Concurrent Separation Logic

Mike Dodds

slides: Matthew J. Parkinson, Alexey Gotsman

Concurrency

Concurrent:

"Running together in space, as parallel lines; going on side by side, as proceedings; occurring together, as events or circumstances; existing or arising together; conjoint"

- Oxford English Dictionary

Programming language

$C ::= \ldots \mid C \parallel C \mid \ldots$

Motivation

- Concurrency is hard:

  > "If you can get away with it, avoid using threads. Threads can be difficult to use, and they make programs harder to debug."
  >
  > Java Sun Tutorial "Threads and Swing"

- Multi-core means concurrency everywhere!

Testing is hard

"Testing concurrent software is hard. Even simple tests require invoking methods from multiple threads and worrying about issues such as timeouts and deadlock. Unlike in sequential programs, many failures are rare, probabilistic events and numerous factors can mask potential errors."

JavaOne Technical session

Testing is hard

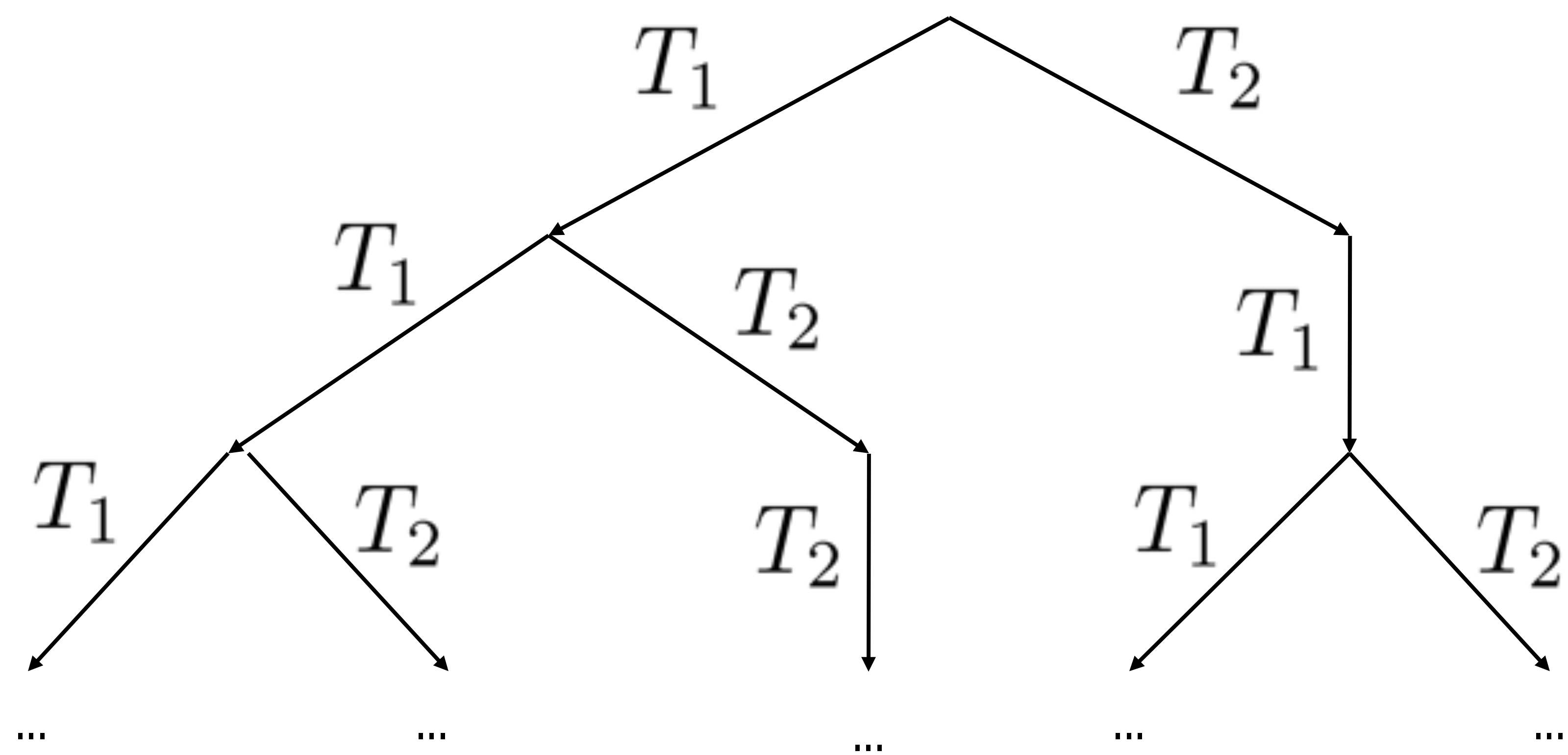
"Testing concurrent software is hard. Even simple tests require invoking methods from multiple threads and worrying about issues such as timeouts and deadlock. Unlike in sequential programs, many failures are rare, probabilistic events and numerous factors can mask potential errors."

JavaOne Technical session

Verification to the rescue?

Verifying concurrent programs is hard

Have to consider all possible interleavings:



Thread-modular reasoning

- Considers every thread in isolation under some assumption on its environment:

Thread-modular reasoning

- Considers every thread in isolation under some assumption on its environment:

Thread-modular reasoning

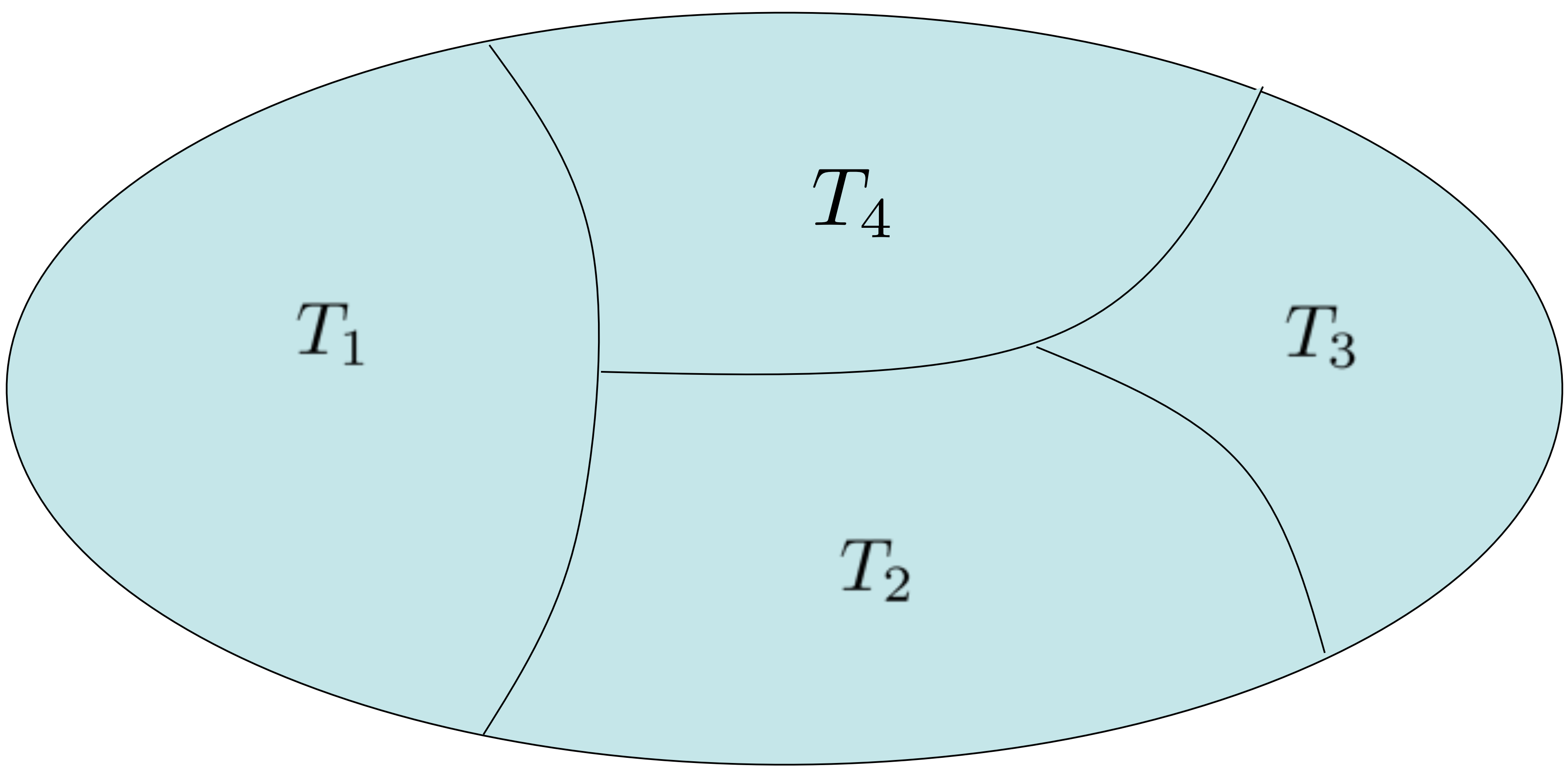
- Considers every thread in isolation under some assumption on its environment:

- Avoids direct reasoning about all interleavings

Disjoint Concurrency

Disjoint concurrency

- Language with parallel composition: $C_1 \parallel C_2$
- Every thread operates on its own part of the heap:



Parallel proof rule

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \quad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * P_2\}\ C_1 \parallel C_2\ \{Q_1 * Q_2\}}$$

variables used in C_1, P_1 and Q_1 not modified by C_2;
variables used in C_2, P_2 and Q_2 not modified by C_1

Parallel proof rule

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \quad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * P_2\}\ C_1 \parallel C_2\ \{Q_1 * Q_2\}}$$

variables used in C_1, P_1 and Q_1 not modified by C_2;
variables used in C_2, P_2 and Q_2 not modified by C_1

Parallel proof rule

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \quad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * P_2\}\ C_1 \parallel C_2\ \{Q_1 * Q_2\}}$$

variables used in C_1, P_1 and Q_1 not modified by C_2;
variables used in C_2, P_2 and Q_2 not modified by C_1

Parallel proof rule

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \quad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * P_2\}\ C_1 \parallel C_2\ \{Q_1 * Q_2\}}$$

variables used in C_1, P_1 and Q_1 not modified by C_2;
variables used in C_2, P_2 and Q_2 not modified by C_1

- Remember semantics of triples: C_1 accesses only the memory in P_1 and the one it allocates itself
- No way to mess up the heap owned by C_2!

Example

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

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

$$[x] := 3 \quad || \quad [y] := 4$$

$$\{ x \mapsto 3 \} \qquad \{ y \mapsto 4 \}$$

$$\{ x \mapsto 3 * y \mapsto 4 \}$$

Parallel Dispose tree

```
struct Tree {
   Tree *Left;
   Tree *Right;   }

disposetree(Tree *x) {
   if (x != NULL) {
      i = x->Left;
      j = x->Right;
      (disposetree(i) || disposetree(j) ||  free(x));
   }
}
```

Parallel Dispose tree

```
struct Tree {
  Tree *Left;
  Tree *Right;   }
```

$$\mathsf{Tree}(x) = (x = \mathsf{NULL} \wedge \mathsf{emp}) \vee (\exists i, j.\, x \mapsto i, j * \mathsf{Tree}(i) * \mathsf{Tree}(j))$$

```
{ tree(x) }
disposetree(Tree *x) {
  if (x != NULL) {
    i = x->Left;
    j = x->Right;
    (disposetree(i) || disposetree(j) ||  free(x));
  }
} { emp }
```

Parallel Dispose tree

$$\mathsf{Tree}(x) = (x = \mathsf{NULL} \wedge \mathsf{emp}) \vee$$
$$(\exists i, j.\, x \mapsto i, j * \mathsf{Tree}(i) * \mathsf{Tree}(j))$$

{ tree(x) ∧ x != NULL}

{ ∃i,j. tree(i) * tree(j) * x ↦ i,j }

i = x->Left;

{ ∃j. tree(i) * tree(j) * x ↦ i,j }

j = x->Right;

{ tree(i) * tree(j) * x ↦ i,j }

(disposetree(i) || disposetree(j) || free(x));

{ emp }

Example

{ tree(i) * tree(j) * x $\mapsto$ i,j }

{ tree(i) } { tree(j) } { x $\mapsto$ i,j }

disposetree(i) || disposetree(j) || dispose x .

{ emp } { emp } { emp }

{ emp * emp * emp }

{ emp }

Can we verify these?

$\{ \mathrm{emp} \}$
x := new;
z := new;
[x]:=4 || [z]:=5;
$\{x \longmapsto 4 * z \longmapsto 5\}$

$\{ \mathrm{emp} \}$
x:=4 || x:=5;
$\{ \mathrm{emp} \}$

$\{ \mathrm{emp} \}$
x := new;
[x]:=4 || [x]:=5;
$\{x \longmapsto \_\}$

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

Merge sort

```
mergesort(x, n)
  if n >1 then
    local m in
    m := n/2;
    mergesort(x,m) || mergesort(x+m,n-m);
    merge(x,m,n-m)
```

Merge sort

{ array(x,n) }
  mergesort(x, n)
{ sorted_array(x,n) }

{ sorted_array(x,m) * sorted_array(x+m,n) }
  merge(x,m,n)
{ sorted_array(x,m+n) }

Merge sort

```
{ array(x,n) }
mergesort(x, n)
  if n > 1 then
    local m in
    m := n/2;
    mergesort(x,m) || mergesort(x+m,n-m);
    merge(x,m,n-m)
{ sorted_array(x,n) }
```

Concurrent Separation Logic

Multiple access

How do we verify a program where several threads want access to the same memory? e.g.

$$[x] := 43 \quad || \quad [x] := 47$$

We protect shared values with locks

Multiple access

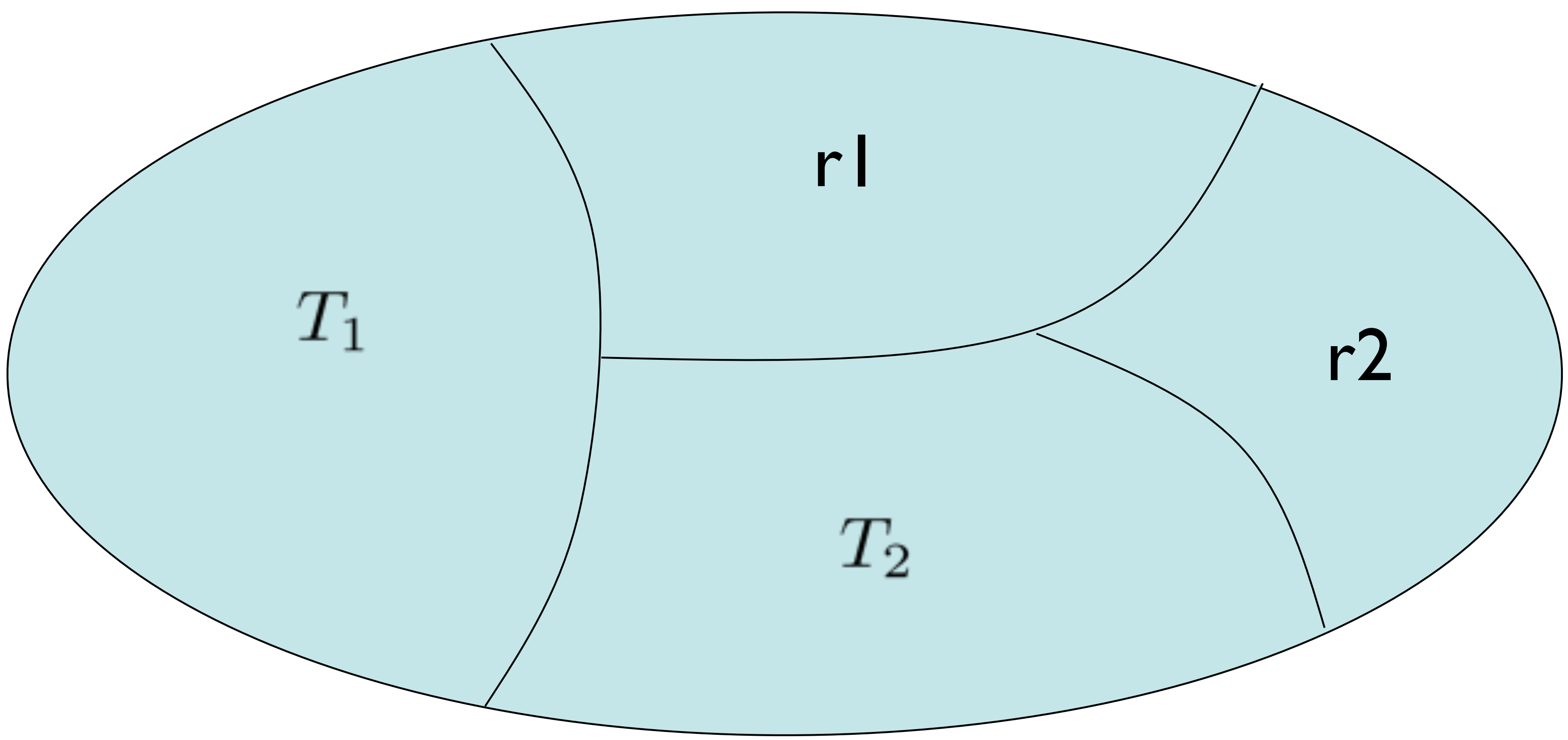
How do we verify a program where several threads want access to the same memory? e.g.

$$[x] := 43 \quad || \quad [x] := 47$$

We protect shared values with locks

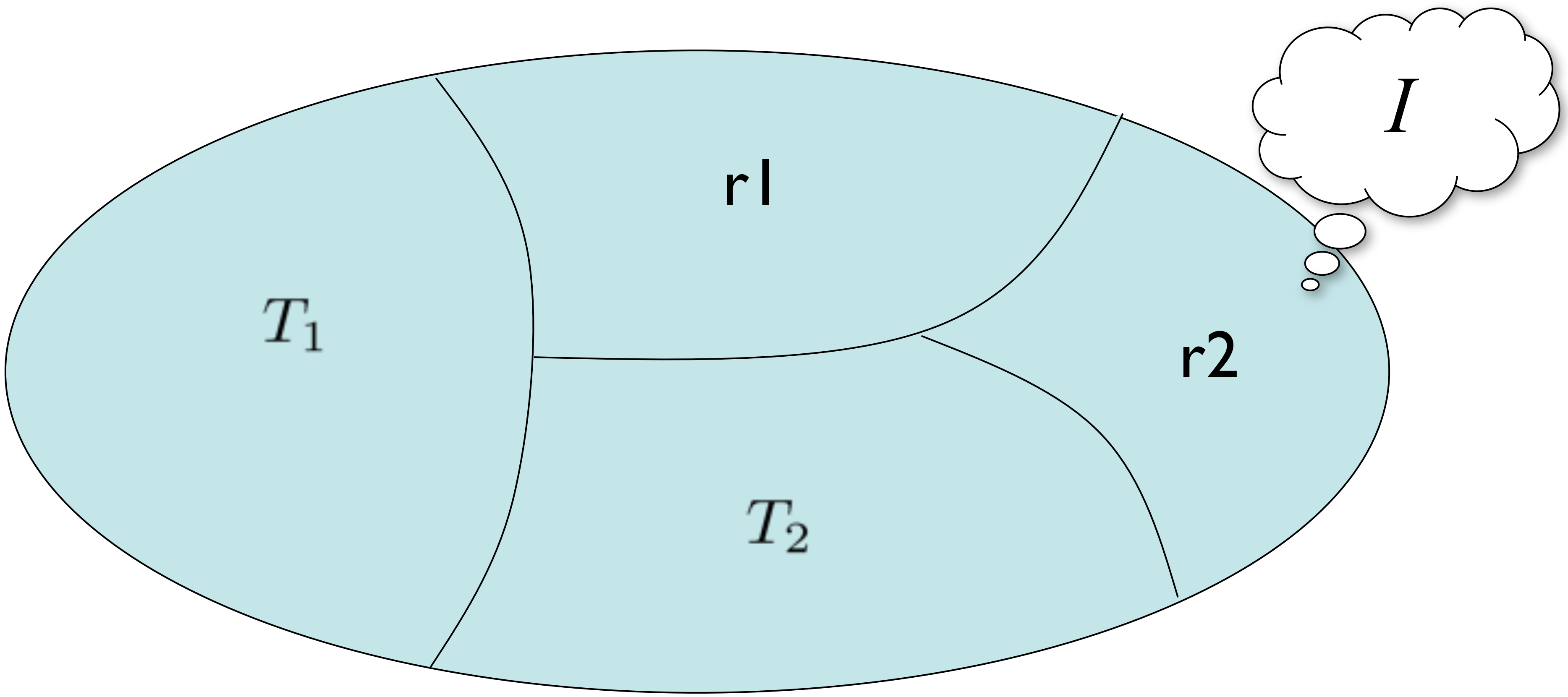
Reasoning principle

Separation property: at any time, the state of the program can be partitioned into that owned by each thread and each free lock



Reasoning principle

Separation property: at any time, the state of the program can be partitioned into that owned by each thread and each free lock



Assign a resource invariant I to every lock r: describes the part of the heap protected by the lock

Programming language

C ::= … | resource r in C | with r when B in C | …

Resource Rule

$$\frac{\Delta, r : I \vdash \{ P \}\ C\ \{ Q \}}{\Delta \vdash \{ P * I \}\ \text{resource}\ r\ \text{in}\ C\ \{ Q * I \}}$$

Lock Rule

$$\frac{\Delta \vdash \{ (P * I) \wedge B \}\ C\ \{ Q * I \}}{\Delta, r : I \vdash \{ P \}\ \text{with } r \text{ when } B \text{ in } C\ \{ Q \}}$$

Caveat: side-conditions

There are subtle variable side-conditions used to allow locks to refer to global variables.

Each variable is either associate to

- a single thread; or
- a single lock.

It can then only be modified and used in assertions by the thread, or while the thread holds the associate lock.

Binary Semaphore

We can encode a semaphore as a critical region

$$P(s) \;=\; \text{with } r_s \text{ when } s{=}1 \text{ do } s := 0$$
$$V(s) \;=\; \text{with } r_s \text{ when } s{=}0 \text{ do } s := 1$$

Resource invariant

$$(s{=}0 \wedge \mathrm{emp}) \quad \vee \quad (s{=}1 \wedge Q)$$

Initially,

$$s{=}0$$

Example

{ emp }
P(s)
[x] := 43
V(s)
{ emp }

$\|$

{ emp }
P(s)
[x] := 47
V(s)
{ emp }

Example

$\{ \text{emp} \}$

P(s)

$\{ x \mapsto \_ \}$

[x] := 43

$\{ x \mapsto \_ \}$

V(s)

$\{ \text{emp} \}$

Example

$\{ \text{emp} \}$
P(s)
$\{ x \mapsto \_ \}$
[x] := 43
$\{ x \mapsto \_ \}$
V(s)
$\{ \text{emp} \}$

$$\frac{\{ \text{emp} * (I_s \wedge s{=}1) \}\ s := 0\ \{ x \mapsto \_ * I_s \}}{\{\text{emp}\}\ \text{with } r_s \text{ when } s{=}1 \text{ do } s{:=}0\ \{ x \mapsto \_ \}}$$

Example

$\{ \mathrm{emp} \}$
P(s)
$\{ x \mapsto \_ \}$
[x] := 43
$\{ x \mapsto \_ \}$
V(s)
$\{ \mathrm{emp} \}$

$$\frac{\{ \mathrm{emp} * (I_s \wedge s=1)\}\ s := 0\ \{ x \mapsto \_ * I_s \}}{\{\mathrm{emp}\}\ \text{with } r_s \text{ when } s=1 \text{ do } s:=0\ \{ x \mapsto \_ \}}$$

$$\frac{\{ x \mapsto \_ * (I_s \wedge s=0)\}\ s := 1\ \{ \mathrm{emp} * I_s \}}{\{x \mapsto \_\}\ \text{with } r_s \text{ when } s=0 \text{ do } s:=1\ \{ \mathrm{emp} \}}$$

One place buffer

full := false

```
with buff when full do
    full := false
    y := c
dispose y
```

||

```
x := new
with buff when ¬full do
    full := true;
    c := x;
```

One place buffer

Resource Invariant

full := false

$\{ (\mathit{full} \wedge c \mapsto \_) \ \vee \ (\neg \mathit{full} \wedge \mathit{emp}) \}$

```
with buff when full do
    full := false
    y := c
dispose y
```

||

```
x := new
with buff when ¬full do
    full := true;
    c := x;
```

One place buffer

$\{ (\mathrm{full} \wedge c \mapsto \_) \vee (\neg\mathrm{full} \wedge \mathrm{emp}) \}$

with buff when full do

$\{ \mathrm{full} \wedge c \mapsto \_ \}$

full := false

y := c

$\{ (\neg\mathrm{full} \wedge \mathrm{emp}) * y \mapsto \_ \}$

$\{ y \mapsto \_ \}$

dispose y

$\|$

x := new

$\{ x \mapsto \_ \}$

with buff when ¬full do

$\{ (\neg\mathrm{full} \wedge \mathrm{emp}) * x \mapsto \_ \}$

full := true;

c := x;

$\{ \mathrm{full} \wedge c \mapsto \_ \}$

Ownership is in the eye of the assertor

full := false

with buff when full do
    full := false
    y := c

||

x := new
with buff when ¬full do
    full := true;
    c := x;
dispose x

Can we verify the following?

Ownership is in the eye of the assertor

Resource Invariant

full := false

{ emp }

with buff when full do
    full := false
    y := c

||

x := new
with buff when ¬full do
    full := true;
    c := x;
dispose x

Can we verify the following?

next time:
Semantics