Programming and Proving with Concurrent Resources

Ilya Sergey

UCL
PPLV

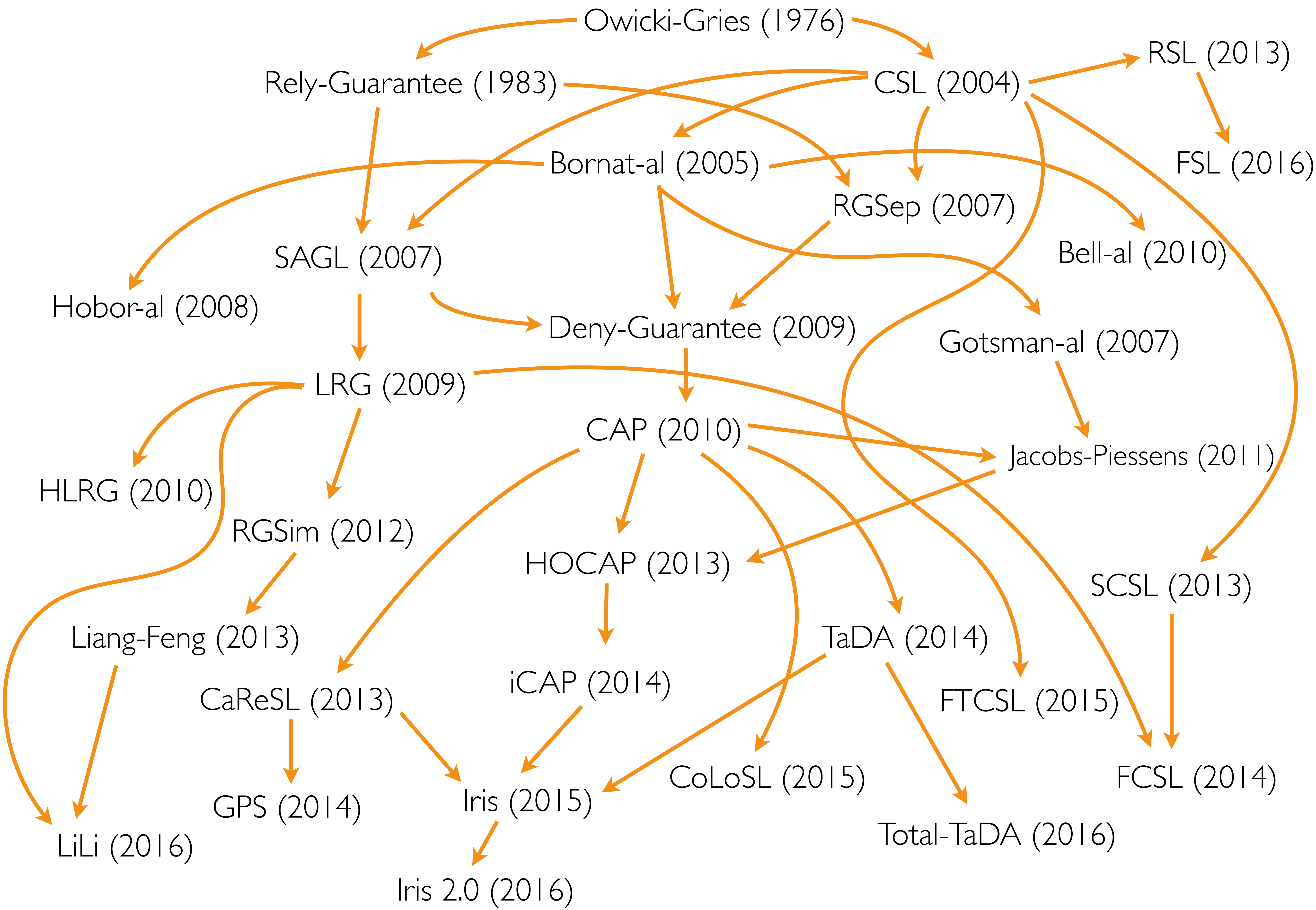
joint work with
Aleks Nanevski, Anindya Banerjee, Ruy Ley-Wild and Germán Delbianco

What and why

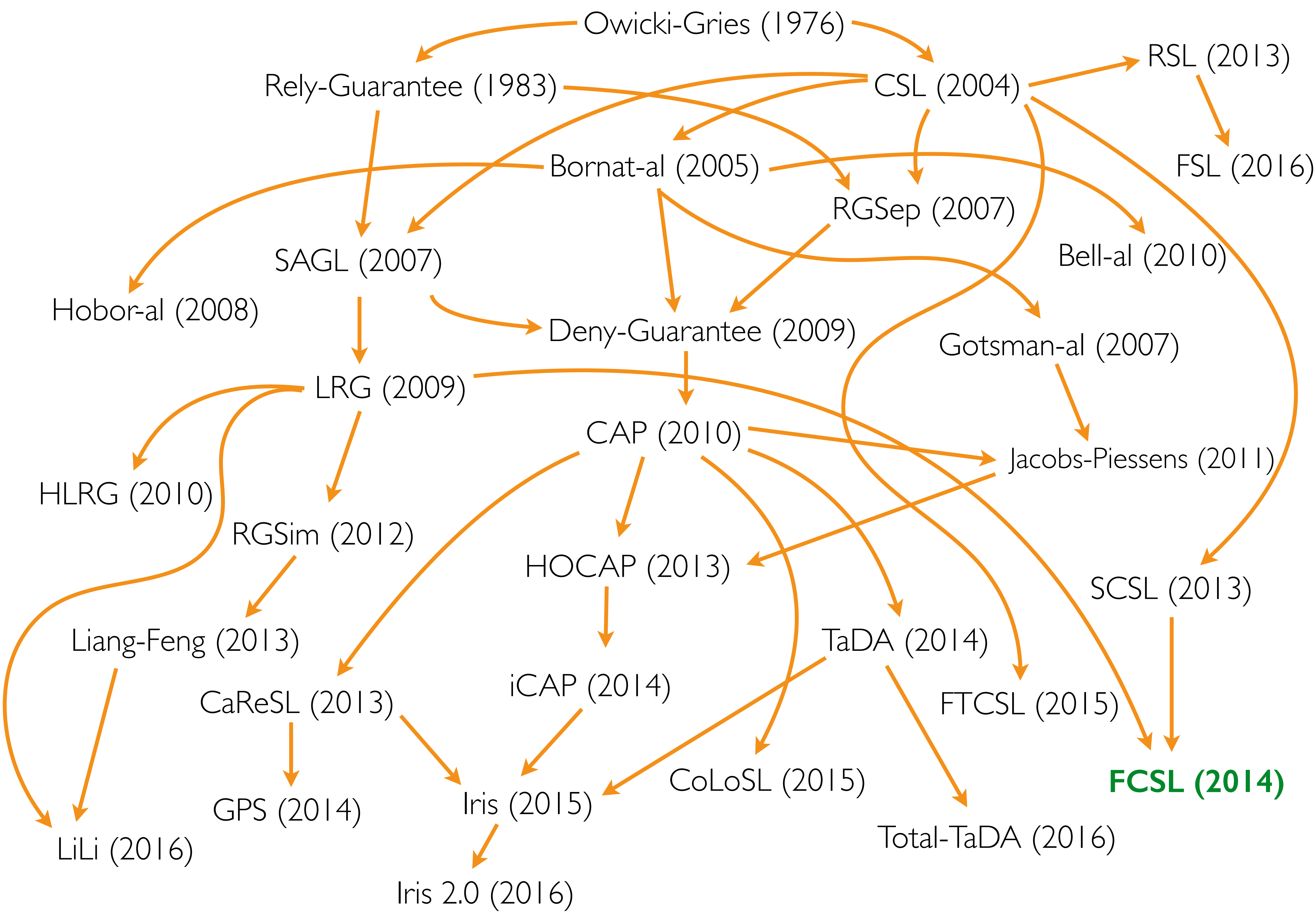
- Concurrency $\Rightarrow$ parallelism $\Rightarrow$ efficiency
- A gap between *informal* and *formal* reasoning
- Scalable formalisation requires *compositionality*

This talk

A logical framework
for *implementation*
and compositional *verification*
of *concurrent* programs.



Owicki-Gries (1976)
Rely-Guarantee (1983)
CSL (2004)
RSL (2013)
Bornat-al (2005)
FSL (2016)
RGSep (2007)
SAGL (2007)
Bell-al (2010)
Hobor-al (2008)
Deny-Guarantee (2009)
Gotsman-al (2007)
LRG (2009)
CAP (2010)
Jacobs-Piessens (2011)
HLRG (2010)
RGSim (2012)
HOCAP (2013)
SCSL (2013)
Liang-Feng (2013)
TaDA (2014)
iCAP (2014)
CaReSL (2013)
FTCSL (2015)
CoLoSL (2015)
FCSL (2014)
GPS (2014)
Iris (2015)
LiLi (2016)
Total-TaDA (2016)
Iris 2.0 (2016)



Owicki-Gries (1976)
Rely-Guarantee (1983)
CSL (2004)
RSL (2013)
Bornat-al (2005)
FSL (2016)
RGSep (2007)
SAGL (2007)
Bell-al (2010)
Hobor-al (2008)
Deny-Guarantee (2009)
Gotsman-al (2007)
LRG (2009)
CAP (2010)
Jacobs-Piessens (2011)
HLRG (2010)
RGSim (2012)
HOCAP (2013)
SCSL (2013)
Liang-Feng (2013)
TaDA (2014)
iCAP (2014)
CaReSL (2013)
FTCSL (2015)
CoLoSL (2015)
FCSL (2014)
GPS (2014)
Iris (2015)
Total-TaDA (2016)
LiLi (2016)
Iris 2.0 (2016)

Key insights

- Subjectivity
- Time-stamped Histories
- Reasoning about Deep Sharing

Key insights

- Subjectivity
- Time-stamped Histories
- Reasoning about Deep Sharing

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

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

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

Suitable for sequential case

Abstract specifications for a stack

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

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

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

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

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

$\{ y = ??? \}$

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

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

$\{ y \in \mathrm{Some}\ \{1, 2\} \vee y = \mathit{None} \}$

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

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

$\{ y \in \mathrm{Some}\ \{1, 2, 3\} \vee y = \mathit{None} \}$

Thread-modular spec for pop?

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

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

$\{\ y = ???\ \}$

Idea

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

(subjective specification)

Subjective stack specifications

- H_s — *history* of my **pushes/pops** to the stack
- H_o — *history* of **pushes/pops** by *all other threads*

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

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

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

Subjective stack specifications

- H_s — *history* of my **pushes/pops** to the stack
- H_o — *history* of **pushes/pops** by *all other threads*

$\{ H_s = \varnothing \}$

```
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

Subjective stack specifications

Valid only if the history is changed
by registering actual push/pops.

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

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

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

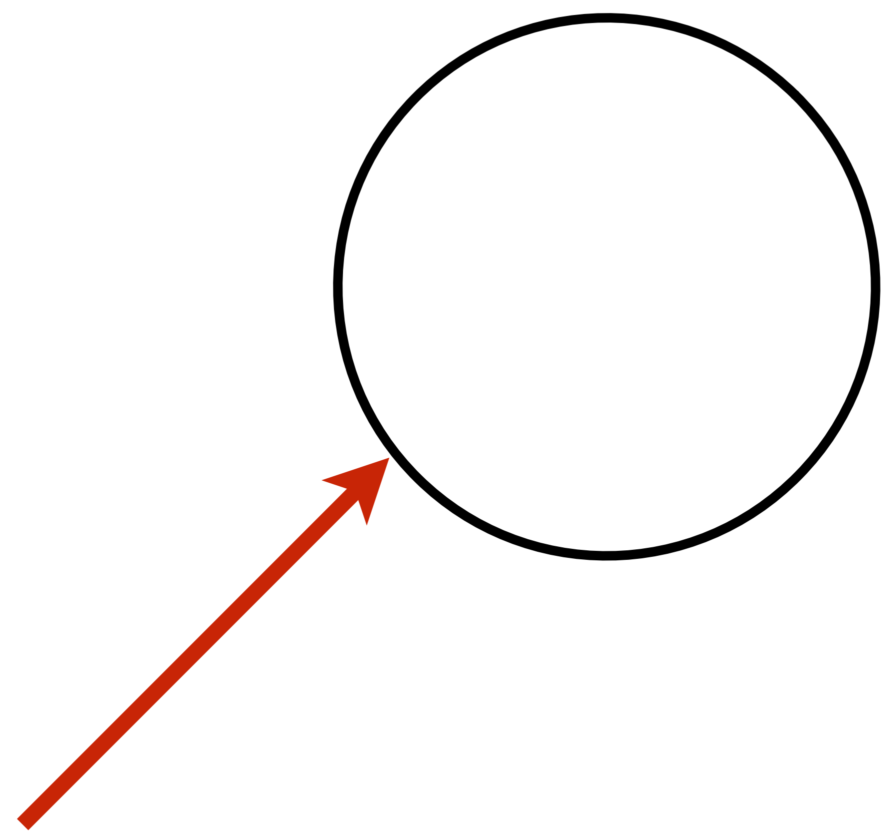
what I popped depends
on what the *others* have pushed

Specifies expected
thread interference

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

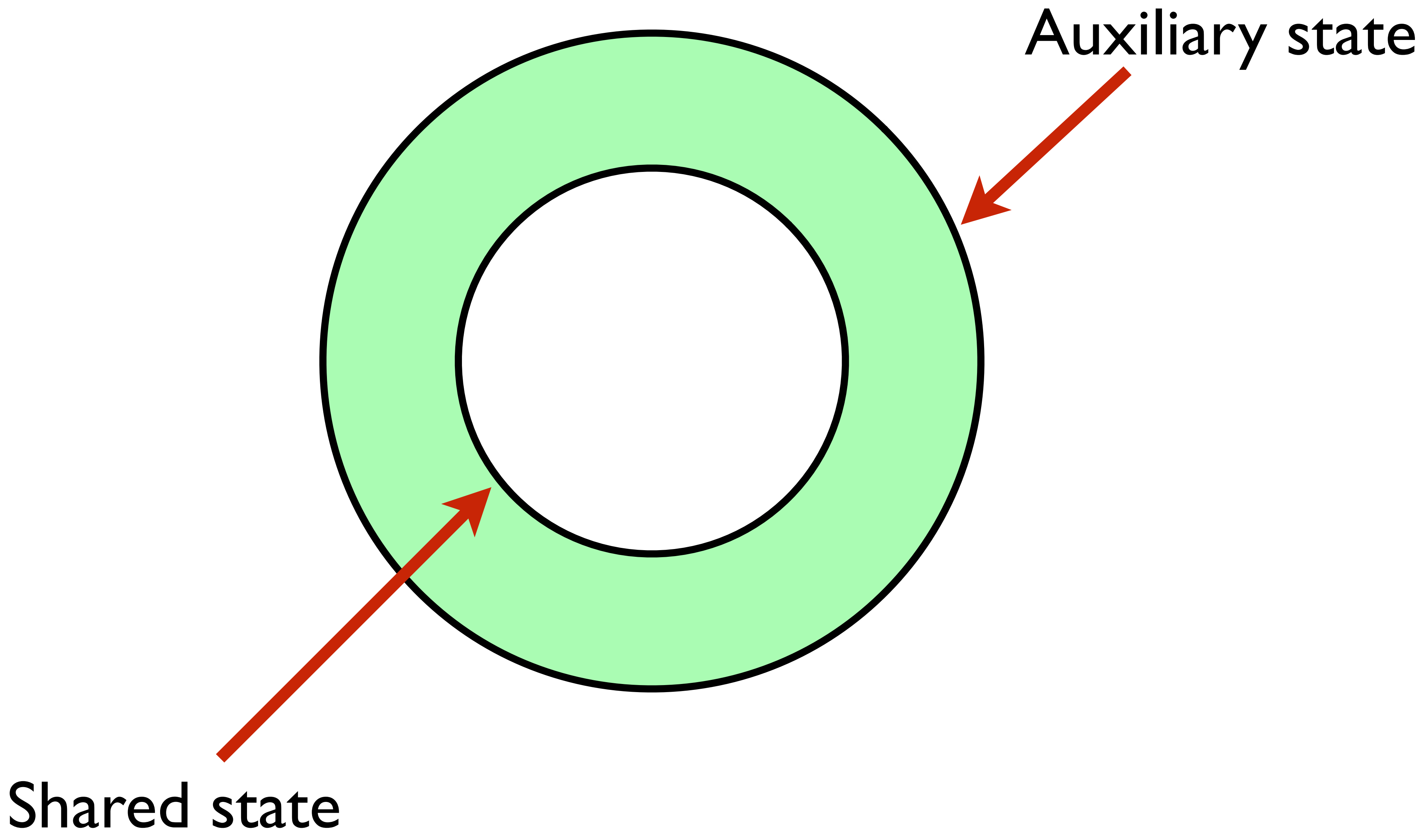
Model of shared state with manifested interference

Concurrent Resources



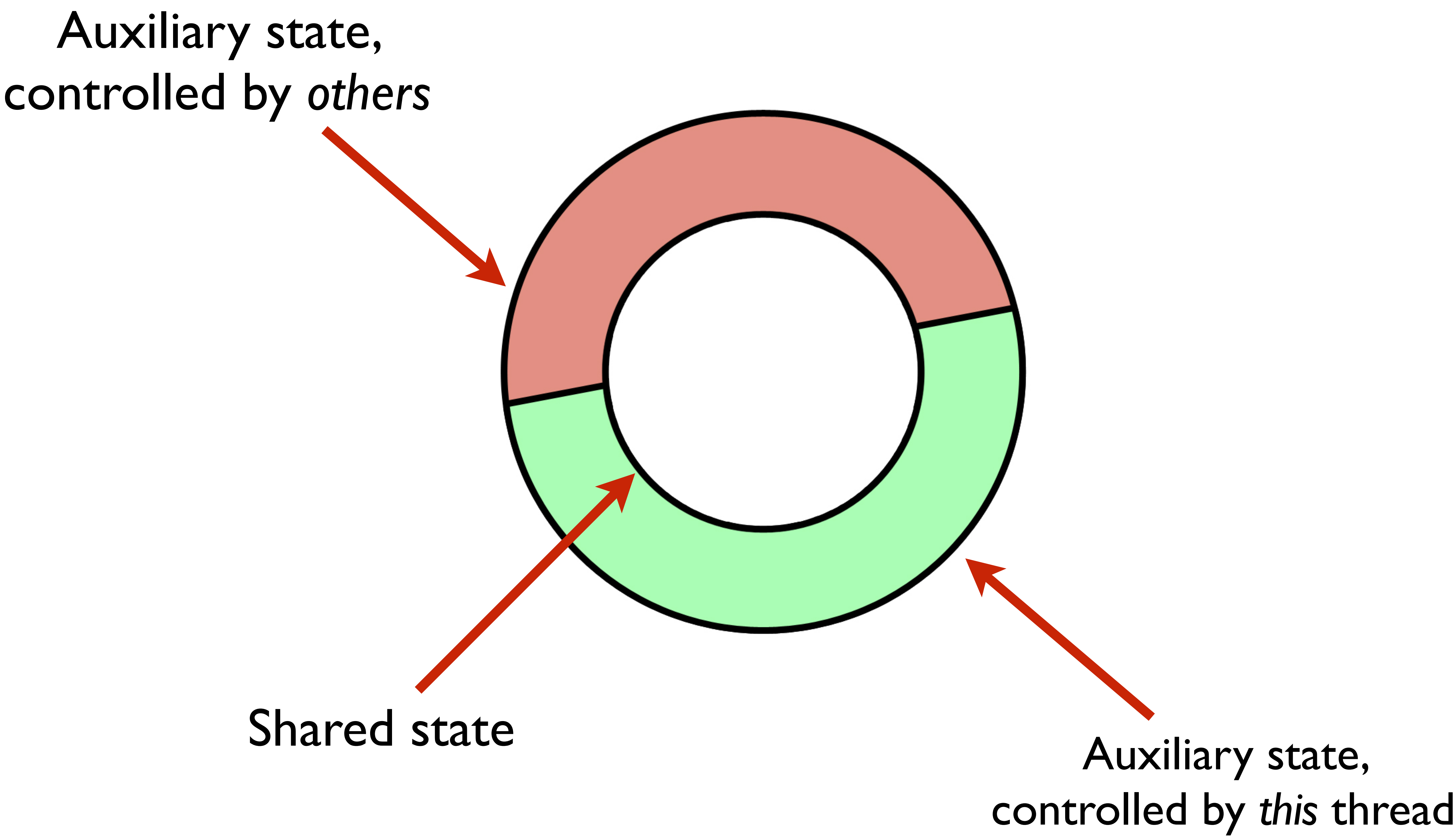
Concurrent Resources

Owicki, Gries [CACM'77]



Subjective Concurrent Resources

Ley-Wild, Nanevski [POPL'13]

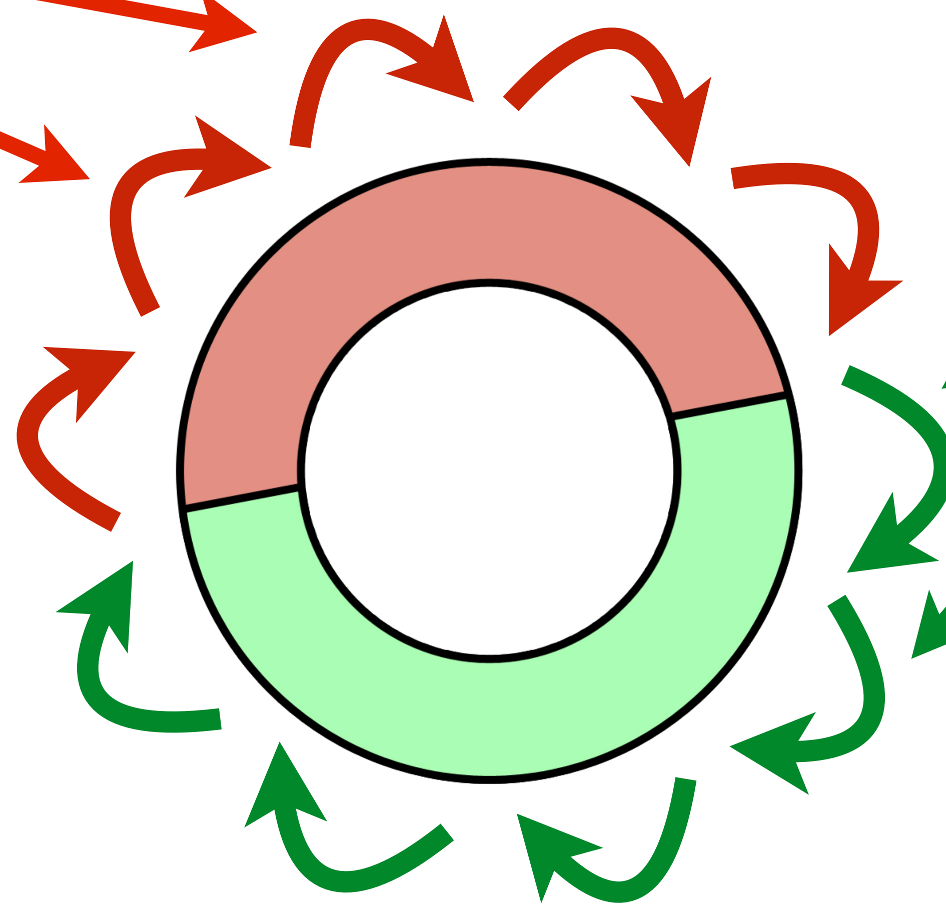


Subjective Concurrent Resources

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**.

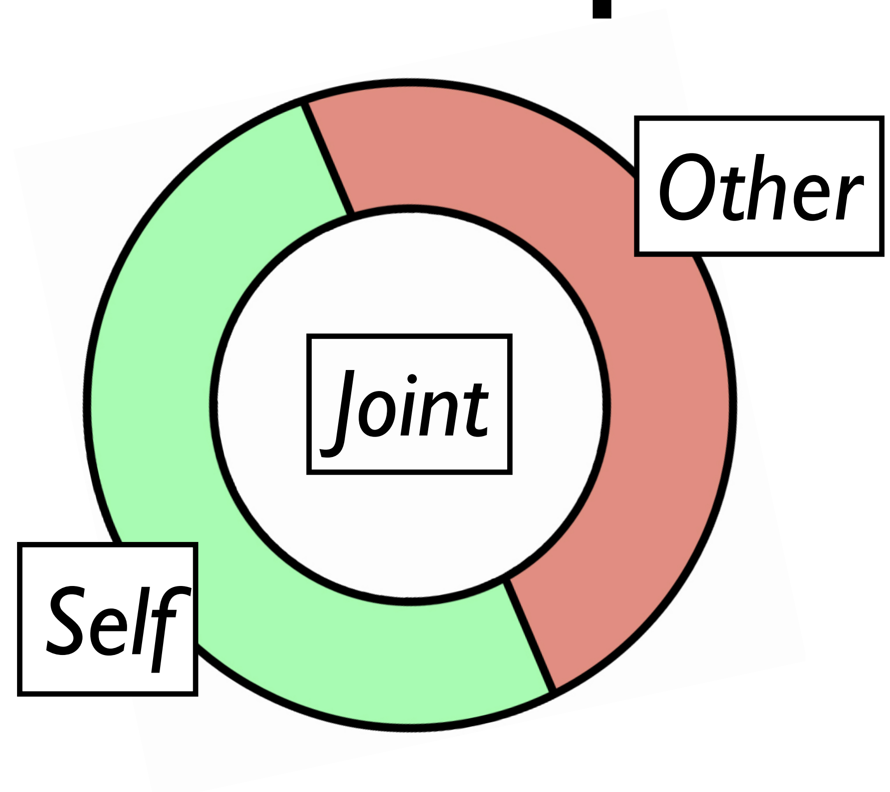
Concurrent Resources

=

State Transition Systems
with
Subjective Auxiliary State

Nanevski et al [ESOP'14]

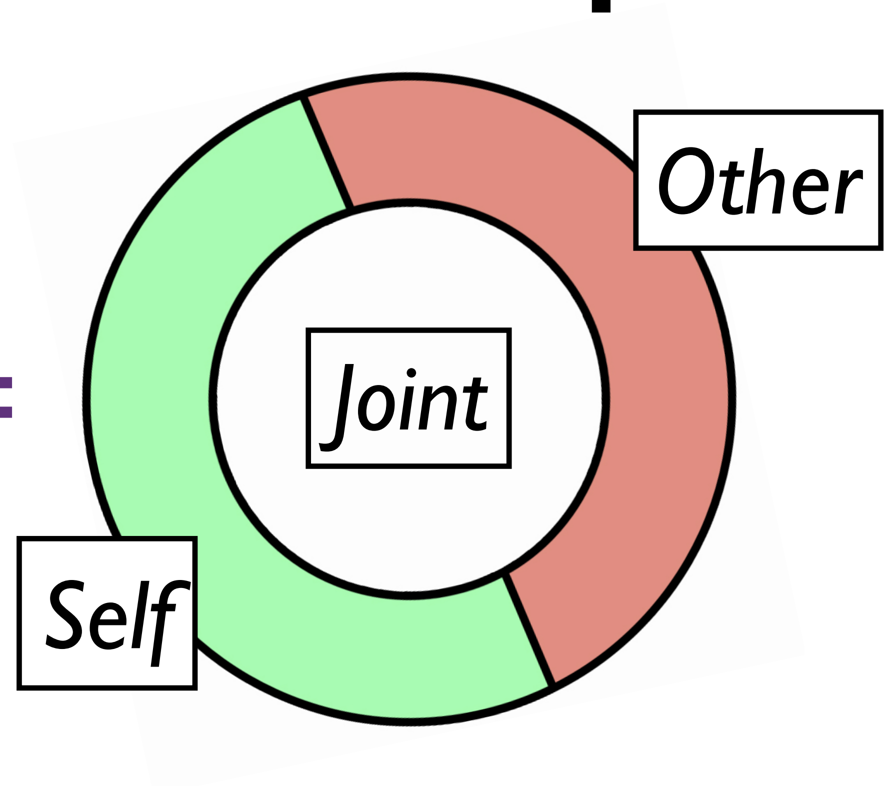
Resource-based specifications



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

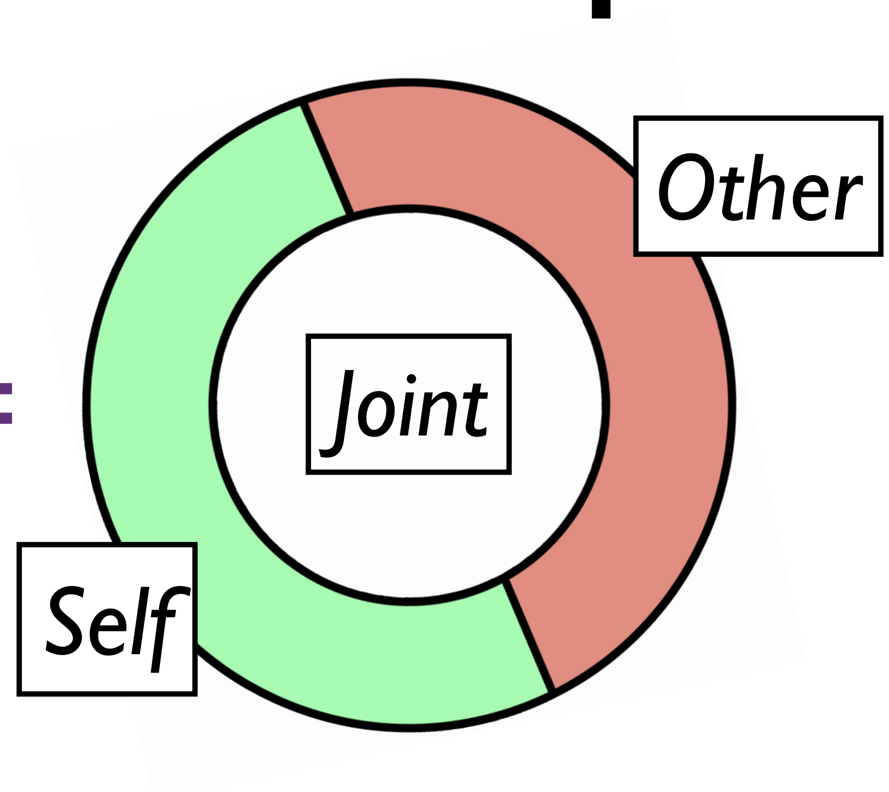
Resource-based specifications

C =



$$C \vdash \{\, P \,\}\ \mathbf{c}\ \{\, Q \,\} \underbrace{@\ C}_{\substack{\text{defines resources, touched by } \mathbf{c}\text{,} \\ \text{their transitions and invariants}}}$$

Resource-based specifications

$C =$ 

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

specify *self/other/joint* parts

Fine-grained Concurrent Separation Logic

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

- Logic for reasoning with (fine-grained) concurrent resources
- Emphasis on *subjective* specifications

Key insights

- Subjectivity
- Histories
- Deep Sharing

Key insights

- Subjectivity — reasoning with *self* and *other*
- Histories
- Deep Sharing

Key insights

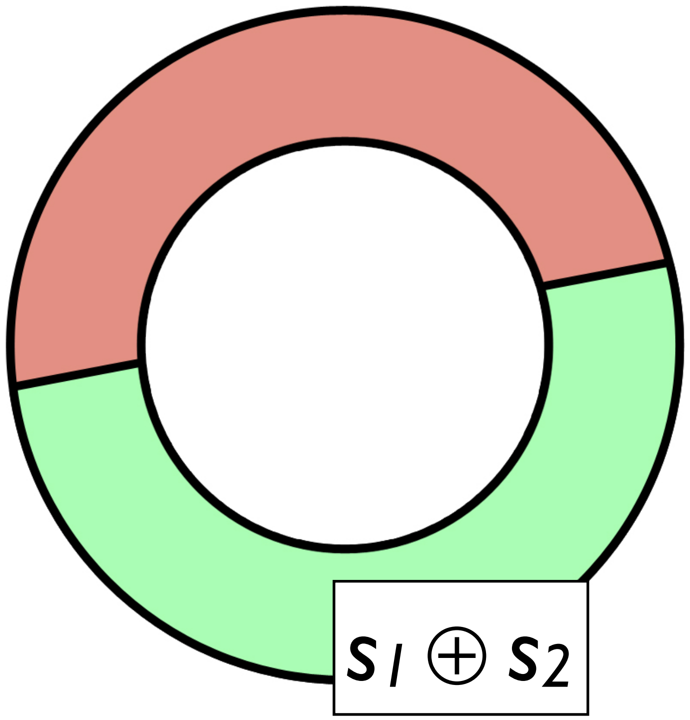
- Subjectivity — reasoning with *self* and *other*
- Histories
- Deep Sharing

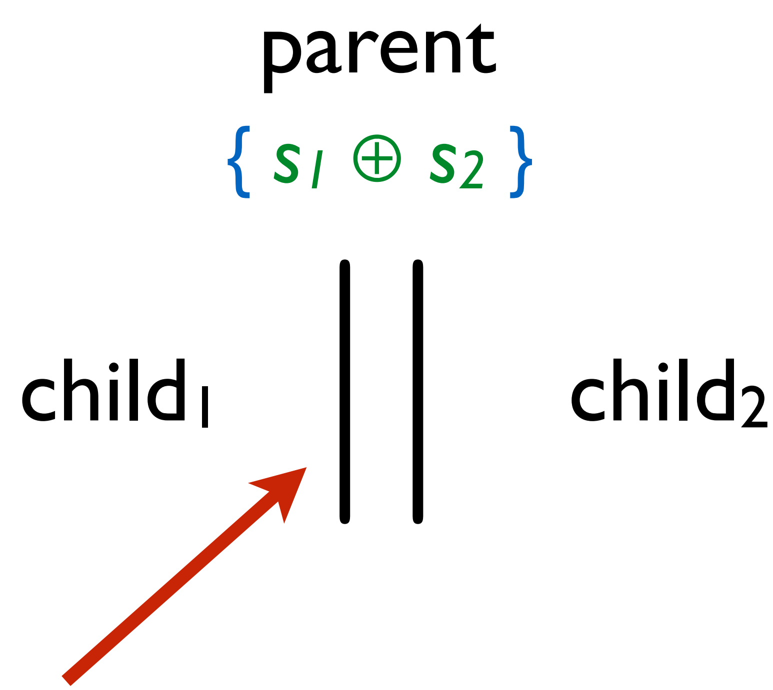
Partial Commutative Monoids

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

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

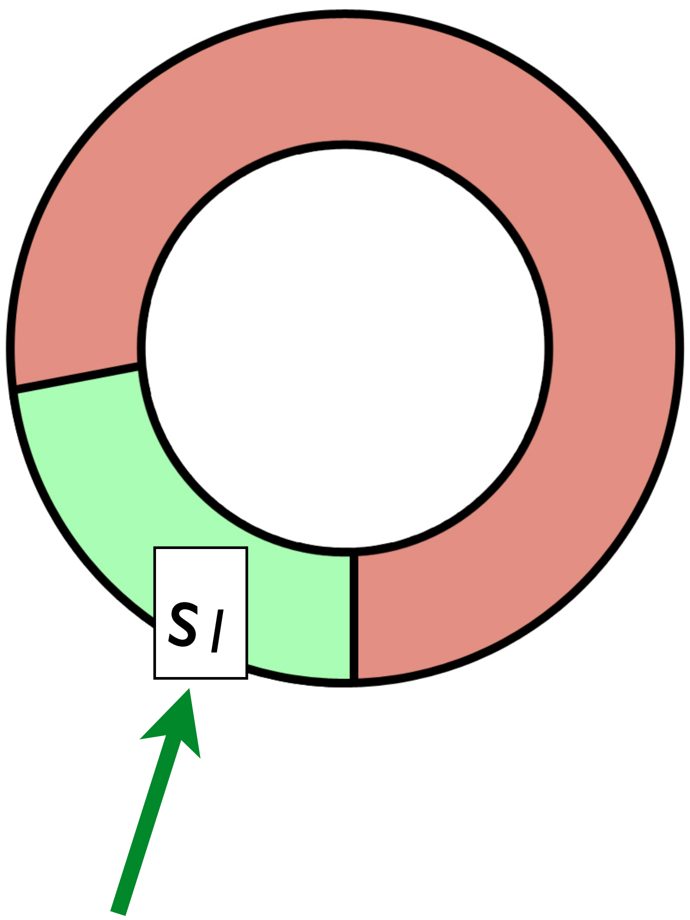
Logical state split





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

Logical state split



State that belongs to child_1

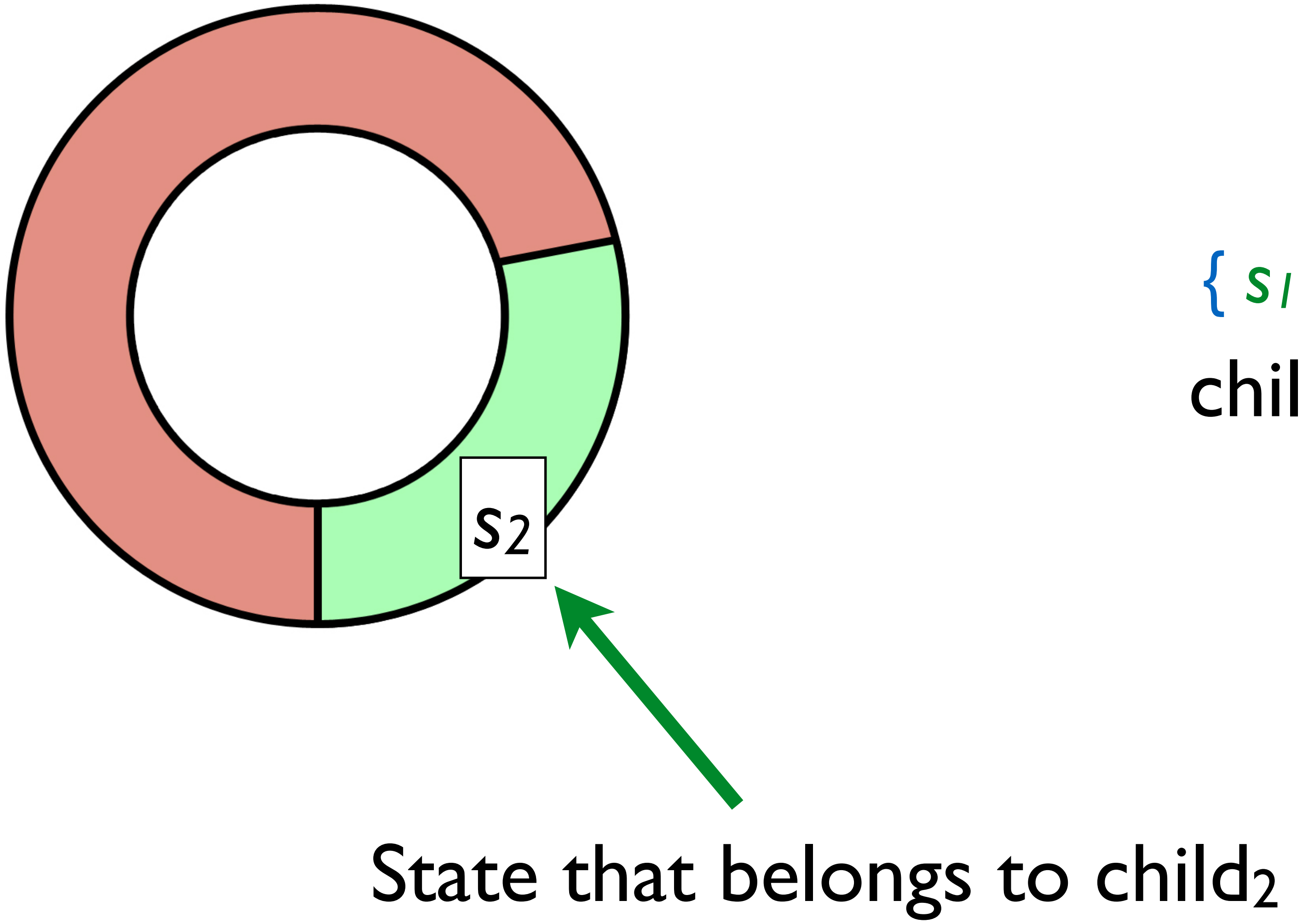
parent

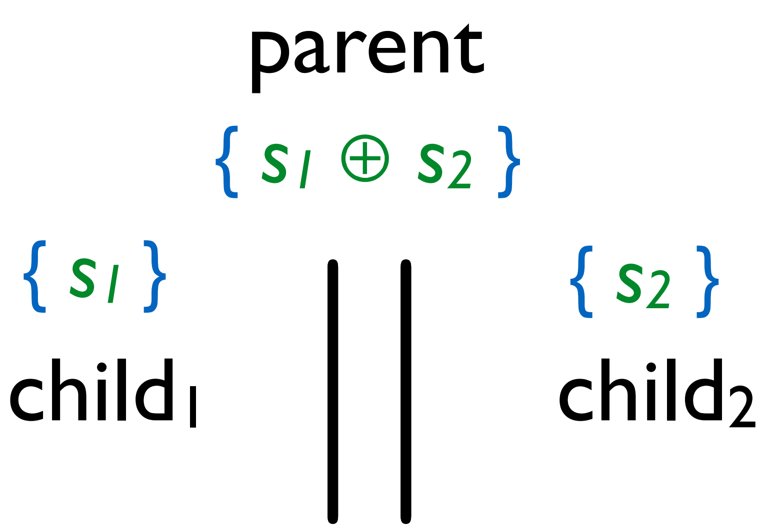
$\{ s_1 \oplus s_2 \}$

$\{ s_1 \}$

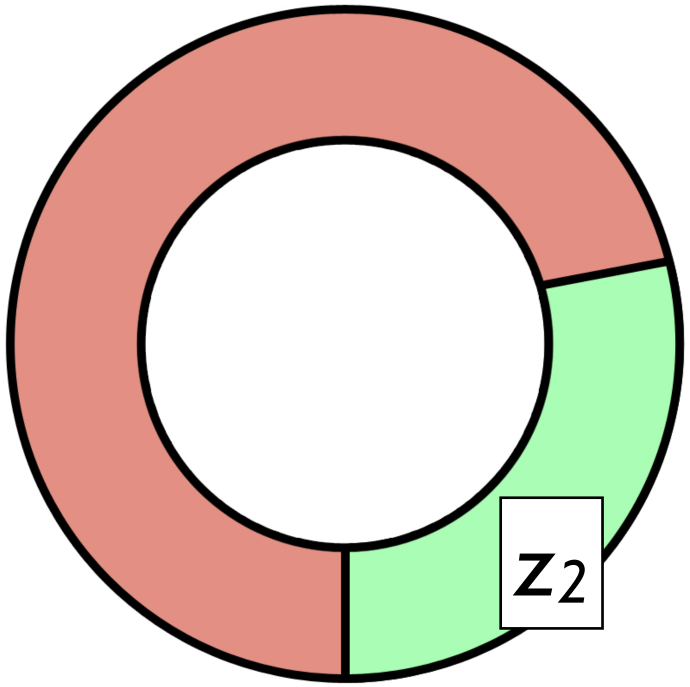
child_1 || child_2

Logical state split





Logical state split



parent

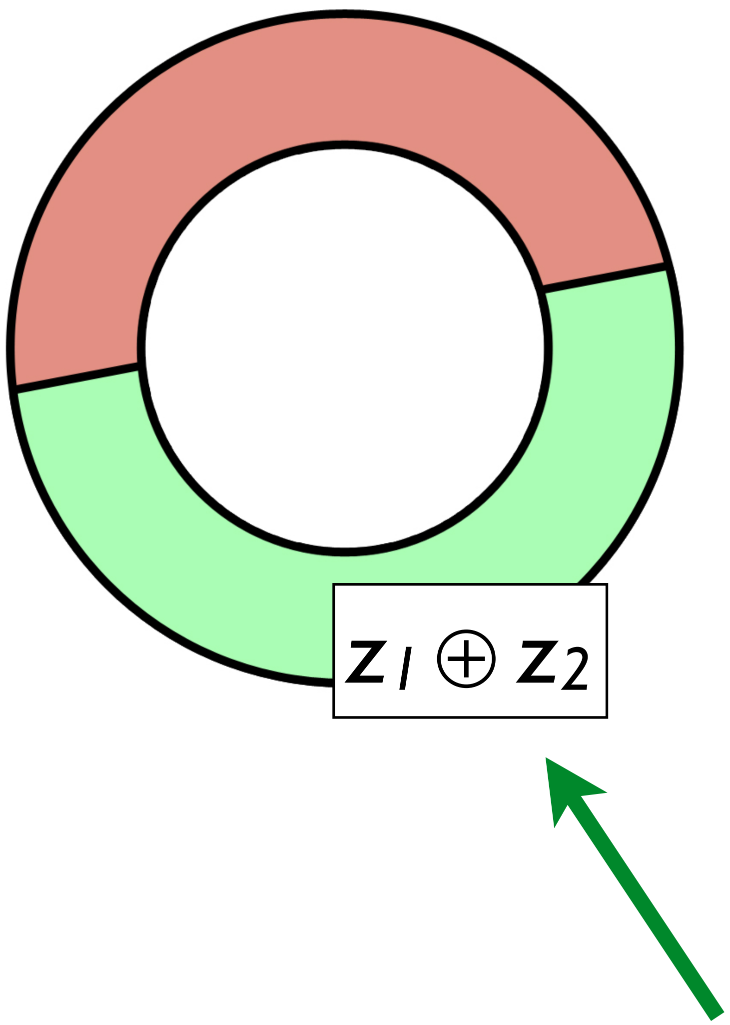
$\{ s_1 \oplus s_2 \}$

$\{ s_1 \}$ child$_1$ $\{ z_1 \}$

||

$\{ s_2 \}$ child$_2$ $\{ z_2 \}$

Logical state split



New state that belongs to parent′

parent

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

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

$\{\, z_1 \oplus z_2 \,\}$

parent′

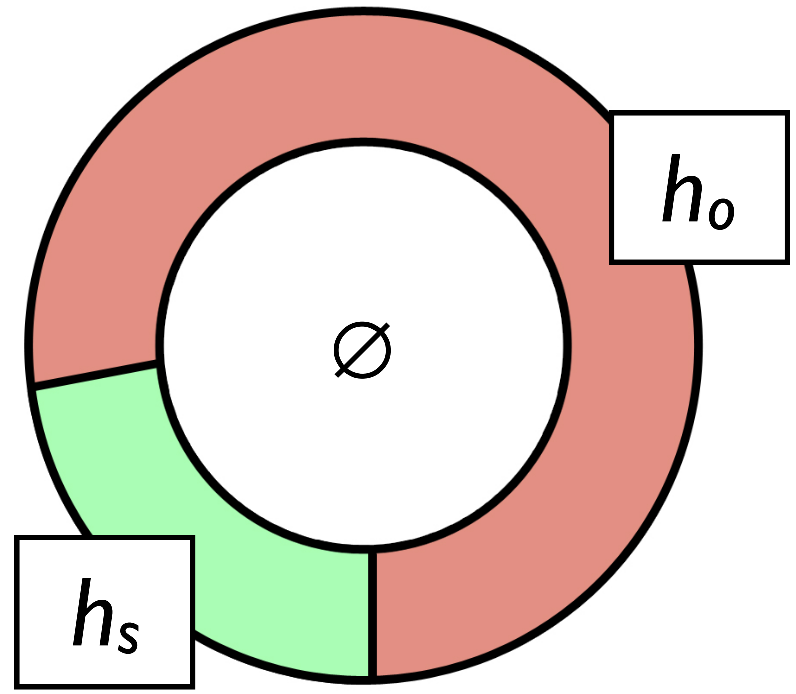
PCMs: a uniform interface for *splittable* state

Familiar PCM: finite heaps

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

Resource 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
- Transitions — writes into h_s

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;
```

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

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

Less familiar PCM: histories

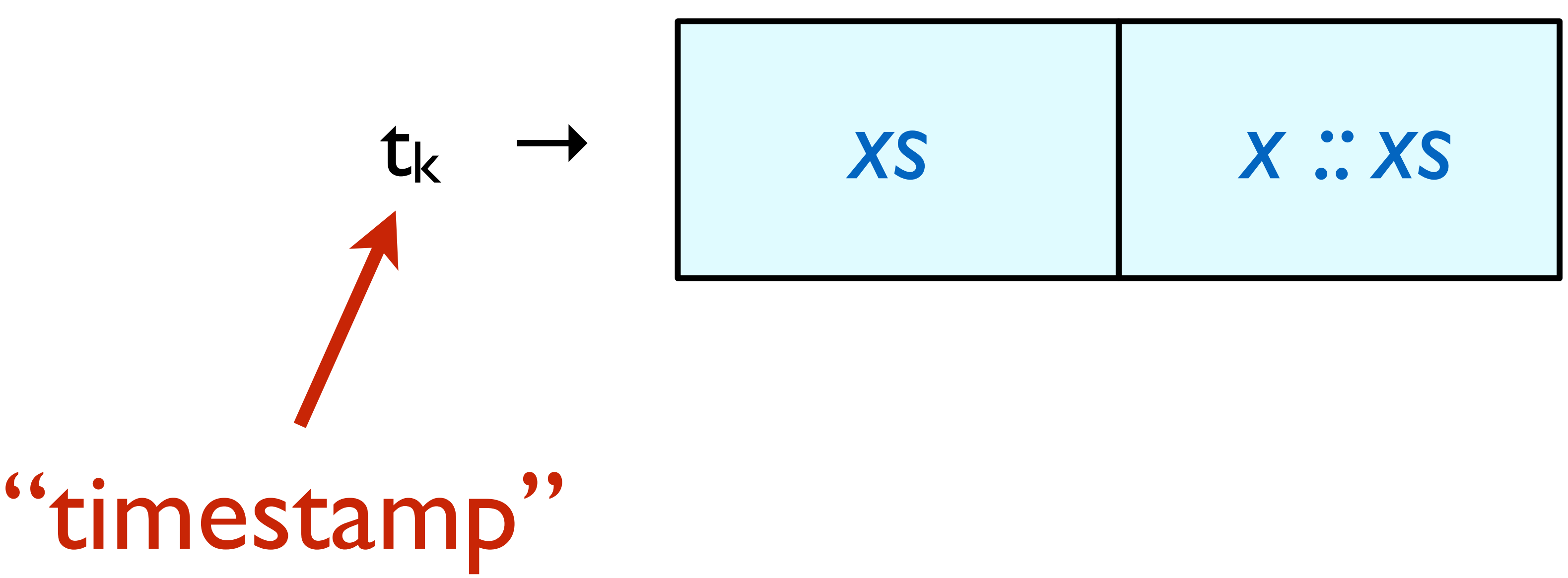
Sergey, Nanevski, Banerjee [ESOP'15]

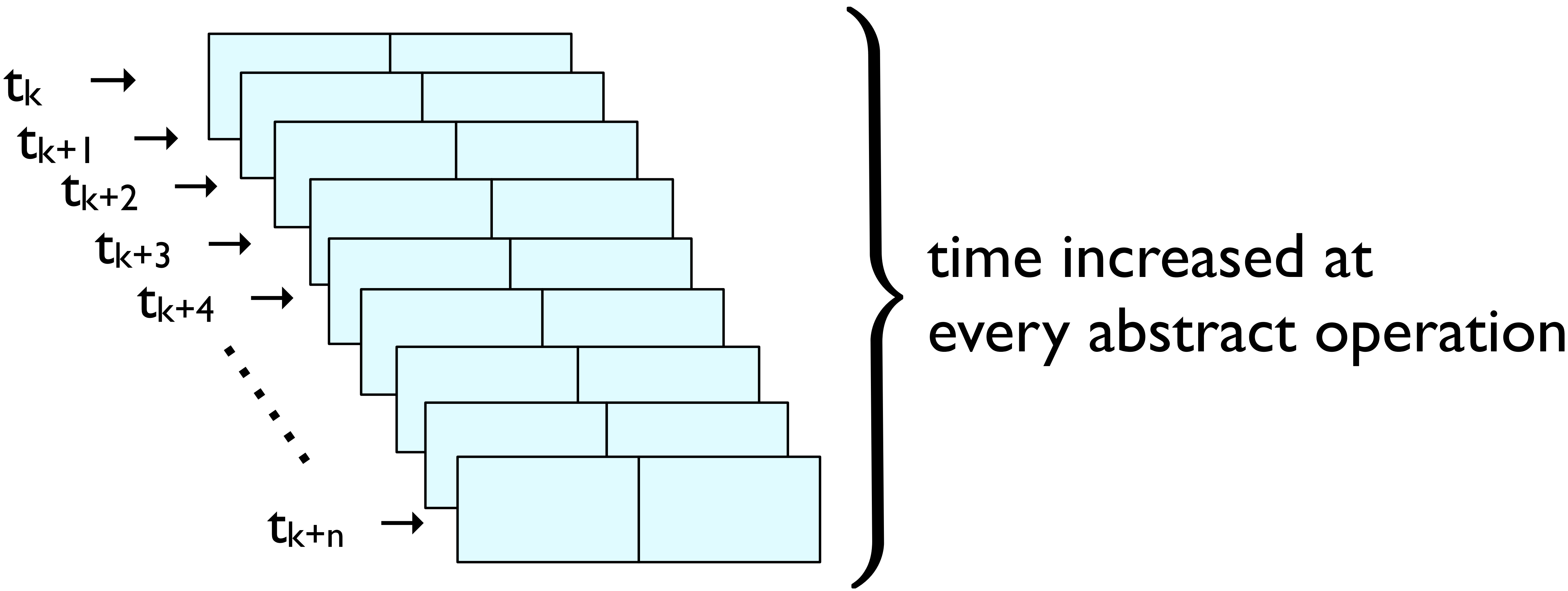
Describing *atomic* state updates via auxiliary state

Atomic stack specifications

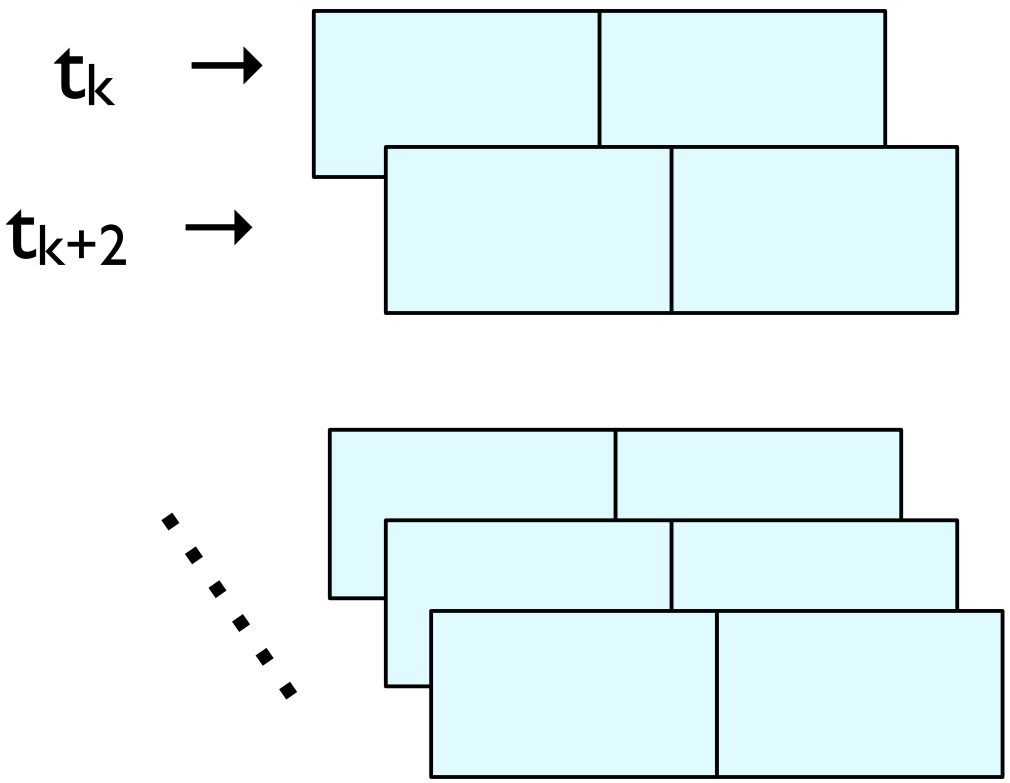
$$\{\ S = xs\ \}\quad \texttt{push(x)}\quad \{\ S' = x :: xs\ \}$$

Atomic stack specifications

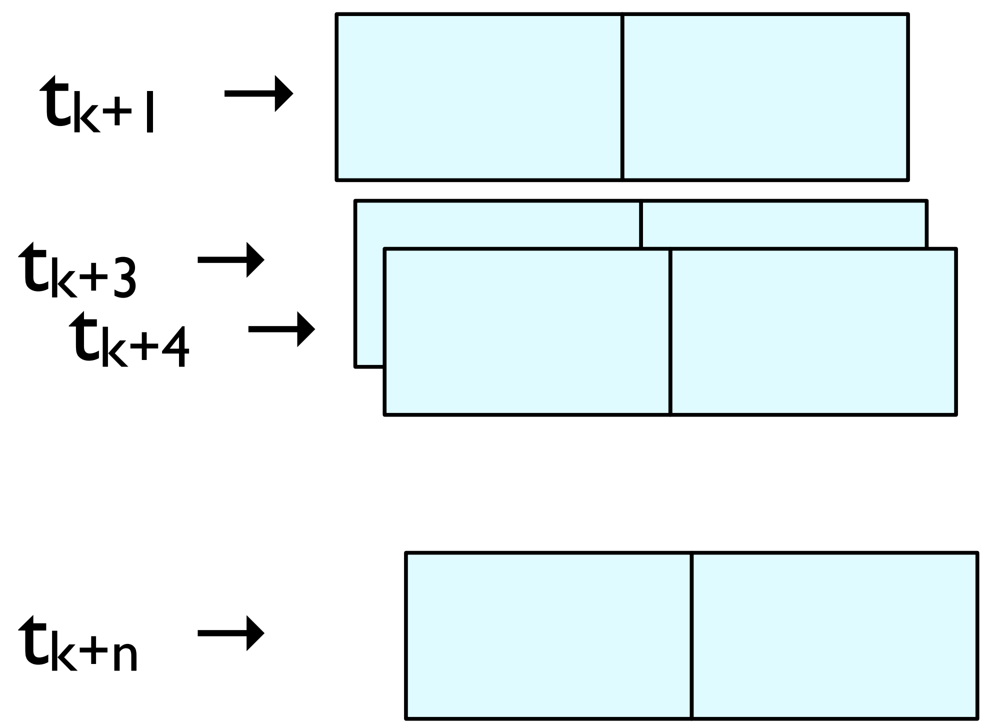


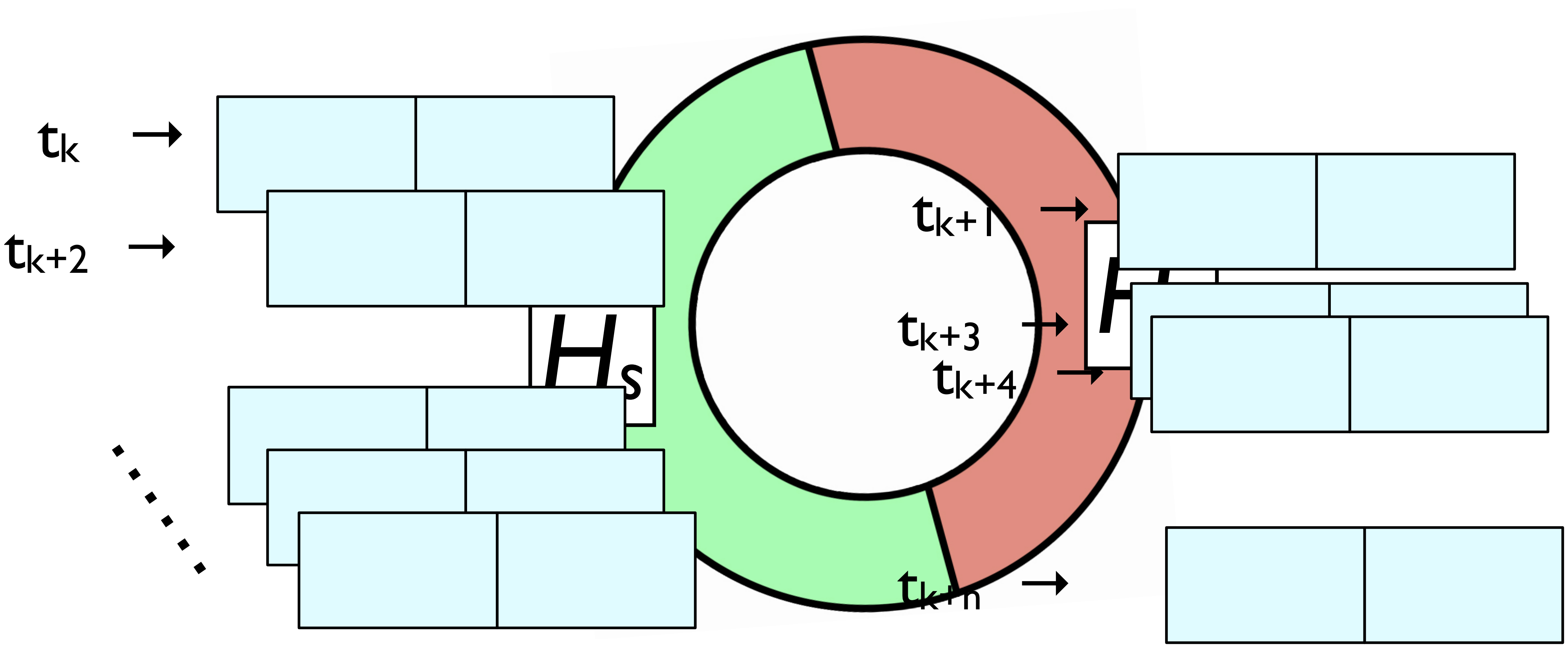

t_k
t_{k+1}
t_{k+2}
t_{k+3}
t_{k+4}
t_{k+n}
time increased at
every abstract operation

Changes by *this* thread



Changes by *other* threads





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

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 $\emptyset$

Stack specification

self-contribution is a single entry

t allocated during the call

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

`push(x)`

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

Stack specification

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

`pop( )`

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

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

Stack specification

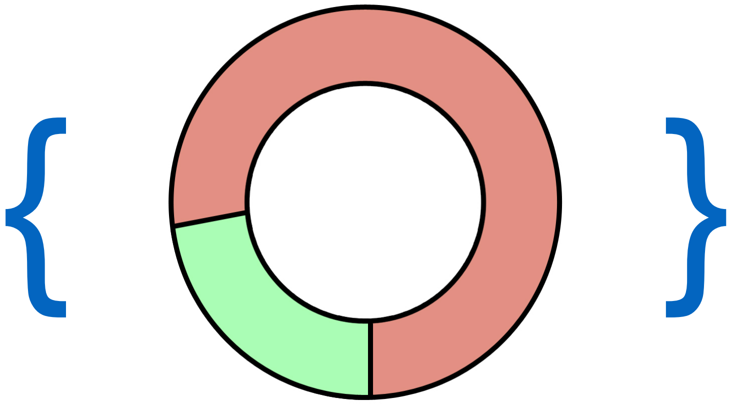
no self-contributions initially?

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

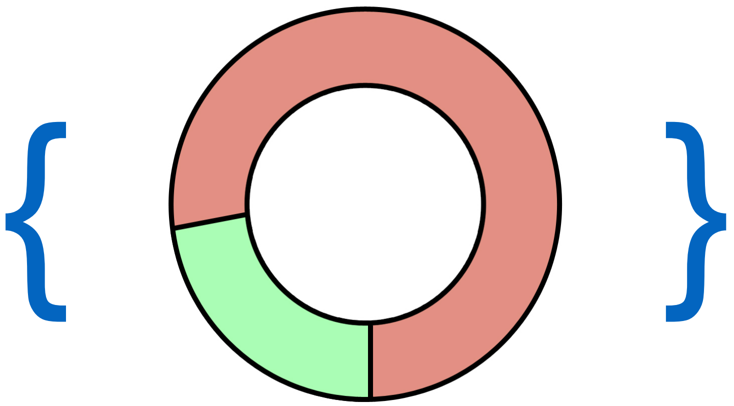
`pop( )`

$$\{ \text{res.}\ \ \textbf{if}\ \ (\mathit{res} = \mathit{Some}\ \mathrm{x})$$
$$\textbf{then}\ \ \exists \mathrm{t}, \mathrm{xs}.\ \mathrm{H} \subseteq H_o \land \mathrm{H} < \mathrm{t} \land H_s = \mathrm{t} \mapsto (\mathrm{x::xs}, \mathrm{xs}))$$
$$\textbf{else}\ \ \exists \mathrm{t}.\ \mathrm{H} \subseteq H_o \land \mathrm{H} \leq \mathrm{t}$$
$$\land H_s = \emptyset \land \mathrm{t} \mapsto (\_, \textbf{Nil}) \subseteq H_o \}@C_{\mathrm{stack}}$$

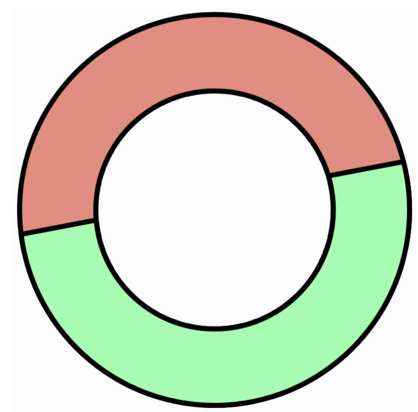
Framing in FCSL



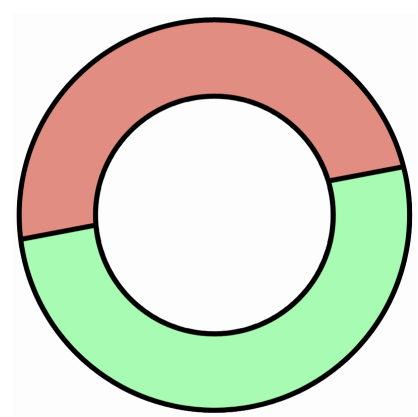
`my_program`



Framing in FCSL

{  }

`my_program`

{  }

Works for *any* PCM, not just heaps (e.g., SL and CSL)!

Framing histories

$$\{\ \mathcal{H}_s = \varnothing \wedge \mathrm{H} \subseteq \mathcal{H}_0\ \}$$

```
push(x)
```

$$\{\ \exists t, xs.\ \mathrm{H} \subseteq \mathcal{H}_0 \wedge \mathrm{H} < t \wedge \mathcal{H}_s = t \mapsto (xs, x{::}xs)\ \}@C_{stack}$$

Framing histories

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

initial self-contribution

`push(x)`

final self-contribution

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

t is later than $H_1 \oplus H_2$

How clients use splittable histories?

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
- **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

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

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

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

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

```
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);
  }
}
```

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

No other threads can interfere on `S`

`hide` C_{stack}`(h`$_S$`)` **`in`**

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

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

`hide` C_{stack}`(`$\mathsf{h_S}$`)` **`in`**

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

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

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

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

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

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

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

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

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

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

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

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

Key insights

- Subjectivity — reasoning with *self* and *other*
- Histories
- Deep Sharing

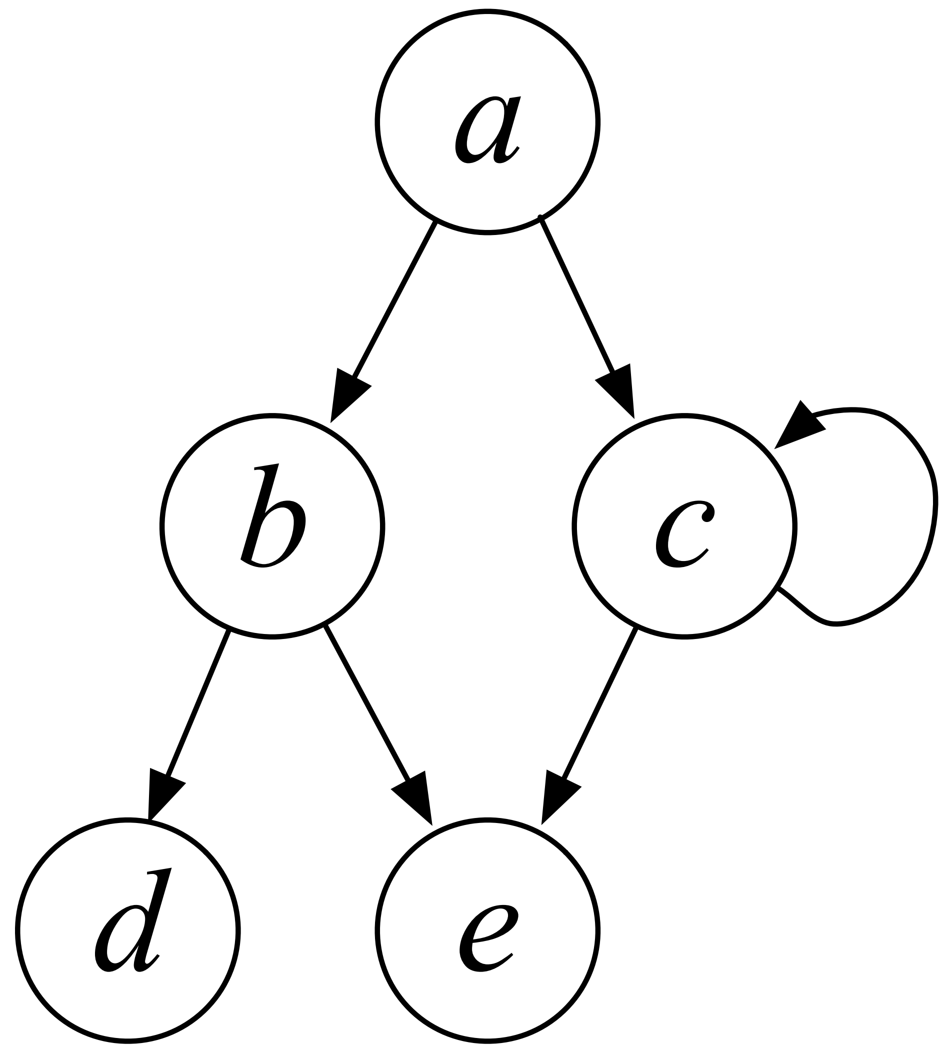
Key insights

- Subjectivity — reasoning with *self* and *other*
- Histories — temporal specification via *state*
- Deep Sharing

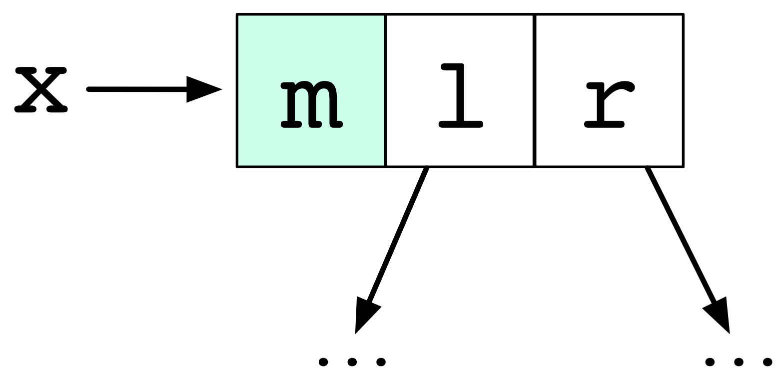
Key insights

- Subjectivity — reasoning with *self* and *other*
- Histories — temporal specification via *state*
- Deep Sharing

Ramified data structures



In-place concurrent spanning tree construction

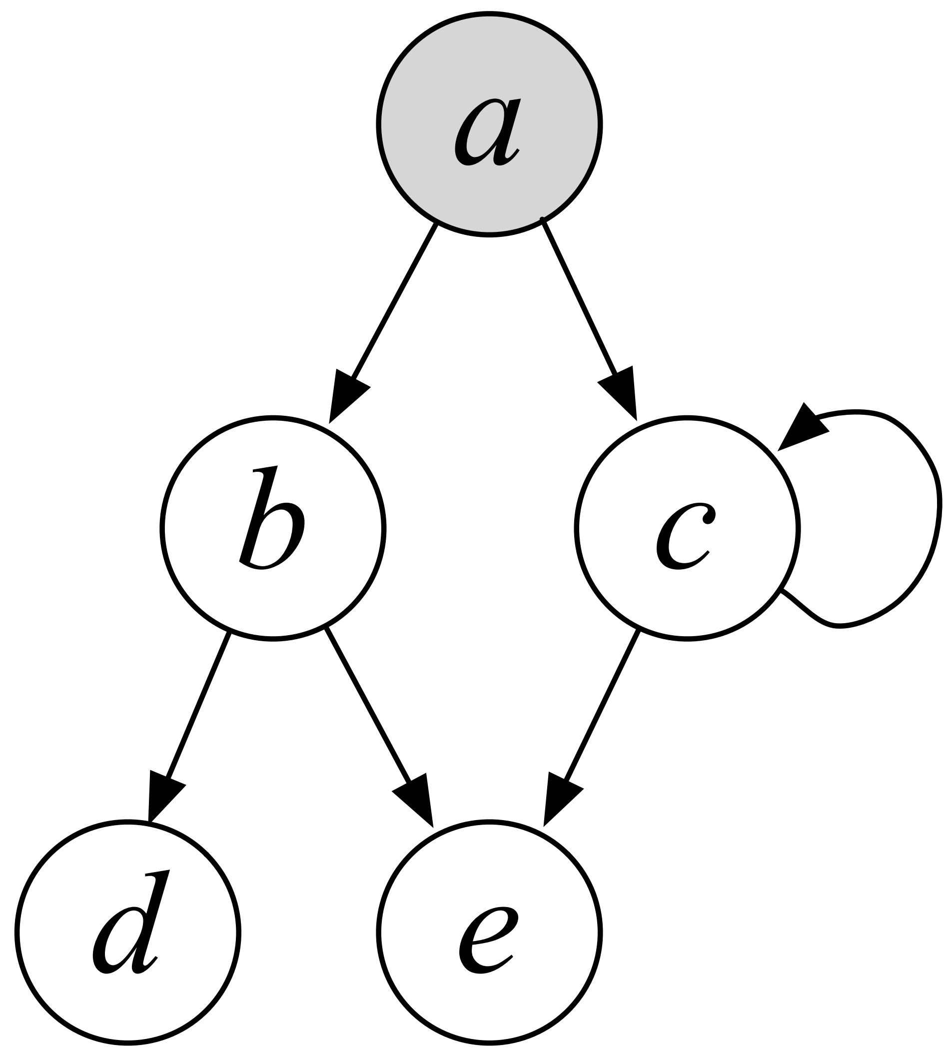


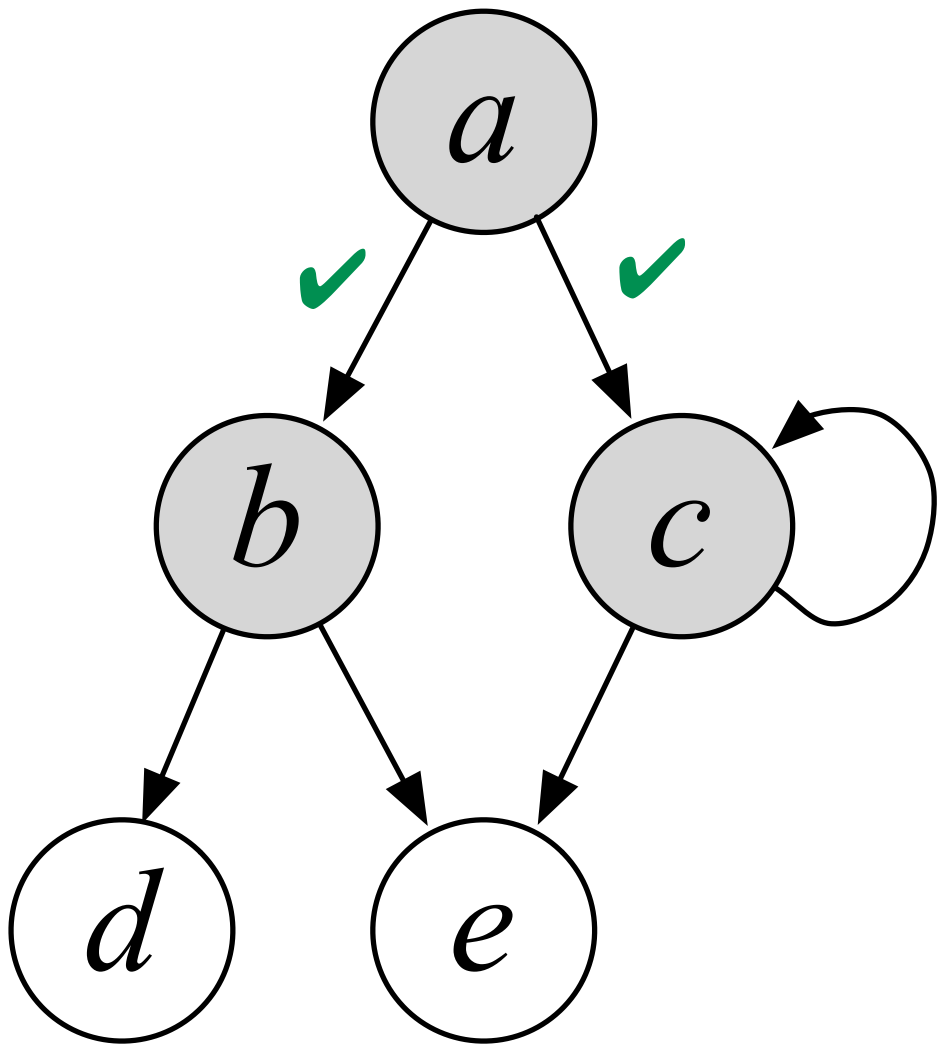
mark the node x

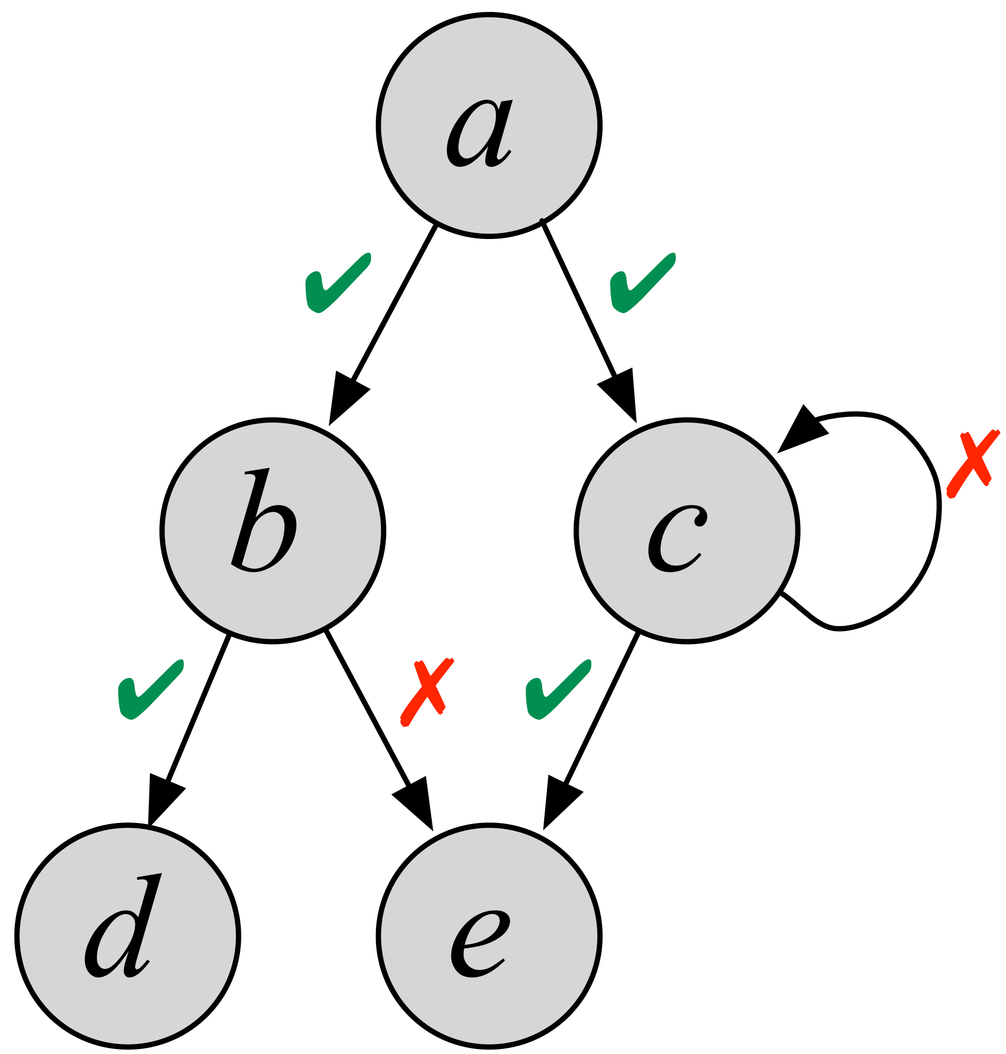
```
letrec span (x : ptr) : bool = {
  if x == null then return false;
  else
    b ← CAS(x->m, 0, 1);
    if b then
      (r_l,r_r) ← (span(x->l) || span(x->r));
      if ¬r_l then x->l := null;
      if ¬r_r then x->r := null;
      return true;
    else return false;
}
```

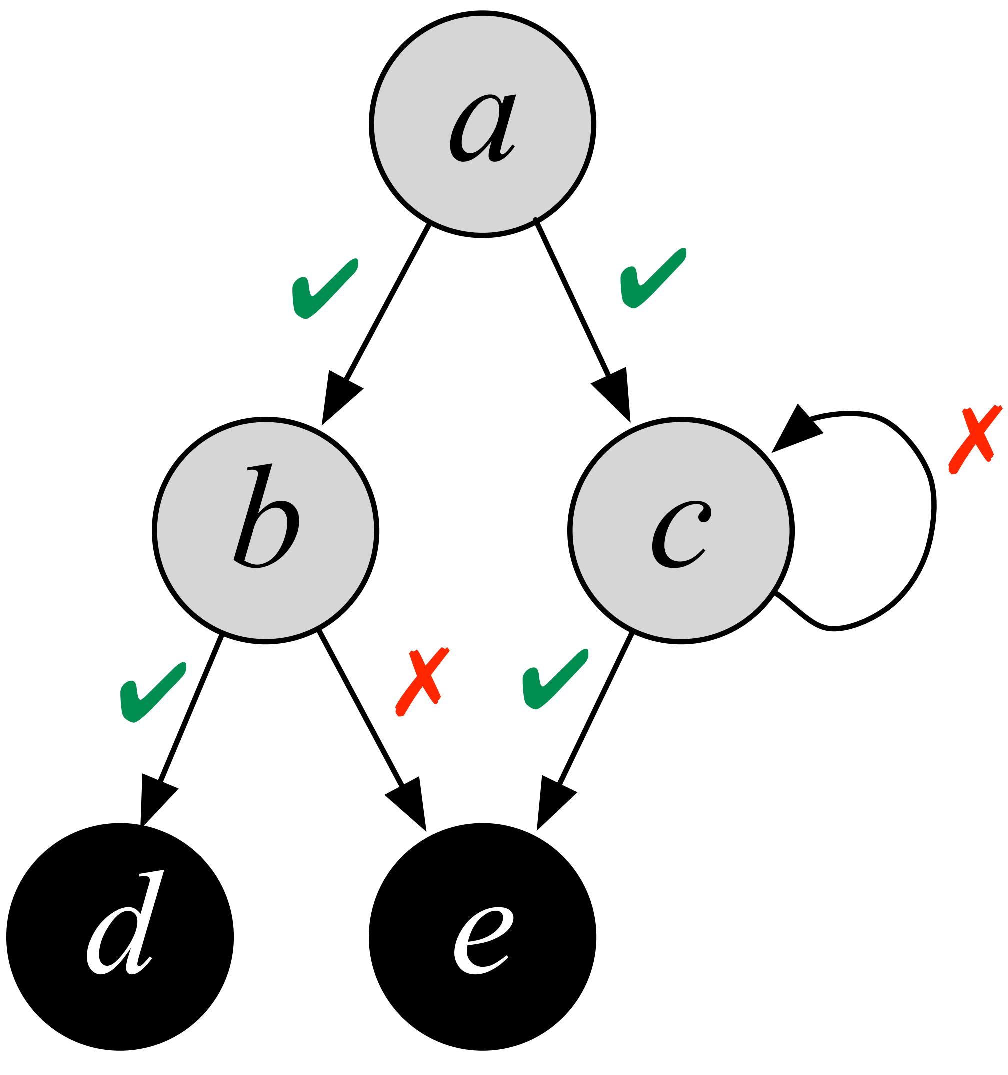
run in parallel for successors

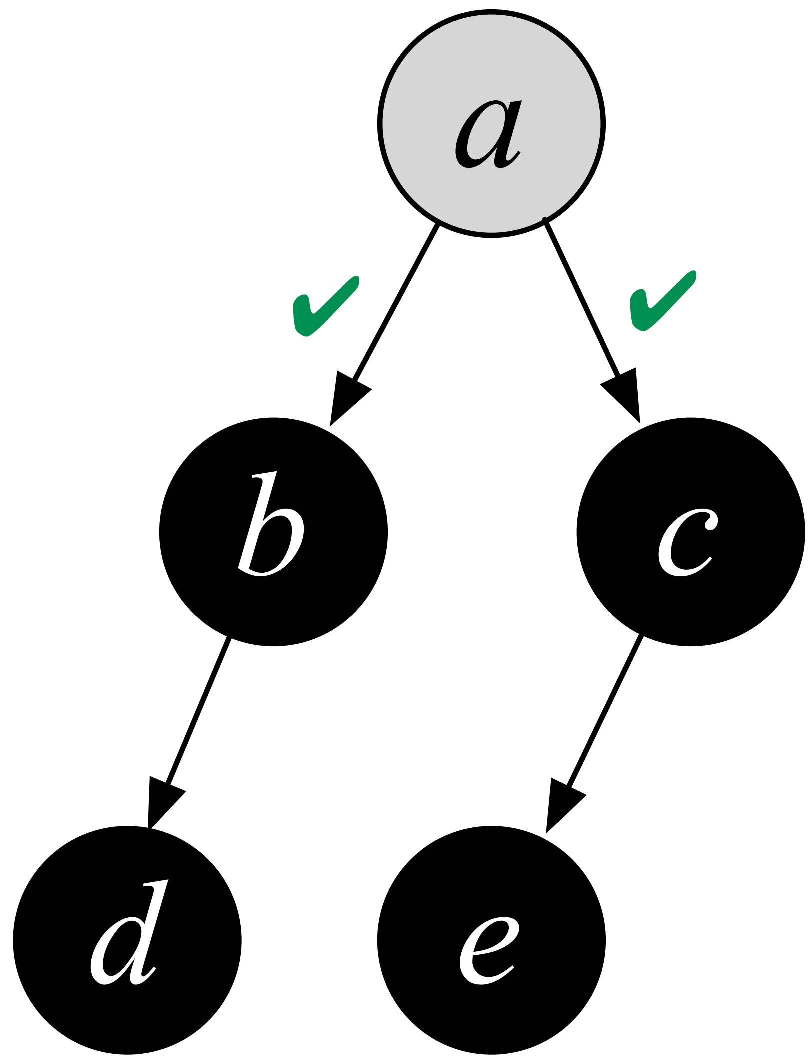
prune redundant edges

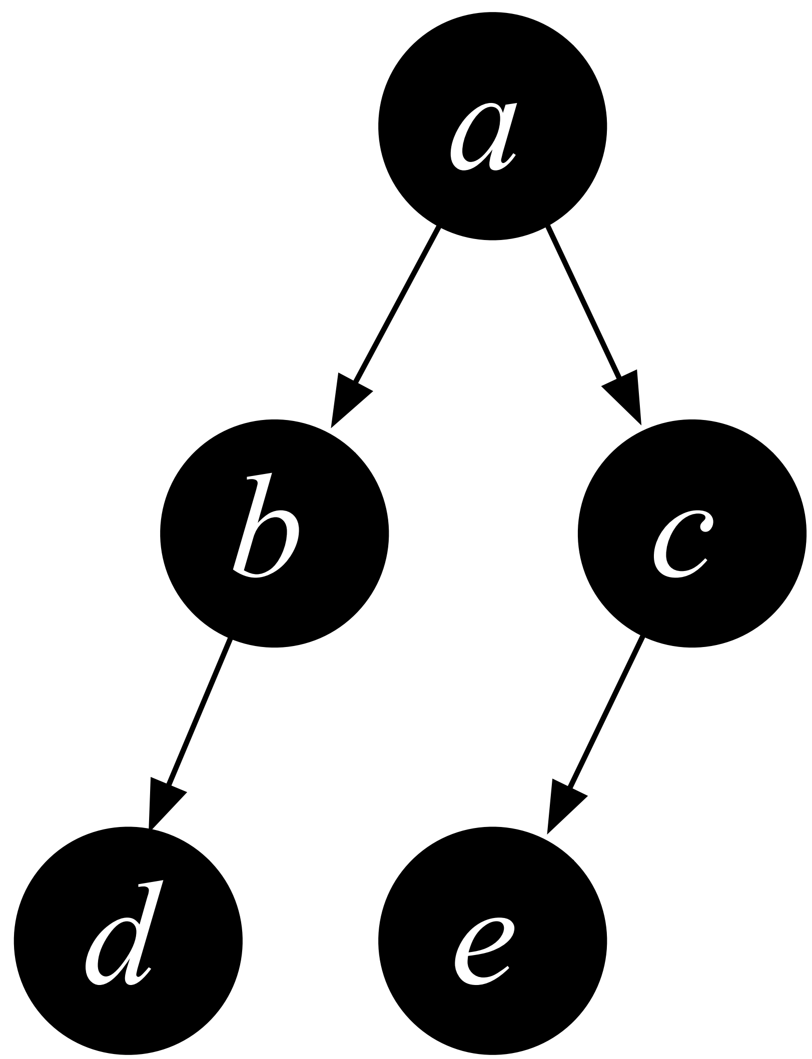

a
b
c
d
e


a
b
c
d
e


a
b
c
d
e
✔
✔
✗
✔
✗
✔

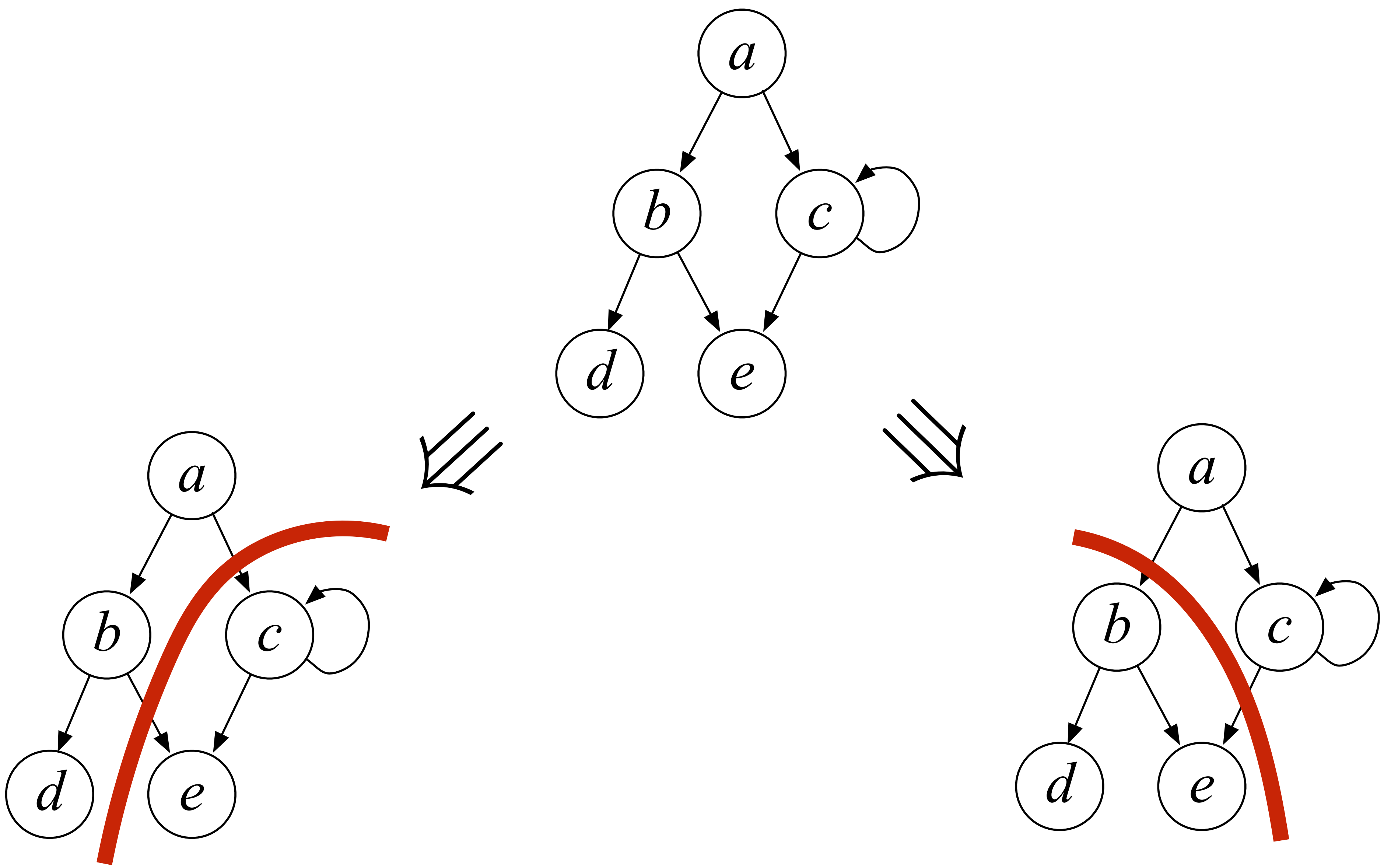

a
b
c
d
e
✔
✔
✗
✔
✗
✔


a
b
c
d
e


a
b
c
d
e

Why verifying span is difficult?

The recursion scheme *does not* follow the shape of the structure.

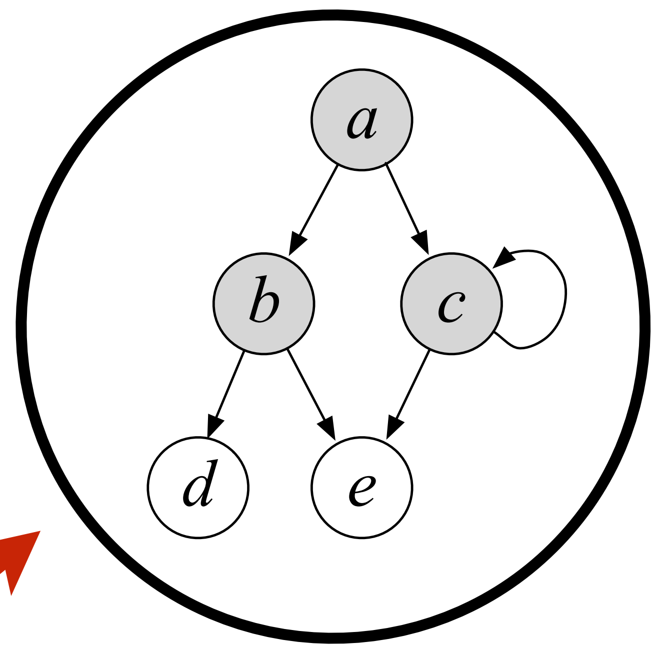


Assumptions for correctness

```
letrec span (x : ptr) : bool = {
  if x == null then return false;
  else
    b ← CAS(x->m, 0, 1);
    if b then
      (rₗ,rᵣ) ← (span(x->l) || span(x->r));
      if ¬rₗ then x->l := null;
      if ¬rᵣ then x->r := null;
      return true;
    else return false;
}
```

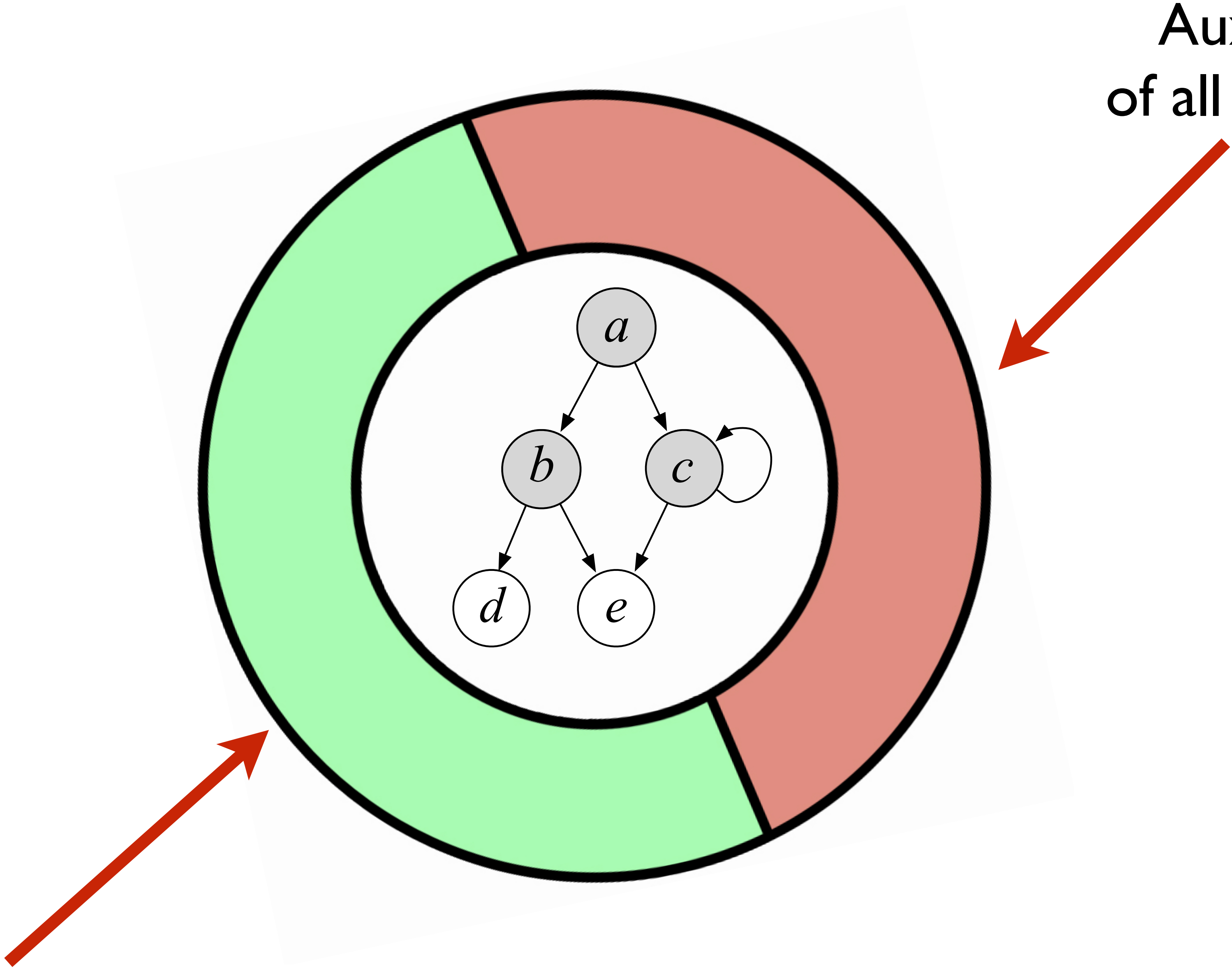
- The graph modified only by the commands of `span`
- The initial call is done from a *root* node

Graph Resource: State



shared state (heap)

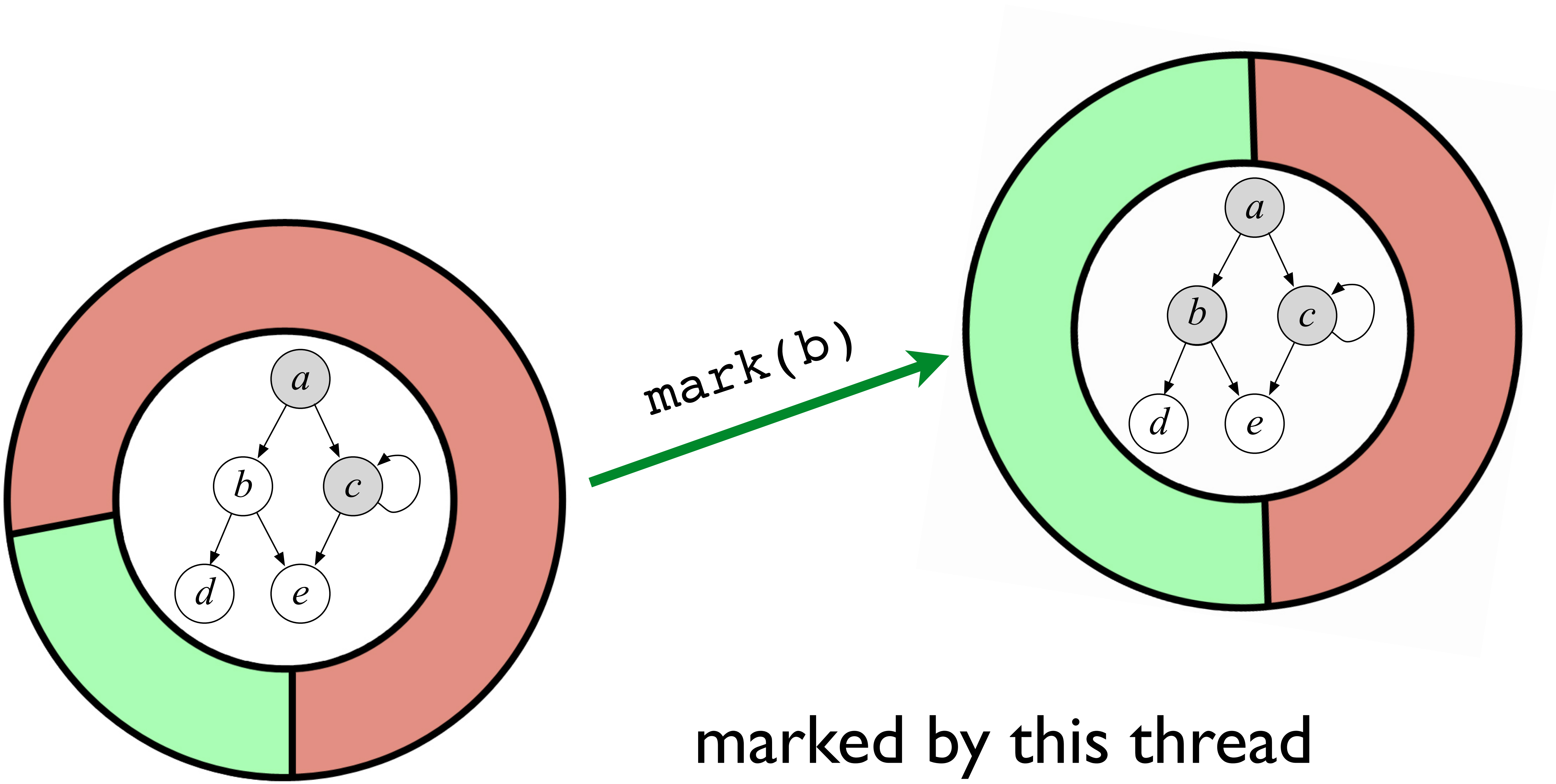
Graph Resource: State



Auxiliary state
of all other threads

Auxiliary state
of this thread

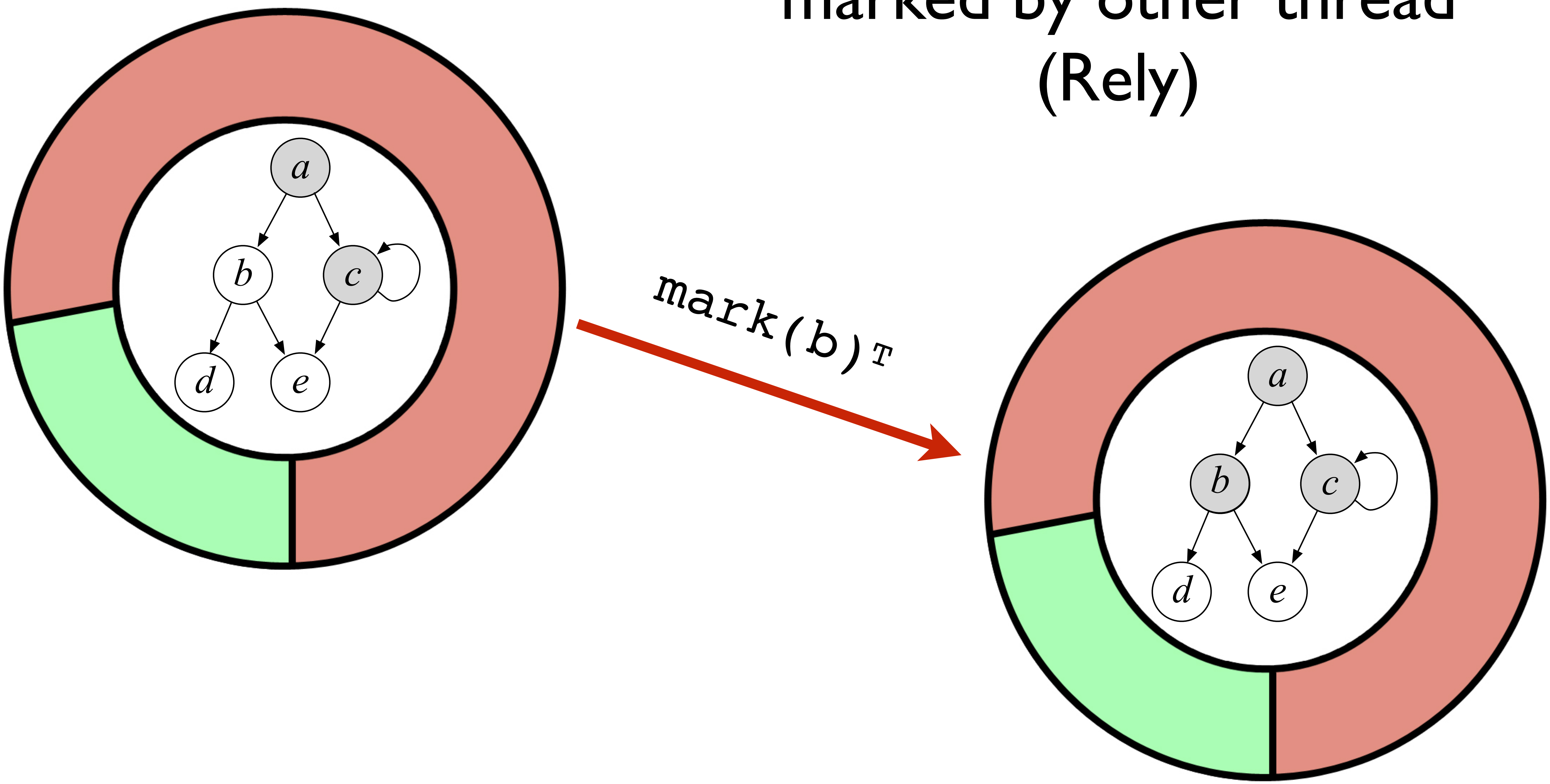
Graph Resource: marking a node



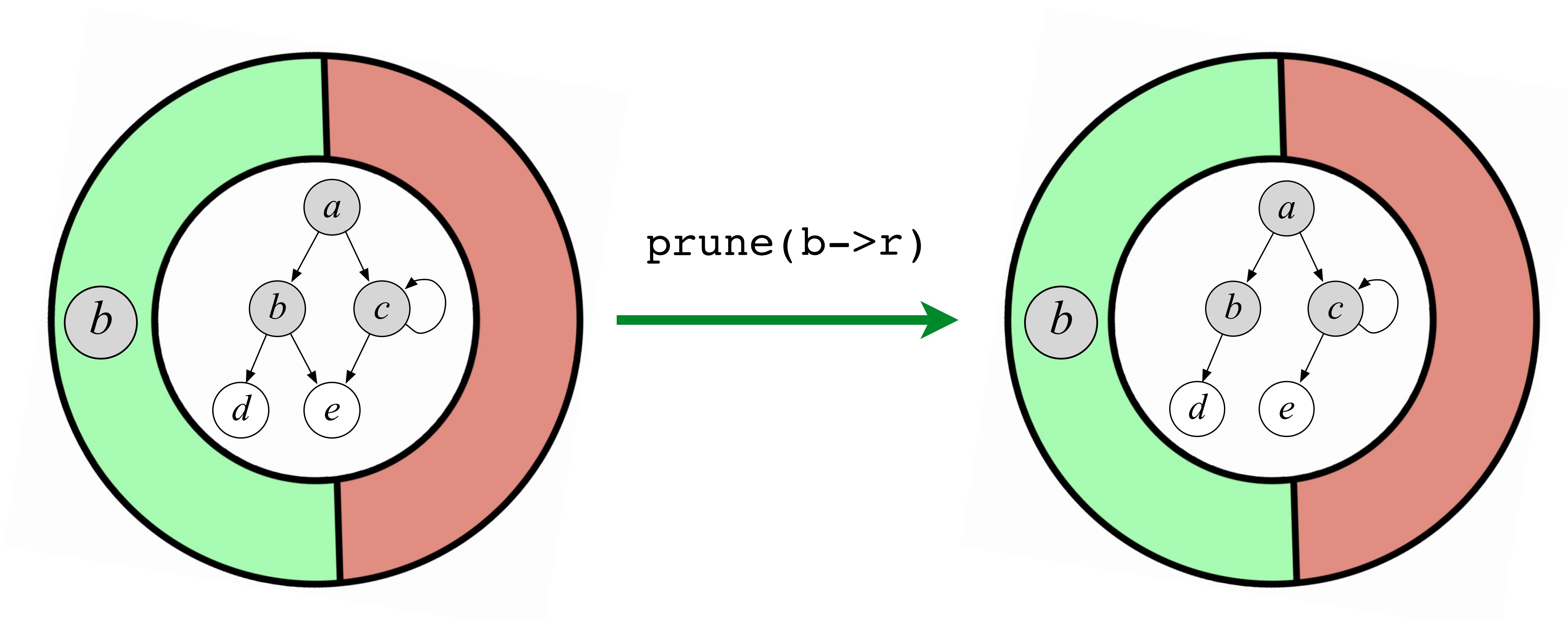
marked by this thread
(Guarantee)

Graph Resource: marking a node

marked by other thread
(Rely)



Graph Resource: pruning an edge



No other thread can do it!

Specification for `span`

$$\{\, P \,\}\ \texttt{span(x)}\ \{\, Q \,\} @ C_{\texttt{SpanTree}}$$

Specification for `span`

$$\texttt{span(x) : span\_tp}\ (x, C_{\texttt{SpanTree}}, P\ , Q)$$

Specification for `span`

```
Definition span_tp (x : ptr) :=
  {i (g1 : graph (joint i))}, STsep [SpanTree]

    (fun s1 => i = s1 ∧ (x == null ∨ x ∈ dom (joint s1)),

     fun (r : bool) s2 => exists g2 : graph (joint s2),
       subgraph g1 g2 ∧
       if r then x != null ∧
                 exists (t : set ptr),
                   self s2 = self i ⊕ t ∧
                   tree g2 x t ∧
                   maximal g2 t ∧
                   front g1 t (self s2 ⊕ other s2)
       else (x == null ∨ mark g2 x) ∧
            self s2 = self i).
```

Specification for `span`

concurrent resource

```
Definition span_tp (x : ptr) :=
  {i (g1 : graph (joint i))}, STsep [SpanTree]

    (fun s1 => i = s1 ∧ (x == null ∨ x ∈ dom (joint s1)),

     fun (r : bool) s2 => exists g2 : graph (joint s2),
       subgraph g1 g2 ∧
       if r then x != null ∧
                 exists (t : set ptr),
                   self s2 = self i ⊕ t ∧
                   tree g2 x t ∧
                   maximal g2 t ∧
                   front g1 t (self s2 ⊕ other s2)
       else (x == null ∨ mark g2 x) ∧
            self s2 = self i).
```

Specification for `span`

precondition

```
Definition span_tp (x : ptr) :=
  {i (g1 : graph (joint i))}, STsep [SpanTree]

    (fun s1 => i = s1 ∧ (x == null ∨ x ∈ dom (joint s1))),

     fun (r : bool) s2 => exists g2 : graph (joint s2),
       subgraph g1 g2 ∧
       if r then x != null ∧
                 exists (t : set ptr),
                   self s2 = self i ⊕ t ∧
                   tree g2 x t ∧
                   maximal g2 t ∧
                   front g1 t (self s2 ⊕ other s2)
       else (x == null ∨ mark g2 x) ∧
            self s2 = self i).
```

Specification for `span`

```
Definition span_tp (x : ptr) :=
  {i (g1 : graph (joint i))}, STsep [SpanTree]

    (fun s1 => i = s1 ∧ (x == null ∨ x ∈ dom (joint s1)),

     fun (r : bool) s2 => exists g2 : graph (joint s2),
       subgraph g1 g2 ∧
       if r then x != null ∧
                 exists (t : set ptr),
                   self s2 = self i ⊕ t ∧
                   tree g2 x t ∧
                   maximal g2 t ∧
                   front g1 t (self s2 ⊕ other s2)
       else (x == null ∨ mark g2 x) ∧
            self s2 = self i).
```

postcondition

Specification for `span`

```
Definition span_tp (x : ptr) :=
  {i (g1 : graph (joint i))}, STsep [SpanTree]

    (fun s1 => i = s1 ∧ (x == null ∨ x ∈ dom (joint s1)),

     fun (r : bool) s2 => exists g2 : graph (joint s2),
       subgraph g1 g2 ∧
       if r then x != null ∧
                 exists (t : set ptr),
                   self s2 = self i ⊕ t ∧
                   tree g2 x t ∧
                   maximal g2 t ∧
                   front g1 t (self s2 ⊕ other s2)
       else (x == null ∨ mark g2 x) ∧
            self s2 = self i).
```

logical variables

Specification for `span`

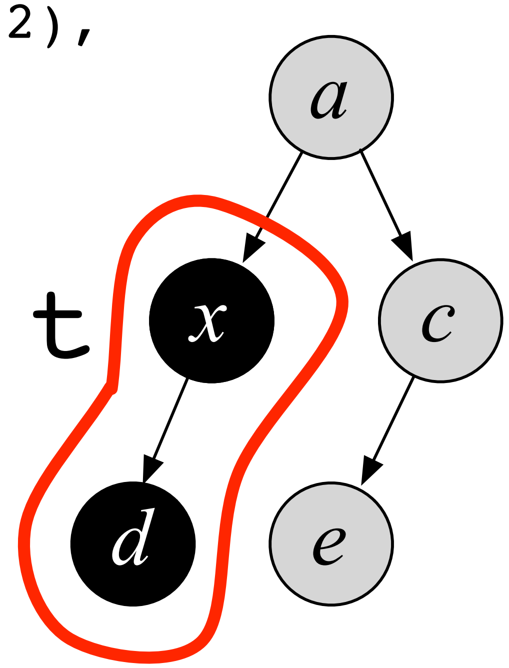
```
Definition span_tp (x : ptr) :=
  {i (g1 : graph (joint i))}, STsep [SpanTree]

    (fun s1 => i = s1 ∧ (x == null ∨ x ∈ dom (joint s1)),

     fun (r : bool) s2 => exists g2 : graph (joint s2),
       subgraph g1 g2 ∧
       if r then x != null ∧
                exists (t : set ptr),
                  self s2 = self i ⊕ t ∧
                  tree g2 x t ∧
                  maximal g2 t ∧
                  front g1 t (self s2 ⊕ other s2)
       else (x == null ∨ mark g2 x) ∧
            self s2 = self i).
```

Specification for `span`

```
Definition span_tp (x : ptr) :=
  {i (g1 : graph (joint i))}, STsep [SpanTree]

    (fun s1 => i = s1 ∧ (x == null ∨ x ∈ dom (joint s1)),

     fun (r : bool) s2 => exists g2 : graph (joint s2),
       subgraph g1 g2 ∧
       if r then x != null ∧
                 exists (t : set ptr),
                   self s2 = self i ⊕ t ∧
                   tree g2 x t ∧
                   maximal g2 t ∧
                   front g1 t (self s2 ⊕ other s2)
       else (x == null ∨ mark g2 x) ∧
            self s2 = self i).
```

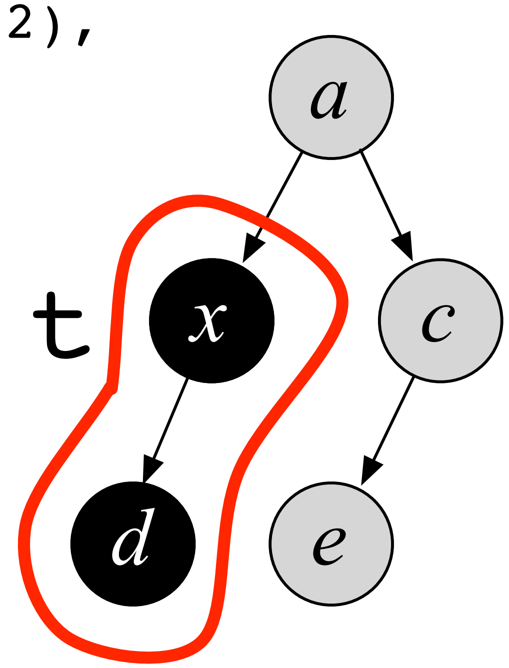
Specification for `span`

```
Definition span_tp (x : ptr) :=
  {i (g1 : graph (joint i))}, STsep [SpanTree]

    (fun s1 => i = s1 ∧ (x == null ∨ x ∈ dom (joint s1)),

     fun (r : bool) s2 => exists g2 : graph (joint s2),
       subgraph g1 g2 ∧
       if r then x != null ∧
                 exists (t : set ptr),
                   self s2 = self i ⊕ t ∧
                   tree g2 x t ∧
                   maximal g2 t ∧
                   front g1 t (self s2 ⊕ other s2)
       else (x == null ∨ mark g2 x) ∧
            self s2 = self i).
```



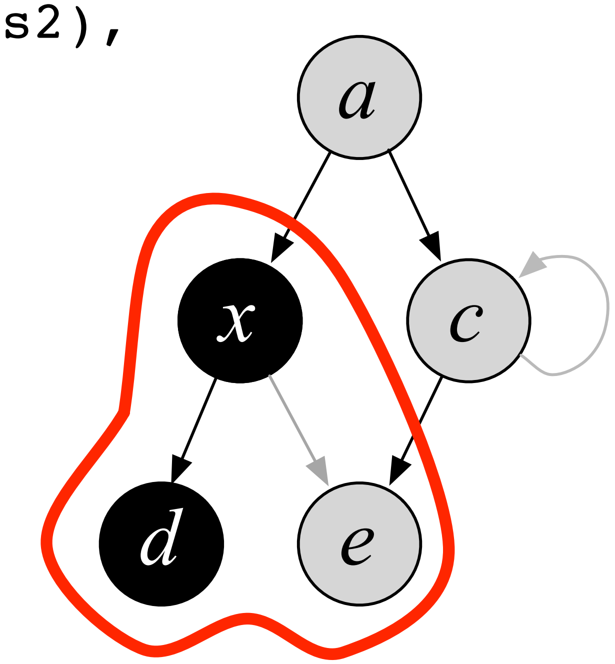
Specification for `span`

```
Definition span_tp (x : ptr) :=
  {i (g1 : graph (joint i))}, STsep [SpanTree]

    (fun s1 => i = s1 ∧ (x == null ∨ x ∈ dom (joint s1)),

     fun (r : bool) s2 => exists g2 : graph (joint s2),
       subgraph g1 g2 ∧
       if r then x != null ∧
                 exists (t : set ptr),
                   self s2 = self i ⊕ t ∧
                   tree g2 x t ∧
                   maximal g2 t ∧
                   front g1 t (self s2 ⊕ other s2)
       else (x == null ∨ mark g2 x) ∧
            self s2 = self i).
```



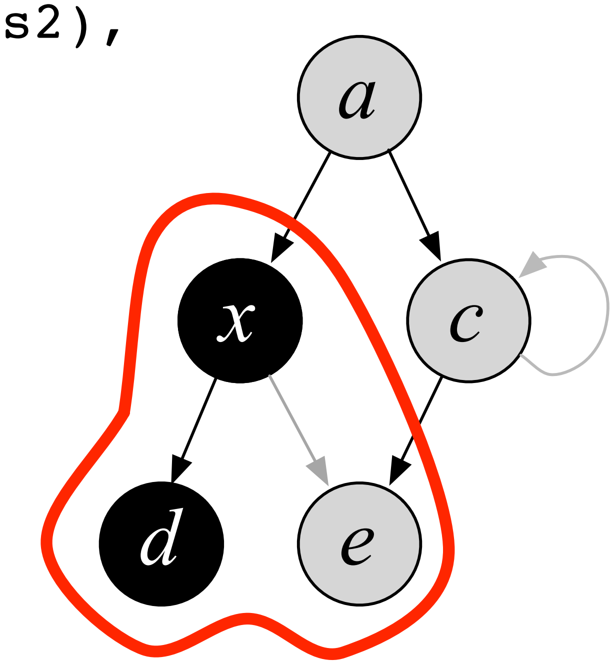
Specification for `span`

```
Definition span_tp (x : ptr) :=
  {i (g1 : graph (joint i))}, STsep [SpanTree]

    (fun s1 => i = s1 ∧ (x == null ∨ x ∈ dom (joint s1)),

     fun (r : bool) s2 => exists g2 : graph (joint s2),
       subgraph g1 g2 ∧
       if r then x != null ∧
                exists (t : set ptr),
                  self s2 = self i ⊕ t ∧
                  tree g2 x t ∧
                  maximal g2 t ∧
                  front g1 t (self s2 ⊕ other s2)
       else (x == null ∨ mark g2 x) ∧
            self s2 = self i).
```



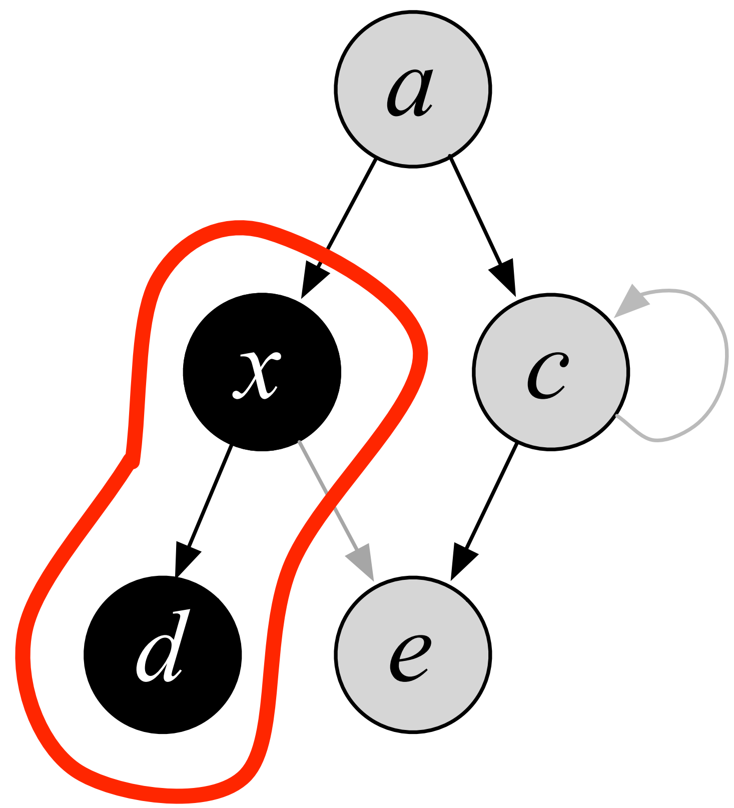
Specification for `span`

```
Definition span_tp (x : ptr) :=
  {i (g1 : graph (joint i))}, STsep [SpanTree]

    (fun s1 => i = s1 ∧ (x == null ∨ x ∈ dom (joint s1)),

     fun (r : bool) s2 => exists g2 : graph (joint s2),
       subgraph g1 g2 ∧
       if r then x != null ∧
                 exists (t : set ptr),
                   self s2 = self i ⊕ t ∧
                   tree g2 x t ∧
                   maximal g2 t ∧
                   front g1 t (self s2 ⊕ other s2)
       else (x == null ∨ mark g2 x) ∧
            self s2 = self i).
```



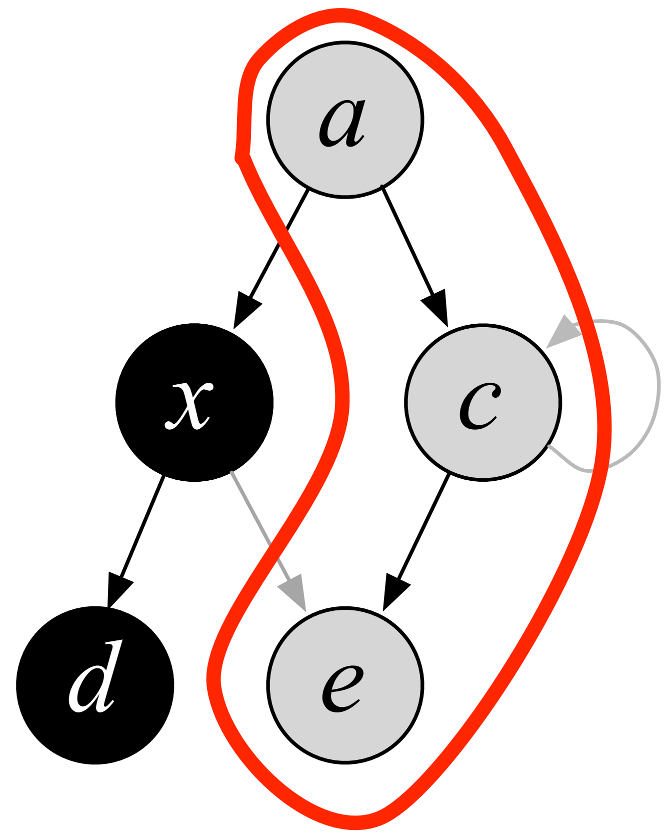
Open world assumption
(assuming other-interference)

Cancelling the interference




```
front g1 t (self s2 ⊕ other s2)
```

Cancelling the interference



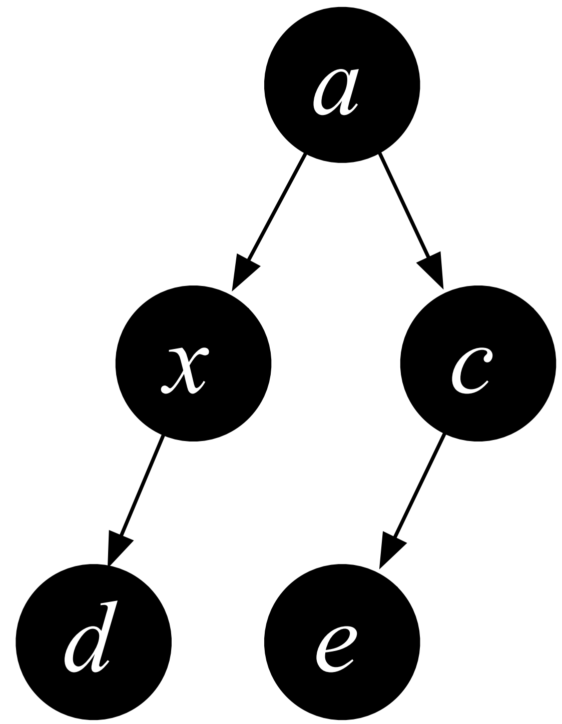
`front g1 t (self s2 ⊕ other s2)`

$\texttt{hide}\ \mathsf{C}_{\texttt{SpanTree}}(\texttt{h}_1)\ \texttt{in}\ \{\ \texttt{span}(a)\ \}$

donated local heap

no *other* threads at the end

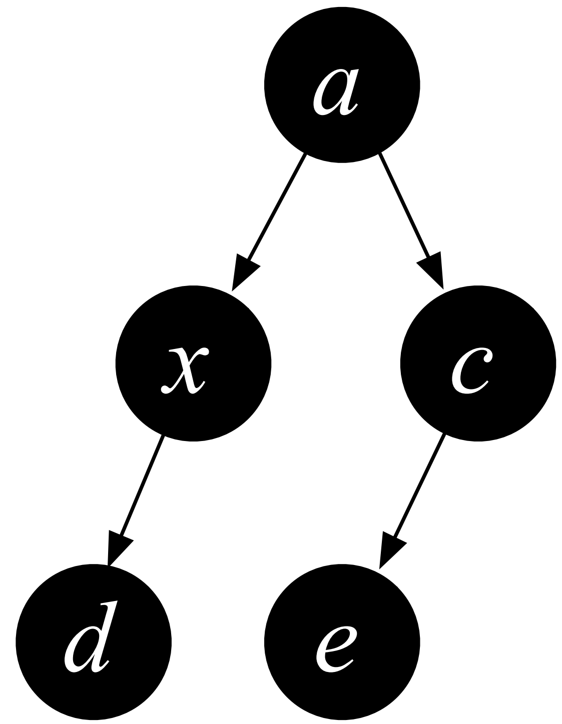
Cancelling the interference




```
front g1 t (self s2)
```

hide $C_{\mathtt{SpanTree}}(\mathtt{h}_1)$ **in** { span(a) }

Cancelling the interference



follow from postcondition and graph connectivity

```
tree g2 a t              ∧ maximal g2 t ∧
front g1 t (self s2)     ∧ t = self s2  ∧
is_root a g1             ∧ subgraph g1 g2
⇒ spanning t g1
```

Key insights

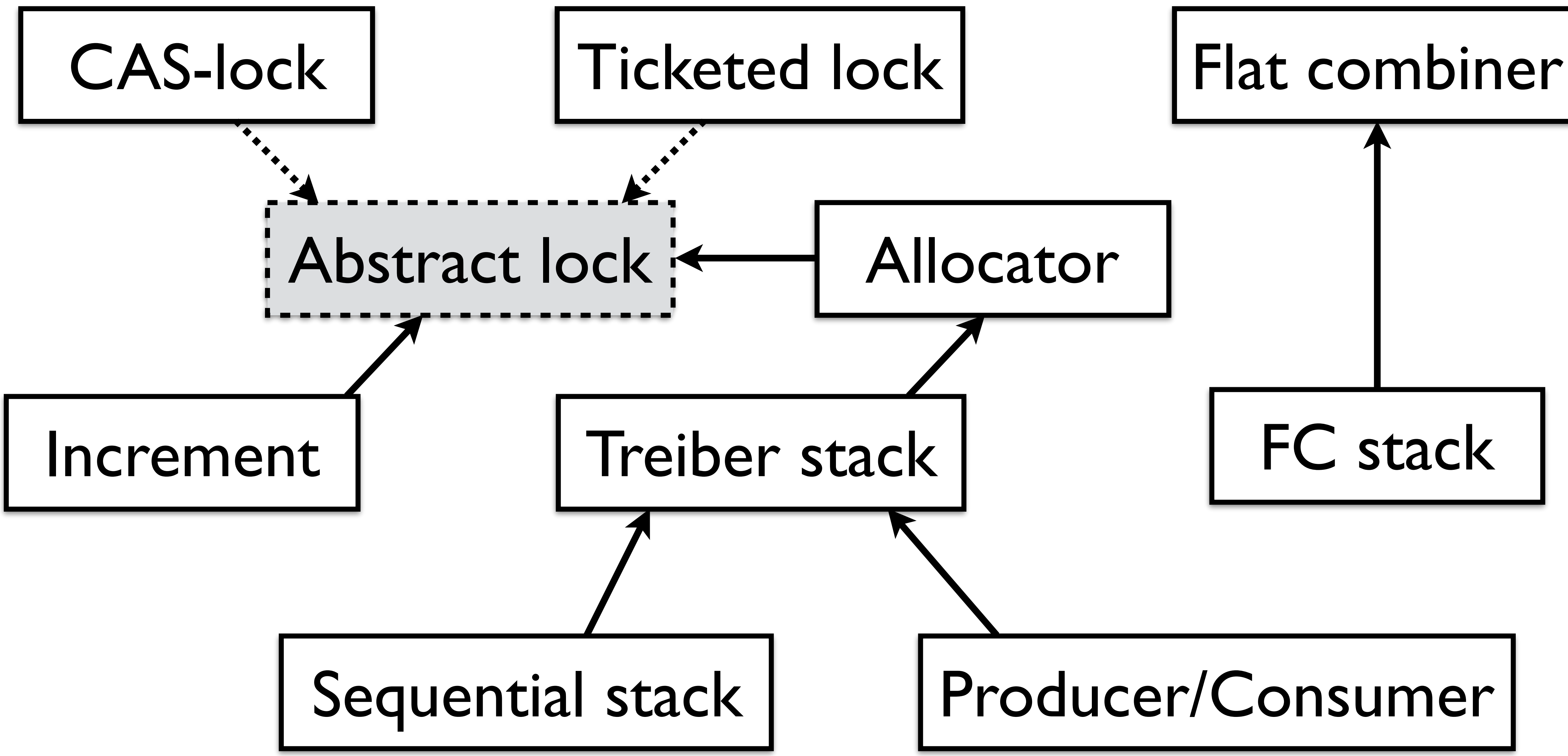
- Subjectivity — reasoning with *self* and *other*
- Histories — temporal specification via *state*
- Deep Sharing

Key insights

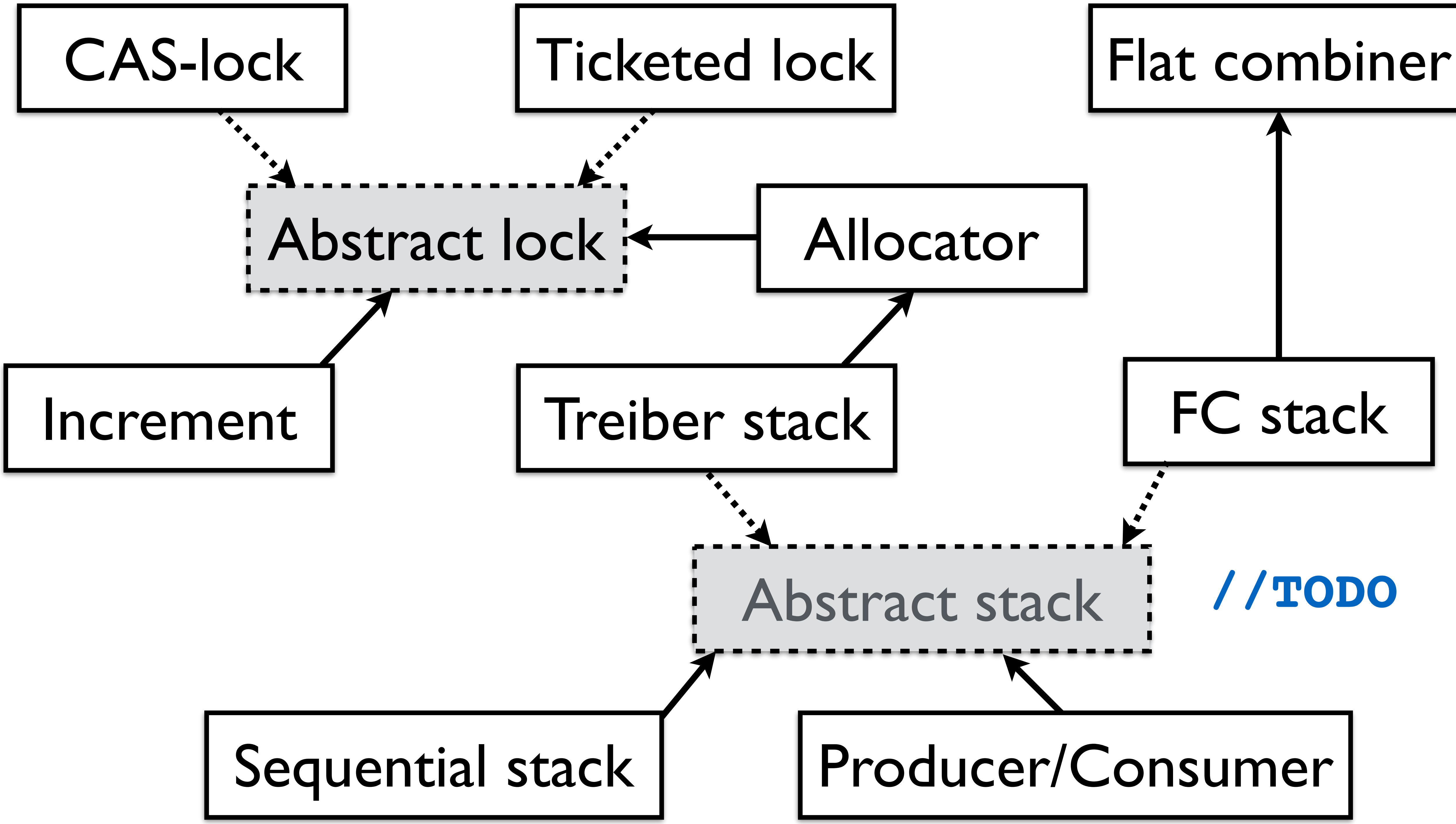
- Subjectivity — reasoning with *self* and *other*
- Histories — temporal specification via *state*
- Deep Sharing — splitting *auxiliary* state

Bonus

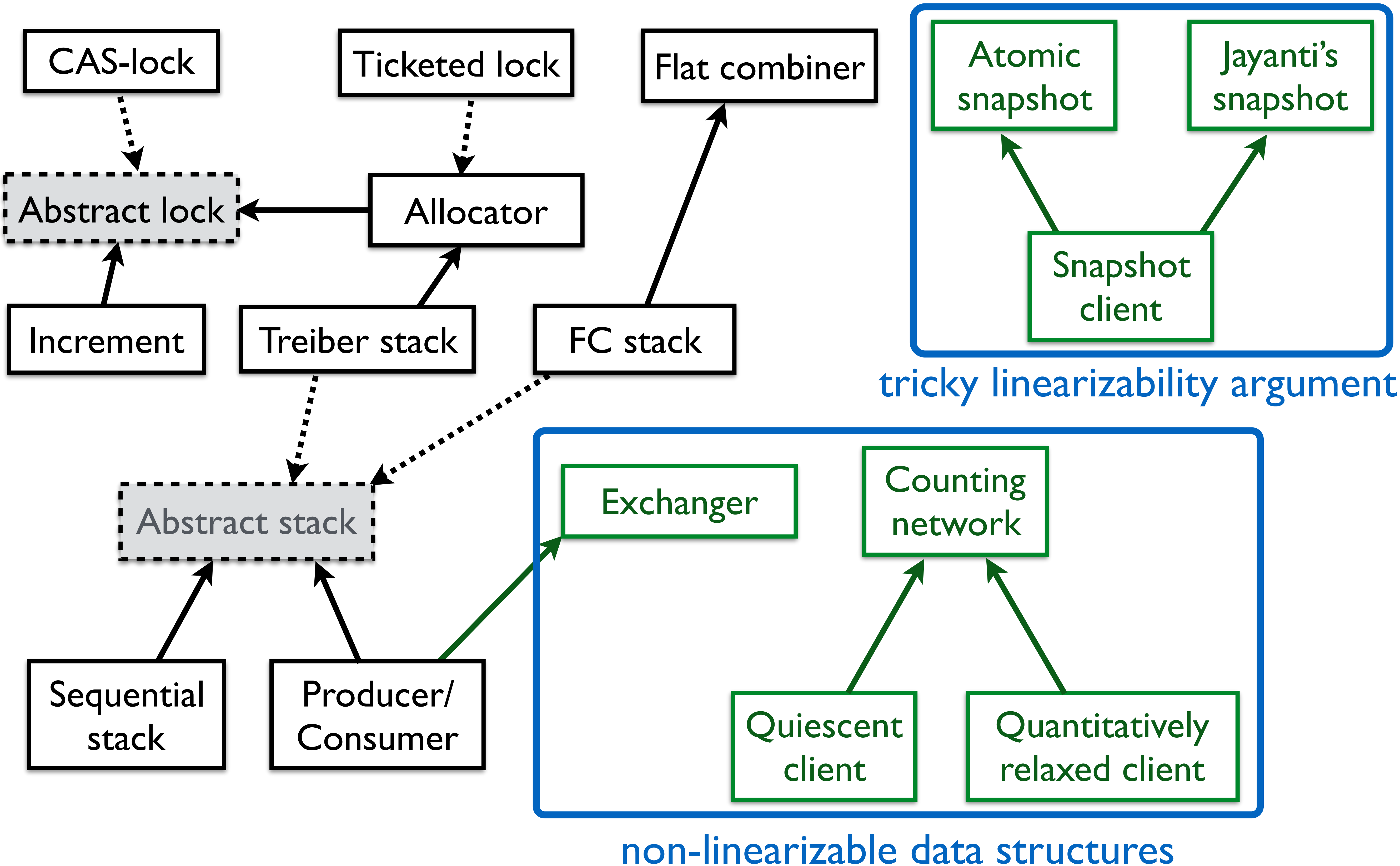
Composing programs and proofs



Composing programs and proofs



Composing programs and proofs



To take away

- **Subjectivity**:
  thread-modularity = reasoning in terms of *self* and *other*
- **Histories**: capturing *temporal* aspects via *auxiliary* state
- **Deep Sharing**: reasoning about ramified data structures by splitting not *real*, but *auxiliary* state

Thanks!