \mapsto \_ \}\ [x] := 4 \parallel [y] := 5\ \{ x \mapsto 4 * y \mapsto 5 \}}$$

Example: parallel disposal of a tree

tree $p \equiv (p = \mathtt{nil} \wedge \text{empty}) \vee (\exists j, k.\ p \mapsto j * p{+}1 \mapsto k * \text{tree } j * \text{tree } k)$

```
procedure DispTree(p) {
   if p != nil then {
     a := [p];
     b := [p+1];
     DispTree(a) || DispTree(b);
     dispose(p+1);
     dispose(p);
}
```

This is an example of a shape predicate: it only describes the memory layout of the data structure, not the actual content.

This is a recursive procedure: to prove its correctness you can assume that the specification holds for the recursive calls. Cheating: in these lectures we won't formalise procedure calls...

This is a bad implementation of parallel disposal of a tree, why?

Exercise: assume that { tree `p` } `DispTree(p)` { empty } holds.

Prove that the body of `DispTree` satisfies { tree `p` } *body* { empty }.

Concurrent separation logic

2. synchronising racy threads


Racy programs

In current practice, most programs of interest are racy, e.g.:

$$
\begin{array}{ccc}
\{\, x \mapsto \_ \,\} & & \{\, x \mapsto \_ \,\} \\
\texttt{[x] := 10} & \| & \texttt{[x] := 20} \\
\{\, x \mapsto 10 \,\} & & \{\, x \mapsto 20 \,\}
\end{array}
$$

But we cannot send $x \mapsto \_$ to both threads:

$$(x \mapsto \_ * x \mapsto \_) \;\Leftrightarrow\; F$$

Racy programs

In current practice, most programs of interest are racy, e.g.:

$$\begin{array}{ccc} \{\, x \mapsto \_ \,\} & & \{\, x \mapsto \_ \,\} \\ [x] := 10 & \| & [x] := 20 \\ \{\, x \mapsto 10 \,\} & & \{\, x \mapsto 20 \,\} \end{array}$$

But we cannot send $x \mapsto \_$ to both threads:

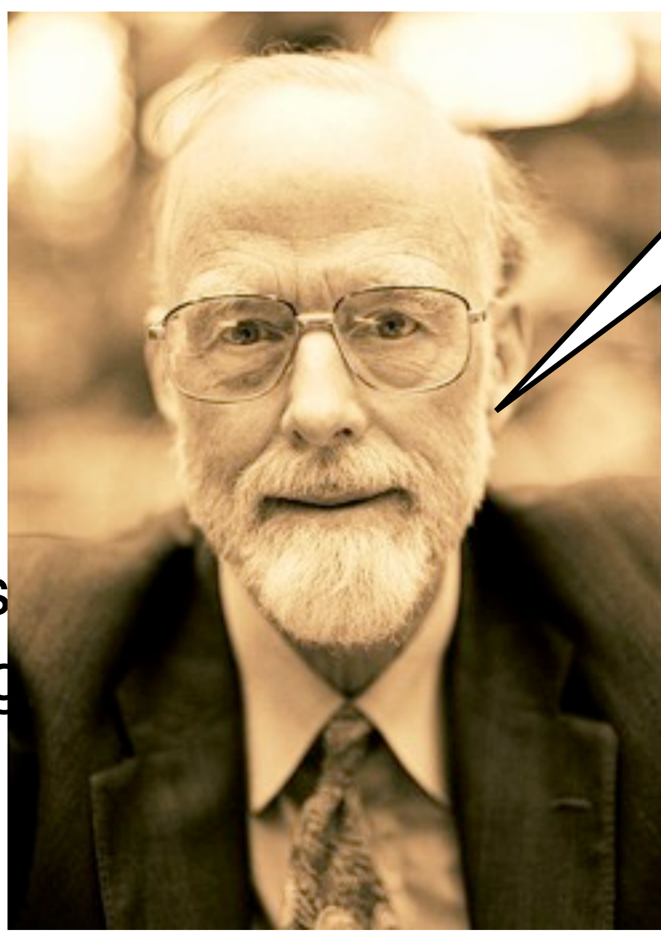
$$(x \mapsto \_ * x \mapsto \_) \Leftrightarrow F$$

This program does not have a race-free start state… to reason about such programs we must be explicit about the granularity of the interactions.

Racy programs

In current practice, most programs of interest are racy, e.g.:

$\{ x \mapsto \_ \} \qquad \{ x \mapsto \_ \}$



But _ to both threads:

$$(x \mapsto \_ * x \mapsto \_) \Leftrightarrow F$$

This ave a race-free start state… to reason about such

prog plicit about the granularity of the interactions.

Conditional critical regions (Hoare, 72)

A program is a collection of resources shared by concurrent threads:

```
init
resource r1 (list of variables) … resource rn (list of variables)
C1 || … || Cm
```

A thread can obtain an exclusive access to a resource:

```
with r when B do C
```

Constraints:

- if a variable belongs to a resource, it cannot appear in a parallel process except in a critical section for that resource;
- if a variable is changed in one process, it cannot appear in another unless it belongs to a resource.

Examples of racy programs

- if a variable belongs to a resource, it cannot appear in a parallel process except in a critical section for that resource;
- if a variable is changed in one process, it cannot appear in another unless it belongs to a resource.

These programs do not respect the constraints above:

- `x := 3` || `x := x + 1`
- `x := 3` || `with r when true do x := x + 1`

In general, races depend on aliasing:

- `[x] := 3` || `[y] := 4`

Concurrent separation logic will rule out all the races!

(Informal) semantics of CCRs

- The `init` command is executed first (and allocates some resources).

- A declaration

  ```
  resource r (x₁,…,xₙ)
  ```

  states that the variables $x_1,\ldots,x_n$ can only be accessed while holding the resource `r`.

- The command

  ```
  with r when B do C
  ```

  executes `C` while holding the resource `r`: no other thread can access the variables $x_1,\ldots,x_n$ while `C` executes. However the execution of `C` is postponed until the statement `B` is true.

Programming a bounded buffer with CCRs

Remark: to simplify notations we use arrays instead of pointers.

Buffer space and pointers are encapsulated in the *buffer* resource:

```
resource buffer (
  item pool[n];
  int count, in, out;
)
```

and here the producer and consumer code:

Producer:

```
with buffer when (count < n) {
  pool[in] = nextp;
  in = (in+1) % n;
  count++;
}
```

Consumer:

```
with buffer when (count > 0) {
  nextc = pool[out];
  out = (out+1) % n;
  count--;
}
```

Example: semaphores

We can program binary semaphores with CCRs:

```
s := 1;

resource s (s)

P(s)  =  with s when s = 1 do s := 0

V(s)  =  with s when s = 0 do s := 1
```

Remark: usually CCRs are implemented using semaphores, not the other way round. However CCRs are simpler from a logical point of view.

Can we associate some property (some invariant?) to a semaphore `s`?

Example: semaphores

Typical use of a semaphore `s`:

$$\begin{array}{lcl} \texttt{P(s)} & & \texttt{P(s)} \\ \texttt{[10] := 43} & \| & \texttt{[10] := 57} \\ \texttt{V(s)} & & \texttt{V(s)} \end{array}$$

The location `[10]` is protected by / associated to / (*owned by*) the semaphore.

Idea:

$$RI_s = (\texttt{s = 0} \wedge \text{empty}) \vee (\texttt{s = 1} \wedge \texttt{10} \mapsto \_\,)$$

when the semaphore is held, the location `[10]` is owned by the thread that holds the semaphore.

When the semaphore is not held, no thread can access the location `[10]`. (because no thread can have $\texttt{10} \mapsto \_$ in its precondition)

Axioms for CCRs

$$\frac{\{ P \}\ \texttt{init}\ \{ RI_1 * \ldots RI_n * P' \} \qquad \{ P' \}\ \texttt{C}_1 \parallel \ldots \parallel \texttt{C}_n\ \{ Q \}}{\{P\}\ \texttt{init; resource}\ \texttt{r}_1\ (\ldots)\ \ldots\ \texttt{resource}\ \texttt{r}_n\ (\ldots);\ \texttt{C}_1 \parallel \ldots \parallel \texttt{C}_n\ \{ RI_1 * \ldots RI_n * Q \}}$$

The init code allocates the resources stored in the resource invariants; the threads are then executed. Threads grab control of the resource invariants when entering the CCRs:

$$\frac{\{ (P * RI_r) \wedge \texttt{S} \}\ \texttt{C}\ \{ Q * RI_r \}}{\{ P \}\ \texttt{with r when S do C}\ \{ Q \}}$$

Idea: inside the critical region, the threads has visibility of the state associated to (protected by) the resource.

Example: semaphores

Exercise: suppose that

$$\mathrm{RI}_{\mathtt{s}} = (\mathtt{s} = \mathtt{0} \wedge \mathrm{empty}) \vee (\mathtt{s} = \mathtt{1} \wedge \mathtt{10} \mapsto \_\,)$$

Can we prove that the following holds?

$$\{\ \mathrm{empty}\ \}$$

$$\begin{array}{lcl} \mathtt{P(s)} & & \mathtt{P(s)} \\ \mathtt{[10]\ :=\ 43} & \| & \mathtt{[10]\ :=\ 57} \\ \mathtt{V(s)} & & \mathtt{V(s)} \end{array}$$

$$\{\ \mathrm{empty}\ \}$$

Example: semaphores

Reminder: $RI_s = (s = 0 \wedge \text{empty}) \vee (s = 1 \wedge 10 \mapsto \_)$

Zoom on the proof of thread 1:

{ empty }
 P(S)
$\{ 10 \longmapsto \_ \}$
 [10] := 43
$\{ 10 \longmapsto \_ \}$
 V(s)
{ empty }

$$\frac{\{ \text{empty} * s = 1 * RI_s \}\ s := 0\ \{ s = 0 * 10 \mapsto \_ * RI_s \}}{\{ \text{empty} \}\ \texttt{with s when s=1 do s := 0}\ \{ 10 \mapsto \_ \}}$$

$$\frac{\{ 10 \mapsto \_ * s = 0 * RI_s \}\ s := 1\ \{ s = 1 * RI_s \}}{\{ 10 \mapsto \_ \}\ \texttt{with s when s = 0 do s := 1}\ \{ \text{empty} \}}$$

Key observation: the resource $10 \mapsto \_$ "flows" from the RI to the thread and back!

Example: semaphores

Reminder: $RI_s = (s = 0 \wedge \text{empty}) \vee (s = 1 \wedge 10 \longmapsto \_)$

Since $s = 0$ the RI cannot hold any resource (empty).

Zoom on the proof of thread 1:

$$\frac{\{\text{empty} * s = 1 * RI_s\}\ s := 0\ \{s = 0 * 10 \longmapsto \_ * RI_s\}}{\{\text{empty}\}\ \texttt{with s when s=1 do s := 0}\ \{10 \longmapsto \_\}}$$

$\{\text{empty}\}$
`P(S)`
$\{10 \longmapsto \_\}$
`[10] := 43`
$\{10 \longmapsto \_\}$
`V(s)`
$\{\text{empty}\}$

The resource $10 \longmapsto \_$ gets into the scope of the thread.

$$\frac{\{10 \ \ldots\ 1 * RI_s\}}{\{10 \longmapsto \_\}\ \texttt{with s when s = 0 do s := 1}\ \{\text{empty}\}}$$

Key observation: the resource $10 \longmapsto \_$ "flows" from the RI to the thread and back!

Example: semaphores

Reminder: RI_s = (s = 0 ∧ empty) ∨ (s = 1 ∧ 10 ↦ _)

Zoom on the proof of thread 1:

{ empty }
P(S)
{ 10 ↦ _ }
[10] := 43
{ 10 ↦ _ }
V(s)
{ empty }

$$\frac{\{\ \text{empty} * \texttt{s = 1} * RI_s\ \}\ \texttt{s := 0}\ \{\ \texttt{s = 0} * \texttt{10} \mapsto \_ * RI_s\ \}}{\{\ \text{empty}\ \}\ \texttt{with s when s=1 do s := 0}\ \{\ \texttt{10} \mapsto \_\ \}}$$

$$\frac{\{\ \texttt{10} \mapsto \_ * \texttt{s = 0} * RI_s\ \}\ \texttt{s := 1}\ \{\ \texttt{s = 1} * RI_s\ \}}{\{\ \texttt{10} \mapsto \_\ \}\ \texttt{with s when s = 0 do s := 1}\ \{\ \text{empty}\ \}}$$

Key observation: the resource 10 ↦ _ "flows" from the RI to the thread and back!

Example: semaphores

Exercise: can you prove the triple below under the stronger invariant?

$$RI_s = (\mathtt{s = 0} \land \text{empty}) \lor (\mathtt{s = 1} \land \mathtt{10} \mapsto \mathtt{57})$$

{ empty }

```
P(s)            P(s)
[10] := 43  ||  [10] := 57
V(s)            V(s)
```

{ empty }

With the stronger invariant we cannot prove that the invariant holds after executing `V(s)`!

Remark: a dull specification?

$$\vdash \{\text{ empty }\}\ \texttt{P(s); [10]:=43; V(s)}\ \|\ \texttt{P(s); [10]:=57; V(s)}\ \{\text{ empty }\}$$

holds under the invariant $RI_s = (\texttt{s = 0} \wedge \text{empty}) \vee (\texttt{s = 1} \wedge \texttt{10} \mapsto \_\,)$.

This guarantees that if the program is executed in a state that satisifies { empty * RI_s }, if the program terminates, it ends in a state that satisfies { empty * RI_s }.

Even if the precondition and the postcondition does not look very interesting, the triple (also) guarantees that:

- the program is race-free;
  all the accesses to the shared resource were correctly protected by locks
- the resource invariant was preserved by all the threads;
- no memory leaks occurred.

Producer/consumer via semaphores

Two semaphores. Initially `free = 1` and `busy = 0`.

{ empty }

```
P(free)            P(busy)
[10] := 43    ||   x := [10]
V(busy)            V(free)
```

{ empty }

For `s` being either `free` or `busy`, the semaphore invariant is:

$$RI_s = (\mathtt{s = 0} \wedge \mathrm{empty}) \vee (\mathtt{s = 1} \wedge \mathtt{10} \mapsto \_\,)$$

Exercise: prove the triple above.

Producer/consumer via semaphores

Initially `free = 1` and `busy = 0`.

{ empty }

{ empty } { empty }

P(free)

{ 10 ↦ _

[10] :=

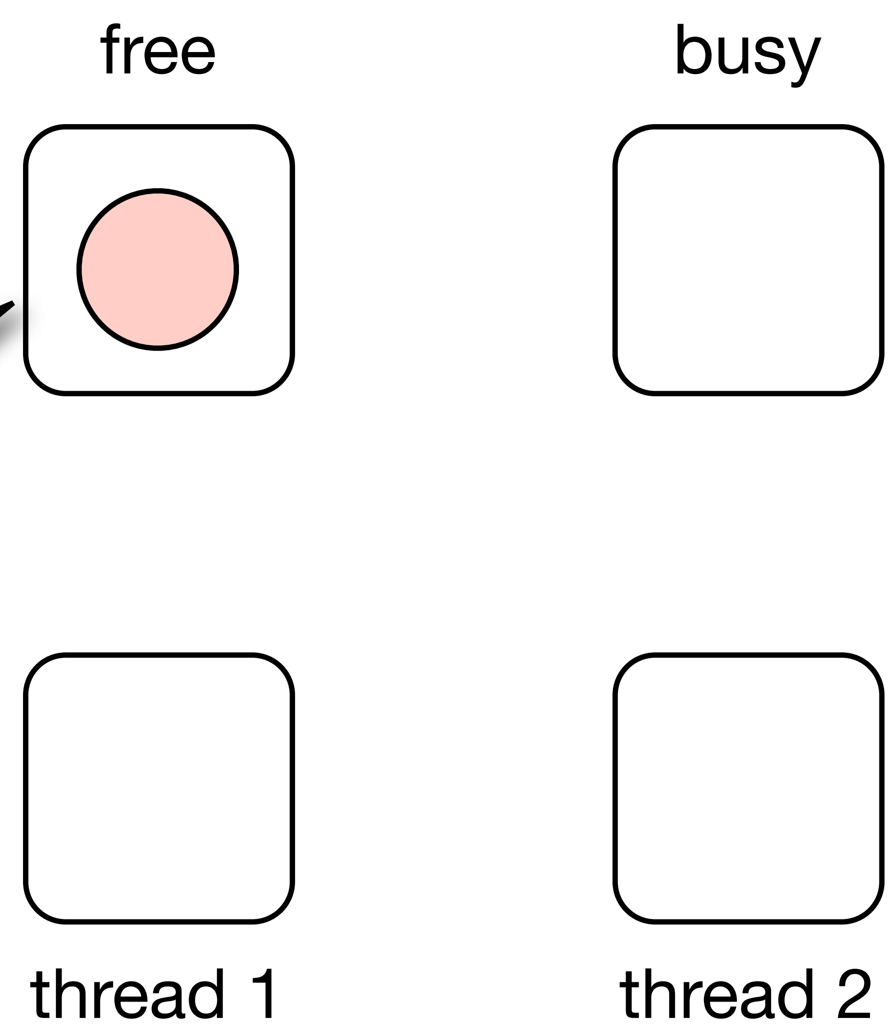
{ 10 ↦ _ } { 10 ↦ _ }

V(busy) V(free)

{ empty } { empty }

{ empty }

This diagram records the current owner of the ressource [10].



$RI_s = (s = 0 \wedge \text{empty}) \vee (s = 1 \wedge 10 \mapsto \_\,)$

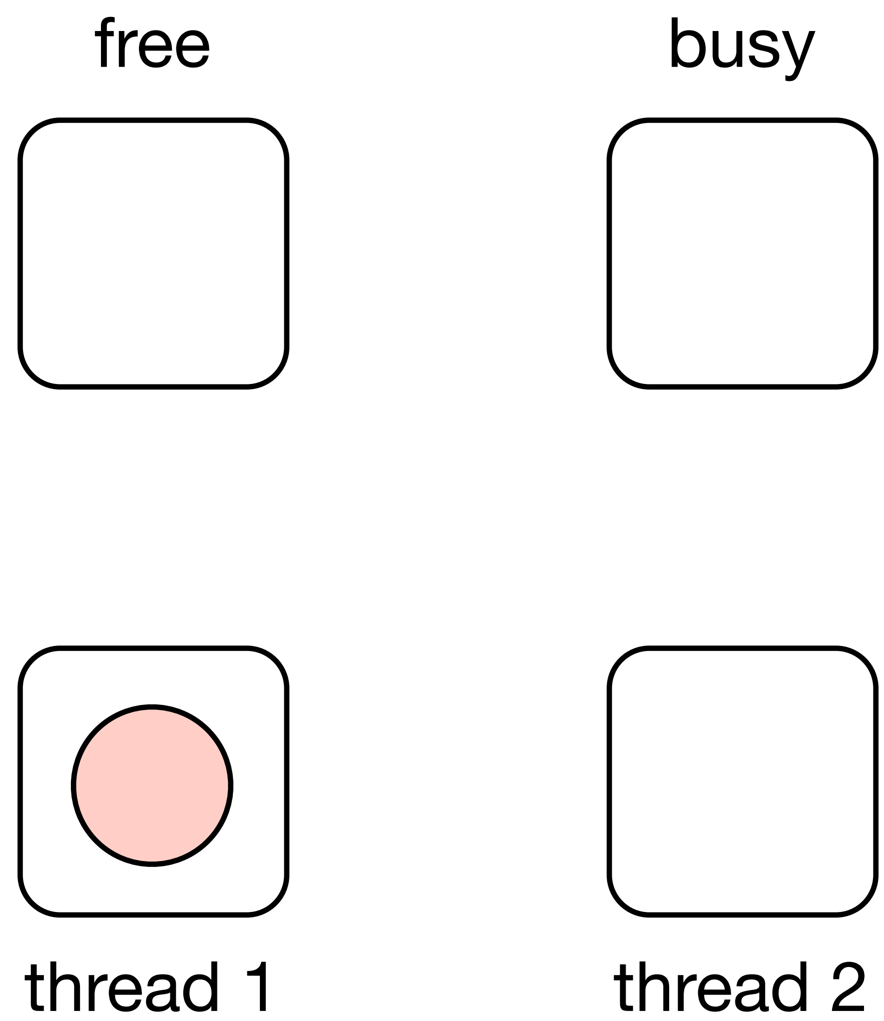
Producer/consumer via semaphores

Initially `free = 1` and `busy = 0`.

{ empty }

{ empty }		{ empty }
`P(free)`		`P(busy)`
{ $10 \mapsto \_$ }		{ $10 \mapsto \_$ }
`[10] := 43`	$\Vert$	`x := [10]`
{ $10 \mapsto \_$ }		{ $10 \mapsto \_$ }
`V(busy)`		`V(free)`
{ empty }		{ empty }

{ empty }

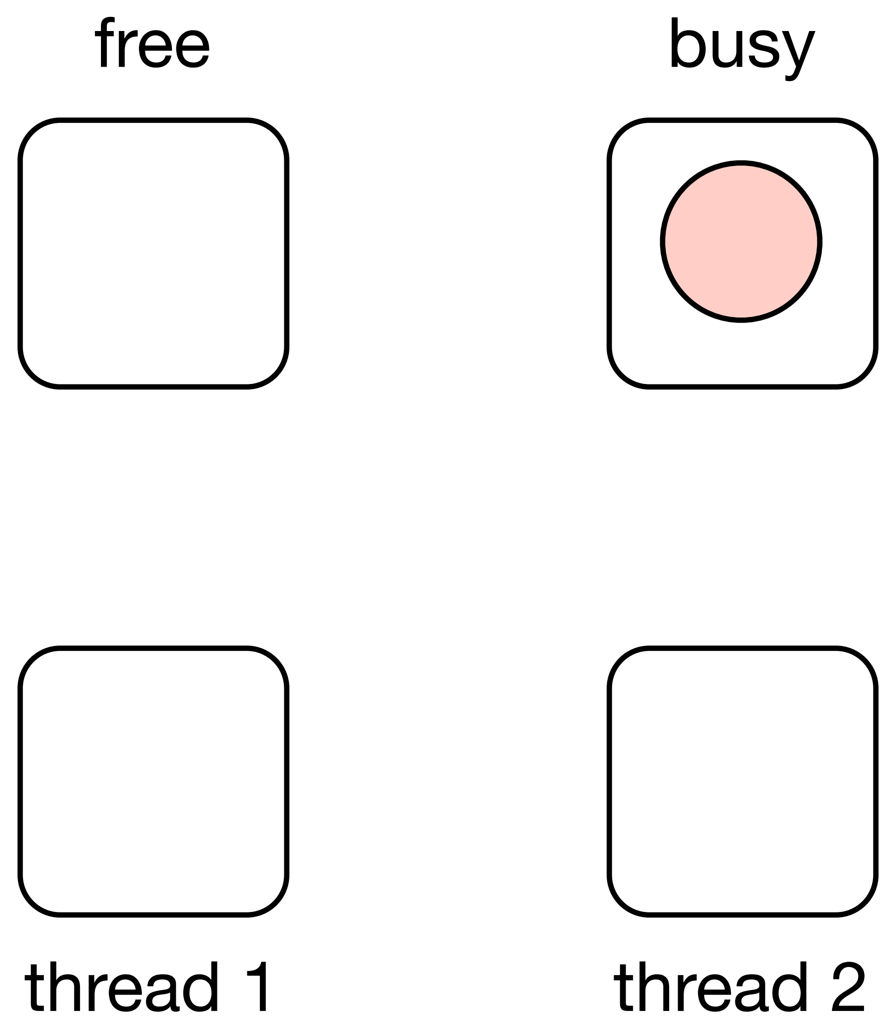


$\mathrm{RI}_s = (s = 0 \wedge \mathrm{empty}) \vee (s = 1 \wedge 10 \mapsto \_\,)$

Producer/consumer via semaphores

Initially `free = 1` and `busy = 0`.

```
            { empty }
{ empty }              { empty }
P(free)                P(busy)
{ 10 ↦ _ }             { 10 ↦ _ }
[10] := 43      ||     x := [10]
{ 10 ↦ _ }             { 10 ↦ _ }
V(busy)                V(free)
{ empty }              { empty }
            { empty }
```



$RI_s = (s = 0 \wedge \text{empty}) \vee (s = 1 \wedge 10 \mapsto \_)$

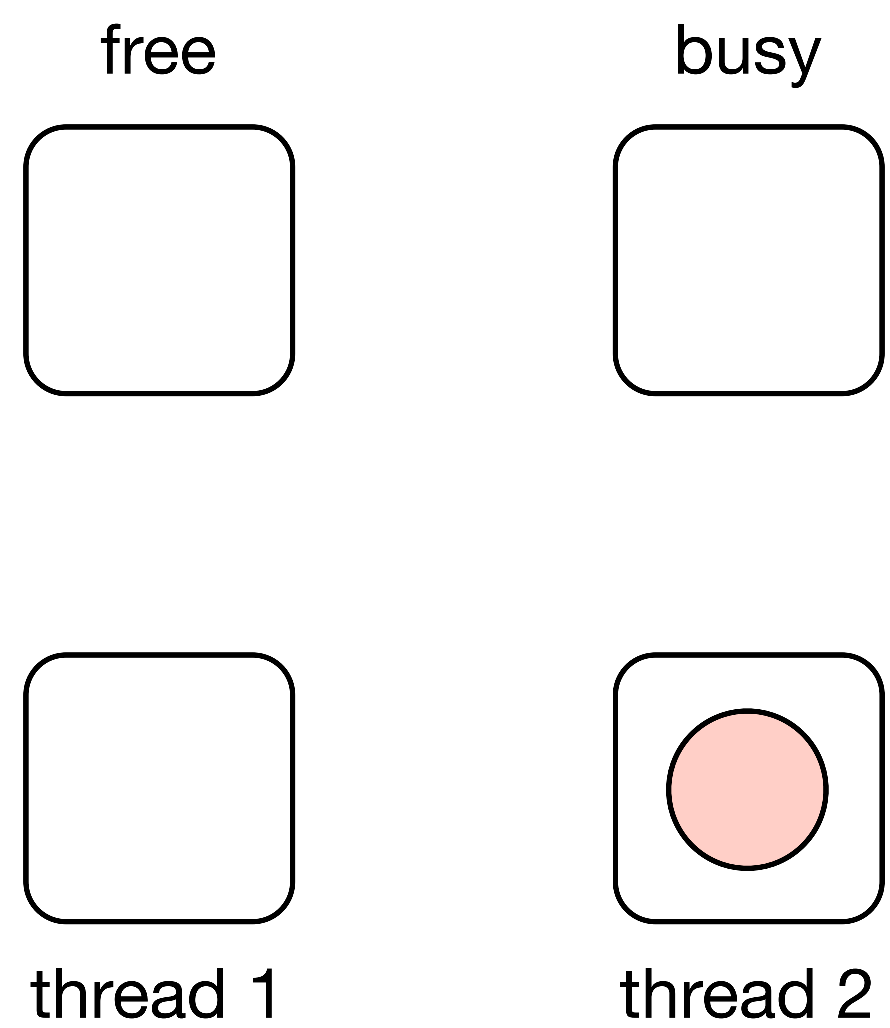
Producer/consumer via semaphores

Initially `free = 1` and `busy = 0`.

{ empty }

{ empty }		{ empty }
`P(free)`		`P(busy)`
{ $10 \mapsto \_$ }		{ $10 \mapsto \_$ }
`[10] := 43`	$\|$	`x := [10]`
{ $10 \mapsto \_$ }		{ $10 \mapsto \_$ }
`V(busy)`		`V(free)`
{ empty }		{ empty }

{ empty }



$RI_s = (s = 0 \wedge \text{empty}) \vee (s = 1 \wedge 10 \mapsto \_)$

Producer/consumer via semaphores

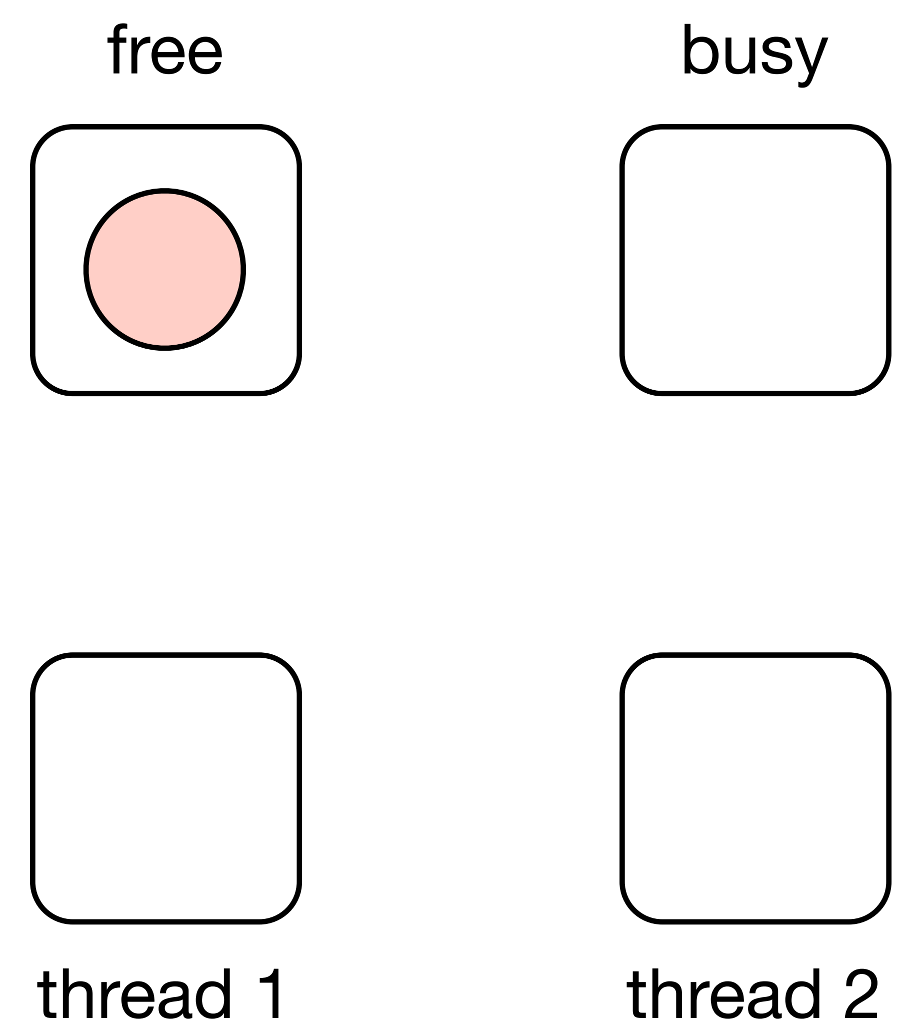
Initially `free = 1` and `busy = 0`.

{ empty }

{ empty } P(free) { $10 \mapsto \_$ } [10] := 43 { $10 \mapsto \_$ } V(busy) { empty } || { empty } P(busy) { $10 \mapsto \_$ } x := [10] { $10 \mapsto \_$ } V(free) { empty }

{ empty }		{ empty }
P(free)		P(busy)
{ $10 \mapsto \_$ }		{ $10 \mapsto \_$ }
[10] := 43	$\|$	x := [10]
{ $10 \mapsto \_$ }		{ $10 \mapsto \_$ }
V(busy)		V(free)
{ empty }		{ empty }

{ empty }



RI_s = (s = 0 ∧ empty) ∨ (s = 1 ∧ $10 \mapsto \_$)

Remarks

- Each semaphore invariant talks only about itself, not about other semaphores or processes.

- Each assertion within a process talks about only its own state, not the state of the other process or even the semaphores.

- We do not maintain `0 ≤ free + busy ≤ 1` as a global invariant.

- Semaphores are “logically attached” to resources. `P` and `V` are *ownership transformers*.

Example: a single-place buffer

Initially `full := false`.

```
resource buf(c, full)
```

Filling the buffer:

```
put (m)  =  with buf when ¬full do {
                c := m;
                full := true;
            }
```

passing a pointer, not the value.

Emptying the buffer:

```
get (n)  =  with buf when full do {
                n := c;
                full := false;
            }
```

Invariant: RI = (empty $\wedge$ ¬`full`) $\vee$ (`c` $\mapsto$ _ $\wedge$ `full`)

Example: a single-place buffer

RI = (empty ∧ ¬`full`) ∨ (`c` ↦ _ ∧ `full`)

{ empty }

```
full := false;
```

{ empty ∧ ¬`full` }

```
resource buf (c, full)
```

{ empty * empty * RI }

{ empty }

```
x := cons(3);
```

{ `x` ↦ `3` }

```
with buf when ¬full do
```

{ (`x` ↦ `3`)*(empty ∧ ¬`full`) }

```
  c := x; full := true;
```

{ `c` ↦ `3` * (empty ∧ `full`) }

{ empty * RI }

put(x)

||

{ empty }

```
with buf when full do
```

{ empty * ((`c` ↦ _) ∧ `full`) }

```
  y := c; full := false;
```

{ (`y` ↦ _) * (empty ∧ ¬`full`) }

```
dispose (y);
```

{ empty }

get(y)

{ empty * empty * RI }

{ RI }

Ownership *is in the eye of the asserter*

Transfer of ownership is not determined operationally:

whatever we transfer depends on what we want to prove.

In the last example ownership of the location allocated by thread 1 had to be transferred to thread 2, so that thread 2 could safely dispose it:

`x:=cons(3);`		`get(y);`
`put(x);`	‖	*use(y);*
		`dispose(y)`

Reminder: RI = (emp $\wedge$ $\neg$`full`) $\vee$ (`c` $\mapsto$ _ $\wedge$ `full`)

Ownership *is in the eye of the asserter*

Transfer of ownership is not determined operationally:

whatever we transfer depends on what we want to prove.

The code below is silly, but should be provable:

```
x:=cons(3);        get(y);
put(x);        ||
dispose(x);
```

Exercise: prove the code above, using the invariant

$$RI = (\mathrm{emp} \wedge \neg \mathtt{full}) \vee (\mathrm{emp} \wedge \mathtt{full})$$

Ownership *is in the eye of the asserter*

Transfer of ownership is not determined operationally:

whatever we transfer depends on what we want to prove.

However you won't be able to prove:

```
x:=cons(3);        get(y);
put(x);       ||   dispose(y);
dispose(x);
```

because ownership cannot flow both to thread 1 and thread 2.

This is fortunate: this program attempts to dispose the same pointer twice.

Simple exercises

- Is the triple below derivable?

```
{ empty }
  x := cons(3);
  z := cons(3);
  [x] := 4 || [z] := 5;
{ x ↦ 4 * z ↦ 5 }
```

- And this?

```
{ empty }
  x := cons(3);
  [x] := 4 || [x] := 5;
{ x ↦ _ }
```

- And this?

```
{ empty }
    x := 4 || x := 5;
{ empty }
```

- And this?

```
{ y = x+1 }
    x := 4 || y := y+1;
{ y = x+2 }
```

Simple exercises

- Is the triple below derivable?

{ empty }
`x := cons(3);`
`z := cons(3);`
`[x] := 4 || [z]`
{ $x \mapsto 4 * z \mapsto 5$ }

This is a stack race!

- And

Here the race is betwen `x := 4` and the proof of { `y = x+1` } `y := y+1` { `y = x+2` }

{
`x := cons(3);`
`[x] := 4 || [x] := 5;`
{ $x \mapsto \_$ }

- And this?

{ empty }
`x := 4 || x := 5;`
{ empty }

- And this?

{ `y = x+1` }
`x := 4 || y := y+1;`
{ `y = x+2` }

Simple exercises

- Is the triple below derivable?

```
{ empty }
  x := cons(3);
  z := cons(3);
  [x] := 4 || [z]
{ x ↦ 4 * z ↦ 5 }
```

This is a stack race!

- And

Here the race is betwen `x := 4` and the proof of
`{ y = x+1 } y := y+1 { y = x+2 }`

```
{
  x := cons(3);
  [x] := 4 || [x] := 5;
{ x ↦ _ }
```

- And this?

```
{ empty }
    x := 4 || x := 5;
{ empty }
```

- And this?

```
{ y = x+1 }
    x := 4 || y := y+1;
{ y = x+2 }
```

The logic forbids all kinds of races.

Exercise: parallel mergesort

Let ls(p) = (empty $\wedge$ `p = nil`) $\vee$ $\exists$ j. p $\longmapsto$ _ $*$ (p+1 $\longmapsto$ j) $*$ ls(j).

Suppose that the functions `split` and `merge` obey to the specifications

{ ls(p) } `split(r,p)` { ls(r) }

{ ls(p) * ls(q) } `merge(r,p,q)` { ls(r) }

Prove that:

{ ls(p) }

```
mergesort(r,p) {
    if p = Nil then r := p;
    else {
      split(q,p);
      mergesort(q1,q) || mergesort(p1,p);
      merge(r,p1,q1)
```

{ ls(r) }

Exercise: parallel mergesort

Let ls(p) = (empty $\wedge$ `p = nil`) $\vee$ $\exists$ j. $p \longmapsto \_ * (p{+}1 \longmapsto j) *$ ls(j).

Suppose th[at] the functions `split` and `merge` obey to the specifications

{ ls(p) } `split`

{ ls(p) * ls(q) }

This is another example of a shape predicate: it only describes the memory layout of the data structure, not the actual content.

Prove that:

{ ls(p) }

```
mergesort(r,p) {
```

```
p);
```

Concurrent separation logic is *decidable* for shape predicates.
(well, you still have to supply the loop invariants).

Check out the SmallFoot tool.

Exercise: parallel mergesort

Let ls(p) = (empty $\wedge$ `p = nil`) $\vee$ $\exists$ j. p $\longmapsto$ _ $*$ (p+1 $\longmapsto$ j) $*$ ls(j).

Suppose that the functions `split` and `merge` obey to the specifications

{ ls(p) } `split(r,p)` { ls(r) }

{ ls(p) * ls(q) } `merge(r,p,q)` { ls(r) }

Prove that:

{ ls(p) }

```
mergesort(r,p) {
    if p = Nil then r := p;
    else {
      split(q,p);
      mergesort(q1,q) || mergesort(p1,p);
      merge(r,p1,q1)
```

{ ls(r) }

Dynamic partitioning idioms

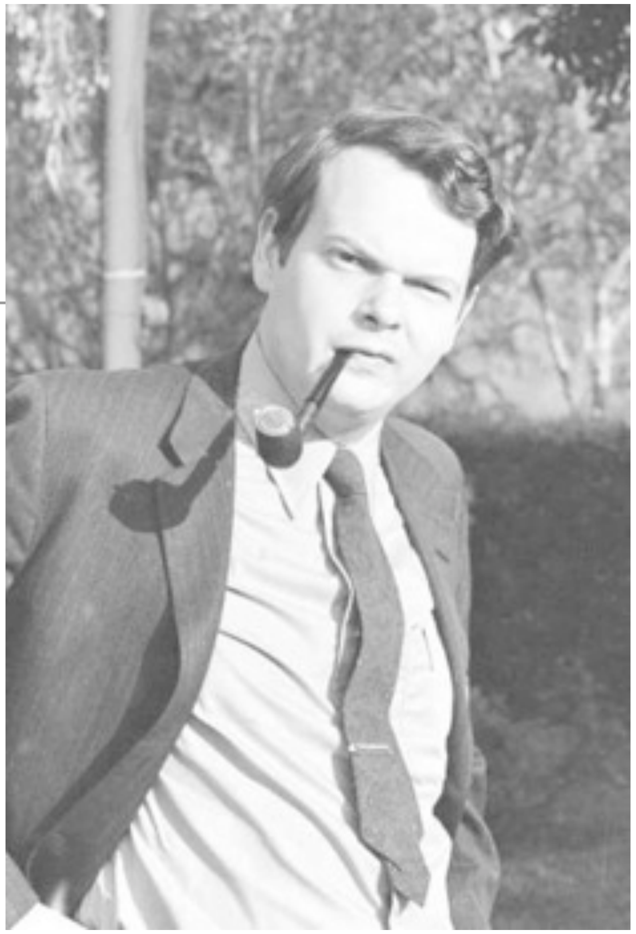
- Memory Managers, Thread Pools, Connection Pools;
- efficient Message Passing (copy avoiding);
- double-buffered I/O;
- many semaphore programs.

These idioms underlie much fundamental code: Microkernel OS designs, web servers, network packet processing, etc...

Old program design ideas, reflected in concurrent separation logic.

Question: are we done?

No: Reynolds counterexample



This logic is *inconsistent*! We can derive:

$$\{\, x \mapsto \_ \,\}\ \texttt{with r when true do skip}\ \{\, F \,\}$$

where the resource `r()` has invariant RI = T.

The triple states that the program diverges, while obviously it does not.

Exercise: can you find such derivation?

No: Reynolds counterexample

This logic is *inconsistent*! We can derive:

$$\{\, x \mapsto \_ \,\}\ \mathtt{with\ r\ when\ true\ do\ skip}\ \{\, \mathrm{F} \,\}$$

where the resource r ( ) has invariant RI = T.

Let one be a shorthand for $x \mapsto \_$. From:

$$\dfrac{\dfrac{\{\ T\ \}\ \text{skip}\ \{\ T\ \}}{\{\ (\text{emp} \lor \text{one}) * \text{True}\ \}\ \text{skip}\ \{\ \text{emp} * \text{True}\ \}}}{\{\ \text{emp} \lor \text{one}\ \}\ \text{with r when true do skip}\ \{\ \text{emp}\ \}}$$

we can derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\{\ \text{emp} \lor \text{one}\ \}\ \text{with … }\{\ \text{emp}\ \}}{\{\ \text{emp}\ \}\ \text{with … }\{\ \text{emp}\ \}}}{\{\ \text{emp} * \text{one}\ \}\ \text{with … }\{\ \text{emp} * \text{one}\ \}}}{\{\ \text{one}\ \}\ \text{with … }\{\ \text{one}\ \}} \qquad \dfrac{\{\ \text{emp} \lor \text{one}\ \}\ \text{with … }\{\ \text{emp}\ \}}{\{\ \text{one}\ \}\ \text{with … }\{\ \text{emp}\}}}{\dfrac{\{\ \text{one} \land \text{one}\ \}\ \text{with r when true do skip}\ \{\ \text{emp} \land \text{one}\ \}}{\{\ \text{one}\ \}\ \text{with r when true do skip}\ \{\ \text{false}\ \}}}$$

What the Reynolds counterexample implies

Trouble if you have all of:

$$\frac{\{\,P_1\,\}\ \mathtt{C}\ \{\,Q_1\,\} \qquad \{\,P_2\,\}\ \mathtt{C}\ \{\,Q_2\,\}}{\{\,P_1 \wedge P_2\,\}\ \mathtt{C}\ \{\,Q_1 \wedge Q_2\,\}} \qquad \frac{\{\,(P * RI_r) \wedge \mathtt{S}\,\}\ \mathtt{C}\ \{\,Q * RI_r\,\}}{\{\,P\,\}\ \mathtt{with\ r\ when\ S\ do\ C}\ \{\,Q\,\}}$$

$$\frac{\{\,P\,\}\ C\ \{\,Q\,\}}{\{\,P * R\,\}\ C\ \{\,Q * R\,\}}$$

The semantics of P * Q is nondeterministic:

$$\exists\ h_1, h_2.\ \mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \varnothing \wedge h_1 \oplus h_2 = h \wedge (s,h_1) \models P \wedge (s,h_2) \models Q$$

The resource invariant T does not precisely nail down the storage owned by the resource; it is ambiguous. And the connective * can be satisfied with different splittings.

What the Reynolds counterexample implies

Trouble if you have all of:

$$\frac{\{\, P_1 \,\}\ \mathtt{C}\ \{\, Q_1 \,\} \qquad \{\, P_2 \,\}\ \mathtt{C}\ \{\, Q_2 \,\}}{\{\, P_1 \wedge P_2 \,\}\ \mathtt{C}\ \{\, Q_1 \wedge Q_2 \,\}} \qquad\qquad \frac{\{\, (\, P * RI_r) \wedge \mathtt{S} \,\}\ \mathtt{C}\ \{\, Q * RI_r \,\}}{\{\, P \,\}\ \mathtt{with\ r\ when\ S\ do\ C}\ \{\, Q \,\}}$$

If we can nail down the storage owned more precisely, perhaps we can get around this problem...

The semantics of P * Q is nondeterministic:

$$\exists\, h_1, h_2.\ \mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \varnothing \wedge h_1 \oplus h_2 = h \wedge (s,h_1) \vDash P \wedge (s,h_2) \vDash Q$$

The resource invariant T does not precisely nail down the storage owned by the resource; it is ambiguous. And the connective * can be satisfied with different splittings.

Precise predicates

A predicate P is *precise* if for every state, there is at most one substate satisfying it. Formally:

P is *precise* if for all s, h, there exists at most one $h' \sqsubseteq h$ where $s,h' \models P$.

Examples of *imprecise predicates*:

$$\top \qquad\qquad 10 \mapsto \_ \lor 11 \mapsto \_$$

$$ls(x,y) = (x = y \land \mathrm{empty}) \lor (\exists x'.\, x \mapsto x' * ls(x',y))$$

Examples of *precise predicates*:

$$\mathrm{empty} \qquad 10 \mapsto \_ \qquad (\mathrm{empty} \land \neg \mathtt{full}) \lor (c \mapsto \_ \land \mathtt{full})$$

$$ls(x,y) = \text{ if } (x = y) \text{ then empty else } \exists x'.\, x \longmapsto x' * ls(x',y)$$

A sound separation logic

Consider concurrent separation logic with the restriction that

all the resource invariants are precise.

Notation: let Γ define all the resources, and let inv(Γ) *-conjunction of all the resource invariants.

Theorem:

If { P } c { Q } is provable, every finite computation from a state satisfying P * inv Γ,

- is *error free*; and
- ends in a state satisfying Q * inv Γ.

A sound separation logic

Consider concurrent separation logic with the restriction that

all the resource invariants are precise.

Notation: let Γ define all the resources, and let inv(Γ) *-conjunction of all the resource invariants.

Theorem:

If { P } `c` { Q } is provable, every finite computation from a state satisfying P * inv Γ,

- is *error free*; and
- ends in a state satisfying Q * inv Γ.

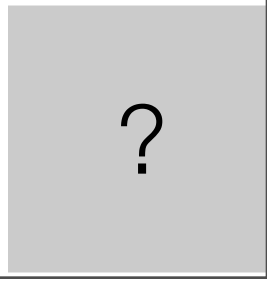
We are not done: we must define what is a *computation* of `c`.

Soundness of concurrent separation logic

Brookes proof

Alternative proofs:

- Vafeiadis, *Concurrent separation logic and operational semantics*
- Hayman, Winskel, *Independence and concurrent separation logic* (LICS 06)



Disclaimer

The purpose of the following section is only to give an overview of the proof of soundness of concurrent separation logic, to characterise what *error-free* means operationally, and discover which is the role of precise predicates.

A simpler and more elegant proof (which unfortunately does not prove that programs verified using CSL do not have data-races) can be found here:

Vafeiadis, *Concurrent separation logic and operational semantics.*

http://www.mpi-sws.org/~viktor/cslsound/

Brookes's semantic analysis: the big picture

1. The denotation of a command is a set of traces:
   - traces captures all the interactions of the command with an arbitrary state;
   - the trace *abort* captures a *race*.
2. An LTS defines the action of a trace on a state:
   - the LTS goes to the state *abort* if the trace performs an access outside of the domain of the state, or if the trace is *abort.*
3. Command C is race-free from state s, if for all traces $\alpha \in [[\mathtt{C}]]$, $\neg\ \mathtt{s} \stackrel{\alpha}{\Rightarrow} \mathtt{abort}$.
4. Intuition (but we'll need one more idea):

   if {P} `C` {Q}, then every *finite computation* of `C` from a state satisfying P * inv Γ, is race-free, and ends in a state satisfying Q * inv Γ.

1. denotation of commands

1. The denotation of a command is a set of traces

- traces captures all the interactions of a command with an arbitrary state:

e.g. $[\![$ `x := i+1` $]\!] = \{ i{=}v\ .\ x{:=}v{+}1 \mid v \in \text{Value} \}$.

- the trace *abort* captures a race:

e.g. $[\![$ `x := 1` $\|$ `x := 2` $]\!] = \{ x{:=}1\ .\ x{:=}2\ ,\ x{:=}2\ .\ x{:=}1,\ \text{abort} \}$

Special care required to define the denotation of || .

1. actions and traces

A command denotes a set of *traces*. A trace is a *sequence* of *actions*:

δ	idle
i=v, i:=v	read, write
[v]=v', [v]:=v'	lookup, update
alloc(v, L), disp(v)	allocate, dispose
try(r), acq(r), rel(r)	try, acquire, release
abort	race detected

λ ranges over actions. A trace can be finite or infinite. α, β range over traces.

Concatenation of traces is defined modulo:

$$\alpha.\delta.\beta = \alpha.\beta \qquad \alpha.\mathrm{abort}.\beta = \alpha.\mathrm{abort}$$

1. clauses (1)

$$\left.\begin{aligned}
&[\![\mathbf{skip}]\!] = \{\delta\}\\
&[\![i{:=}e]\!] = \{\rho\, i{:=}v \mid (\rho, v) \in [\![e]\!]\}\\
&[\![c_1; c_2]\!] = [\![c_1]\!]\,[\![c_2]\!]\\
&[\![\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2]\!] = [\![b]\!]_{\mathit{true}}\,[\![c_1]\!] \cup [\![b]\!]_{\mathit{false}}\,[\![c_2]\!]\\
&[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] = ([\![b]\!]_{\mathit{true}}\,[\![c]\!])^{*}\,[\![b]\!]_{\mathit{false}} \cup ([\![b]\!]_{\mathit{true}}\,[\![c]\!])^{\omega}
\end{aligned}\right\}\ \text{sequential constructs}$$

$$\left.\begin{aligned}
&[\![i{:=}[e]]\!] = \{\rho\, [v]{=}v'\, i{:=}v' \mid (\rho, v) \in [\![e]\!]\}\\
&[\![i{:=}\mathbf{cons}(E)]\!] = \{\rho\, \mathit{alloc}(l, L)\, i{:=}l \mid (\rho, L) \in [\![E]\!]\}\\
&[\![[e]{:=}e']\!] = \{\rho\, \rho'\, [v]{:=}v' \mid (\rho, v) \in [\![e]\!]\ \&\ (\rho', v') \in [\![e']\!]\}\\
&[\![\mathbf{dispose}\, e]\!] = \{\rho\, \mathit{disp}\, l \mid (\rho, l) \in [\![e]\!]\}
\end{aligned}\right\}\ \text{pointer operations}$$

1. clauses (2)

$$[\![\mathbf{with}\ r\ \mathbf{when}\ b\ \mathbf{do}\ c]\!] = wait^*\ enter\ \cup\ wait^\omega$$ synchronisation

where

$$wait = \{try\ r\}\ \cup\ acq\ r\ [\![b]\!]_{false}\ rel\ r$$

$$enter = acq\ r\ [\![b]\!]_{true}\ [\![c]\!]\ rel\ r$$

$$[\![c_1 \| c_2]\!] = [\![c_1]\!]_{\{\}} \|_{\{\}} [\![c_2]\!]$$ parallel composition

Key ideas:

1) processes start with no resources;
2) resources are mutually exclusive;
3) races produce abort.

1. clauses (3)

$$[\![c_1 \| c_2]\!] = [\![c_1]\!]_{\{\}} \|_{\{\}} [\![c_2]\!]$$

We rely on an auxiliary operator on traces:

- it builds all the traces obtained by interleaving the actions of the two threads
- in doing so, it keeps track of the resources allocated by each thread, and looks for data races.

Key ideas:

1) processes start with no resources;
2) resources are mutually exclusive;
3) races produce abort.

1. (3) resource enabling

Let A_1 and A_2 be sets of resources.

What a process can do depends on its resources and those of its environment. For that, we define the *resource enabling relation*:

$$(A_1, A_2) \xrightarrow{\lambda} (A_1', A_2)$$

Intuition: a process holding the resources A_1 can do λ in an environment that holds the resources A_2.

$$(A_1, A_2) \xrightarrow{acq\, r} (A_1 \cup \{r\}, A_2) \quad \text{if } r \notin A_1 \cup A_2$$

$$(A_1, A_2) \xrightarrow{rel\, r} (A_1 - \{r\}, A_2) \quad \text{if } r \in A_1$$

$$(A_1, A_2) \xrightarrow{\lambda} (A_1, A_2) \quad \text{if } r \neq acq\, r,\ rel\, r$$

1. (3) interference and interleaving

Two actions *interfere* if one write to a variable or a cell used by the other:

$\lambda_1 \bowtie \lambda_2$ iff $\mathit{free}(\lambda_1) \cap \mathit{writes}(\lambda_2) \neq \{\}$ or $\mathit{free}(\lambda_2) \cap \mathit{writes}(\lambda_1) \neq \{\}$

We can then define (fair, resource sensitive, race-detecting) interleaving:

$$\alpha\,{}_{A_1}\|_{A_2}\,\epsilon \;=\; \{\alpha \mid (A_1, A_2) \xrightarrow{\alpha} \cdot\}$$

$$\epsilon\,{}_{A_1}\|_{A_2}\,\alpha \;=\; \{\alpha \mid (A_2, A_1) \xrightarrow{\alpha} \cdot\}$$

$$\begin{aligned}
(\lambda_1\alpha_1)\,{}_{A_1}\|_{A_2}\,(\lambda_2\alpha_2) \;=\; & \\
& \{\lambda_1\beta \mid (A_1, A_2) \xrightarrow{\lambda_1} (A_1', A_2)\ \&\ \beta \in \alpha_1\,{}_{A_1'}\|_{A_2}\,(\lambda_2\alpha_2)\} \\
\cup\ & \{\lambda_2\beta \mid (A_2, A_1) \xrightarrow{\lambda_2} (A_2', A_1)\ \&\ \beta \in (\lambda_1\alpha_1)\,{}_{A_1}\|_{A_2'}\,\alpha_2\} \\
\cup\ & \{\mathit{abort} \mid \lambda_1 \bowtie \lambda_2\}
\end{aligned}$$

1. Examples

$[\![$ `x := 1` $\|$ `y := 1` $]\!] = \{$ x:=1 y:=1, y:=1 x:=1 $\}$

$[\![$ `x := 1` $\|$ `x := 1` $]\!] = \{$ x:=1 x:=1, abort $\}$

$[\![$ `with r do x := 1` $]\!]$ = (try r)* acq r x:=1 rel r $\cup$ (try r)$^\omega$

$[\![$ `with r do x := 1` $\|$ `with r do x := 2` $]\!]$ =

(try r)* acq r x:=1 rel r $\cup$ (try r)$^\omega$ $_{\{\}}\|_{\{\}}$ (try r)* acq r x:=2 rel r $\cup$ (try r)$^\omega$ =

(try r)* acq r (try r)* x:=1 (try r)* rel r (try r)* acq r x:=2 rel r

$\cup$ (try r)* acq r (try r)* x:=2 (try r)* rel r (try r)* acq r x:=1 rel r

$\cup$ (try r)* acq r (try r)* x:=1 (try r)* rel r (try r)$^\omega$

$\cup$ (try r)* acq r (try r)* x:=2 (try r)* rel r (try r)$^\omega$

$\cup$ (try r)$^\omega$

1. Examples

[[`x := 1` || `y := 1`]] = { x:=1 y:=1, y:=1 x:=1 }

[[`x := 1` || `x := 1`]] = { x:=1 x:=1, abort }

[[`with r do x := 1`]] = (try r)* acq r x:=1 rel r ∪ (try r)$^\omega$

[[`with r do x := 1` || `with r d`

here acq r by thread 2 is not enabled

(try r)* acq r x:=1 rel r ∪ (try r)$^\omega$ {}||{} (try r)* acq r x:=2 rel r ∪ (try r)$^\omega$ =

(try r)* acq r (try r)* x:=1 (try r)* rel r (try r)* acq r x:=2 rel r

∪ (try r)* acq r (try r)* x:=2 (try r)* rel r (try r)* acq r x:=1 rel r

∪ (try r)* acq r (try r)* x:=1 (try r)* rel r (try r)$^\omega$

∪ (try r)* acq r (try r)* x:=2 (try r)* rel r (try r)$^\omega$

∪ (try r)$^\omega$

1. Examples

[[`x := 1` || `y := 1`]] = { x:=1 y:=1, y:=1 x:=1 }

[[`x := 1` || `x := 1`]] = { x:=1 x:=1, abort }

[[`with r do x := 1`]] = (try r)* acq r x:=1 rel r ∪ (try r)$^{\omega}$

[[`with r do x := 1` || `with r d`

here acq r by thread 2 is not enabled

now, acr r by thread 2 is enabled

(try r)* acq r x:=1 rel r ∪ (try r) {}||{} (try r)* acq r x r)$^{\omega}$ =

(try r)* acq r (try r)* x:=1 (try r)* rel r (try r)* acq r x:=2 rel r

∪ (try r)* acq r (try r)* x:=2 (try r)* rel r (try r)* acq r x:=1 rel r

∪ (try r)* acq r (try r)* x:=1 (try r)* rel r (try r)$^{\omega}$

∪ (try r)* acq r (try r)* x:=2 (try r)* rel r (try r)$^{\omega}$

∪ (try r)$^{\omega}$

2. The action of a trace on a state

The state is store + heap + resource:

$$(s, h, A)$$

- global store: s : var ⇀ value ;
- global heap: h : loc ⇀ value ;
- resources A held by the process.

Actions cause *state change*, and either end in a new state, or abort.

$$(s, h, A) \xRightarrow{\alpha} (s', h', A')$$

$$(s, h, A) \xRightarrow{\alpha} \mathbf{abort}$$

2. the LTS that relates states and actions (1)

$$(s, h, A) \xRightarrow{\delta} (s, h, A)$$

$$(s, h, A) \xRightarrow{i=v} (s, h, A) \quad \mathit{if}\ (i, v) \in s$$

$$(s, h, A) \xRightarrow{i:=v} ([s \mid i : v], h, A) \quad \mathit{if}\ i \in \mathit{dom}\, s$$

$$(s, h, A) \xRightarrow{[v]=v'} (s, h, A) \quad \mathit{if}\ (v, v') \in h$$

$$(s, h, A) \xRightarrow{[v]:=v'} (s, [h \mid v : v'], A) \quad \mathit{if}\ v \in \mathit{dom}\, h$$

$$(s, h, A) \xRightarrow{alloc(v,[v_0,\ldots,v_n])} (s, [h \mid v : v_0, \ldots, v+n : v_n], A)$$

$$\mathit{if}\ v, v+1, \ldots, v+n \notin \mathit{dom}\, h$$

$$(s, h, A) \xRightarrow{disp\ v} (s, h \backslash v, A) \quad \mathit{if}\ v \in \mathit{dom}\, h$$

$$(s, h, A) \xRightarrow{acq\ r} (s, h, A \cup \{r\}) \quad \mathit{if}\ r \notin A$$

$$(s, h, A) \xRightarrow{rel\ r} (s, h, A - \{r\}) \quad \mathit{if}\ r \in A$$

$$(s, h, A) \xRightarrow{try\ r} (s, h, A)$$

2. the LTS that relates states and actions (2)

$$\textit{if } i \notin \textit{dom } s$$
$$(s,h,A) \xrightarrow{i=v} \textbf{abort}$$
$$(s,h,A) \xrightarrow{i:=v} \textbf{abort}$$

$$\textit{if } v \notin \textit{dom } h$$
$$(s,h,A) \xrightarrow{[v]=v'} \textbf{abort}$$
$$(s,h,A) \xrightarrow{[v]:=v'} \textbf{abort}$$
$$(s,h,A) \xrightarrow{\textit{disp } v} \textbf{abort}$$

$$(s,h,A) \xrightarrow{\textit{abort}} \textbf{abort}$$
$$\textbf{abort} \xrightarrow{\lambda} \textbf{abort}$$

A global computation is an *executable sequence of actions*:

$$(s,h,A) \xrightarrow{\alpha} (s',h',A')$$
$$(s,h,A) \xrightarrow{\alpha} \textbf{abort}$$

3. Error freedom

Definition: a command `C` is *error-free* if from (`s`,`h`) iff

$$\text{forall } \alpha \in [[C]].\ \neg\ ((s,h,\{\}) \overset{\alpha}{\Rightarrow} \text{abort})\ .$$

Example:

`dispose x` || `dispose y`

is error-free from all the states `s` such that $\neg(s(x) = s(y)) \wedge s(x),\ s(y) \in \text{dom}(h)$.

3.

Defi

Exa

dis

is er

4. A theorem?

It would be natural to define validity of { P } c { Q } as:

Theorem:

{ P } c { Q } if every *finite computation* of c from a state satisfying P * inv Γ,

1) is error free,

2) ends in a state satisfying Q * inv Γ.

Proof: It is natural to proceed by induction on the derivation of {P} c {Q}. But… can you prove the case where c is $c_1 \parallel c_2$ and the last rule is the rule for parallel composition?

4. A theorem? Not yet!

It would be natural to define validity of { P } c { Q } as:

Theorem:

{ P } c { Q } if every *finite computation* of c from a state satisfying P * inv Γ,

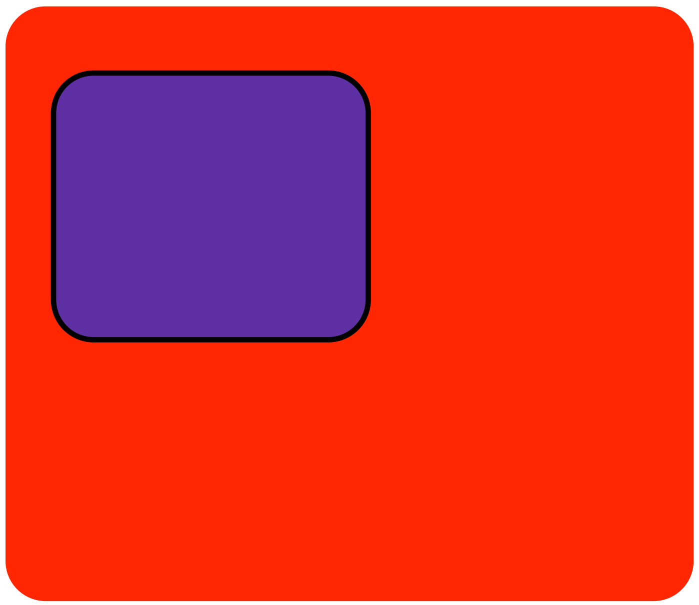
1) is error free,

2) ends in a state satisfying Q * inv Γ.

Proof: It is natural to proceed by induction on the derivation of {P} c {Q}. But… can you prove the case where c is $c_1 \parallel c_2$ and the last rule is the rule for parallel composition?

NO! This definition is not compositional: finite computations of c look at c in its entirety, and do not give enough informations about the computations performed by c_1 and c_2.

Compositionality



heap

C1 || C2

imagine C1 owns the blue part of the heap...

- C1 behaviour defines the evolution of the blue part of the heap;
- but all the red part is not constrained at all by C1, and might change under the influence of C2;
- if the semantics of C1 is defined in terms of the whole heap, it is tricky to derive it from the behaviour of C1 || C2…

Idea: define the semantics of the thread only in terms of the heap it owns!

Local computations

Idea: keep track of the *local state* of each thread. The local state is defined by:

$$(s, h, A)$$

subject the condition

$$\mathrm{dom}(s) \cap \mathrm{owned}(\Gamma) = \mathrm{owned}(\Gamma|A).$$

Local store only contains protected variables for which the process has resources.

A process starts with only non-critical data in its local state:

- local state grows when resource is acquired;
- local state shrinks when resource is released;
- error if program breaks design rules.

We can define another LTS, that captures the *local effects.*

Local effects: acquire and release

The rule for acq r imports into the local state the part of the stack and of the heap protected by r.

$$(s, h, A) \xrightarrow[\Gamma]{acq\, r} (s \cdot s', h \cdot h', A \cup \{r\})$$

$$\begin{array}{c} \textit{if } r(X){:}R \in \Gamma \\ \textit{and } s \perp s', h \perp h', \textit{dom}\, s' = X, \\ (s \cdot s', h') \models R \end{array}$$

The heap h' is uniquely determined because the invariant R is precise.

$$(s, h, A) \xrightarrow[\Gamma]{rel\, r} (s \backslash X, h - h', A - \{r\})$$

Similarly for rel r.

$$\begin{array}{c} \textit{if } r(X){:}R \in \Gamma \\ h' \subseteq h, \ (s, h') \models R \end{array}$$

Local effects: other transitions

All the other transitions are inherited from the global semantics. E.g. (excerpt):

$$(s, h, A) \xrightarrow[\Gamma]{\delta} (s, h, A)$$

$$(s, h, A) \xrightarrow[\Gamma]{i=v} (s, h, A) \qquad \mathit{if}\ i \in \mathit{dom}\ s$$

$$(s, h, A) \xrightarrow[\Gamma]{i:=v} ([s \mid i : v], h, A)$$

$$\mathit{if}\ i \in \mathit{dom}\ s - \mathit{free}(\Gamma \backslash A)$$

(in the last rule, the extra condition ensures that the variable being updated is does not belong to a resource).

Local effects: abort transitions

Two new abort rules (plus all the cases as in the global semantics):

- cannot update a variable protected by a resource not owned.

$$(s, h, A) \xrightarrow[\Gamma]{i:=v} \mathbf{abort}$$

$$\textit{if } i \in \textit{free}(\Gamma \backslash A) \textit{ or } i \notin \textit{dom } s$$

- when releasing a resource, the associated invariant must hold.

$$(s, h, A) \xrightarrow[\Gamma]{\textit{rel } r} \mathbf{abort}$$

$$\textit{if } r(X){:}R \in \Gamma$$
$$\textit{and } \forall h' \subseteq h.\ (s, h') \models \neg R$$

Local computations

$$(s, h, A) \xrightarrow[\Gamma]{\alpha} (s', h', A')$$

$$(s, h, A) \xrightarrow[\Gamma]{\alpha} \mathbf{abort}$$

Local computation captures what a *thread* sees of a computation.

Assumes that the environment:

- respects the resource rules;
- interferes only on synchronisation.

A local computation of put || (get ; dispose y)

$$\Gamma \;=\; buf(c, full) : (full \wedge c \mapsto \_) \vee (\neg full \wedge \mathbf{emp})$$

$$\begin{array}{rl}
 & ([x:v, y:_], [v:_], \{\}) \\
\xrightarrow[\Gamma]{acq\ buf} & ([x:v, y:_, full: false, c:_], [v:_], \{buf\}) \\
\xrightarrow[\Gamma]{full=false\ \mathrm{put}\ v} & ([x:v, y:_, full: true, c:v], [v:_], \{buf\}) \\
\xrightarrow[\Gamma]{rel\ buf} & ([x:v, y:_], [\,], \{\}) \\
\xrightarrow[\Gamma]{acq\ buf} & ([x:v, y:_, full: true, c:v], [v:_], \{buf\}) \\
\xrightarrow[\Gamma]{full=true\ \mathrm{get}\ v} & ([x:v, y:v, full: false, c:v], [v:_], \{buf\}) \\
\xrightarrow[\Gamma]{rel\ buf} & ([x:v, y:v], [v:_], \{\}) \\
\xrightarrow[\Gamma]{y=v\ \mathrm{disp}\ v} & ([x:v, y:v], [\,], \{\})
\end{array}$$

This can be decomposed into local computations of put and of get; dispose y...

The local computations of put and get || dispose y

put

$$([x:v],[v:\_],\{\})$$

$$\xrightarrow[\Gamma]{acq\ buf} ([x:v, full:false, c:\_],[v:\_],\{buf\})$$

$$\xrightarrow[\Gamma]{full=false\ \mathrm{put}\ v} ([x:v, full:true, c:v],[v:\_],\{buf\})$$

$$\xrightarrow[\Gamma]{rel\ buf} ([x:v],[\ ],\{\})$$

get || dispose y

$$([y:\_],[\ ],\{\})$$

$$\xrightarrow[\Gamma]{acq\ buf} ([y:\_, full:true, c:v],[v:\_],\{buf\})$$

$$\xrightarrow[\Gamma]{full=true\ \mathrm{get}\ v} ([y:v, full:false, c:v],[v:\_],\{buf\})$$

$$\xrightarrow[\Gamma]{rel\ buf} ([y:v],[v:\_],\{\})$$

$$\xrightarrow[\Gamma]{y=v\ \mathrm{disp}\ v} ([y:v],[\ ],\{\})$$

Validity

{ P } C { Q } is *valid* if every *finite local computation* of C from a state satisfying P * inv Γ, is 1) error free and 2) ends in a state satisfying Q * inv Γ.

Theorem: all provable formulas are valid.

Proof: uses local states and local effects, shows that each rule preserves validity, for parallel uses the parallel lemma:

- a local computation of $C_1 \| C_2$ decomposes into local computations of C_1 and C_2;
- A local error of $C_1 \| C_2$ is caused by a local error of C_1 or C_2 (not by interference);
- A successful local computation of $C_1 \| C_2$ is consistent with all successful local computations of C_1 and C_2.

Local vs. global

1. Soundness shows that *provable formulas* are *valid*;

2. *Validity* referes to *local* computations.

Need to connect local computations with conventional notions: global state, traditional partial correctness.

Theorem: Suppose $\alpha \in [\![c]\!],\ h = h_1 \cdot h_2,\ (s, h_2) \models inv(\Gamma)$.

1. If $(s, h) \overset{\alpha}{\Longrightarrow} \mathbf{abort}$ then $(s \backslash owned\,\Gamma, h_1) \xrightarrow[\Gamma]{\alpha} \mathbf{abort}$

2. If $(s, h) \overset{\alpha}{\Longrightarrow} (s', h')$ then $(s \backslash owned\,\Gamma, h_1) \xrightarrow[\Gamma]{\alpha} (s'_1, h'_1)$ where

$s'_1 = s' \backslash owned\,\Gamma$ and $\exists h'_2.\ h' = h'_1 \cdot h'_2\ \&\ (s', h'_2) \models inv(\Gamma)$

Corollary: validity implies error freedom

{ P } C { Q } if every *finite computation* of C from a state satisfying P * inv Γ,

1) is error free,

2) ends in a state satisfying Q * inv Γ.

Many concurrent separation logics?

The logic presented here is not entirely realistic, and is not as expressive as one might hope/desire/expect.



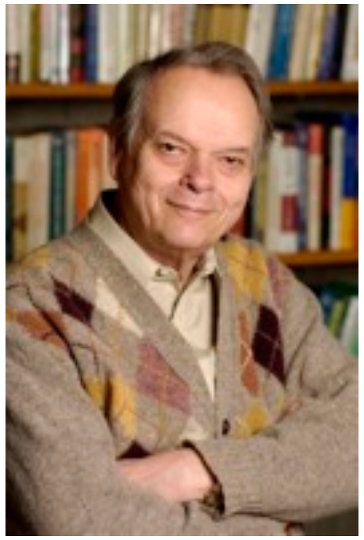

Several variants have been proposed, including:

- Gotsman, Berdine, Cook, Rinetzky and Sagiv (APLAS 07)
- … *plenty of other logics…*
- Hobor, Appel and me (ESOP 08)







Thanks to:

Stephen Brookes

John Reynolds

Tony Hoare

Edgser Dijkstra

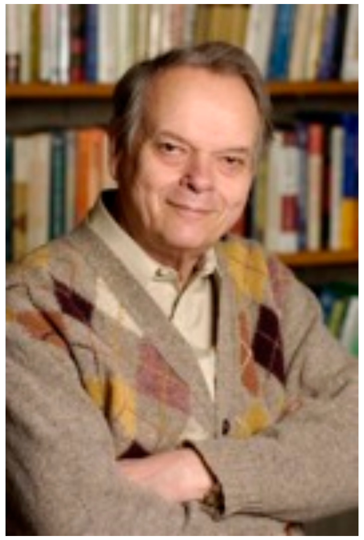
Peter O'Hearn

Per Brinch Hansen

Exercise: associate each picture with its owner...







Thanks to:

Stephen Brookes

John Reynolds

Tony Hoare

Edgser Dijkstra

Peter O'Hearn

Per Brinch Hansen

Exercise: associate each picture with its owner...

References:

Peter O'Hearn, *Resources, concurrency and local reasoning*;

Stephen Brookes, *A semantics for concurrent separation logic;*

Viktor Vafeiadis, *Concurrent separation logic and operational semantics*

all available from http://moscova.inria.fr/~zappa/teaching/mpri/2010/ .

Next lecture:

can we reason about racy programs?