Concurrent Separation Logic

Mike Dodds

slides: Matthew J. Parkinson, Alexey Gotsman

This lecture

- The problems of concurrency
- Disjoint concurrency
- Concurrent separation logic

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 ::= … | C || C | …

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

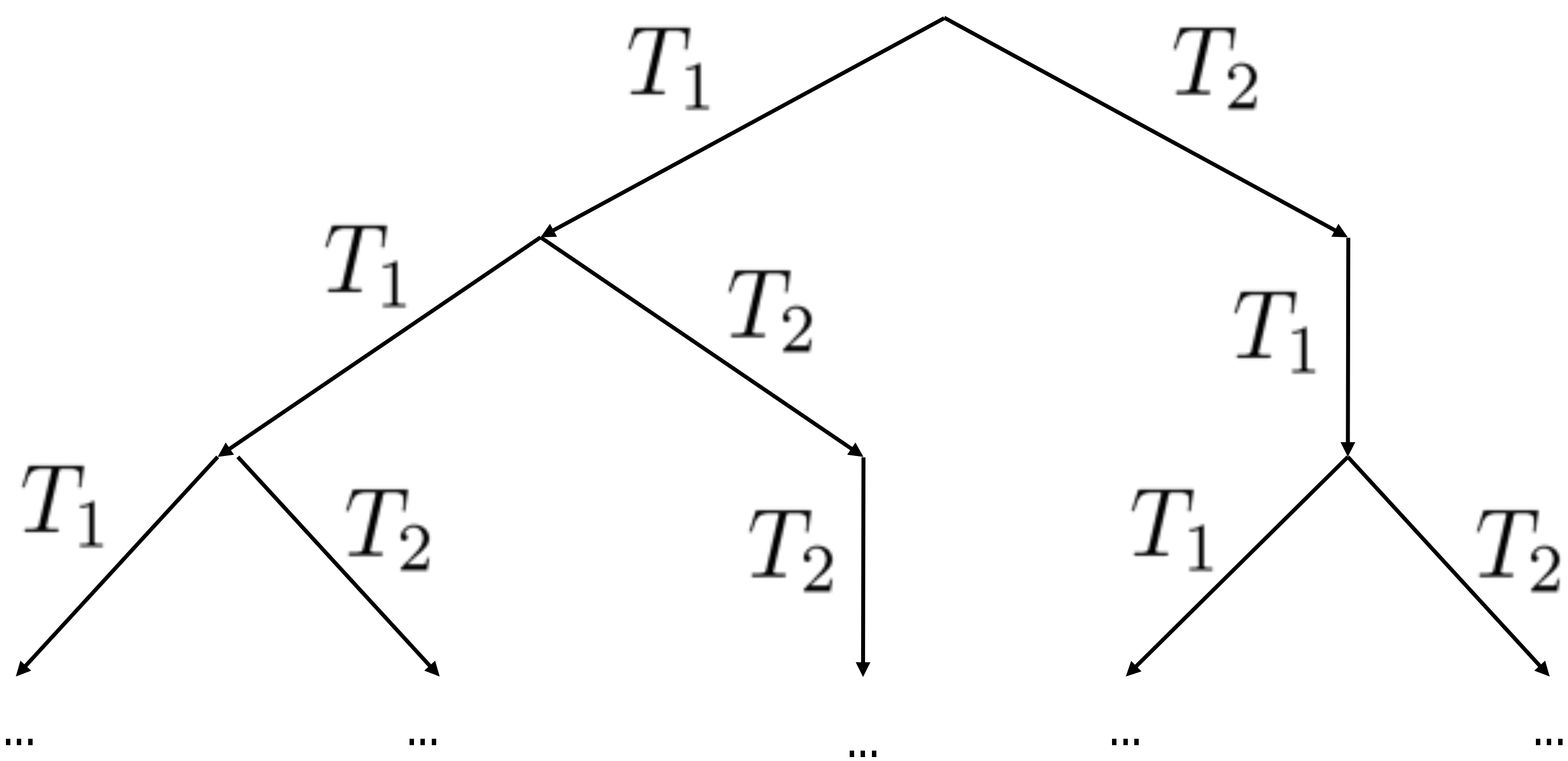
"Testing concurrent software is hard. Even simple tests require invoking methods from multiple threads and worrying about issues such as timeouts and deadlock. Unlike in sequential programs, many failures are rare, probabilistic events and numerous factors can mask potential errors."

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

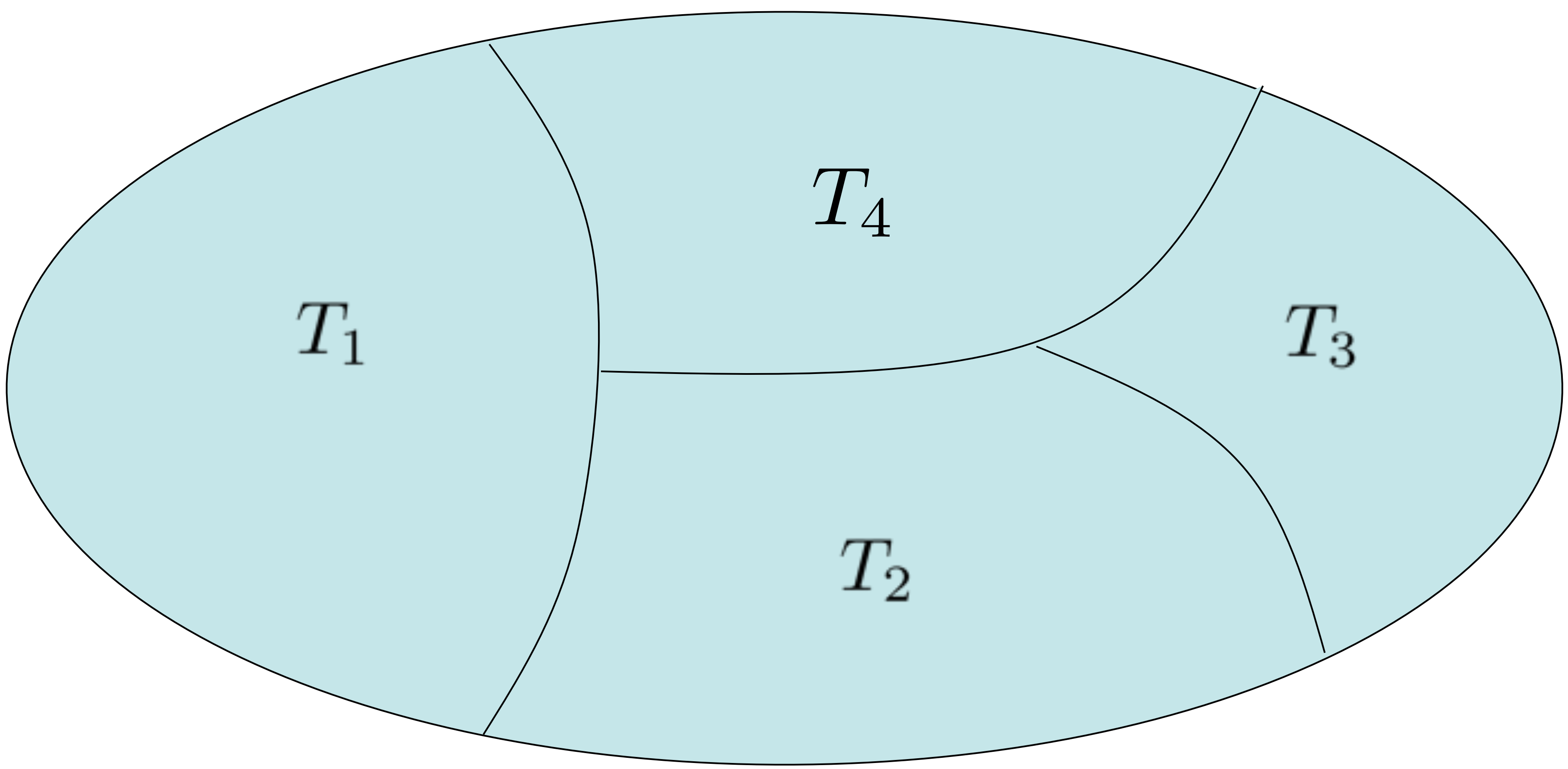
- Considers every thread in isolation under some assumption on its environment:

- 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

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

```
{ 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?

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

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

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

$\{ y = x+1 \}$
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 || [x] := 47

We protect shared values with locks

Multiple access

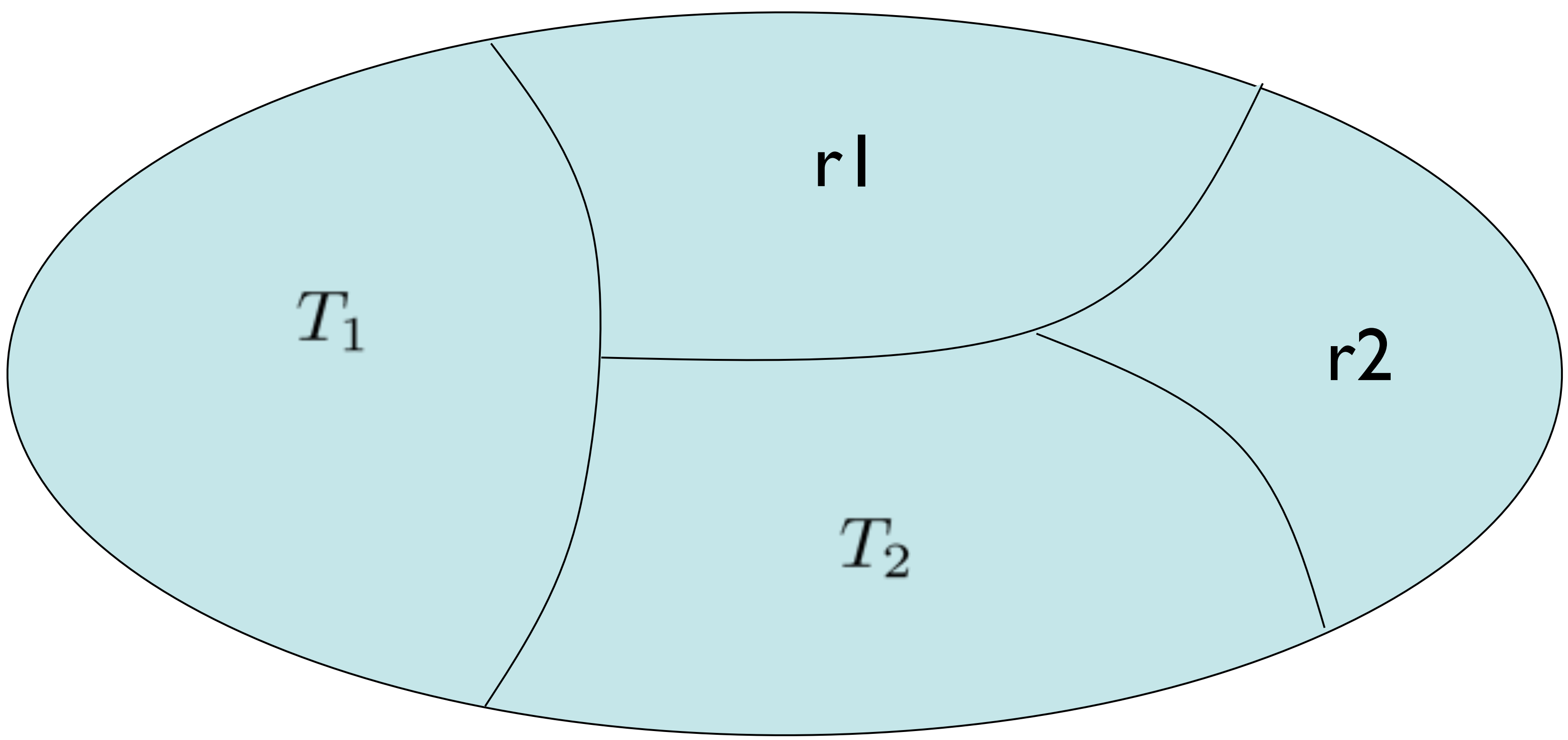
How do we verify a program where several threads want access to the same memory? e.g.

[x] := 43 || [x] := 47

We protect shared values with locks

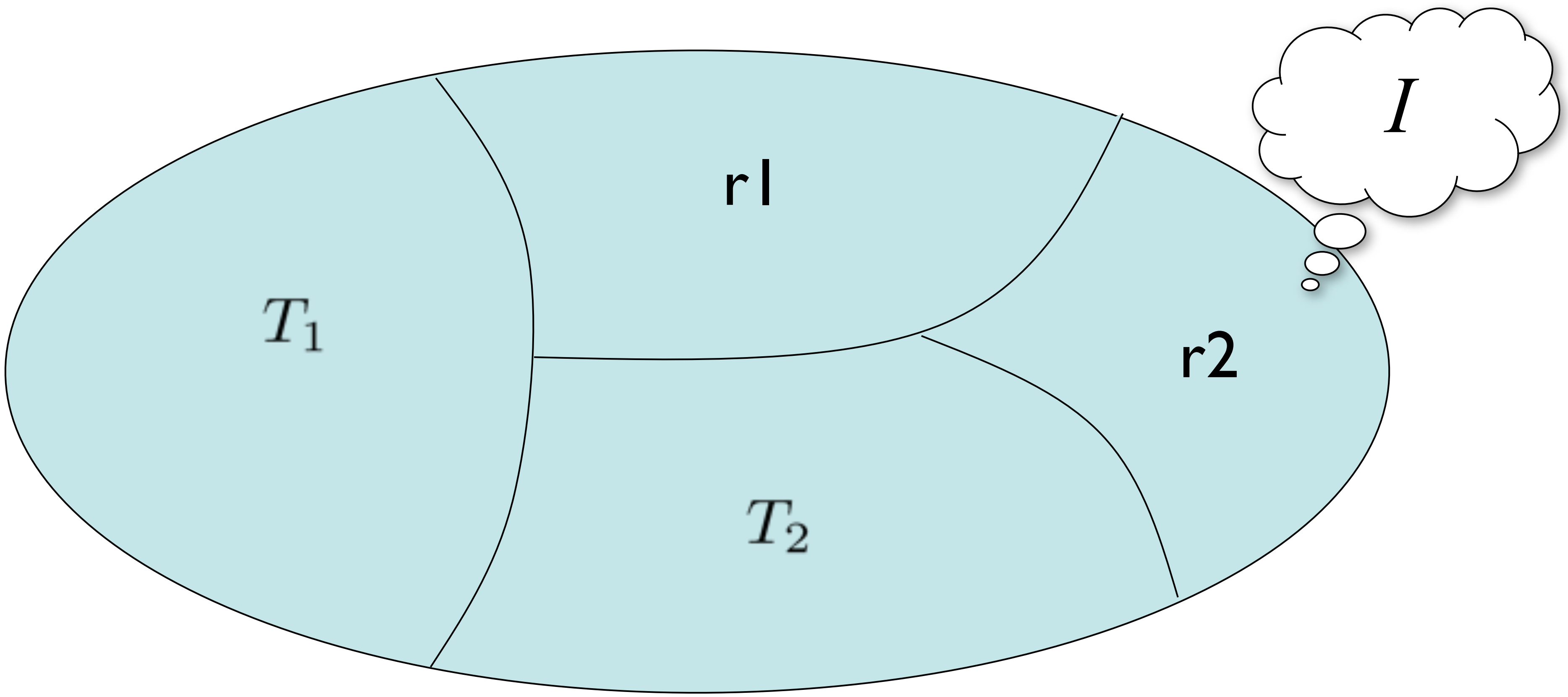
Reasoning principle

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

$(s=0 \wedge \text{emp}) \;\vee\; (s=1 \wedge Q)$

Initially,

$s=0$

Example

{ emp }	‖	{ emp }
P(s)		P(s)
[x] := 43		[x] := 47
V(s)		V(s)
{ emp }		{ emp }

Example

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

Example

{ emp }

P(s)

{ x ↦ _ }

[x] := 43

{ x ↦ _ }

V(s)

{ emp }

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

Example

$\{ emp \}$

P(s)

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

$\{ x \mapsto \_ \}$

[x] := 43

$\{ x \mapsto \_ \}$

V(s)

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

$\{ emp \}$

More Concurrent Separation Logic

Mike Dodds

slides: Matthew J. Parkinson, Alexey Gotsman

This lecture

- Recap: CSL
- Ownership
- Precision
- Read-sharing
- Auxiliary state

Programming language

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

Resource sharing

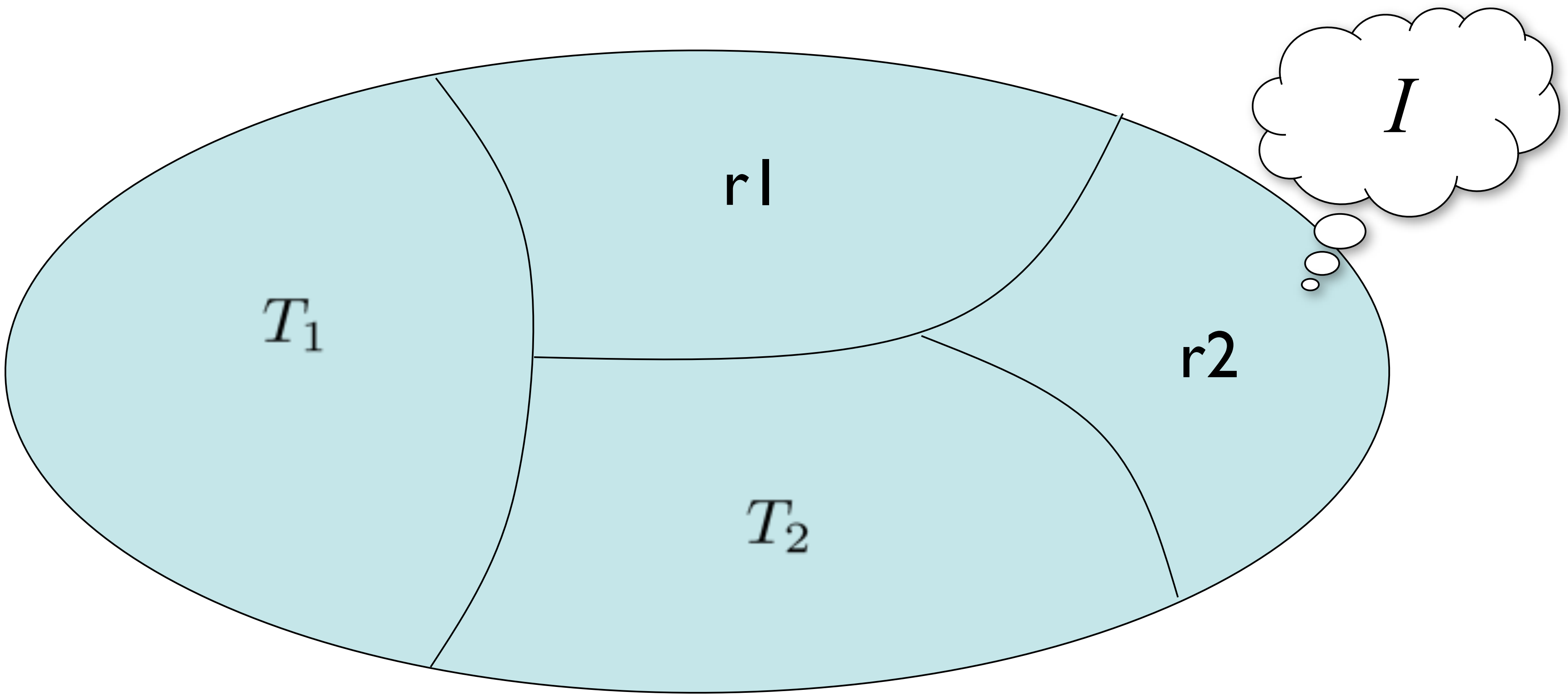
with l when true do
[x] := 56

$\|$

with l when true do
[x] := 42

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

Parallel proof rule

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

Resource Rule

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

Lock Rule

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

Ownership

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

full := false

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

Resource Invariant

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

||

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

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

with buff when full do	∥	x := new
$\{ \text{full} \wedge c \mapsto \_ \}$		$\{ x \mapsto \_ \}$
full := false		with buff when ¬full do
y := c		$\{ (\neg\text{full} \wedge \text{emp}) * x \mapsto \_ \}$
$\{ (\neg\text{full} \wedge \text{emp}) * y \mapsto \_ \}$		full := true;
$\{ y \mapsto \_ \}$		c := x;
dispose y		$\{ \text{full} \wedge c \mapsto \_ \}$
$\{ \text{emp} \}$		

full := false

```
with buff when full do          ||    x := new
    full := false               ||    with buff when ¬full do
    y := c                      ||        full := true;
                                ||        c := x;
                                ||    dispose x
```

Can we verify this?

full := false

{ emp }

Resource Invariant

```
with buff when full do          ||   x := new
    full := false               ||   with buff when ¬full do
    y := c                      ||       full := true;
                                ||       c := x;
                                ||   dispose x
```

Can we verify this?

$\{ \text{emp} \}$

with buff when full do

$\{ \text{full} \wedge \text{emp} \}$

full := false

y := c

$\{\neg \text{full} \wedge \text{emp} \wedge y = c\}$

$\{ \text{emp} \wedge y = c \}$

$\|$

x := new

$\{ x \mapsto \_ \}$

with buff when ¬full do

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

full := true;

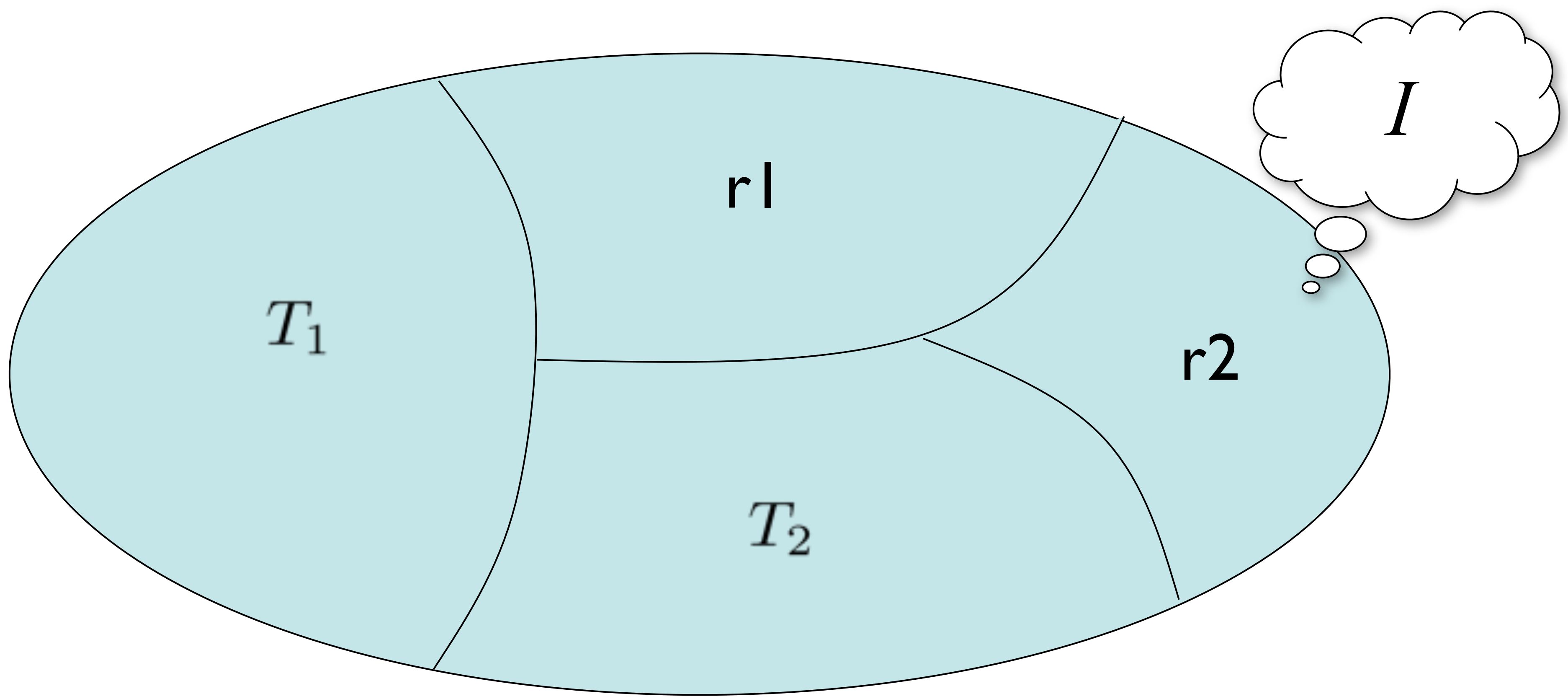
c := x;

$\{ \text{full} \wedge x \mapsto \_ \}$

dispose x

$\{ \text{emp} \}$

Ownership is in the Eye of the Asserter



The important thing is that threads have *compatible assumptions* about each other.

Precision

Precise invariants

A formula, P, is *precise* iff:

for any heap, there is at most one subheap satisfying the formula

$$\forall h_1, h_2, h. \quad h_1 \leq h \wedge h_2 \leq h \wedge h_1 \models P \wedge h_2 \models P \Rightarrow h_1 = h_2$$

Precise predicates?

$\forall h_1, h_2, h. \quad h_1 \leq h \wedge h_2 \leq h \wedge h_1 \models P \wedge h_2 \models P \Rightarrow h_1 = h_2$

emp

emp $\vee$ (x $\mapsto$ null * y $\mapsto$ null)

x $\mapsto$ null * true

true

ls(x,null)

$\exists$y. ls(x,y)

$$\mathsf{ls}(E, F) \Leftrightarrow (E = F \wedge \mathsf{emp}) \vee (\exists x'. E \mapsto x' * \mathsf{ls}(x', F))$$

Rule of conjunction

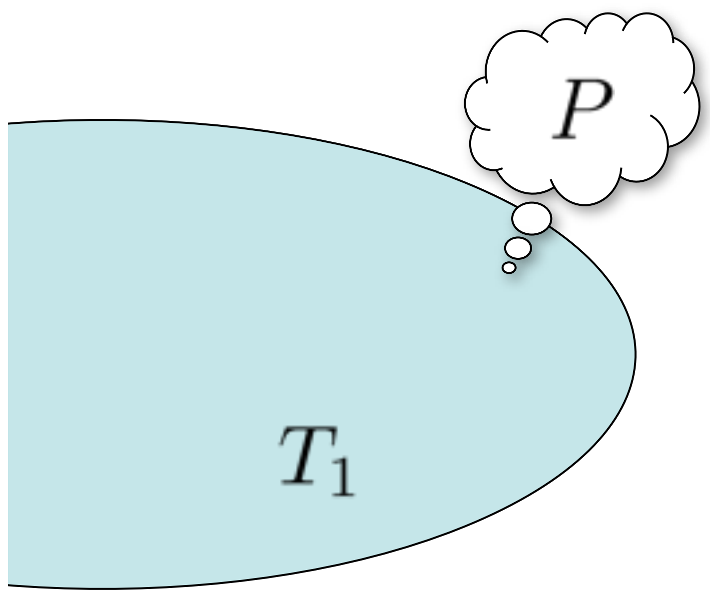
$$\frac{\{ P_1 \}\ C\ \{ Q_1 \} \quad \{ P_2 \}\ C\ \{ Q_2 \}}{\{ P_1 \wedge P_2 \}\ C\ \{ Q_1 \wedge Q_2 \}}$$

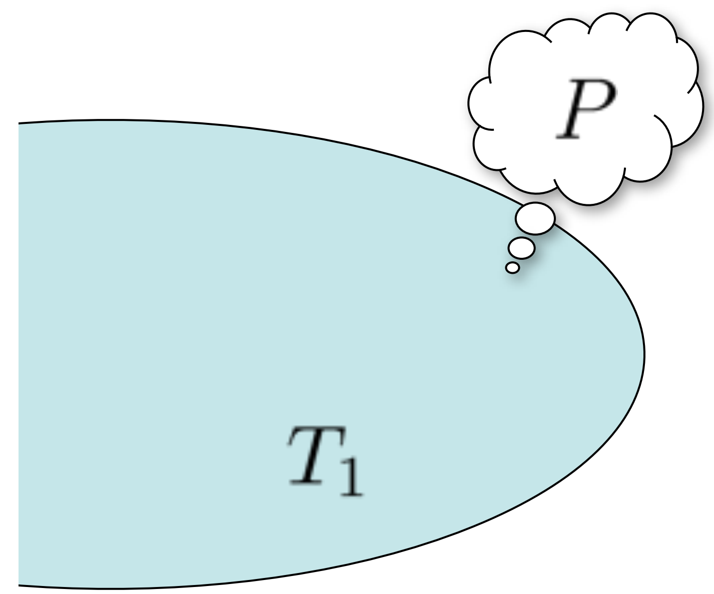
Subtle soundness

Without rule of conjunction Concurrent Separation Logic is sounds with arbitrary resource invariants.

With precise invariants and the rule of conjunction the logic is sound.

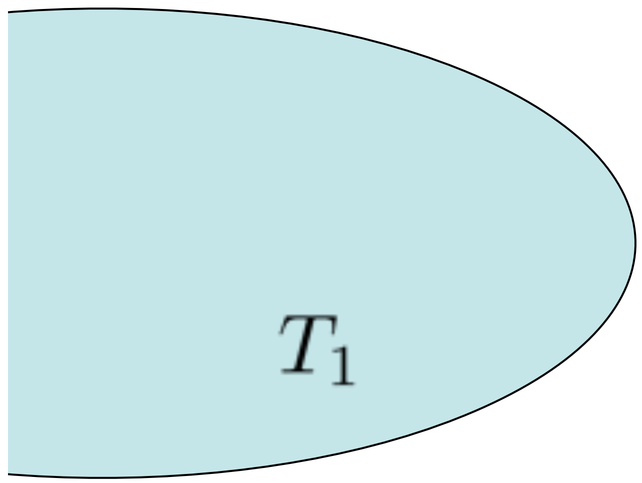
Unsoundness of conjunction rule

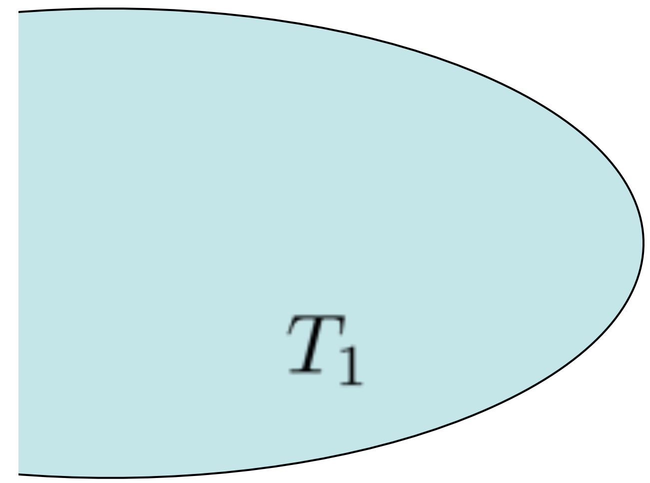




Imprecise invariants allow different heap splittings at unlock

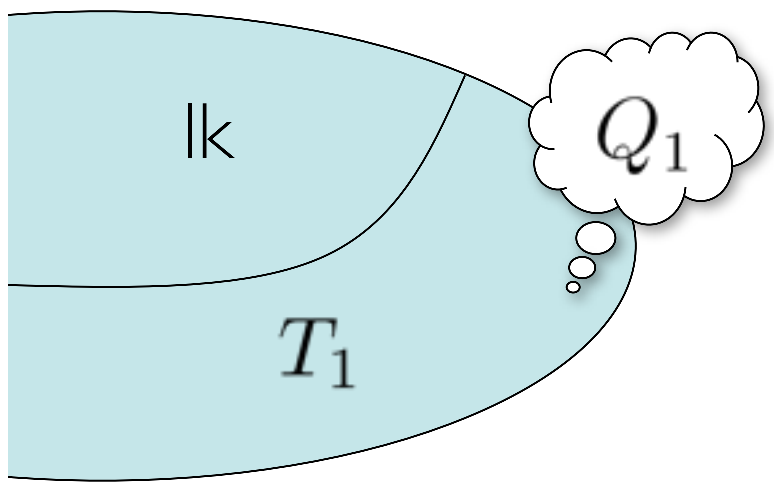
Unsoundness of conjunction rule

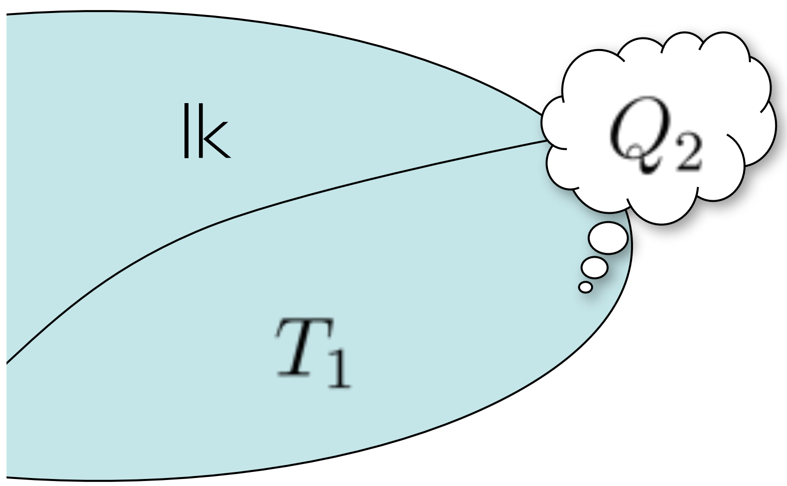




Imprecise invariants allow different heap splittings at unlock

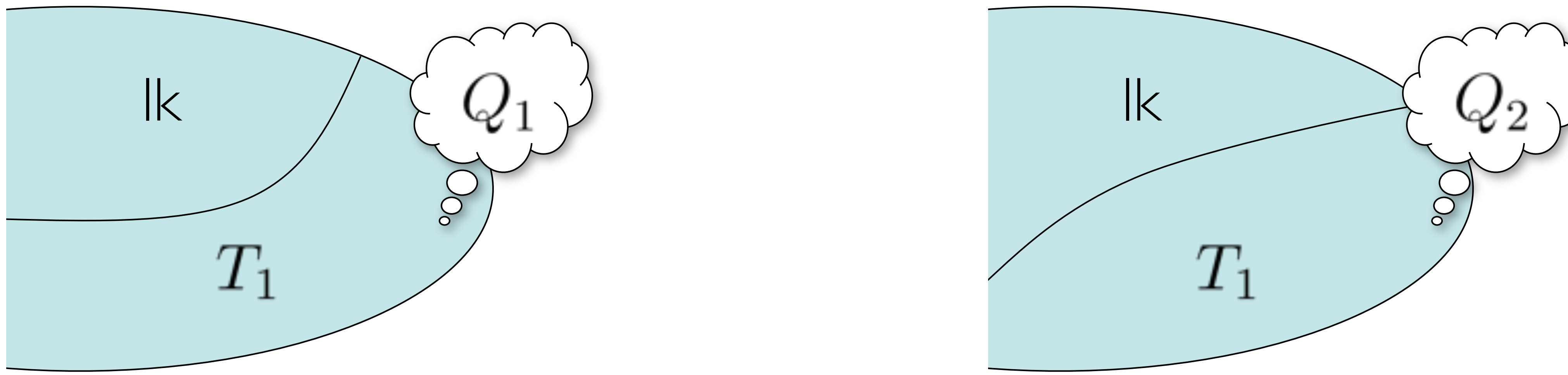
Unsoundness of conjunction rule





Imprecise invariants allow different heap splittings at unlock

Unsoundness of conjunction rule



Imprecise invariants allow different heap splittings at unlock

Q_1 and Q_2 describe different parts of the heap, yet can be conjoined:

$$\frac{\{P\}\ C\ \{Q_1\} \quad \{P\}\ C\ \{Q_2\}}{\{P\}\ C\ \{Q_1 \wedge Q_2\}}$$

Reynolds' counter example

We can prover the following holds with the resource invariant r:true

$$\frac{\dfrac{\{\text{true}\}\ \text{skip}\ \{\text{true}\}}{\{(\text{emp} \vee x \mapsto \_) * \text{true}\}\ \text{skip}\ \{\text{emp} * \text{true}\}}}{\{\text{emp} \vee x \mapsto \_\}\ \text{with r when true do skip}\ \{\text{emp}\}}$$

Reynolds' counter example

From the previous proof we can derive both

{ emp * x ↦ _ } with… { x ↦ _ }

{ emp * x ↦ _ } with… { emp }

which with the rule of conjunction leads to a contradiction.

Read Sharing

Races

Data race - a read and write, or two writes, to the same shared data at the same time.

Do we want to forbid all races?

Does concurrent separation logic forbid all races?

Read sharing

{ emp }

x = new

t1 = [x] || t2 = [x]

free(x)

{ emp }

Read sharing

Replace $\mapsto$ with $\overset{\pi}{\mapsto}$, where $\pi \in (0;1]$ is a permission

- $\pi < 1$ allows only read access
- $\pi = 1$ allows read and write access

Read sharing

Replace $\mapsto$ with $\overset{\pi}{\mapsto}$, where $\pi \in (0;1]$ is a permission

- $\pi < 1$ allows only read access
- $\pi = 1$ allows read and write access

$$x \overset{\pi_1+\pi_2}{\mapsto} E \quad \Leftrightarrow \quad x \overset{\pi_1}{\mapsto} E \ * \ x \overset{\pi_2}{\mapsto} E$$

$$x \mapsto E \quad \Leftrightarrow \quad x \overset{1}{\mapsto} E$$

{ emp }

x = new

t1 = [x] || t2 = [x]

free(x)

{ emp }

$\{ \mathrm{emp} \}$

x = new

$\{ x \overset{1}{\mapsto} \_ \}$

$\{ x \overset{0.5}{\mapsto} \_ * x \overset{0.5}{\mapsto} \_ \}$

$\{ x \overset{0.5}{\mapsto} \_ \}$	||	$\{ x \overset{0.5}{\mapsto} \_ \}$
t1 = [x]	||	t2 = [x]
$\{ x \overset{0.5}{\mapsto} \_ \}$	||	$\{ x \overset{0.5}{\mapsto} \_ \}$

$\{ x \overset{0.5}{\mapsto} \_ * x \overset{0.5}{\mapsto} \_ \}$

$\{ x \overset{1}{\mapsto} \_ \}$

free(x)

$\{ \mathrm{emp} \}$

Auxiliary State

Doing Something Twice

$\{ x \mapsto 0 \}$

resource r in

$$\left(\begin{array}{l} \text{with r when true} \\ [x] := [x] + 1; \end{array} \;\middle\|\; \begin{array}{l} \text{with r when true} \\ [x] := [x] + 1; \end{array} \right)$$

$\{ x \mapsto 2 \}$

Auxiliary State

The problem: the invariant hides the fact that there are just two threads.

We add *auxiliary state* to track this.

Can add extra state as long as it doesn't affect the control-flow of the program.

resource r in

with r when true
[x] := [x] + 1

||

with r when true
[x] := [x] + 1

resource r in

with r when true
[x] := [x] + 1
[y] := 1

||

with r when true
[x] := [x] + 1
[z] := 1

Auxiliary assignment

$\{\ \exists i,j.\ y \overset{0.5}{\mapsto} i * z \overset{0.5}{\mapsto} j * x \mapsto i+j\ \}$

Resource Invariant

resource r in

with r when true [x] := [x] + 1 [y] := 1	‖	with r when true [x] := [x] + 1 [z] := 1

Auxiliary assignment

$$\{ y \mapsto 0 * z \mapsto 0 * x \mapsto 0 \}$$

resource r in

$$\{ y \overset{0.5}{\mapsto} 0 * z \overset{0.5}{\mapsto} 0 \}$$

with r when true	‖	with r when true
[x] := [x] + 1	‖	[x] := [x] + 1
[y] := 1	‖	[z] := 1

$$\{\ y \mapsto 0 * z \mapsto 0 * x \mapsto 0\ \}$$

resource r in

$$\{\ y \overset{0.5}{\mapsto} 0 * z \overset{0.5}{\mapsto} 0\ \}$$

$$\{\ y \overset{0.5}{\mapsto} 0\ \}$$

with r when true

$$\{\ \exists j.\ y \mapsto 0 * z \overset{0.5}{\mapsto} j * x \mapsto j\ \}$$

[x] := [x] + 1

[y] := 1

$$\{\ y \overset{0.5}{\mapsto} 1\ \}$$

||

with r when true

[x] := [x] + 1

[z] := 1

$$\{ y \mapsto 0 * z \mapsto 0 * x \mapsto 0 \}$$

resource r in

$$\{ y \overset{0.5}{\mapsto} 0 * z \overset{0.5}{\mapsto} 0 \}$$

$\{ y \overset{0.5}{\mapsto} 0 \}$	‖	$\{ z \overset{0.5}{\mapsto} 0 \}$
with r when true	‖	with r when true
$\{ \exists j.\, y \mapsto 0 * z \overset{0.5}{\mapsto} j * x \mapsto j \}$	‖	$\{ \exists i.\, y \overset{0.5}{\mapsto} i * z \mapsto 0 * x \mapsto i \}$
[x] := [x] + 1	‖	[x] := [x] + 1
[y] := 1	‖	[z] := 1
$\{ y \overset{0.5}{\mapsto} 1 \}$	‖	$\{ z \overset{0.5}{\mapsto} 1 \}$

$$\{\, y \mapsto 0 * z \mapsto 0 * x \mapsto 0 \,\}$$

resource r in

$$\{\, y \overset{0.5}{\mapsto} 0 * z \overset{0.5}{\mapsto} 0 \,\}$$

$\{\, y \overset{0.5}{\mapsto} 0 \,\}$

with r when true

$\{\, \exists j.\, y \mapsto 0 * z \overset{0.5}{\mapsto} j * x \mapsto j \,\}$

[x] := [x] + 1

[y] := 1

$\{\, y \overset{0.5}{\mapsto} 1 \,\}$

||

$\{\, z \overset{0.5}{\mapsto} 0 \,\}$

with r when true

$\{\, \exists i.\, y \overset{0.5}{\mapsto} i * z \mapsto 0 * x \mapsto i \,\}$

[x] := [x] + 1

[z] := 1

$\{\, z \overset{0.5}{\mapsto} 1 \,\}$

$$\{\, (\exists i,j.\, y \overset{0.5}{\mapsto} i * z \overset{0.5}{\mapsto} j) * x \mapsto i{+}j * y \overset{0.5}{\mapsto} 1 * z \overset{0.5}{\mapsto} 1 \,\}$$

Concurrent Separation Logic References

- O'Hearn. Resources, concurrency and local reasoning. TCS, 2007
- Bornat et al. Permission accounting in separation logic. POPL'05