Specifying and Verifying Concurrent Algorithms with Histories and Subjectivity

Ilya Sergey Aleks Nanevski Anindya Banerjee

IMDEA Software Institute

ESOP 2015

A logic-based approach for Specifying and Verifying Concurrent Algorithms

An approach, which is

An approach, which is

- Natural
  - captures intuition behind realistic algorithms

An approach, which is

- Natural
  - captures intuition behind realistic algorithms
- Powerful
  - enables compositional verification of concurrency

An approach, which is

- Natural
  - captures intuition behind realistic algorithms
- Powerful
  - enables compositional verification of concurrency
- Lightweight
  - does not require to engineer a new logical framework

Key ideas

- Subjectivity
- Partial Commutative Monoids (PCMs)
- Histories

Key ideas

- Subjectivity
- Partial Commutative Monoids (PCMs)

Nanevski et al. [ESOP'14]

- Histories

Key ideas

- Subjectivity
- Partial Commutative Monoids (PCMs)
- Histories

Hoare-style program specifications

Hoare-style program specifications

$$\{\ P\ \}\ \ c\ \ \{\ Q\ \}$$

Hoare-style program specifications

$$\{\, P \,\}\ c\ \{\, Q \,\}$$

precondition

Hoare-style program specifications

$$\{\, P \,\}\ \ c\ \ \{\, Q \,\}$$

precondition postcondition

Hoare-style program specifications

$$\{\, P \,\}\ \ c\ \ \{\, Q \,\}$$

precondition postcondition

If the *initial* state satisfies P,
then, after **c** terminates,
the *final* state satisfies Q.

Abstract specifications for a stack

```
push(x)

pop()
```

Abstract specifications for a stack

```
push(x)

pop()
```

Abstract specifications for a stack

$\{ S = xs \}$ `push(x)` $\{ S' = x :: xs \}$

`pop()`

Abstract specifications for a stack

$\{\, S = xs \,\}$ `push(x)` $\{\, S' = x :: xs \,\}$

$$\{\, S = xs \,\} \;\; \texttt{pop()} \;\; \{\; \begin{aligned} & res = None \wedge S = \mathbf{Nil} \\ \vee\ & \exists x, xs'.\ res = Some\ x \wedge \\ & \qquad xs = x :: xs' \wedge S' = xs' \;\} \end{aligned}$$

Abstract specifications for a stack

$\{\, S = xs \,\}$ `push(x)` $\{\, S' = x :: xs \,\}$

$\{\, S = xs \,\}$ `pop()` $\{\, res = None \wedge S = \mathbf{Nil}$
$\vee\ \exists x, xs'.\ res = Some\ x \wedge$
$xs = x :: xs' \wedge S' = xs' \,\}$

Suitable for sequential case

Abstract specifications for a stack

$\{\ S = xs\ \}$ `push(x)` $\{\ S' = x :: xs\ \}$

$$\{\ S = xs\ \}\ \texttt{pop()}\ \{\ \begin{array}{l} res = None \land S = \mathbf{Nil} \\ \lor\ \exists x, xs'.\ res = Some\ x\ \land \\ \qquad xs = x :: xs' \land S' = xs'\ \} \end{array}$$

Not so good for concurrent use:
useless in the presence of interference

```
y := pop();
```

$\{ S = \mathbf{Nil} \}$

```
y := pop();
```

$\{ S = \textbf{Nil} \}$

```
y := pop();
```

$\{ y = ??? \}$

{ S = Nil }

`y := pop();`	$\Vert$	`push(1);` `push(2);`

$\{ S = \mathbf{Nil} \}$

```
y := pop();
```

$\{ y = 1 \lor y = 2 \lor y = \mathit{None} \}$

$\Vert$

```
push(1);

push(2);
```

{ S = Nil }

```
              ||   push(1);   ||
y := pop();   ||              ||   push(3);
              ||   push(2);   ||
```

$\{\ S = \mathbf{Nil}\ \}$

`y := pop();`	∥	`push(1);` `push(2);`	∥	`push(3);`
$\{\ y = 1 \lor y = 2 \lor y = 3 \lor y = \mathit{None}\ \}$				

Thread-modular spec for pop?

{ S = **Nil** }

```
y := pop();
```

{ y = ??? }

Idea

Idea

Capture the effect of *self*, abstract over the *others*.

Idea

Capture the effect of *self*, abstract over the *others*.

(subjective specification)

Subjective stack specifications

```
y := pop();
```

Subjective stack specifications

- H_s — **pushes/pops** to the stack by *this* thread

```
y := pop();
```

Subjective stack specifications

- H_s — **pushes/pops** to the stack by *this* thread
- H_o — **pushes/pops** by *all other threads*

```
y := pop();
```

Subjective stack specifications

- H_s — **pushes/pops** to the stack by *this* thread
- H_o — **pushes/pops** by *all other threads*

$$\{ H_s = \emptyset \}$$

```
y := pop();
```

Subjective stack specifications

- H_s — **pushes/pops** to the stack by *this* thread
- H_o — **pushes/pops** by *all other threads*

$$\{ H_s = \varnothing \}$$

```
y := pop();
```

$$\{ y = \mathit{None} \lor y = \mathit{Some}(v), \text{ where } v \in H_o \}$$

Subjective stack specifications

- H_s — **pushes/pops** to the stack by *this* thread
- H_o — **pushes/pops** by *all other threads*

$$\{\ H_s = \emptyset\ \}$$

```
y := pop();
```

$$\{\ y = \mathit{None} \vee y = \mathit{Some}(v), \text{ where } \underbrace{v \in H_o}_{}\ \}$$

what I popped depends
on what the *others* have pushed

Subjective stack specifications

Valid only if the stack is changed
only by push/pops.

$\{ H_s = \varnothing \}$

```
y := pop();
```

$\{ y = None \vee y = Some(v), \text{ where } \underbrace{v \in H_o} \}$

what I popped depends
on what the *others* have pushed

$\{ P \}$ `y := pop();` $\{ Q \}$

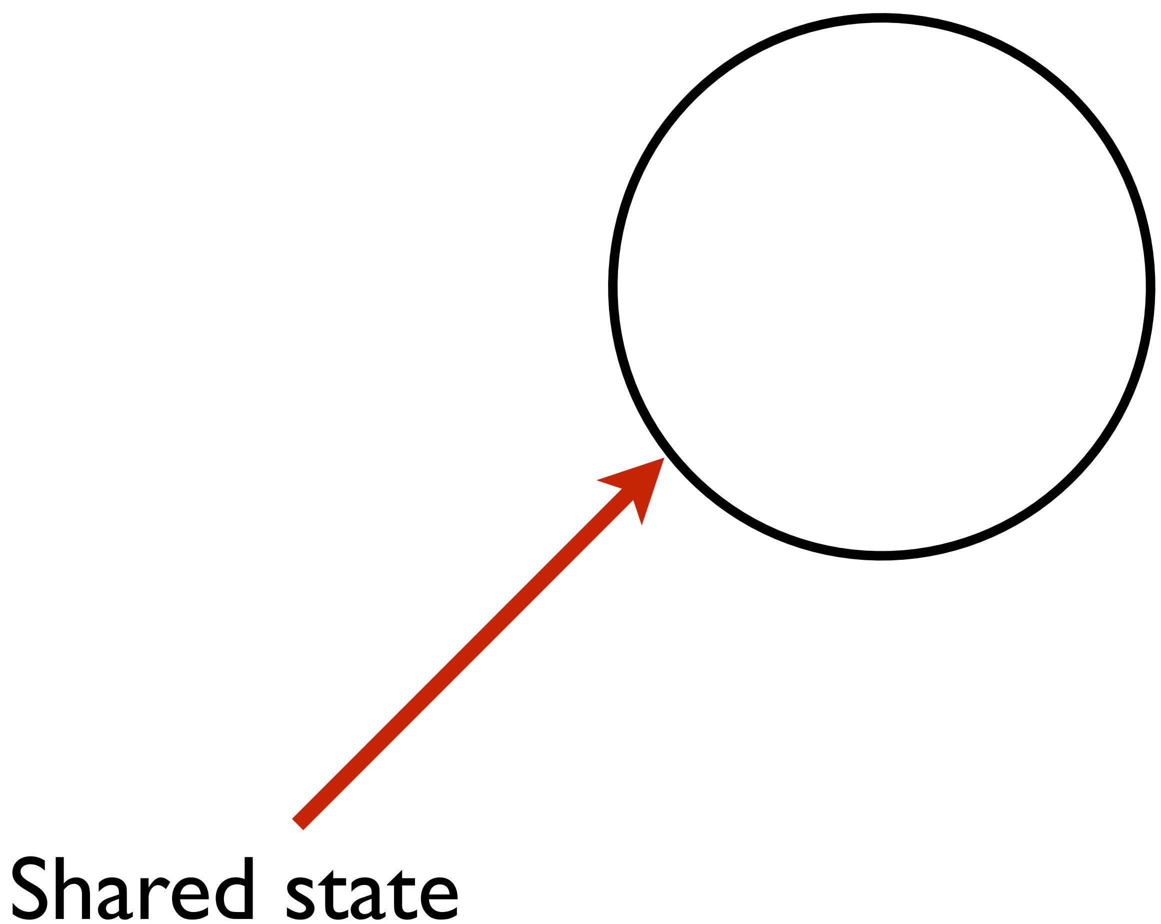
$$C \vdash \{P\}\ \texttt{y := pop();}\ \{Q\}$$

Specifies expected thread interference

$C \vdash \{ P \}$ `y := pop();` $\{ Q \}$

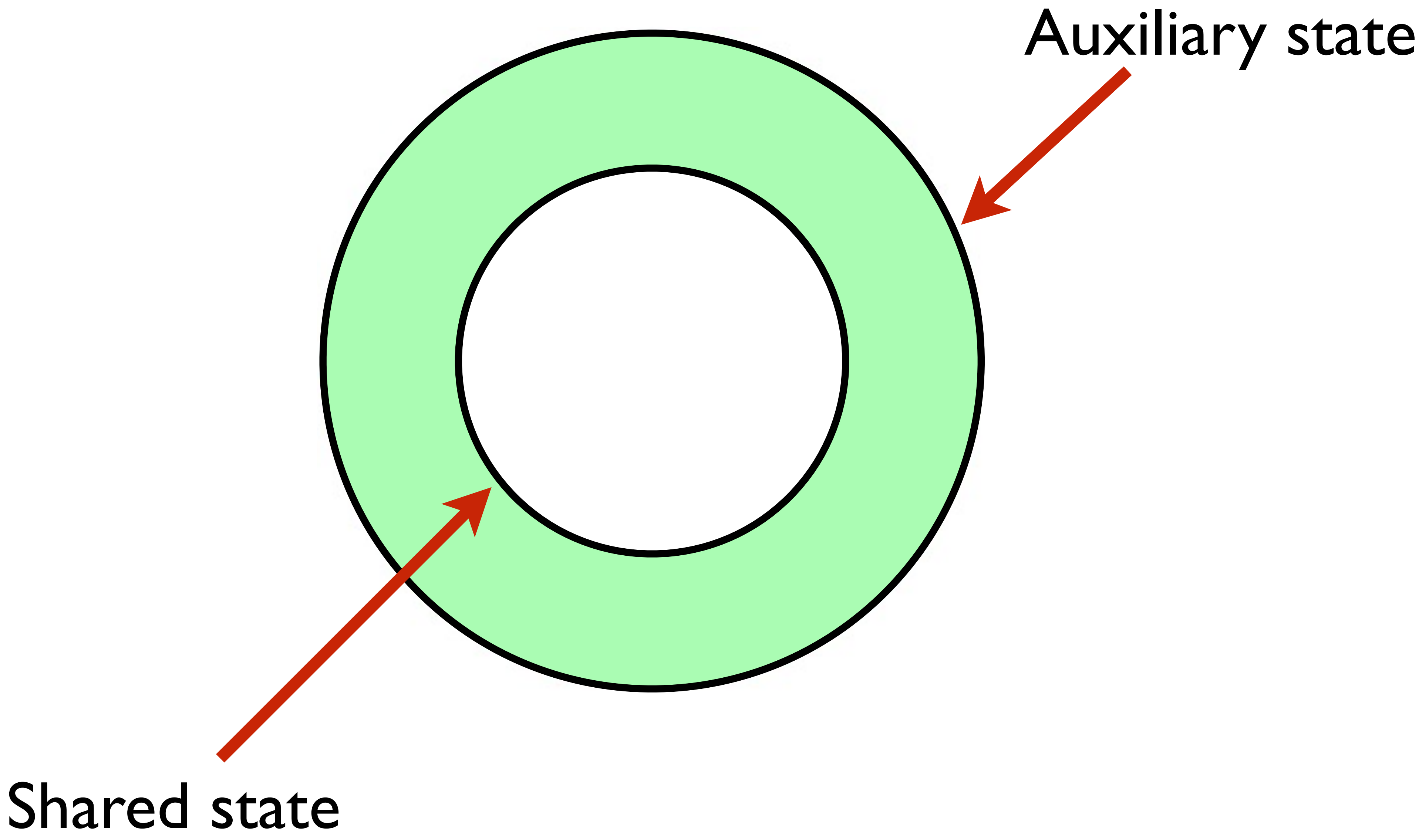
Concurrent Resources

Concurrent Resources



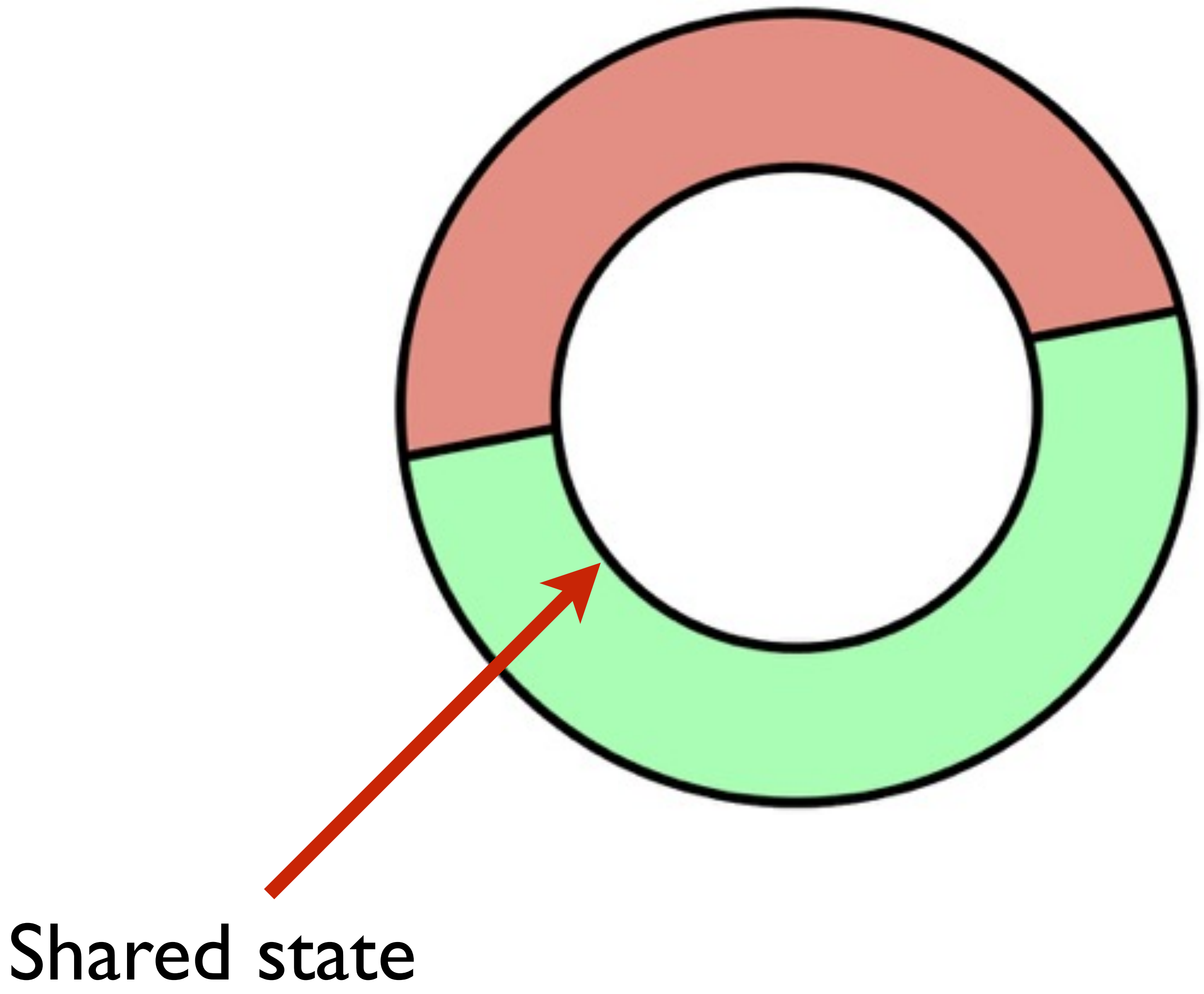
Concurrent Resources

Owicki, Gries [CACM'77]



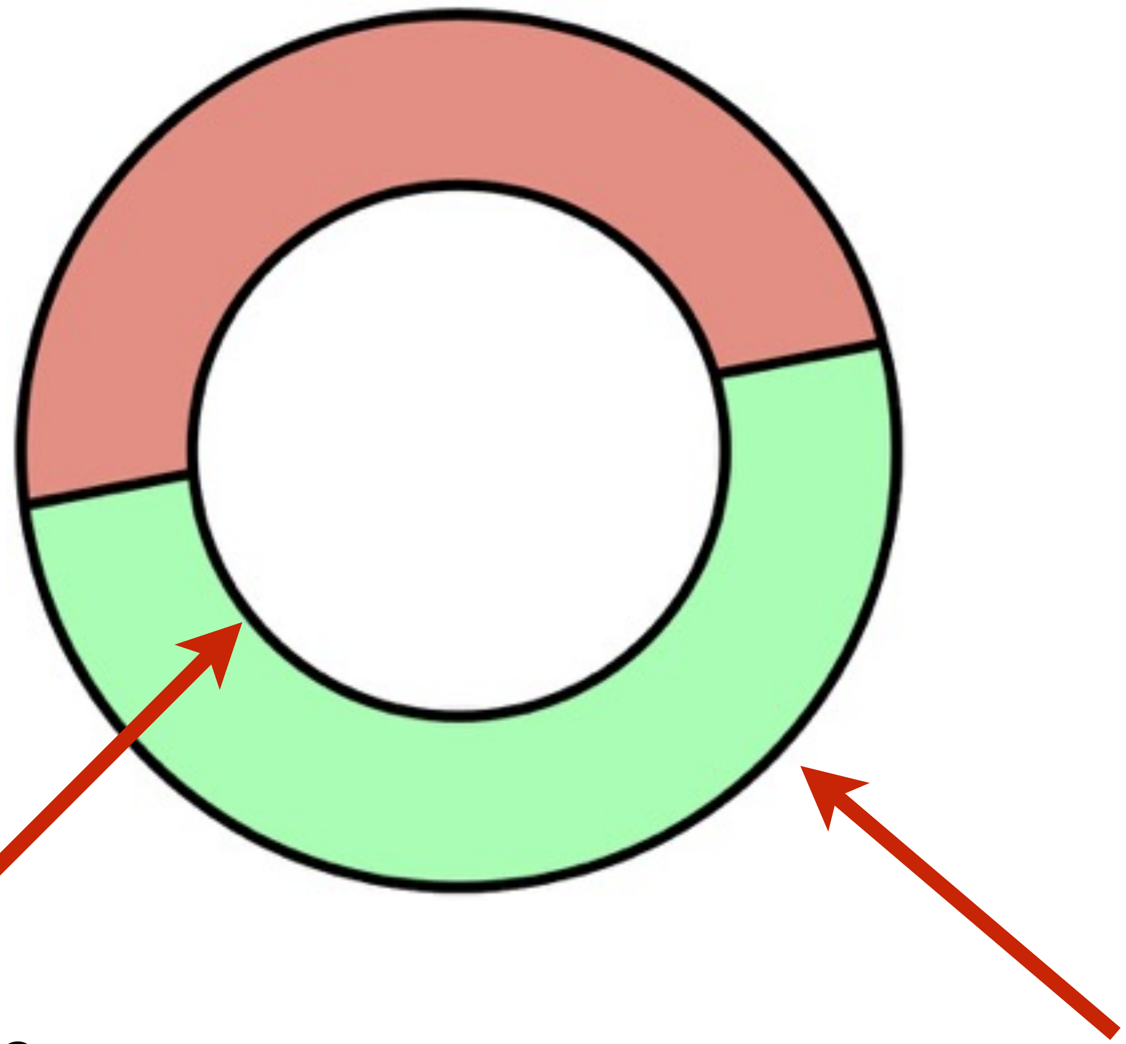
Subjective Concurrent Resources

Ley-Wild, Nanevski [POPL'13]



Subjective Concurrent Resources

Ley-Wild, Nanevski [POPL'13]



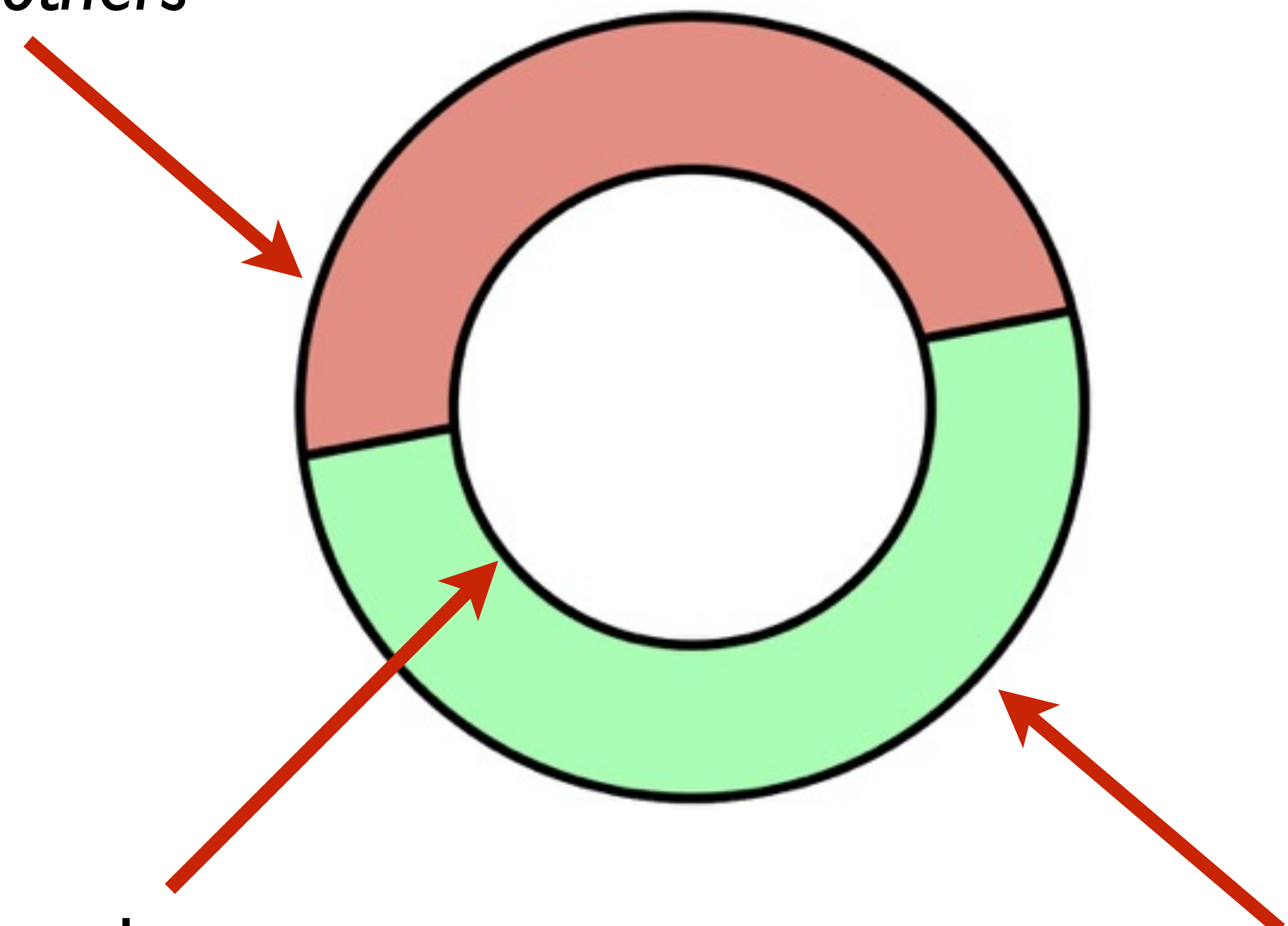
Shared state

Auxiliary state,
controlled by *this* thread

Subjective Concurrent Resources

Ley-Wild, Nanevski [POPL'13]

Auxiliary state,
controlled by *others*

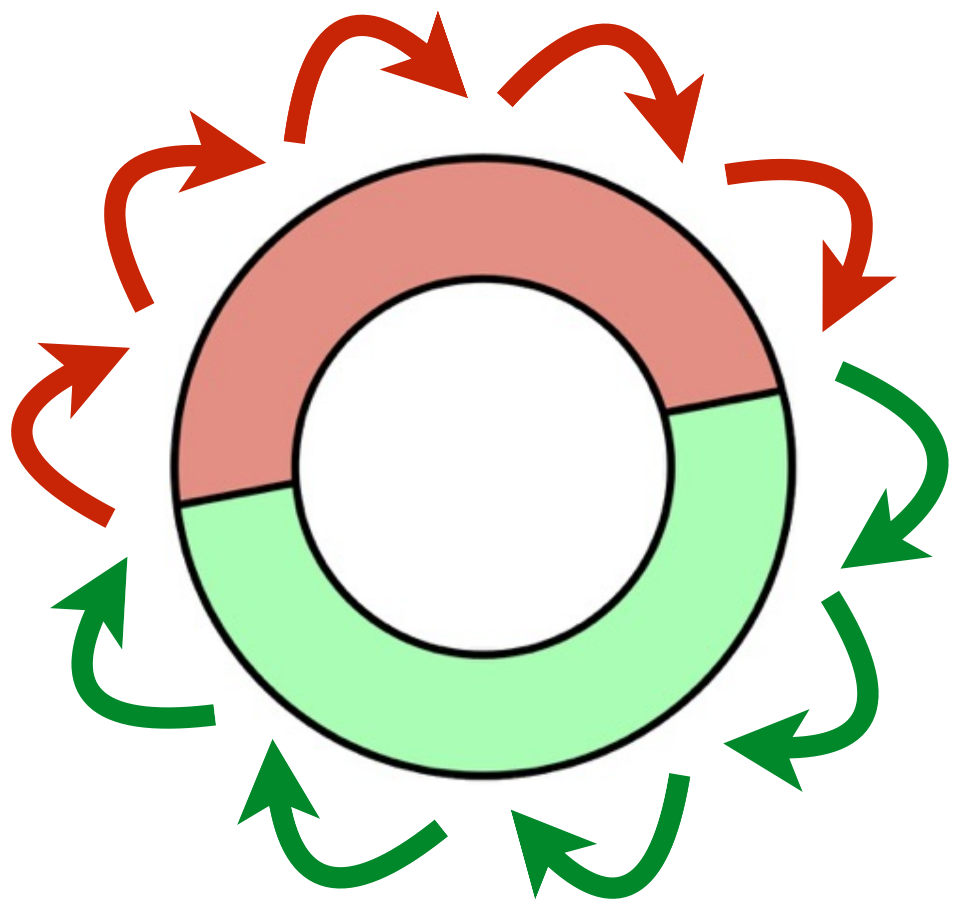


Shared state

Auxiliary state,
controlled by *this* thread

Subjective Concurrent Resources

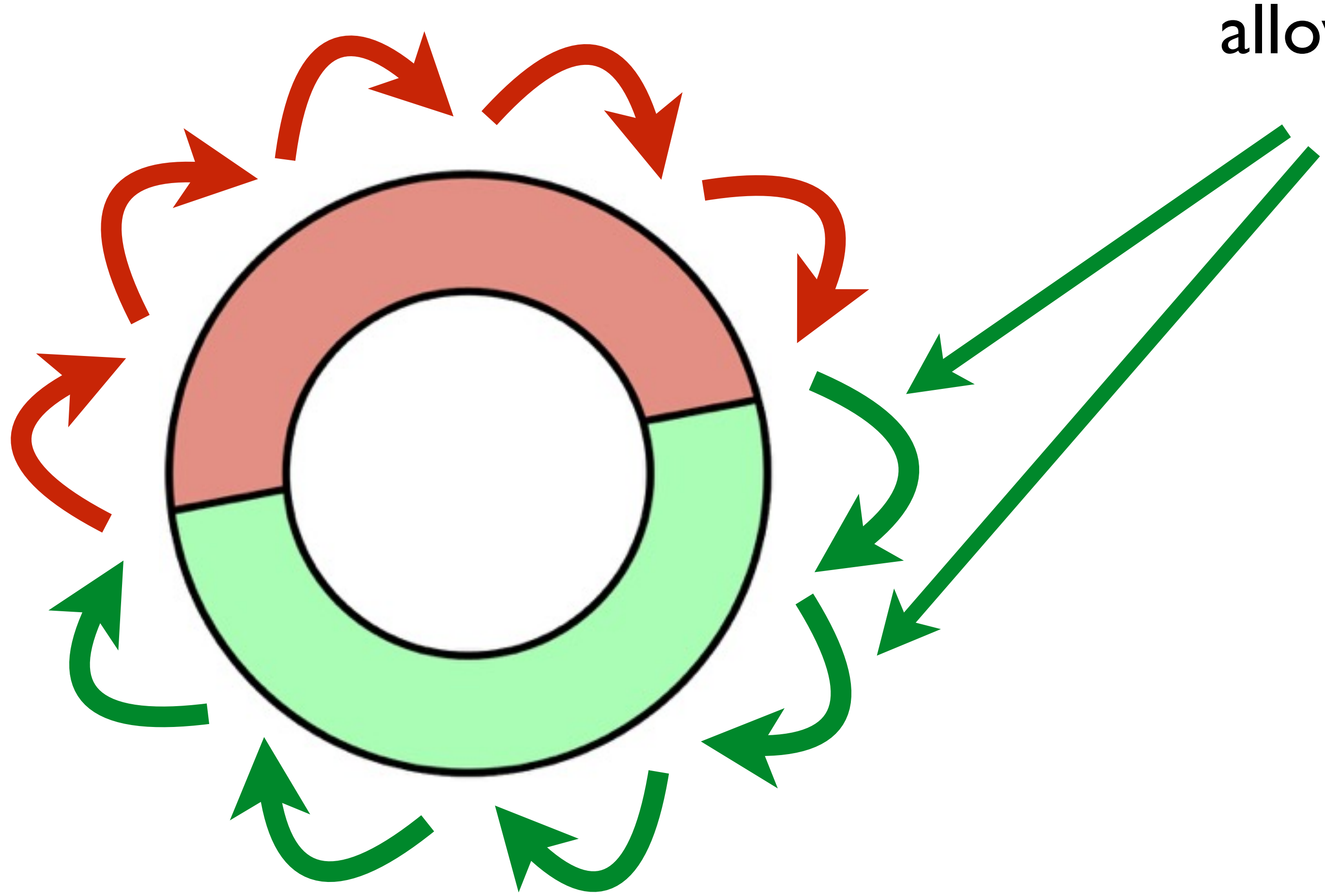
Jones [TOPLAS'83]



Subjective Concurrent Resources

Jones [TOPLAS'83]

Changes (transitions)
allowed to *myself*
(***Guarantee***)

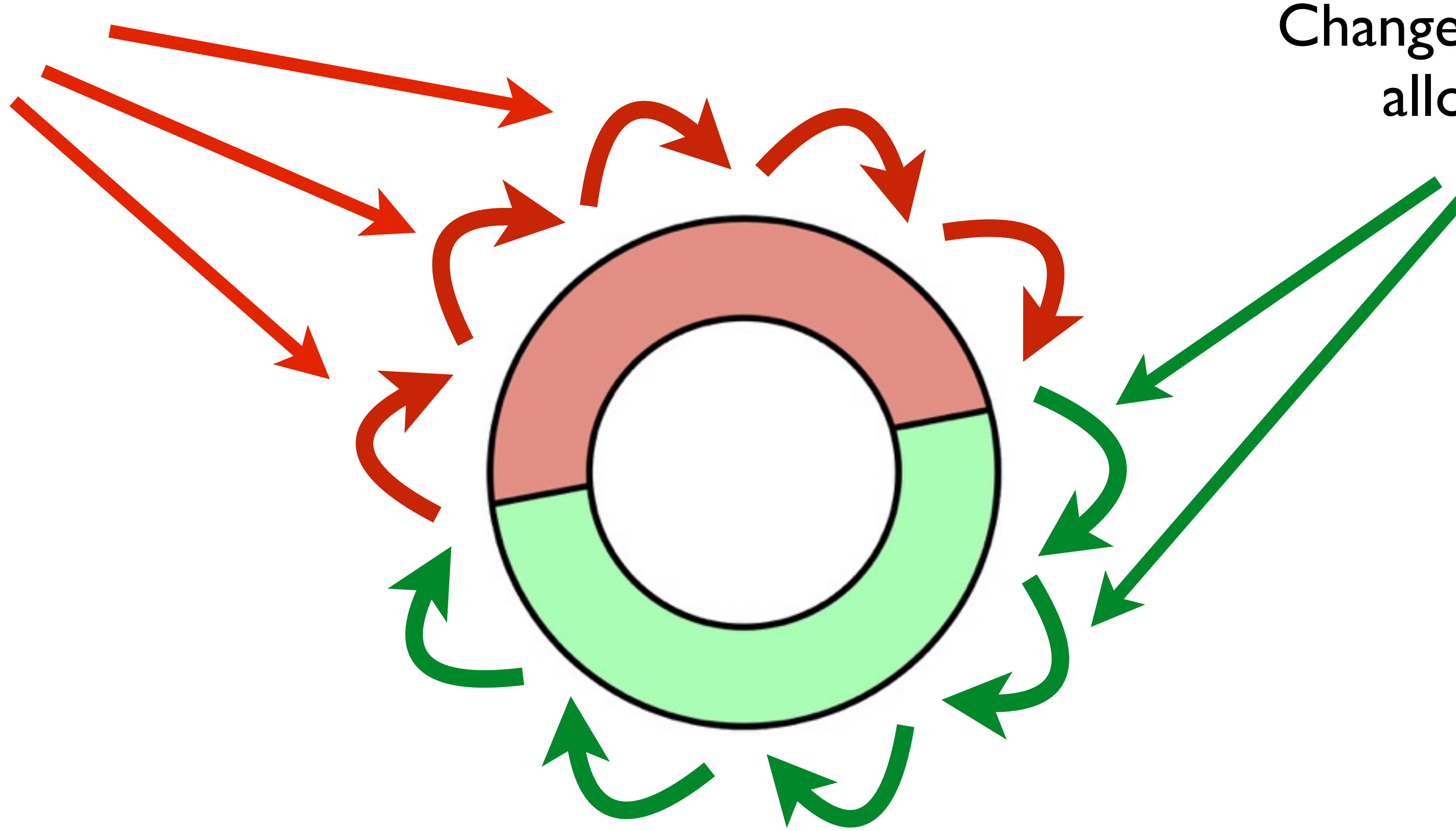


Subjective Concurrent Resources

Jones [TOPLAS'83]

Transitions, allowed to the *others* (***Rely***)

Changes (transitions) allowed to *myself* (***Guarantee***)

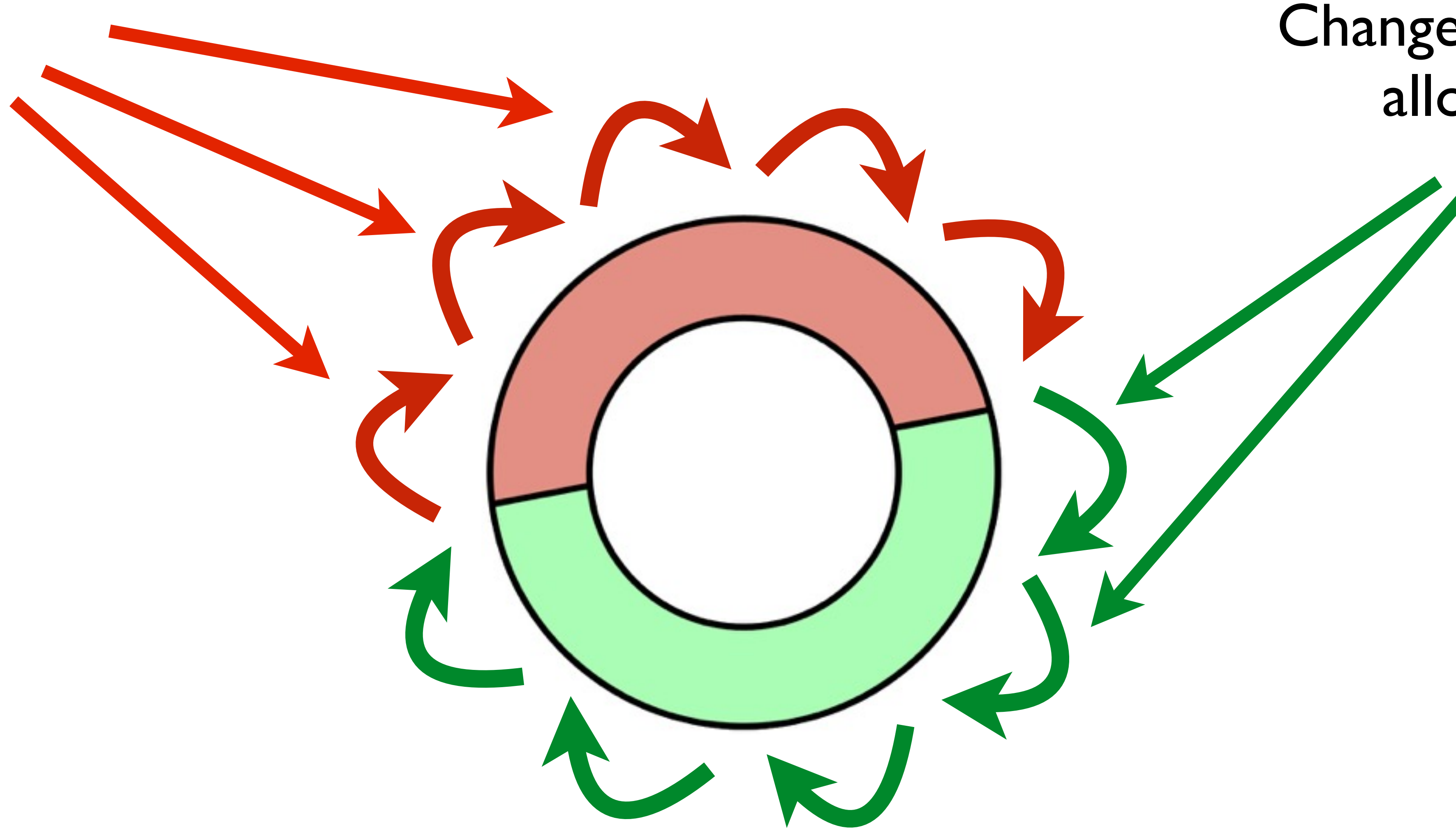


Subjective Concurrent Resources

Jones [TOPLAS'83]

Transitions, allowed to the *others* (***Rely***)

Changes (transitions) allowed to *myself* (***Guarantee***)



What I **have** = what I **can do** and what I **have done**.

Nanevski, Ley-Wild, Sergey, Delbianco [ESOP'14]

Concurrent Resources

=

State Transition Systems with Subjective Auxiliary State

Nanevski, Ley-Wild, Sergey, Delbianco [ESOP'14]

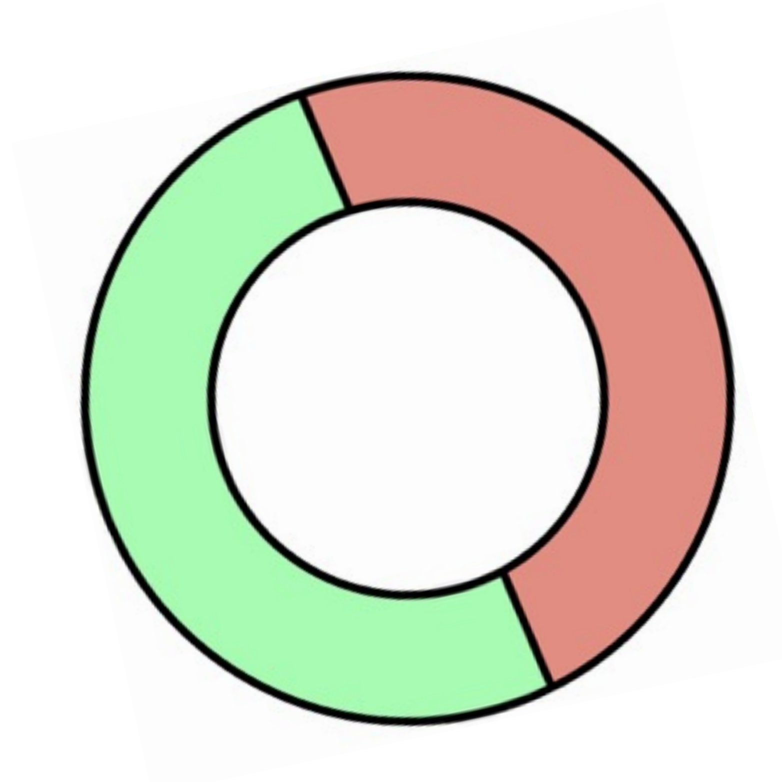
Concurrent Resources

=

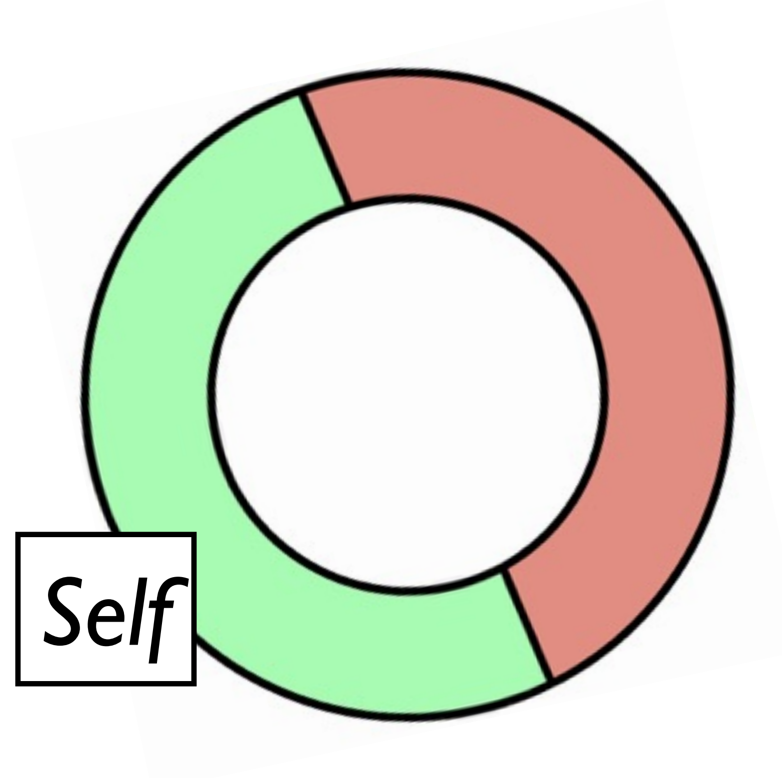
State Transition Systems
with
Subjective Auxiliary State

(Concurroids)

Specifications with concurroids

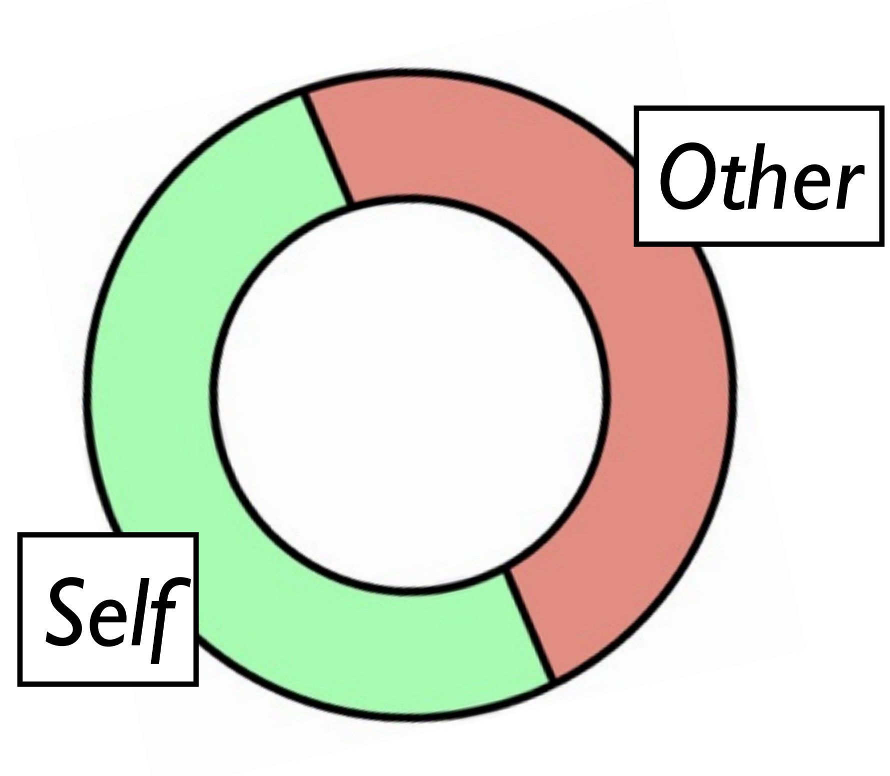


Specifications with concurroids



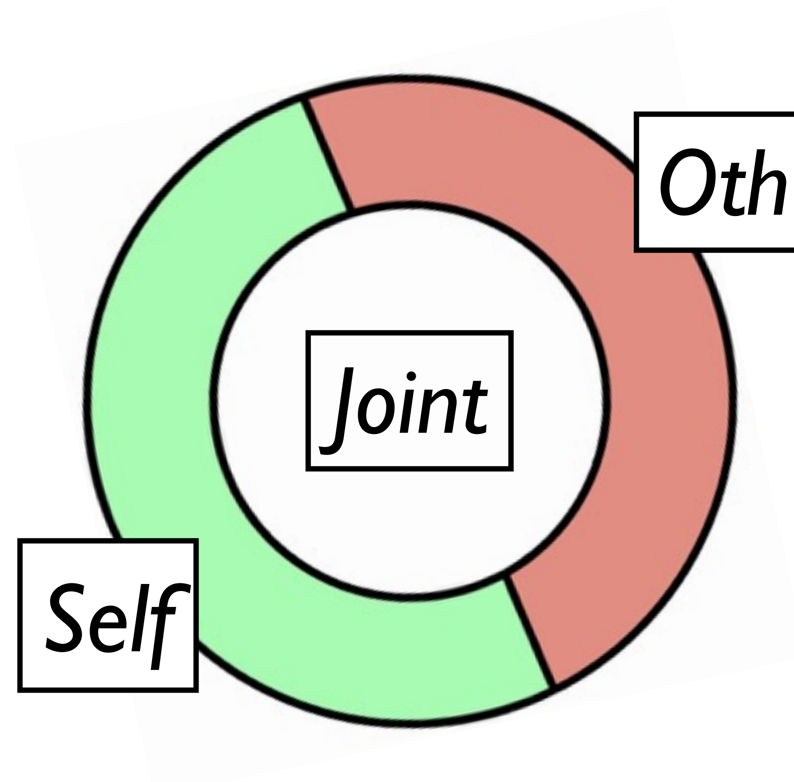
- *Self* — state controlled by *me*

Specifications with concurroids



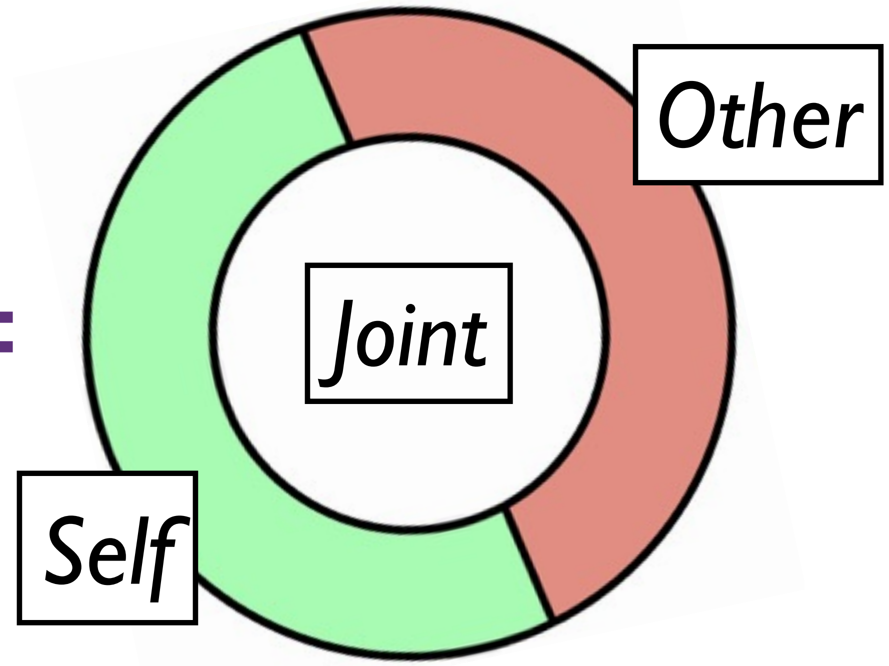
- *Self* — state controlled by *me*
- *Other* — state controlled by *all other threads*

Specifications with concurroids

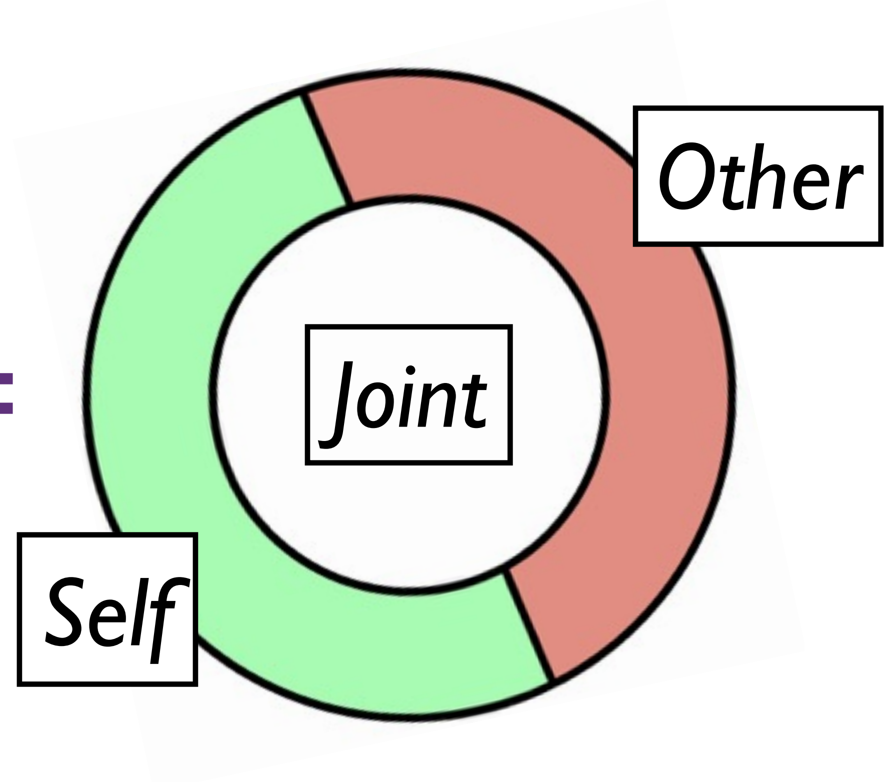


- *Self* — state controlled by *me*
- *Other* — state controlled by *all other threads*
- *Joint* — modified by everyone, *as allowed by transitions*

Specifications with concurroids

C = 

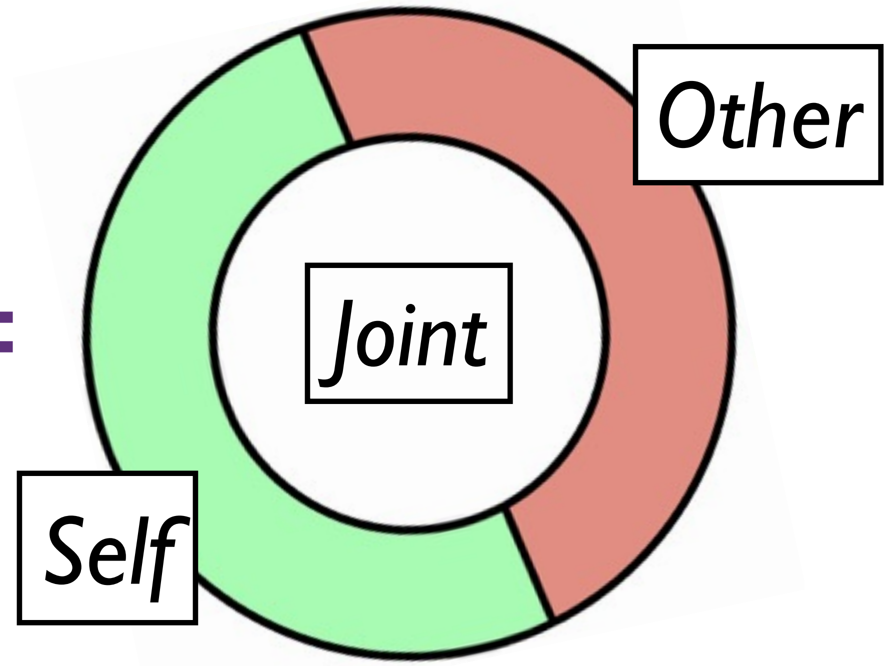
Specifications with concurroids

$C =$ 

$$C \vdash \{\, P \,\}\ c\ \{\, Q \,\}$$

Specifications with concurroids

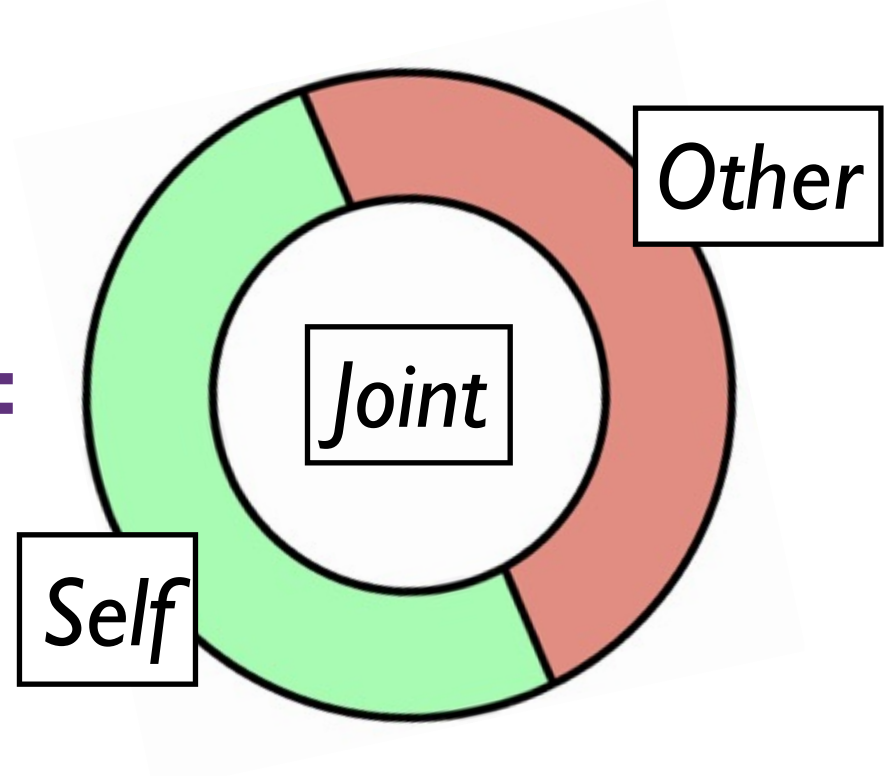
C =



{ P } c { Q }@ C

Specifications with concurroids

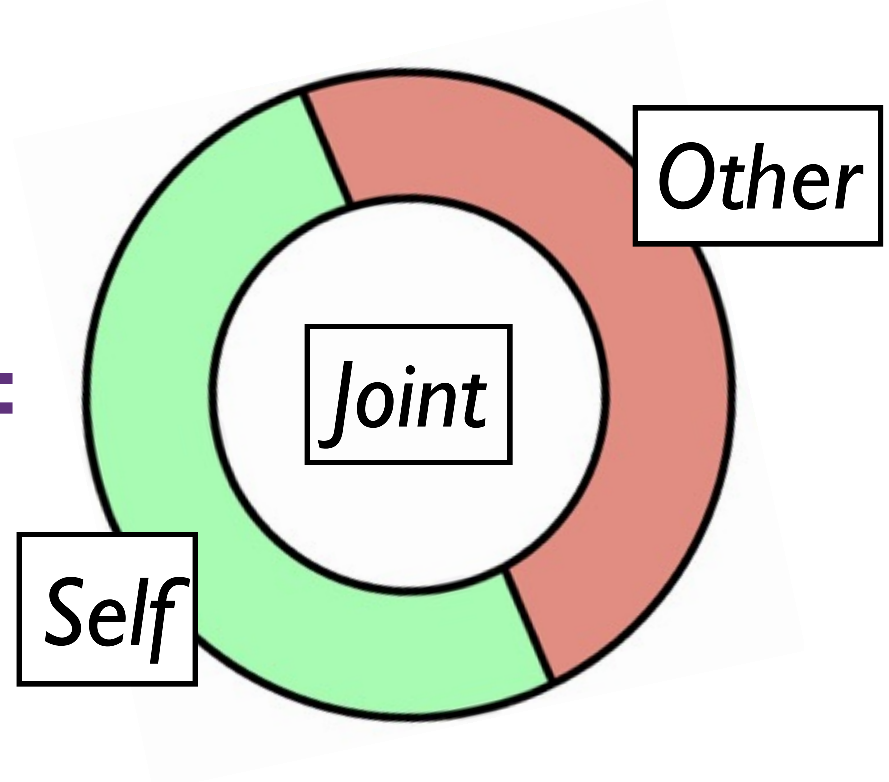
C =



$\{ P \}\ \mathbf{c}\ \{ Q \}\ \underbrace{@\ C}$

defines resources, touched by **c**, their transitions and invariants

Specifications with concurroids

$C =$ 

$\{ P \}\ c\ \{ Q \} @ C$

specify *self/other/joint* parts

FCSL: Fine-grained Concurrent Separation Logic

Nanevski, Ley-Wild, Sergey, Delbianco [ESOP'14]

FCSL: Fine-grained Concurrent Separation Logic

Nanevski, Ley-Wild, Sergey, Delbianco [ESOP'14]

- Logic for reasoning with concurroids

FCSL: Fine-grained Concurrent Separation Logic

Nanevski, Ley-Wild, Sergey, Delbianco [ESOP'14]

- Logic for reasoning with concurroids
- Emphasis on *subjective* specifications

Key ideas

- Subjectivity
- PCMs
- Histories

Key ideas

- Subjectivity — reasoning with *self* and *other*
- PCMs
- Histories

Key ideas

- Subjectivity — reasoning with *self* and *other*
- PCMs
- Histories

Partial Commutative Monoids

$$(S, \oplus, \mathbf{0})$$

- A *set* S of elements
- *Join* ($\oplus$): commutative, associative, partial
- *Unit* element $\mathbf{0}$: $\forall e \in S$, $e \oplus \mathbf{0} = \mathbf{0} \oplus e = e$

Parallel composition

Parallel composition

parent

child$_1$ $\parallel$ child$_2$

Parallel composition

parent

child$_1$ $\parallel$ child$_2$

- commutative
- associative
- unit — *idle* thread
- partial

Logical state split

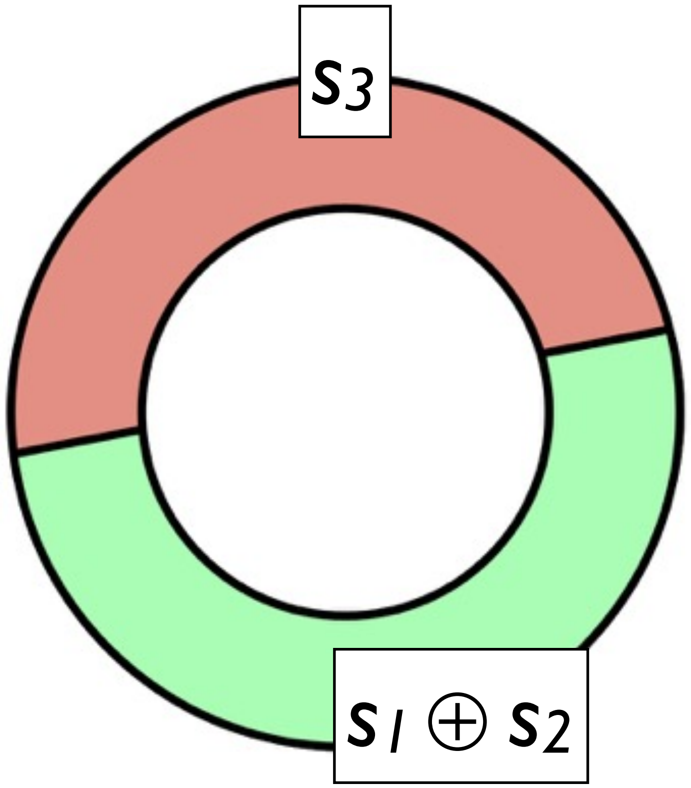
parent

$\{\ s_1 \oplus s_2\ \}$

child$_1$

child$_2$

Logical state split

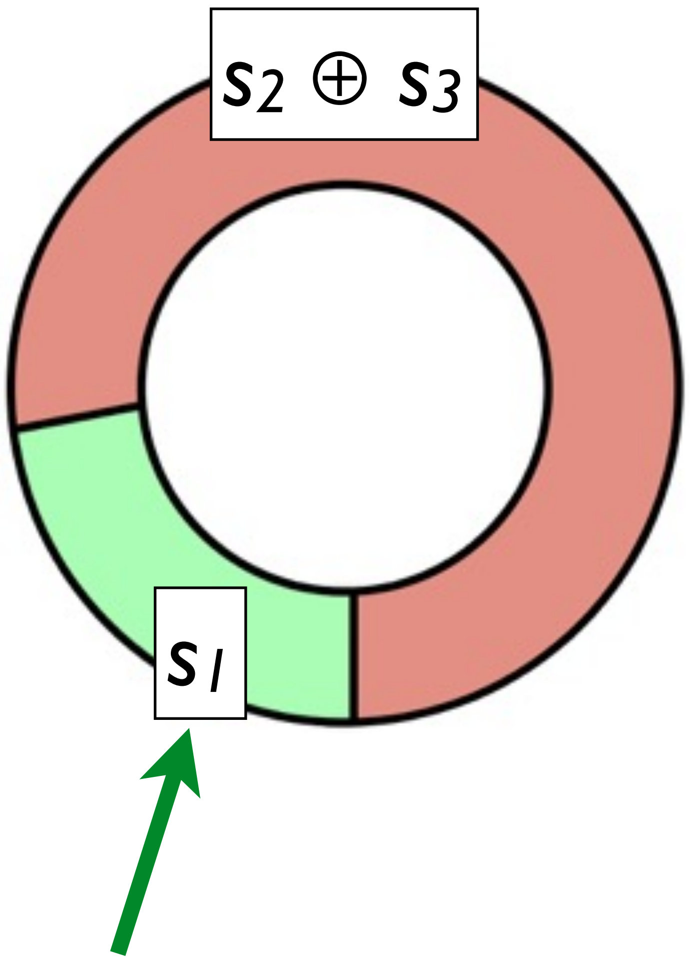


parent

$\{\, s_1 \oplus s_2 \,\}$

$\text{child}_1 \quad \| \quad \text{child}_2$

Logical state split



State that belongs to child_1

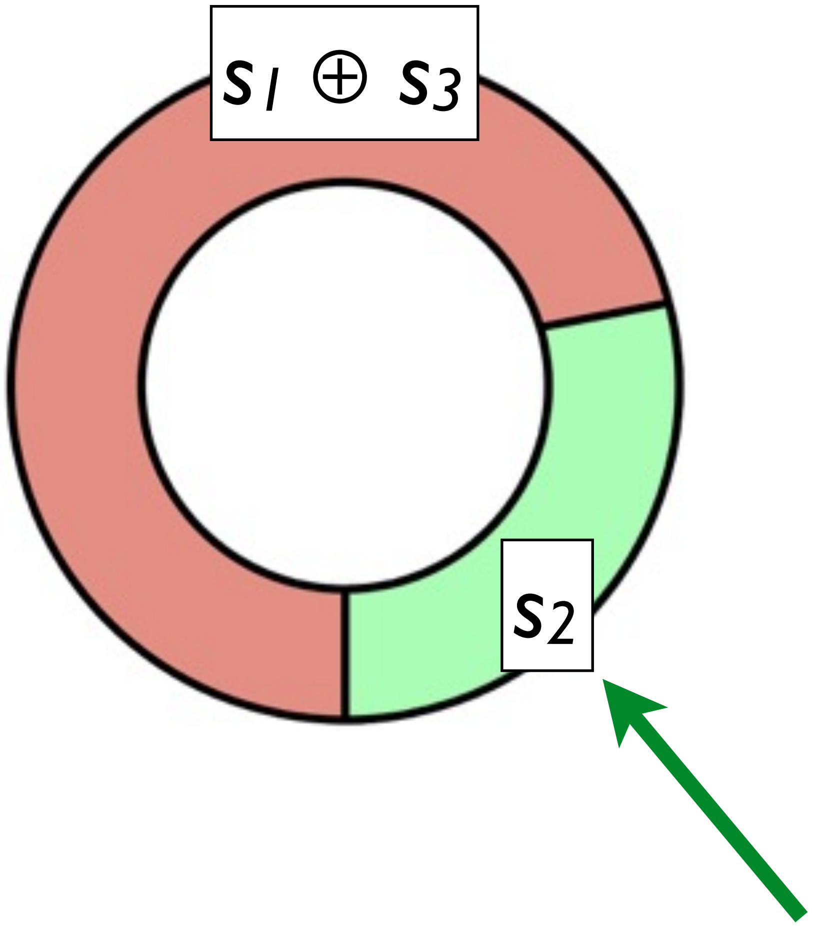
parent

$\{ s_1 \oplus s_2 \}$

$\{ s_1 \}$

child_1 || child_2

Logical state split



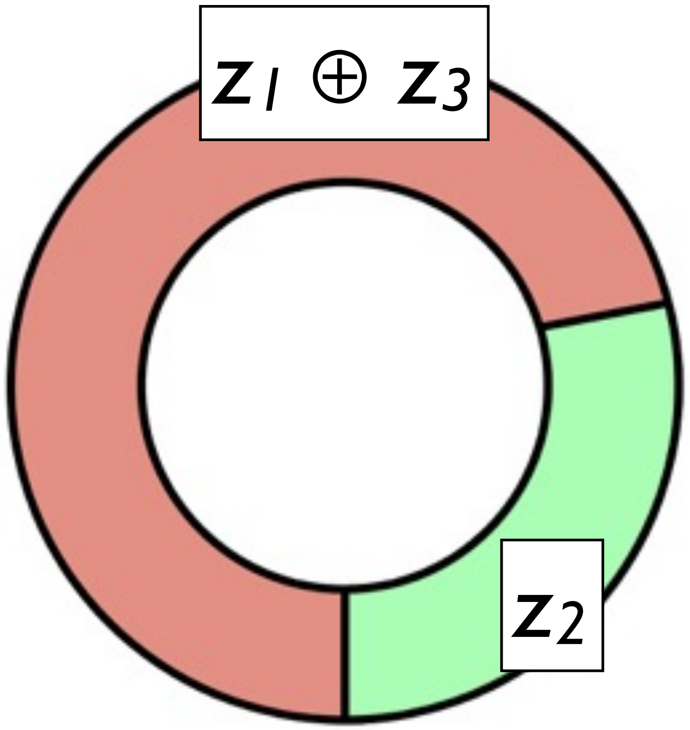
State that belongs to $child_2$

parent

$\{ s_1 \oplus s_2 \}$

$\{ s_1 \}$ $child_1$ || $\{ s_2 \}$ $child_2$

Logical state split

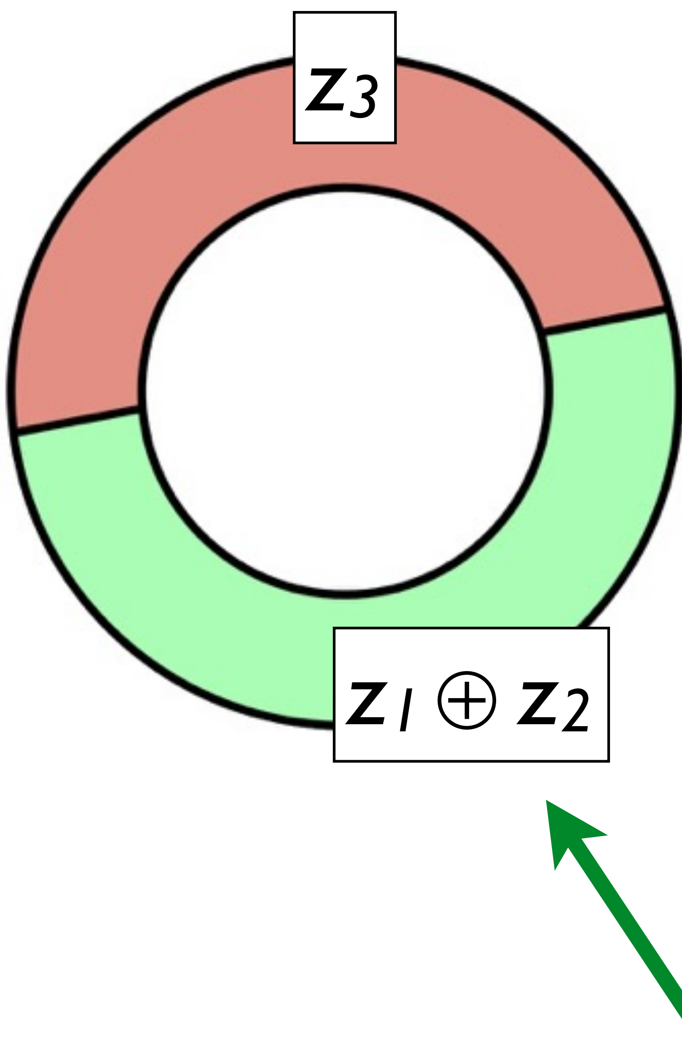


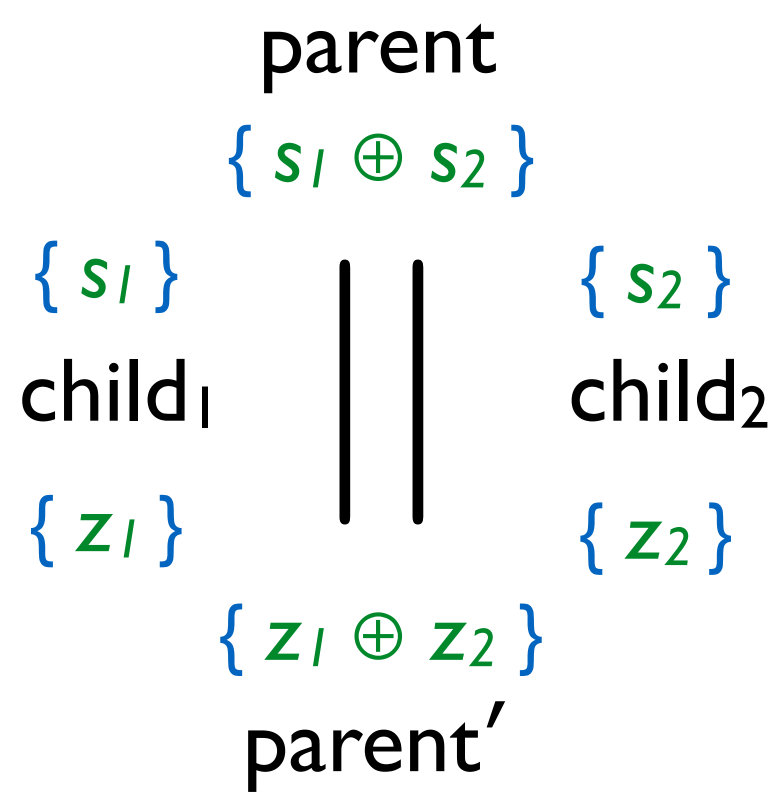
parent

$\{ s_1 \oplus s_2 \}$

$\{ s_1 \}$		$\{ s_2 \}$
child$_1$	$\|\|$	child$_2$
$\{ z_1 \}$		$\{ z_2 \}$

Logical state split





New state that belongs to parent′

Key ideas

- Subjectivity — reasoning with *self* and *other*
- PCMs
- Histories

Key ideas

- Subjectivity — reasoning with *self* and *other*
- PCMs — uniform way to logically *split* state
- Histories

Familiar PCM: finite heaps

Familiar PCM: finite heaps

- Heaps are partial finite maps *nat* → *Val*

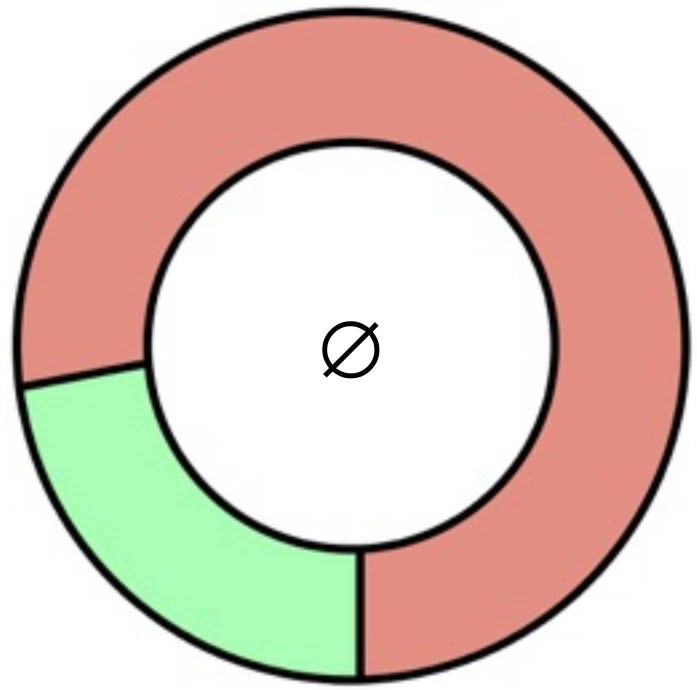
Familiar PCM: finite heaps

- Heaps are partial finite maps *nat* → *Val*
- Join operation $\oplus$ is disjoint union

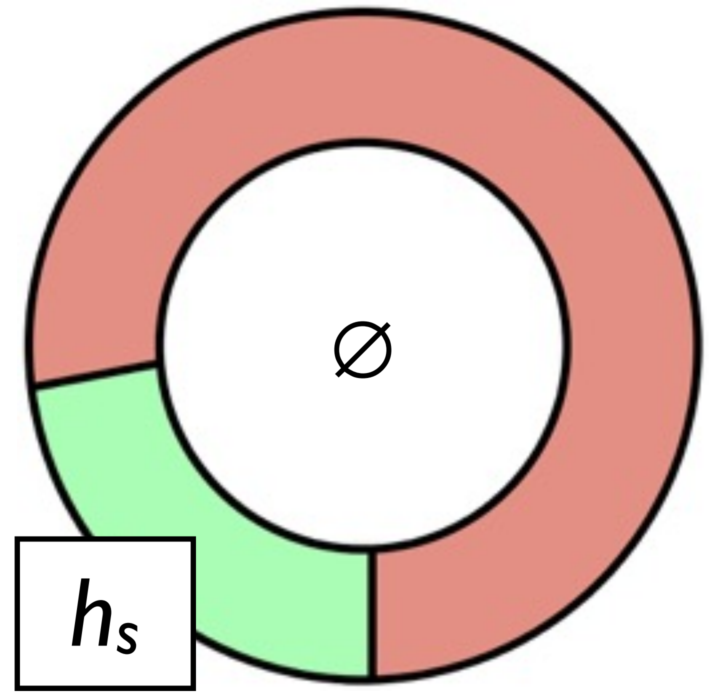
Familiar PCM: finite heaps

- Heaps are partial finite maps $nat \to Val$
- Join operation $\oplus$ is disjoint union
- Unit element $\mathbf{0}$ is the empty heap $\varnothing$

Concurroid for thread-local state

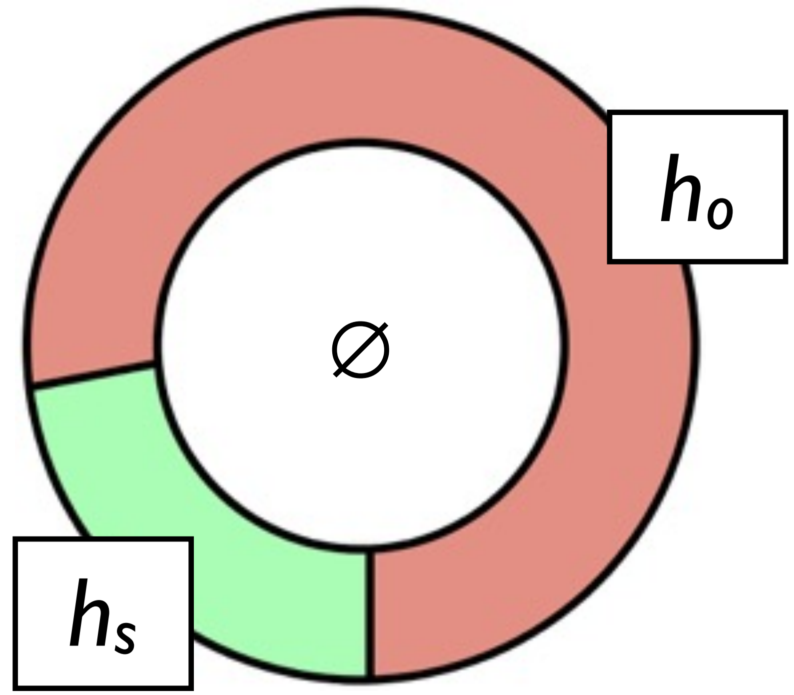


Concurroid for thread-local state



- h_s — heap, logically owned by this thread

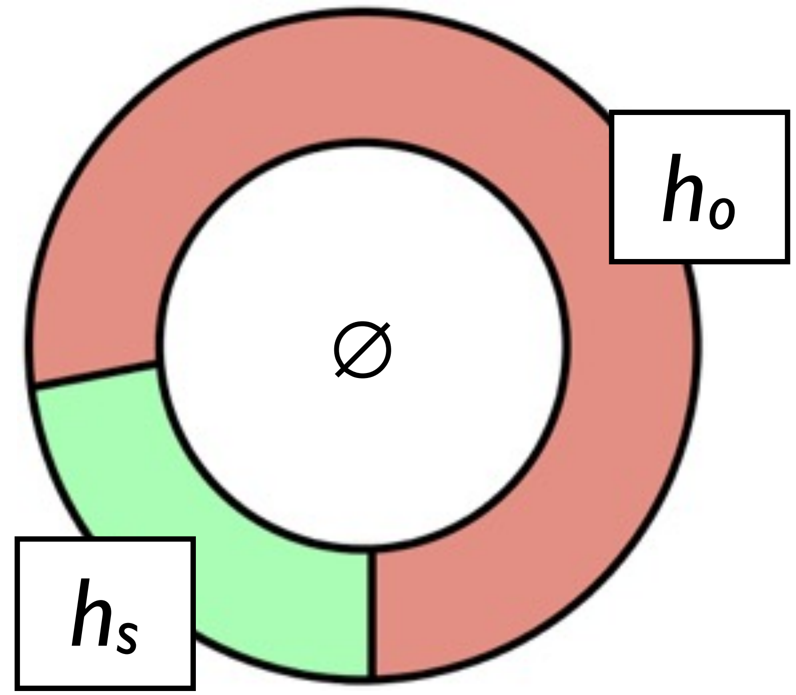
Concurroid for thread-local state



- h_s — heap, logically owned by this thread
- h_o — heap, owned by others

Concurroid for thread-local state

Concurrent Separation Logic
O'Hearn [CONCUR'04]



- h_s — heap, logically owned by this thread
- h_o — heap, owned by others

```
*x := 5;    ||    *y := 7;
```

$$\{\ h_s = x \mapsto - \oplus\ y \mapsto -\ \wedge\ \ h_o = h\ \}$$

```
*x := 5;        ||        *y := 7;
```

disjoint by resource definition

$$\{\; h_s = x \mapsto - \oplus y \mapsto - \;\wedge\; h_o = h \;\}$$

```
*x := 5;        ||        *y := 7;
```

disjoint by resource definition

$$\{\ h_s = x \mapsto - \oplus y \mapsto - \ \wedge\ h_o = h\ \}$$

$$\{\ h_s = x \mapsto - \wedge h_o = y \mapsto ? \oplus h\ \}$$

```
*x := 5;        ||        *y := 7;
```

disjoint by resource definition

$$\{\ h_s = x \mapsto - \oplus y \mapsto - \ \wedge\ h_o = h\ \}$$

$\{\ h_s = x \mapsto - \wedge h_o = y \mapsto ? \oplus h\ \}$

```
*x := 5;
```

$\|$

$\{\ h_s = y \mapsto - \wedge h_o = x \mapsto ? \oplus h\ \}$

```
*y := 7;
```

disjoint by resource definition

$$\{\ h_s = x \mapsto - \oplus y \mapsto - \ \wedge\ h_o = h\ \}$$

$\{\ h_s = x \mapsto - \wedge h_o = y \mapsto ? \oplus h\ \}$

```
*x := 5;
```

$\{\ h_s = x \mapsto 5 \wedge h_o = y \mapsto ? \oplus h\ \}$

$\|$

$\{\ h_s = y \mapsto - \wedge h_o = x \mapsto ? \oplus h\ \}$

```
*y := 7;
```

$\{\ h_s = y \mapsto 7 \wedge h_o = x \mapsto ? \oplus h\ \}$

disjoint by resource definition

$$\{\, h_s = x \mapsto - \oplus y \mapsto - \;\wedge\; h_o = h \,\}$$

$$\begin{array}{c||c}
\{\, h_s = x \mapsto - \wedge h_o = y \mapsto ? \oplus h \,\} & \{\, h_s = y \mapsto - \wedge h_o = x \mapsto ? \oplus h \,\} \\
\texttt{*x := 5;} & \texttt{*y := 7;} \\
\{\, h_s = x \mapsto 5 \wedge h_o = y \mapsto ? \oplus h \,\} & \{\, h_s = y \mapsto 7 \wedge h_o = x \mapsto ? \oplus h \,\}
\end{array}$$

$$\{\, h_s = x \mapsto 5 \oplus y \mapsto 7 \;\wedge\; h_o = h \,\}$$

Key ideas

- Subjectivity — reasoning with *self* and *other*
- PCMs — uniform way to logically *split* state
- Histories

Key ideas

- Subjectivity — reasoning with *self* and *other*
- PCMs — uniform way to logically *split* state
- Histories

Key ideas

- Subjectivity — reasoning with *self* and *other*

- PCMs — uniform way to logically *split* state

- Histories

Sergey et al. [ESOP'15]

Atomic stack specifications

`push(x)`

Atomic stack specifications

$\{\, S = xs \,\}$ `push(x)` $\{\, S' = x :: xs \,\}$

Atomic stack specifications

$t_k \rightarrow$	*xs*	*x* :: *xs*

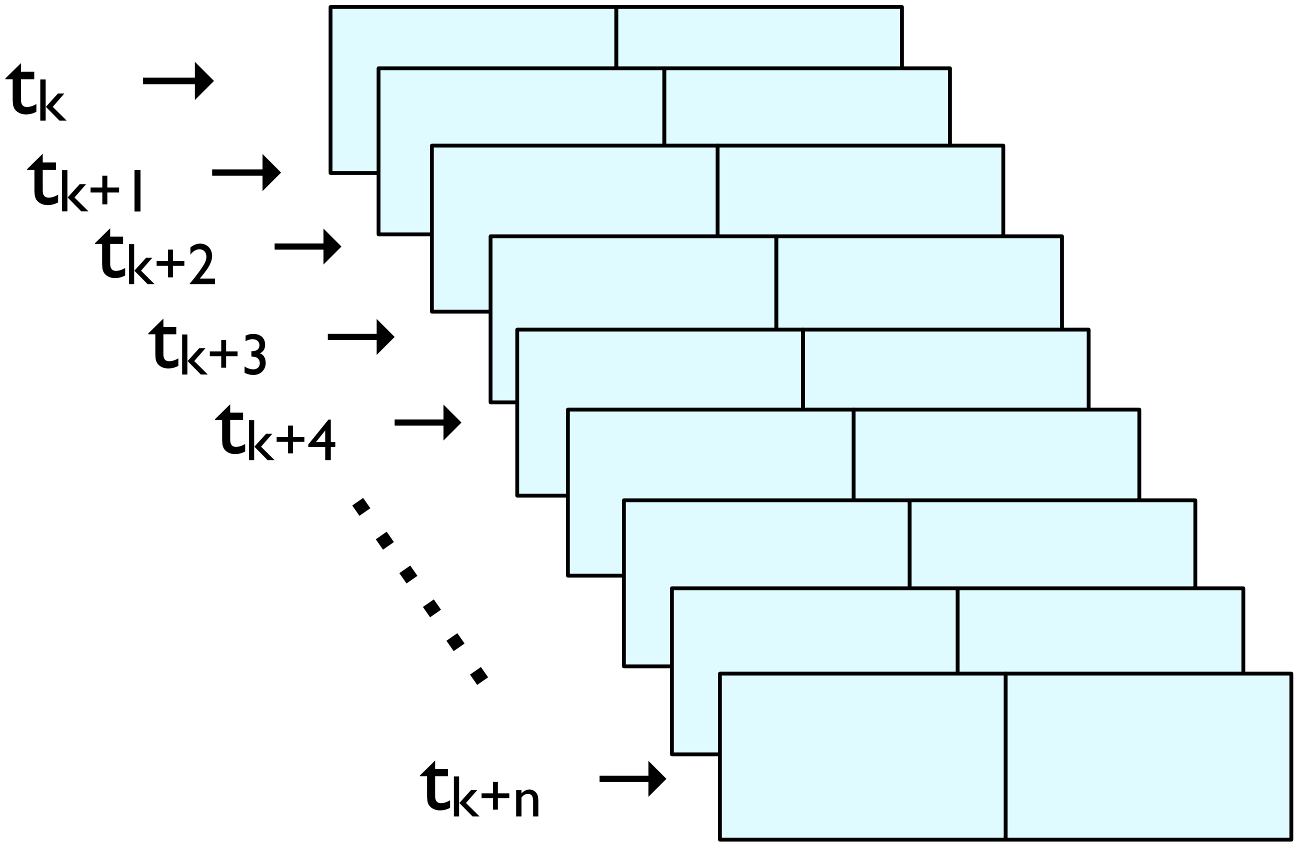
Atomic stack specifications

$t_k \rightarrow$

xs	$x :: xs$

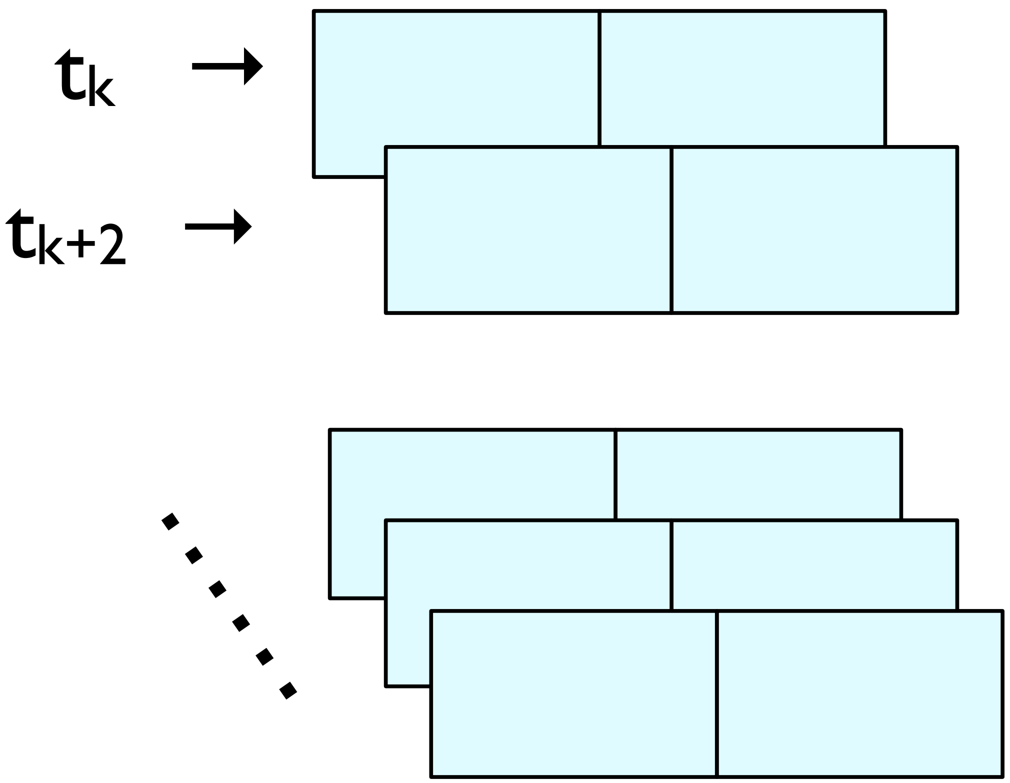
“timestamp”

$t_k \rightarrow$ 

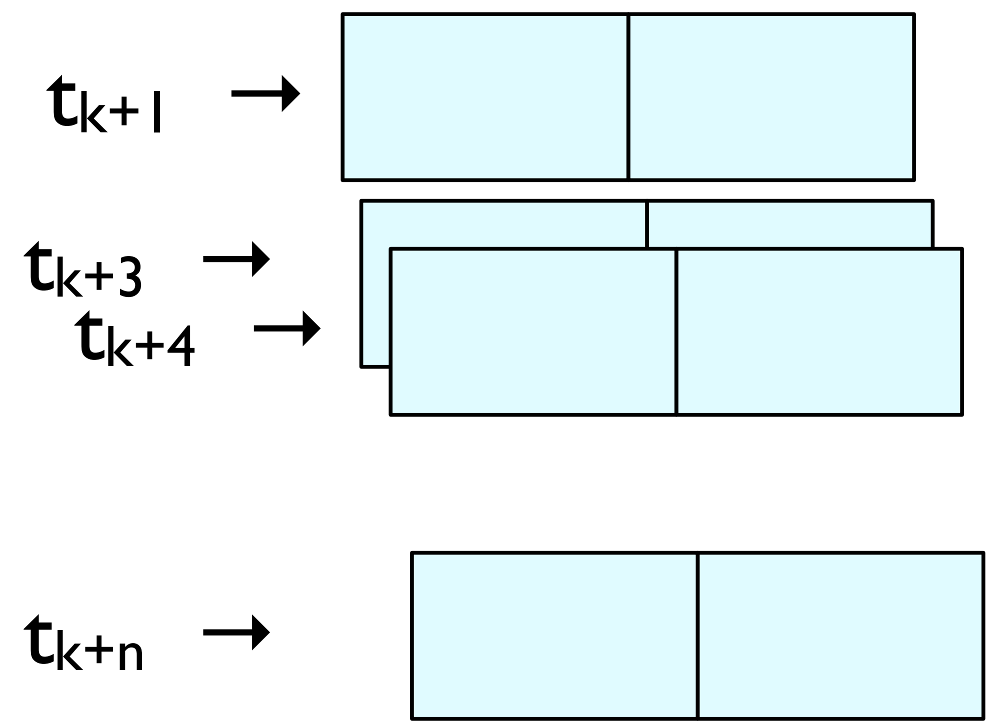


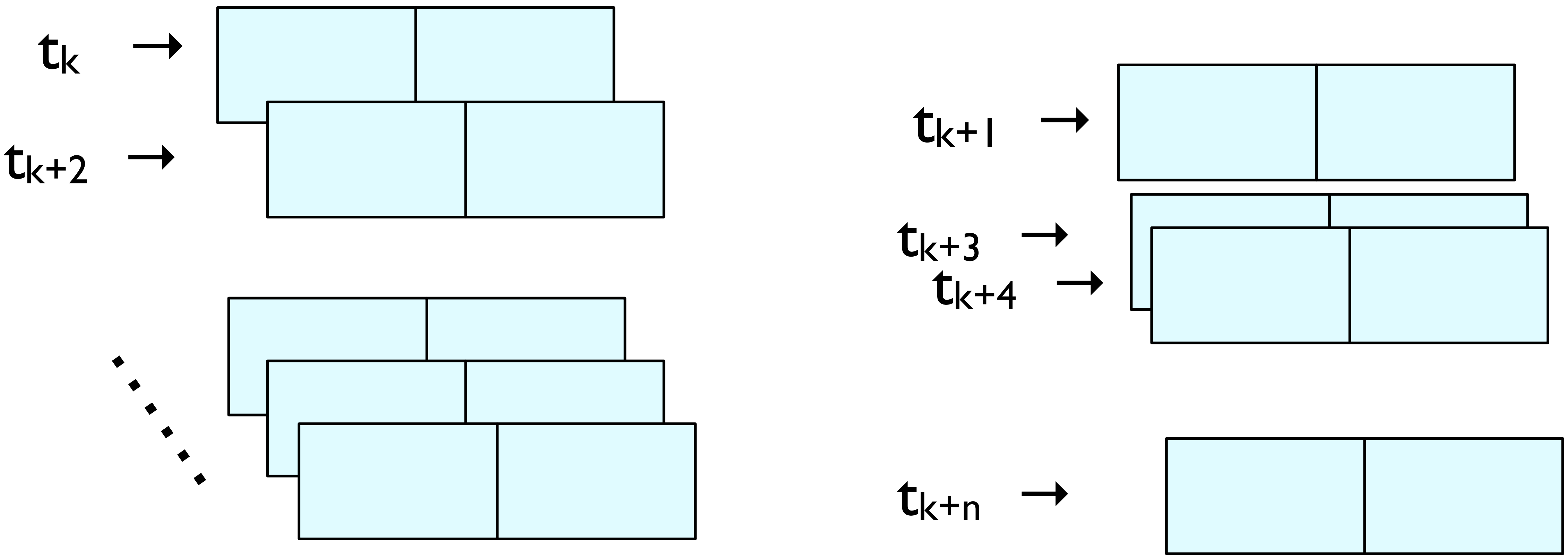
t_k
t_{k+1}
t_{k+2}
t_{k+3}
t_{k+4}
t_{k+n}

Changes by *this* thread

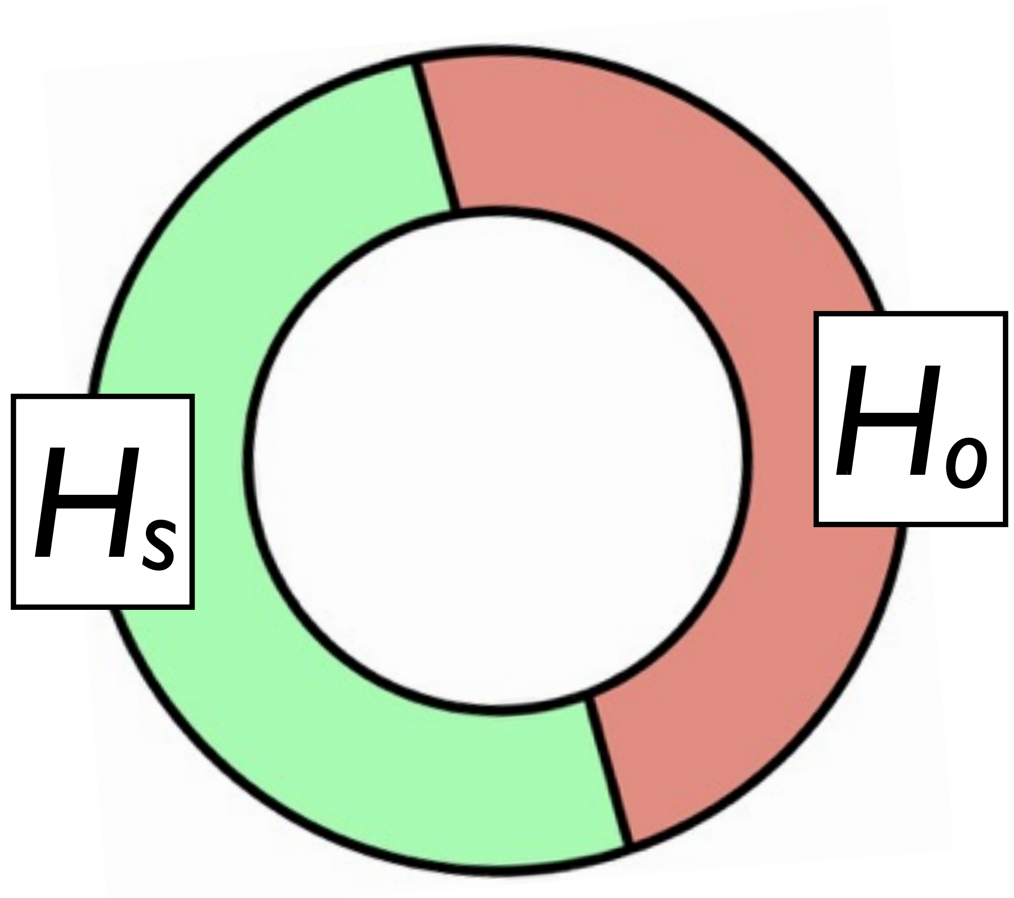


Changes by *other* threads





t_k
t_{k+2}
t_{k+1}
t_{k+3}
t_{k+4}
t_{k+n}



H_s, H_o — *self/other* contributions to the resource history

Histories are like heaps!

Histories are like heaps!

- Histories are partial finite maps *nat* → *AbsOp*

Histories are like heaps!

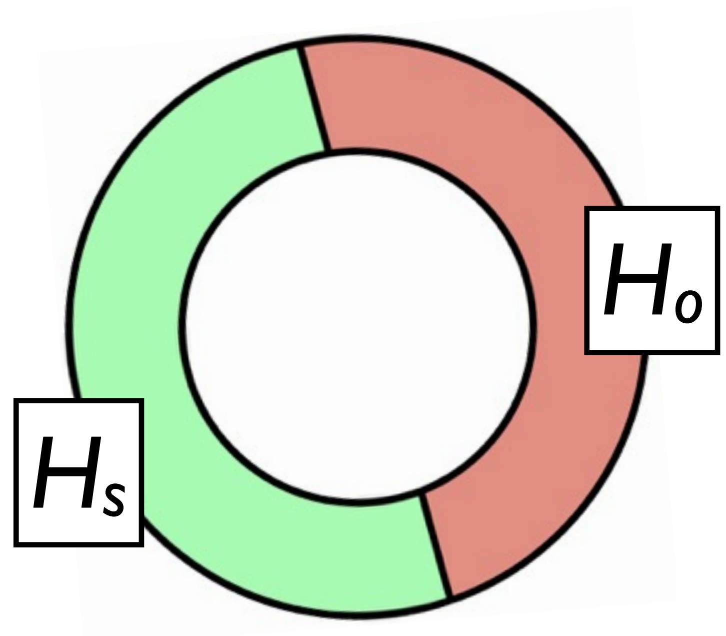
- Histories are partial finite maps $\mathit{nat} \rightharpoonup \mathit{AbsOp}$
- Join operation $\oplus$ is disjoint union

Histories are like heaps!

- Histories are partial finite maps $\mathit{nat} \to \mathit{AbsOp}$
- Join operation $\oplus$ is disjoint union
- Unit element $\mathbf{0}$ is the empty history $\varnothing$

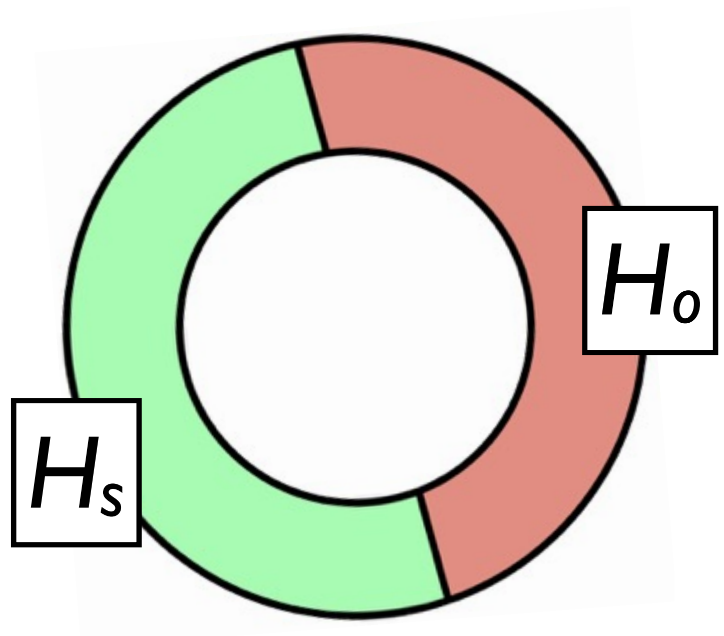
Specifying stacks with histories

C_{stack} =



Specifying stacks with histories

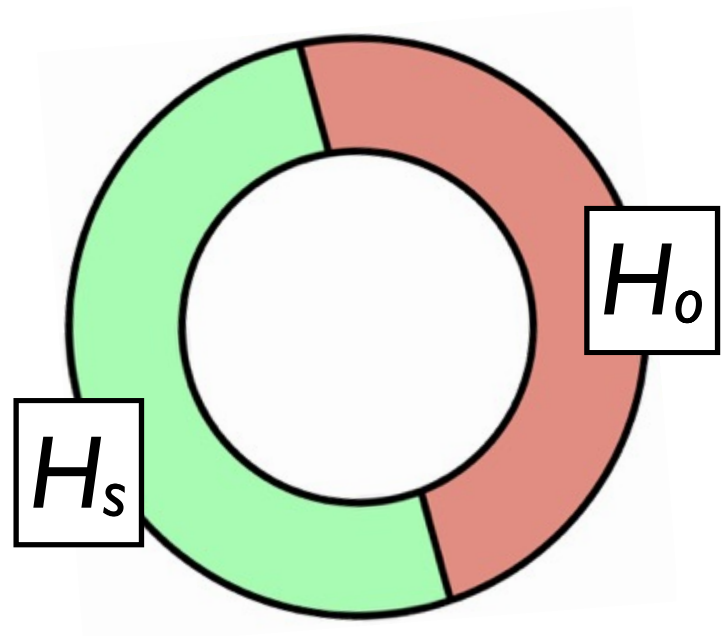
C_{stack} =



- H_s, H_o = { $t_k \mapsto$ (xs, x::xs), $t_n \mapsto$ (x::xs, xs), … }

Specifying stacks with histories

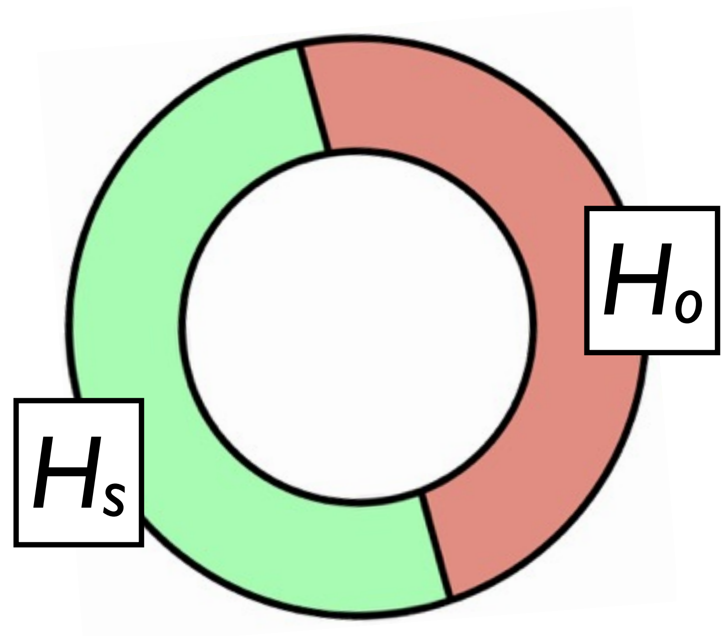
C_{stack} =



- H_s, H_o = { $t_k \mapsto$ (xs, x::xs), $t_n \mapsto$ (x::xs, xs), … }
- *Joint* part is specific for each implementation

Specifying stacks with histories

C_{stack} =



- H_s, H_o = { $t_k \mapsto$ (xs, x::xs), $t_n \mapsto$ (x::xs, xs), … }
- *Joint* part is specific for each implementation
- Adjacent history entries agree on overlapping abstract states

Stack specification

$$\{\ H_s = \varnothing \wedge \mathrm{H} \subseteq H_o\ \}$$

```
push(x)
```

$$\{\ \exists \mathrm{t}, \mathrm{xs}.\ \ H_s = \mathrm{t} \mapsto (\mathrm{xs}, \mathrm{x::xs}) \wedge \mathrm{H} \subseteq H_o \wedge \mathrm{H} < \mathrm{t}\ \}@\mathrm{C}_{\mathrm{stack}}$$

Stack specification

self-contribution is a single entry

$$\{\ \boxed{H_s = \emptyset} \wedge \mathrm{H} \subseteq H_o\ \}$$

`push(x)`

$$\{\ \exists t, xs.\ \boxed{H_s = t \mapsto (xs, x{::}xs)} \wedge \mathrm{H} \subseteq H_o \wedge \mathrm{H} < t\ \}@C_{stack}$$

Stack specification

t allocated during the call

$$\{\, H_s = \emptyset \;\wedge\; \boxed{H \subseteq H_0} \,\}$$

```
push(x)
```

$$\{\, \exists t, xs.\;\; H_s = t \mapsto (xs, x{::}xs) \;\wedge\; \boxed{H \subseteq H_0} \wedge \boxed{H < t} \,\} @ C_{stack}$$

Stack specification

$$\{\, H_s = \varnothing \wedge \mathrm{H} \subseteq H_o \,\}$$

`pop( )`

$$\begin{aligned}
\{\ \mathrm{res}.\ \ &\mathbf{if}\ \ (res = Some\ \mathrm{x}) \\
&\mathbf{then}\ \ \exists \mathrm{t}, \mathrm{xs}.\ \mathrm{H} \subseteq H_o \wedge \mathrm{H} < \mathrm{t} \wedge\ H_s = \mathrm{t} \mapsto (\mathrm{x{::}xs}, \mathrm{xs})) \\
&\mathbf{else}\ \ \ \exists \mathrm{t}.\ \mathrm{H} \subseteq H_o \wedge \mathrm{H} \leq \mathrm{t} \\
&\qquad\qquad \wedge\ H_s = \varnothing \wedge \mathrm{t} \mapsto (_, \mathbf{Nil}) \subseteq H_o\ \}@C_{\mathrm{stack}}
\end{aligned}$$

Stack specification

$$\{\, H_s = \varnothing \wedge \mathrm{H} \subseteq H_o \,\}$$

```
pop( )
```

$$\begin{aligned}
\{\ \mathrm{res}.\ \ &\textbf{if}\ \ (\mathit{res} = \mathit{Some}\ \mathrm{x})\\
&\textbf{then}\ \ \exists \mathrm{t}, \mathrm{xs}.\ \mathrm{H} \subseteq H_o \wedge \mathrm{H} < \mathrm{t} \wedge H_s = \mathrm{t} \mapsto (\mathrm{x}::\mathrm{xs}, \mathrm{xs}))\\
&\textbf{else}\ \ \exists \mathrm{t}.\ \mathrm{H} \subseteq H_o \wedge \boxed{\mathrm{H} \leq \mathrm{t}}\\
&\qquad \wedge H_s = \varnothing \wedge \boxed{\mathrm{t} \mapsto (_, \mathbf{Nil}) \subseteq H_o}\ \}@C_{stack}
\end{aligned}$$

- pop has hit **Nil** during its execution at the moment t

Stack specification

no self-contributions initially?

$$\{\, \boxed{H_s = \varnothing} \wedge \mathsf{H} \subseteq H_o \,\}$$

```
pop()
```

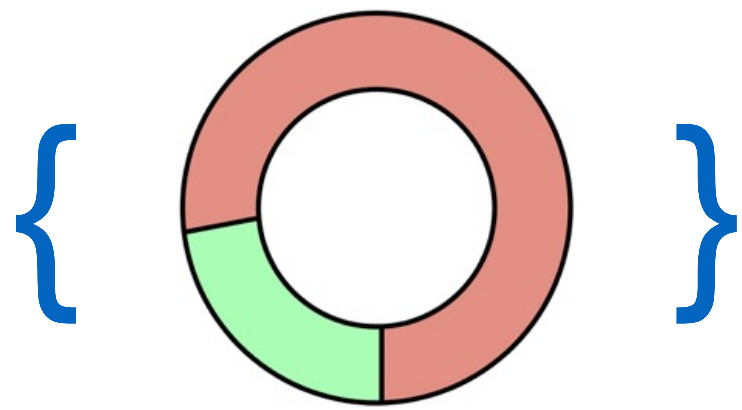
$$\{\, \mathsf{res}.\ \ \textbf{if}\ \ (res = Some\ \mathsf{x})$$
$$\textbf{then}\ \ \exists \mathsf{t}, \mathsf{xs}.\ \mathsf{H} \subseteq H_o \wedge \mathsf{H} < \mathsf{t} \wedge H_s = \mathsf{t} \mapsto (\mathsf{x::xs}, \mathsf{xs}))$$
$$\textbf{else}\ \ \exists \mathsf{t}.\ \mathsf{H} \subseteq H_o \wedge \mathsf{H} \leq \mathsf{t}$$
$$\wedge\ H_s = \varnothing \wedge \mathsf{t} \mapsto (\_, \mathbf{Nil}) \subseteq H_o \,\}@C_{stack}$$

Framing in FCSL

Framing in FCSL

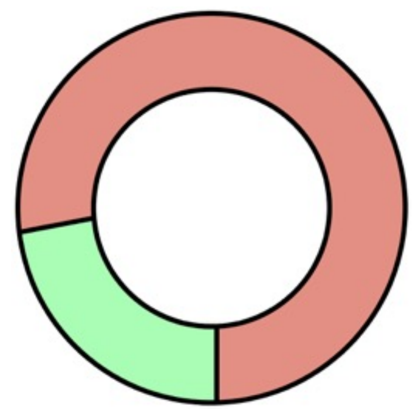
`my_program`

Framing in FCSL

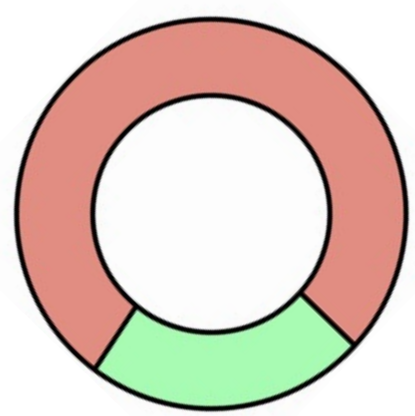


`my_program`

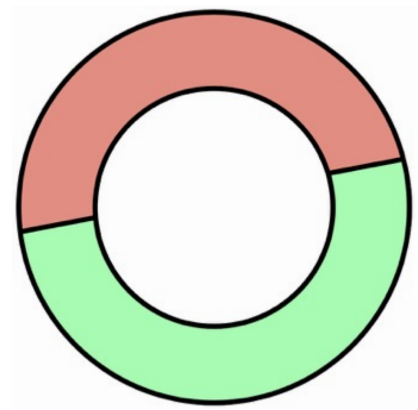
Framing in FCSL

{  }

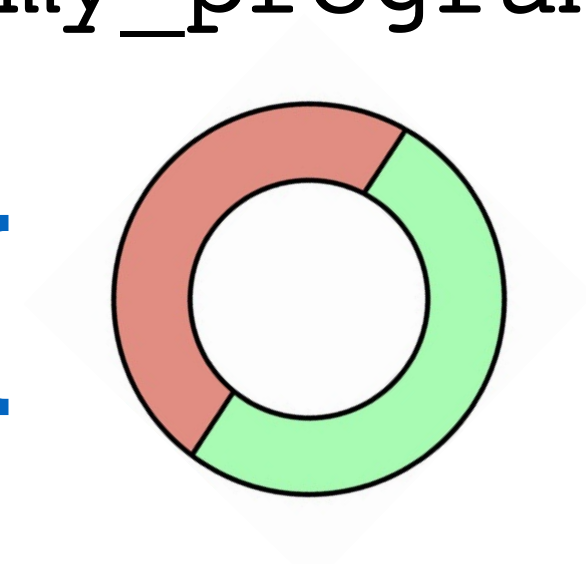
`my_program`

{  }

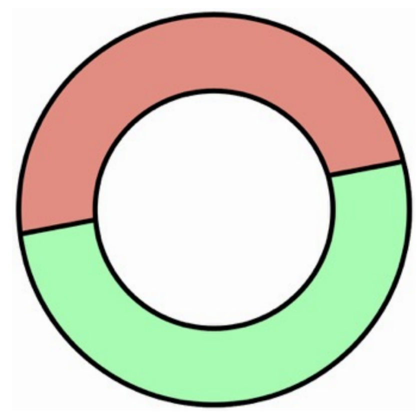
Framing in FCSL

{  }

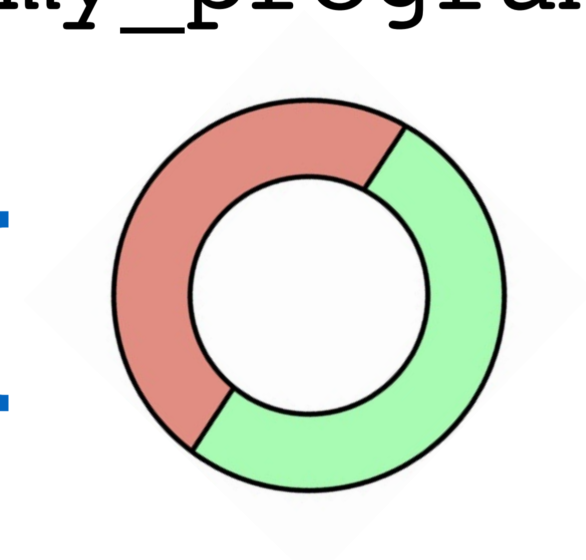
`my_program`

{  }

Framing in FCSL

{  }

`my_program`

{  }

Works for *any* PCM, not just heaps!

Framing histories

$\{ H_s = \varnothing \wedge \mathrm{H} \subseteq H_o \}$

```
push(x)
```

$\{ \exists \mathrm{t}, \mathrm{xs}.\ \mathrm{H} \subseteq H_o \wedge \mathrm{H} < \mathrm{t} \wedge H_s = \mathrm{t} \mapsto (\mathrm{xs}, \mathrm{x::xs}) \}@C_{\mathrm{stack}}$

Framing histories

$$\{\ \mathit{H_s} = \mathrm{H_1} \wedge \mathrm{H_2} \subseteq \mathit{H_o}\ \}$$

```
push(x)
```

$$\{\ \exists t, xs.\ \mathrm{H_2} \subseteq \mathit{H_o} \wedge \mathrm{H_1} \oplus \mathrm{H_2} < t \wedge \mathit{H_s} = \mathrm{H_1} \oplus t \mapsto (xs, x{::}xs)\ \}@C_{stack}$$

Framing histories

$$\{\ \boxed{H_s = \mathrm{H}_1} \wedge \mathrm{H}_2 \subseteq H_o\ \}$$

initial self-contribution

```
push(x)
```

$$\{\ \exists t, xs.\ \mathrm{H}_2 \subseteq H_o \wedge \mathrm{H}_1 \oplus \mathrm{H}_2 < t \wedge H_s = \mathrm{H}_1 \oplus t \mapsto (xs, x{::}xs)\ \}@C_{stack}$$

Framing histories

$$\{\ \mathcal{H}_s = H_1 \wedge H_2 \subseteq \mathcal{H}_o\ \}$$

`push(x)`

final self-contribution

$$\{\ \exists t, xs.\ H_2 \subseteq \mathcal{H}_o \wedge H_1 \oplus H_2 < t \wedge \boxed{\mathcal{H}_s = H_1 \oplus t \mapsto (xs, x{::}xs)}\ \}@C_{stack}$$

Key ideas

- Subjectivity — reasoning with *self* and *other*
- PCMs — uniform way to logically *split* state
- Histories

Key ideas

- Subjectivity — reasoning with *self* and *other*
- PCMs — uniform way to logically *split* state
- Histories — logical updates via auxiliary state

How useful are histories for clients?

A stack client program

- Two threads: *producer* and *consumer*
- `Ap` — an `n`-element *producer* array
- `Ac` — an `n`-element *consumer* array
- A *shared* concurrent stack `S` is used as a buffer
- The goal: prove the exchange correct

Auxiliary Predicates

- Pushed *H E* *iff*
  E is a multiset of elements, *pushed* in *H*

- Popped *H E* *iff*
  E is a multiset of elements, *popped* in *H*

```
letrec produce(i : nat) = {
  if (i == n)
  then return;
  else {
    S.push(Ap[i]);
    produce(i+1);
  }
}
```

$\{ \texttt{Ap} \mapsto L \land \ \text{Pushed}\ H_s\ L[< i] \ \land\ \text{Popped}\ H_s\ \varnothing \ \}$

```
letrec produce(i : nat) = {
  if (i == n)
  then return;
  else {
    S.push(Ap[i]);
    produce(i+1);
  }
}
```

$\{ \texttt{Ap} \mapsto L \land \ \text{Pushed}\ H_s\ L[< n] \ \land\ \text{Popped}\ H_s\ \varnothing \ \}$

$\{\ \mathtt{Ap} \mapsto L \wedge \boxed{\text{Pushed } H_s\ L[< i]} \wedge \text{ Popped } H_s\ \varnothing\ \}$

```
letrec produce(i : nat) = {
  if (i == n)
  then return;
  else {
    S.push(Ap[i]);
    produce(i+1);
  }
}
```

$\{\ \mathtt{Ap} \mapsto L \wedge \boxed{\text{Pushed } H_s\ L[< n]} \wedge \text{ Popped } H_s\ \varnothing\ \}$

```
letrec consume(i : nat) = {
  if (i == n)
  then return;
  else {
    t ← S.pop();
    if t == Some v
    then {
      Ac[i] := v;
      consume(i+1);
    }
    else consume(i);
  }
}
```

$\{\exists L,\ \texttt{Ac} \mapsto L \wedge \ \text{Pushed}\ H_s\ \varnothing \ \wedge \ \text{Popped}\ H_s\ L[< i]\ \}$

```
letrec consume(i : nat) = {
  if (i == n)
  then return;
  else {
    t ← S.pop();
    if t == Some v
    then {
      Ac[i] := v;
      consume(i+1);
    }
    else consume(i);
  }
}
```

$\{\exists L,\ \texttt{Ac} \mapsto L \wedge \ \text{Pushed}\ H_s\ \varnothing \wedge \ \text{Popped}\ H_s\ L[< n]\ \}$

$$\{\exists L, \mathtt{Ac} \mapsto L \land \text{Pushed } H_s\ \varnothing \land \boxed{\text{Popped } H_s\ L[< i]}\}$$

```
letrec consume(i : nat) = {
  if (i == n)
  then return;
  else {
    t ← S.pop();
    if t == Some v
    then {
      Ac[i] := v;
      consume(i+1);
    }
    else consume(i);
  }
}
```

$$\{\exists L, \mathtt{Ac} \mapsto L \land \text{Pushed } H_s\ \varnothing \land \boxed{\text{Popped } H_s\ L[< n]}\}$$

`produce(0)` || `consume(0)`

$$\textbf{hide}\ C_{stack}(\texttt{h}_{\texttt{s}})\ \textbf{in}$$

$$\left\{ \quad \texttt{produce(0)} \quad \Bigg\| \quad \texttt{consume(0)} \quad \right\}$$

No other threads can interfere on `S`

$$\texttt{hide}\ C_{stack}(\texttt{h}_{\texttt{S}})\ \texttt{in}$$

$$\left\{\quad \texttt{produce(0)} \quad \Big\| \quad \texttt{consume(0)} \quad\right\}$$

$$\{\, \texttt{Ap} \mapsto L \oplus \texttt{Ac} \mapsto L' \oplus \mathsf{h_S} \,\}$$

$$\textbf{hide}\ \ C_{\mathsf{stack}}(\texttt{h}_{\mathsf{S}})\ \ \textbf{in}$$

$$\left\{ \quad \texttt{produce(0)} \quad \Big\| \quad \texttt{consume(0)} \quad \right\}$$

$$\{\, \mathtt{Ap} \mapsto L \oplus \mathtt{Ac} \mapsto L' \oplus \mathsf{h_S} \,\}$$

$$\mathbf{hide}\ \mathsf{C_{stack}}(\mathtt{h_S})\ \mathbf{in}$$

$$\left\{ \begin{array}{c|c} \{\, \mathtt{Ap} \mapsto L & \{\, \mathtt{Ac} \mapsto L' \\ \mathtt{produce(0)} & \mathtt{consume(0)} \end{array} \right\}$$

$$\{\, \texttt{Ap} \mapsto L \oplus \texttt{Ac} \mapsto L' \oplus \mathrm{h_S} \,\}$$

$$\textbf{hide}\ \mathsf{C}_{\mathsf{stack}}(\texttt{h}_{\texttt{S}})\ \textbf{in}$$

$$\left\{ \begin{array}{c|c} \begin{array}{c} \{\, \texttt{Ap} \mapsto L \\ \wedge\ \mathsf{Pushed}\ H_s\,\varnothing\ \ \wedge\ \ \mathsf{Popped}\ H_s\,\varnothing \,\} \\ \texttt{produce(0)} \end{array} & \begin{array}{c} \{\, \texttt{Ac} \mapsto L' \\ \wedge\ \mathsf{Pushed}\ H_s\,\varnothing\ \ \wedge\ \ \mathsf{Popped}\ H_s\,\varnothing \,\} \\ \texttt{consume(0)} \end{array} \end{array} \right\}$$

$$\{\, \texttt{Ap} \mapsto L \oplus \texttt{Ac} \mapsto L' \oplus \mathrm{h_S} \,\}$$

$$\textbf{hide}\ \mathsf{C}_{\mathsf{stack}}(\texttt{h}_{\texttt{S}})\ \textbf{in}$$

$$\left\{ \begin{array}{c|c} \begin{array}{c} \{\, \texttt{Ap} \mapsto L \\ \wedge\ \text{Pushed}\ H_s\ \emptyset\ \ \wedge\ \ \text{Popped}\ H_s\ \emptyset\ \} \\ \\ \texttt{produce(0)} \\ \\ \{\, \texttt{Ap} \mapsto L\ \ \wedge \\ \text{Pushed}\ H_s\ L[< n]\ \ \wedge \\ \text{Popped}\ H_s\ \emptyset\ \} \end{array} & \begin{array}{c} \{\, \texttt{Ac} \mapsto L' \\ \wedge\ \text{Pushed}\ H_s\ \emptyset\ \ \wedge\ \ \text{Popped}\ H_s\ \emptyset\ \} \\ \\ \texttt{consume(0)} \\ \\ \{\, \texttt{Ac} \mapsto L''\ \wedge \\ \text{Pushed}\ H_s\ \emptyset\ \ \wedge \\ \text{Popped}\ H_s\ L''[\text{<n}]\ \} \end{array} \end{array} \right\}$$

$$\{\, \mathtt{Ap} \mapsto L \oplus \mathtt{Ac} \mapsto L' \oplus h_S \,\}$$

hide $C_{\text{stack}}(\mathtt{h_S})$ **in**

$$\left\{\begin{array}{c||c}
\{\, \mathtt{Ap} \mapsto L & \{\, \mathtt{Ac} \mapsto L' \\
\land\ \text{Pushed } H_s\, \varnothing \ \land\ \text{Popped } H_s\, \varnothing \,\} & \land\ \text{Pushed } H_s\, \varnothing \ \land\ \text{Popped } H_s\, \varnothing \,\} \\
\mathtt{produce(0)} & \mathtt{consume(0)} \\
\{\, \mathtt{Ap} \mapsto L \ \land & \{\, \mathtt{Ac} \mapsto L'' \ \land \\
\boxed{\text{Pushed } H_s\, L[<n]} \ \land & \text{Pushed } H_s\, \varnothing \ \land \\
\text{Popped } H_s\, \varnothing \,\} & \boxed{\text{Popped } H_s\, L''[<n]} \,\}
\end{array}\right\}$$

These are the *only* changes
in the stack's history

$$\{\, \texttt{Ap} \mapsto L \oplus \texttt{Ac} \mapsto L' \oplus \mathrm{h_S} \,\}$$

$$\textbf{hide}\ \mathrm{C_{stack}}(\texttt{hs})\ \textbf{in}$$

$$\left\{ \begin{array}{c|c} \begin{array}{c} \{\, \texttt{Ap} \mapsto L \\ \wedge\ \text{Pushed}\ H_s\ \varnothing\ \ \wedge\ \ \text{Popped}\ H_s\ \varnothing \,\} \\[1ex] \texttt{produce(0)} \\[1ex] \{\, \texttt{Ap} \mapsto L\ \ \wedge \\ \text{Pushed}\ H_s\ L[< n]\ \ \wedge \\ \text{Popped}\ H_s\ \varnothing \,\} \end{array} & \begin{array}{c} \{\, \texttt{Ac} \mapsto L' \\ \wedge\ \text{Pushed}\ H_s\ \varnothing\ \ \wedge\ \ \text{Popped}\ H_s\ \varnothing \,\} \\[1ex] \texttt{consume(0)} \\[1ex] \{\, \texttt{Ac} \mapsto L''\ \wedge \\ \text{Pushed}\ H_s\ \varnothing\ \ \wedge \\ \text{Popped}\ H_s\ L''[<\mathrm{n}] \,\} \end{array} \end{array} \right\}$$

$$\{\, \texttt{Ap} \mapsto L \oplus \texttt{Ac} \mapsto L'' \oplus \mathrm{hs}' \wedge L =_{set} L'' \}$$

$$\{\,\mathtt{Ap} \mapsto L \oplus \mathtt{Ac} \mapsto L' \oplus \mathrm{hs}\,\}$$

hide $C_{stack}(\mathtt{h_s})$ **in**

$$\left\{ \begin{array}{c|c} \begin{array}{c} \{\,\mathtt{Ap} \mapsto L \\ \wedge\ \mathrm{Pushed}\ H_s\ \varnothing\ \ \wedge\ \ \mathrm{Popped}\ H_s\ \varnothing\ \} \\ \mathtt{produce(0)} \\ \{\,\mathtt{Ap} \mapsto L\ \ \wedge \\ \mathrm{Pushed}\ H_s\ L[< n]\ \ \wedge \\ \mathrm{Popped}\ H_s\ \varnothing\ \} \end{array} & \begin{array}{c} \{\,\mathtt{Ac} \mapsto L' \\ \wedge\ \mathrm{Pushed}\ H_s\ \varnothing\ \ \wedge\ \ \mathrm{Popped}\ H_s\ \varnothing\ \} \\ \mathtt{consume(0)} \\ \{\,\mathtt{Ac} \mapsto L''\ \wedge \\ \mathrm{Pushed}\ H_s\ \varnothing\ \ \wedge \\ \mathrm{Popped}\ H_s\ L''[<n]\ \} \end{array} \end{array} \right\}$$

$$\{\,\mathtt{Ap} \mapsto L \oplus \mathtt{Ac} \mapsto L'' \oplus \mathrm{hs}' \wedge \boxed{L =_{set} L''}\,\}$$

More use for histories

(see the paper)

More use for histories

(see the paper)

- Verifying atomic snapshots

More use for histories

(see the paper)

- Verifying atomic snapshots
- Instantiating higher-order concurrent structures

More use for histories

(see the paper)

- Verifying atomic snapshots
- Instantiating higher-order concurrent structures
- Deriving sequential specifications via hiding

More use for histories

(see the paper)

- Verifying atomic snapshots
- Instantiating higher-order concurrent structures
- Deriving sequential specifications via hiding

To take away

To take away

- Histories as auxiliary state
  - Expressive abstraction for concurrent specs

To take away

- Histories as auxiliary state
  - Expressive abstraction for concurrent specs
- Histories are a PCM
  - They are subject of the same rules as heaps

To take away

- **Histories as auxiliary state**
  - Expressive abstraction for concurrent specs
- **Histories are a PCM**
  - They are subject of the same rules as heaps
- **Historical reasoning requires subjectivity**
  - History-based specs often talk about the effect of *other* threads

To take away

- **Histories as auxiliary state**
  - Expressive abstraction for concurrent specs
- **Histories are a PCM**
  - They are subject of the same rules as heaps
- **Historical reasoning requires subjectivity**
  - History-based specs often talk about the effect of *other* threads

`software.imdea.org/fcsl`

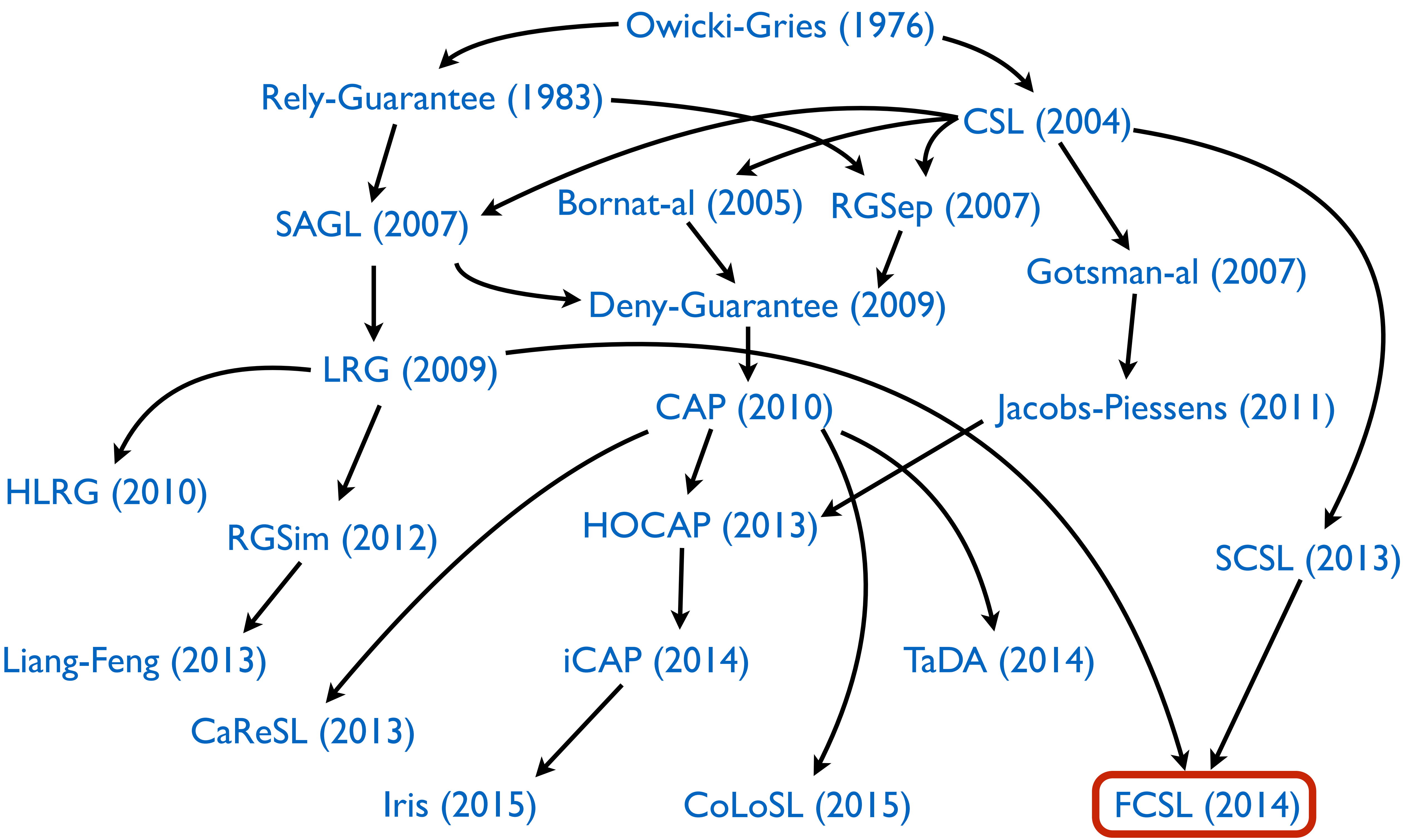
To take away

- **Histories as auxiliary state**
  - Expressive abstraction for concurrent specs
- **Histories are a PCM**
  - They are subject of the same rules as heaps
- **Historical reasoning requires subjectivity**
  - History-based specs often talk about the effect of *other* threads

`software.imdea.org/fcsl`

Thanks!

Q&A slides



Owicki-Gries (1976)
Rely-Guarantee (1983)
CSL (2004)
SAGL (2007)
Bornat-al (2005)
RGSep (2007)
Gotsman-al (2007)
Deny-Guarantee (2009)
LRG (2009)
CAP (2010)
Jacobs-Piessens (2011)
HLRG (2010)
RGSim (2012)
HOCAP (2013)
SCSL (2013)
Liang-Feng (2013)
iCAP (2014)
TaDA (2014)
CaReSL (2013)
Iris (2015)
CoLoSL (2015)
FCSL (2014)

How is your stuff different from other existing concurrent logics?

- [Owicki-Gries:CACM76] - reasoning about parallel composition is not compositional; *subjectivity fixes that*;
- [OHearn:CONCUR04] - only one type of resources — critical sections;
  FCSL allows one to define arbitrary resources;
- [Feng-al:ESOP07, Vafeiadis-Parkinson:CONCUR07] - framing over Rely/Guarantee, but only one shared resource: *FCSL allows multiple ones;*
- [Feng:POPL09] - introduced local Rely/Guarantee; *FCSL improves on it by introducing a subjective state and explicitly identifying resources as STS;*
- [DinsdaleYoung-al:ECOOP10] - first introduced concurred protocols;
  FCSL generalises permissions - self-state defines what a thread is allowed to do with a resource;
- [DinsdaleYoung-al:POPL13] - general framework for concurrency logic;
  FCSL is a particular logic, not clear whether it is an instance of Views;
- [Turon-al:ICFP13] - CaReSL and reasoning about contextual refinement;
  FCSL doesn't address CR, in our experience it's never required for Hoare-style reasoning;
- [Svendsen-al:ESOP13, ESOP14] - use much richer semantic domain,
  FCSL uses transitions and communication instead of view-shifts for changes in state and composition of resources;
- [Raad-al:ESOP15] - different notion of subjectivity, *no self/other dichotomy, no observation made about PCMs.*

FCSL's assertions work explicitly with state variables.

How is your stuff different from Iris?

Jung-al [POPL'15]

- Iris makes the same observations as FCSL did in 2014 (PCMs, Invariants);
- Iris doesn't have hiding and self/other dichotomy;
- It considers more primitive "building blocks" and encodes protocols as STSs + interpretation;
    - This encoding is made default in FCSL, and so far it suffices;
- Currently, FCSL doesn't support abstract atomicity in Iris/iCAP sense (however, it can recover most of it through the choice of PCMs).

Encoding verification in FCSL

Encoding verification in FCSL

```
Program Definition my_prog: STSep (p, q) :=
                   Do c.
```

Encoding verification in FCSL

```
Program Definition my_prog: STSep (p, q) :=
                   Do c.
```

has type `STSep (p*, q*)`

- Program c's *weakest pre-* and *strongest postconditions* `(p*, q*)` *wrt.* safety, inferred from the types of basic commands (`ret`, `par`, `bind`);

Encoding verification in FCSL

```
Program Definition my_prog: STSep (p, q) :=
                    Do c.
```

Notation for `do (_ : (p*, q*) ⊑ (p, q)) c`

- Program c's *weakest pre-* and *strongest postconditions* `(p*, q*)` *wrt.* safety, inferred from the types of basic commands (`ret`, `par`, `bind`);
- `Do` encodes the application of the rule of consequence `(p*, q*) ⊑ (p, q)`;

Encoding verification in FCSL

```
Program Definition my_prog: STSep (p, q) :=
                              Do c.
```

Notation for `do (_ : (p*, q*) ⊑ (p, q)) c`

- Program c's *weakest pre-* and *strongest postconditions* `(p*, q*)` *wrt.* safety, inferred from the types of basic commands (`ret`, `par`, `bind`);
- `Do` encodes the application of the rule of consequence `(p*, q*) ⊑ (p, q)`;
- The client constructs the proof of `(p*, q*) ⊑ (p, q)` interactively;

Encoding verification in FCSL

```
Program Definition my_prog: STSep (p, q) :=
                   Do c.
```

Notation for `do (_ : (p*, q*) ⊑ (p, q)) c`

- Program c's *weakest pre-* and *strongest postconditions* `(p*, q*)` *wrt.* safety, inferred from the types of basic commands (`ret`, `par`, `bind`);
- `Do` encodes the application of the rule of consequence `(p*, q*) ⊑ (p, q)`;
- The client constructs the proof of `(p*, q*) ⊑ (p, q)` interactively;
- The obligations are reduced via *structural lemmas* (inference rules).

Implementation and evaluation

Program	Libs	Conc	Acts	Stab	Main	**Total**	**Build**
CAS-lock	63	291	509	358	27	1248	1m 1s
Ticketed lock	58	310	706	457	116	1647	2m 46s
Increment	26	-	-	-	44	70	8s
Allocator	82	-	-	-	192	274	14s
Pair snapshot	167	233	107	80	51	638	4m 7s
Treiber stack	56	323	313	133	155	980	2m 41s
Spanning tree	348	215	162	217	305	1247	1m 11s
Flat combiner	92	442	672	538	281	2025	10m 55s
Seq. stack	65	-	-	-	125	190	1m 21s
FC-stack	50	-	-	-	114	164	44s
Prod/Cons	365	-	-	-	243	608	2m 43s

Proof of `push` specification

Proof of `push` specification

```
Next Obligation.
apply: gh=>i [h hS][B][v] X C.
case: C (C) X=>_ [_][xa][->{i}][_][xp][xt][->] Cp Ct Ca C [P S H].
rewrite (getC Cp C) !(getC Ct C) /= in P S H.
rewrite -!joinA joinCA in C *.
apply: step; apply: val_extend=>//;apply: (gh_ex h); apply:
val_do=>//.
move=>p' {xt S H C Ct Ca} xt [t][K] S H _ Ct s C M.
case: {M} (menvs_coh M) (M) C=>_ [_][xp1][xa1][->{s}] Cp1 Ca M C.
case/(menvs_split (injLE _ (erefl _)) Cp Cp1): M=>/= M _.
have {M P} P : pv_self xp1 = p :-> v \+ hS by rewrite -(menvs_loc M).
rewrite -joinCA in C *.
apply: step; apply: val_extend=>//.
apply: (gh_ex hS); apply: val_do; first by exists B, v.
case=>{xp xp1 xa C Cp Cp1 Ca P} xp P Cp s /= C M.
case: {M} (menvs_coh M) (M) C=>_ [_][xt1][xa][->{s}] Ct1 Ca M C.
case/(menvs_split (injLA _ (erefl _)) Ct Ct1): M=>/= {xa1} M _.
have {S} S : tb_self xt1 = Unit by rewrite -(menvs_loc M).
have {xt Ct M H} H : [h <<= tb_other xt1]
  by apply: hist_trans H (hist_other M).
rewrite joinA in C *.
apply: step; apply: val_extend; first by apply/(star_coh_prec C).
apply: (gh_ex h); apply: (gh_ex hS); apply: val_do.
- by move=>C'; rewrite (getC Cp C') !(getC Ct1 C').
move=>b s X Y; case: Y (Y) X=>_ [xp1][xt][->{s}] {Cp} Cp1 Ct Y X.
move=>xa1 /= {C} C M; case: (menvs_coh M)=>_ {M xa Ca} Ca1.
rewrite (getC Cp1 Y) !(getC Ct Y) in X.
case: b X; last first.
- case=>{P S H} P S H.
  apply: (gh_ex h); apply: (gh_ex hS); apply: val_do=>//.
  move=>{C} C; exists (prod A ptr), (e, p').
  by rewrite (getC Cp1 C) !(getC Ct C).
case=>{xp xt1 Ct1 P K S H} t' [ls][P K S H].
apply: val_ret=>//= m M; rewrite -(menvs_loc M).
rewrite (getC Cp1 C) (getC Ct C).
exists t', ls; split=>//.
case: {M} (menvs_coh M) (M)=>_ /= X'.
case: X' (X')=>_ [s][xa][->{m}] Y'; case: Y' (Y')=>_ [xp][xt1][->].
move=>Cp Ct1 Y' {Ca1} Ca {C} C.
case/(menvs_split (injLE _ (erefl _)) Y Y').
case/(menvs_split (injLE _ (erefl _)) Cp1 Cp)=> _ M _.
by rewrite (getC Ct1 C); apply: hist_trans H (hist_other M).
apply: gh=>i [h hS] X C.
case: C (C) X =>_ [_][xa][->][_][xp][xt][->] Cp Ct Ca C /= [P Ps H].
rewrite !(getC Ct C) !(getC Cp C) /= in P Ps H.
rewrite joinAC /V /= starAC in C *.
apply: step; apply: val_extend; first by apply/(star_coh_prec C).
apply: (gh_ex (pv_self xp)); apply: val_do.
- by move=>Cpa; rewrite (getC Cp Cpa).
move=>x m [B][v] Y X {Cp Ca C}.
case: X (X) Y=>_ [xp1][xa1][->{m}] /= Cp Ca X S xt1 C M.
rewrite {X} (getC Cp X) in S.
case: (menvs_coh M)=>_ /= {Ct} Ct.
have {P} P : tb_self xt1 = Unit by rewrite -(menvs_loc M).
have {H M} H : [h <<= tb_other xt1] by apply: hist_trans H (hist_o M).
apply: (gh_ex h); apply: (gh_ex (pv_self xp)).
apply: val_do=>[_|]; first by exists B, v; rewrite!(getC Ct C).
case=>m [t][ls][P2 S2 H2 K2]; exists t, ls; rewrite P2; split=>//.
Qed.
```

Proof of `push` specification

```
Next Obligation.
apply: gh=>i [h hS][B][v] X C.
case: C (C) X=>_ [_][xa][->{i}][_][xp][xt][->] Cp Ct Ca C [P S H].
rewrite seq Cp C) !(getC Ct C) /= in P S H.
rewrite A joinCA in C *.
apply: apply: val_extend=>//;apply: (gh_ex h); apply:
fun_call S H C Ct Ca} xt [t][K] S H _ Ct s C M.
case: {M} (menvs_coh M) (M) C=>_ [_][xp1][xa1][->{s}] Cp1 Ca M C.
case/(menvs_split (injLE _ (erefl _)) Cp Cp1): M=>/= M _.
have {M P} P : pv_self xp1 = p :-> v \+ hS by rewrite -(menvs_loc M).
rewrit nCA in C *.
apply: seq apply: va
apply: hS); app fun_call st by exists B, v.
case=>{xp xp1 xa C Cp C Cp s /= C M.
case: {M} (menvs_coh M) (M) C=>_ [_][xt1][xa][->{s}] Ct1 Ca M C.
case/(menvs_split (injLA _ (erefl _)) Ct Ct1): M=>/= {xa1} M _.
have {S} S : tb_self xt1 = Unit by rewrite -(menvs_loc M).
have {xt Ct M H} H : [h <<= tb_other xt1]
  by apply: hist_trans H (hist_other M).
rewrit A in C *.
apply: seq apply: val_extend; first by oh_prec C).
apply: x h); apply: (gh_ex hS); ap fun_call
- by move=>C'; rewrite (getC Cp C') !(g
move=>b s X Y; case: Y (Y) X=>_ [xp1][xt][->{s}] {Cp} Cp1 Ct Y X.
move=>xa1 /= {C} C M; case: (menvs_coh M)=>_ {M xa Ca} Ca1.
rewrite (getC Cp1 Y) !(getC Ct Y) in X.
case: b X; last first.
- case=>{P S H} P S H.
  apply: (gh_ex h); apply: (gh_ex hS); app fun_call .
  move=>{C} C; exists (prod A ptr), (e, p'
  by rewrite (getC Cp1 C) !(getC Ct C).
case= 1 P K S H} t' [ls][P K S H].
apply return //= m M; rewrite -(menvs_loc M).
rewri 1 C) (getC Ct C).
exists t', ls; split=>//.
case: {M} (menvs_coh M) (M)=>_ /= X'.
case: X' (X')=>_ [s][xa][->{m}] Y'; case: Y' (Y')=>_ [xp][xt1][->].
move=>Cp Ct1 Y' {Ca1} Ca {C} C.
case/(menvs_split (injLE _ (erefl _)) Y Y').
case/(menvs_split (injLE _ (erefl _)) Cp1 Cp)=> _ M _.
by rewrite (getC Ct1 C); apply: hist_trans H (hist_other M).
```

```
apply: gh=>i [h hS] X C.
case: C (C) X =>_ [_][xa][->][_][xp][xt][->] Cp Ct Ca C /= [P Ps H].
rewrite !(getC Ct C) !(getC Cp C) /= in P Ps H.
rewrit seq AC /V /= starAC in C *.
apply: apply: val_extend; f fun_call (star_coh_prec C).
apply: (pv_self xp)); appl
- by move=>Cpa; rewrite (getC Cp
move=>x m [B][v] Y X {Cp Ca C}.
case: X (X) Y=>_ [xp1][xa1][->{m}] /= Cp Ca X S xt1 C M.
rewrite {X} (getC Cp X) in S.
case: (menvs_coh M)=>_ /= {Ct} Ct.
have {P} P : tb_self xt1 = Unit by rewrite -(menvs_loc M).
have {H M} H : [h <<= tb_other xt1] by apply: hist_trans H (hist_o M).
ap fun_call apply: (gh_ex (pv_self xp)).
ap ||]; first by exists B, v; rewrite!(getC Ct C).
case=>m [t][ls][P2 S2 H2 K2]; exists t, ls; rewrite P2; split=>//.
Qed.
```

Proof of `push` specification

```
Next Obligation.
apply: gh=>i [h hS][B][v] X C.
case: C (C) X=>_ [_][xa][->{i}][_][xp][xt][->] Cp Ct Ca C [P S H].
rewrite      Cp C) !(getC Ct C) /= in P S H.
rewrite  seq  A joinCA in C *.
apply:       apply: val_extend=>//;apply: (gh_ex h); apply:
 fun_call   S H C Ct Ca} xt [t][K] S H _ Ct s C M.
case: {M} (menvs_coh M) (M) C=>_ [_][xp1][xa1][->{s}] Cp1 Ca M C.
case/(menvs_split (injLE _ (erefl _)) Cp Cp1): M=>/= M _.
have {M P} P : pv_self xp1 = p :-> v \+ hS by rewrite -(menvs_loc M).
rewrit     nCA in C *.
apply: seq  apply: val
apply:     hS); appl  fun_call  st by exists B, v.
case=>{xp xp1 xa C Cp C              Cp s /= C M.
case: {M} (menvs_coh M) (M) C=>_ [_][xt1][xa][->{s}] Ct1 Ca M C.
case/(menvs_spl  proving stability  {xa1} M _.
have {S} S : tb                     oc M).
have {xt Ct M H
  by apply: his
rewrit     A in C *.
apply: seq  apply: val_extend; first by  fun_call  oh_prec C).
apply:     x h); apply: (gh_ex hS); app
- by move=>C'; rewrite (getC Cp C') !(ge
move=>b s X Y; case: Y (Y) X=>_ [xp1][xt][->{s}] {Cp} Cp1 Ct Y X.
 proving stability                                  X.
                                   ; app  fun_call  .
  move=>{C} C; exists (prod A ptr), (e, p'
  by rewrite (getC Cp1 C) !(getC Ct C).
case=  return  1 P K S H} t' [ls][P K S H].
apply            //= m M; rewrite -(menvs_loc M).
rewri            1 C) (getC Ct C).
exists t', ls; split=>//.
case
case  proving stability  Y' (Y')=>_ [xp][xt1][->].
move
case                      ).
case/                         Cp1 Cp)=> _ M _.
by rewrite (getC Ct1 C); apply: hist_trans H (hist_other M).
```

```
apply: gh=>i [h hS] X C.
case: C (C) X =>_ [_][xa][->][_][xp][xt][->] Cp Ct Ca C /= [P Ps H].
rewrite !(getC Ct C) !(getC Cp C) /= in P Ps H.
rewrite  seq  AC /V /= starAC in C *.
apply:  seq  apply: val_extend; f  fun_call  (star_coh_prec C).
apply:       (pv_self xp)); appl
- by move=>Cpa; rewrite (getC Cp
 proving stability  Cp Ca X S xt1 C M.
have {H M} H : [h <<= tb_other xt1] by apply: hist_trans H (hist_o M).
app  fun_call    apply:  proving stability
app              ||]; first
case=>m [t][ls][P2 S2 H2 K                //.
Qed.
```

Why do you need the explicit *other*? Other logics don't have it!

- *Other* makes it possible to state ***open-world assumptions*** in a straightforward way (e.g., in `push`);
- It allows us to use ***hiding*** for uniformly cancelling the interference;
- Some algorithms are given more natural specs via *other*-contributions (e.g., stack's `pop` and atomic snapshots).

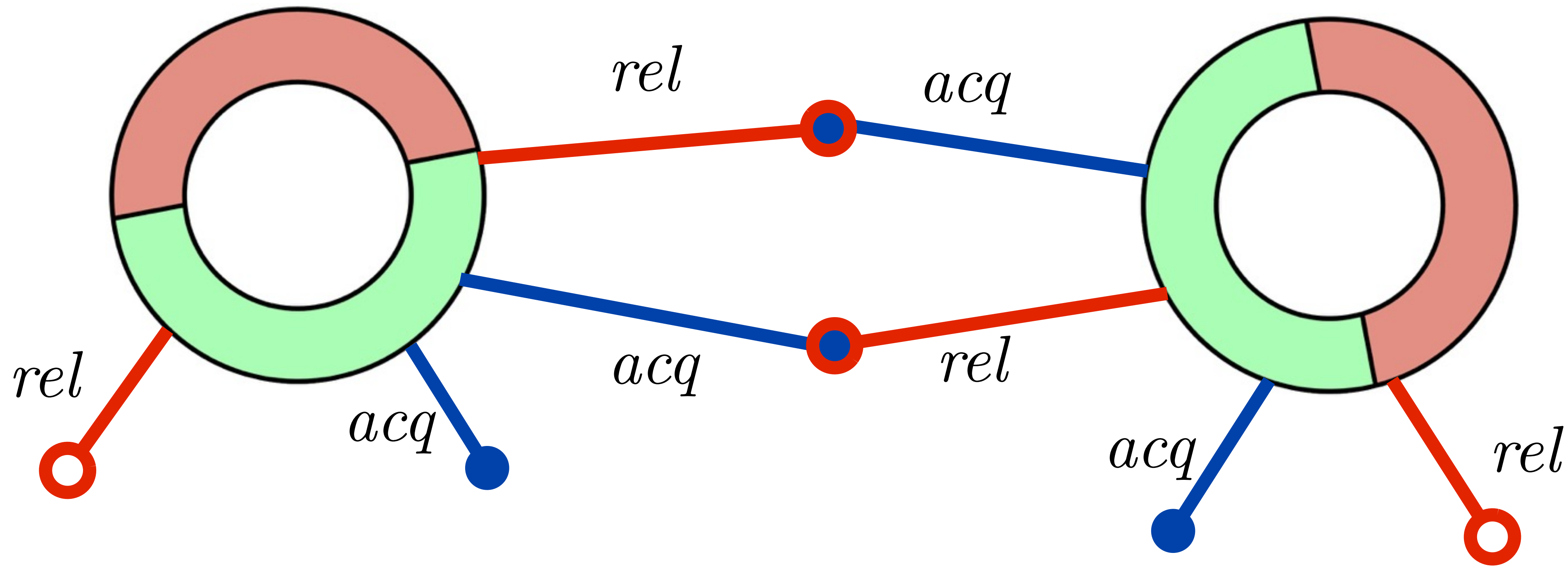
Composing concurrent resources

Composing concurrent resources

Connect ownership-transferring transitions with right polarity

Composing concurrent resources

Connect ownership-transferring transitions with right polarity



- Some channels might be left loose
- Some channels might be shut down
- *Same* channels might be connected several times

Can you extract the verified program from your Coq implementation and run it?

Can you extract the verified program from your Coq implementation and run it?

Not yet.

Can you extract the verified program from your Coq implementation and run it?

Not yet.

- Imperative programs are composed and verified (i.e., type-checked) by means of Coq;
- They cannot be run by means of Gallina's operational semantics;
- The reason for that is the necessity to reason about concurrent computations and potentially diverging programs;
- Extraction will require proving operationally of arbitrary atomic actions.