Mechanized Verification of Fine-grained Concurrent Programs

Ilya Sergey Aleks Nanevski Anindya Banerjee

institute IMDEA software

PLDI 2015

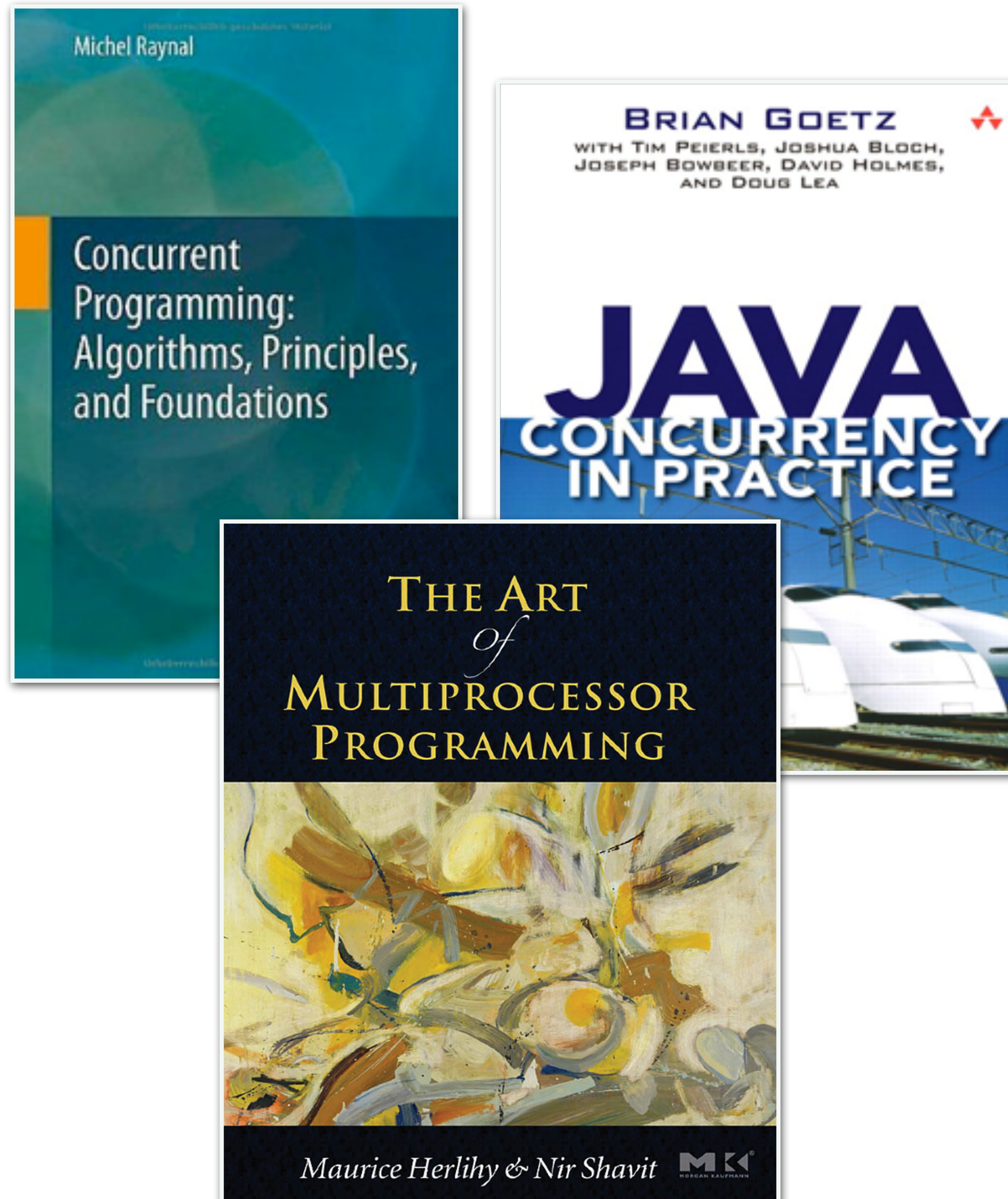
Terminology

- Coarse-grained Concurrency —
  synchronisation between threads via **locks**;

- Fine-grained Concurrency —
  synchronisation via RMW operations (e.g., **CAS**).

Some FG concurrent programs

- Spin-lock
- Ticketed lock
- Bakery lock
- Filter lock
- Lock-free atomic snapshot
- Treiber stack
- Michael stack
- HSY elimination-based stack
- Lock-coupling set
- Optimistic list-based set
- Lazy concurrent list-based set
- Michael-Scott queue
- Harris et al.'s MCAS
- Concurrent counters
- Concurrent allocators
- Flat Combiner
- Concurrent producer/consumer
- Concurrent indices
- Concurrent barriers
- …



Using and verifying FG concurrency

Great scalability —
high performance on multi-core CPU architectures

Sophisticated interference between threads —
difficult to specify and verify formally

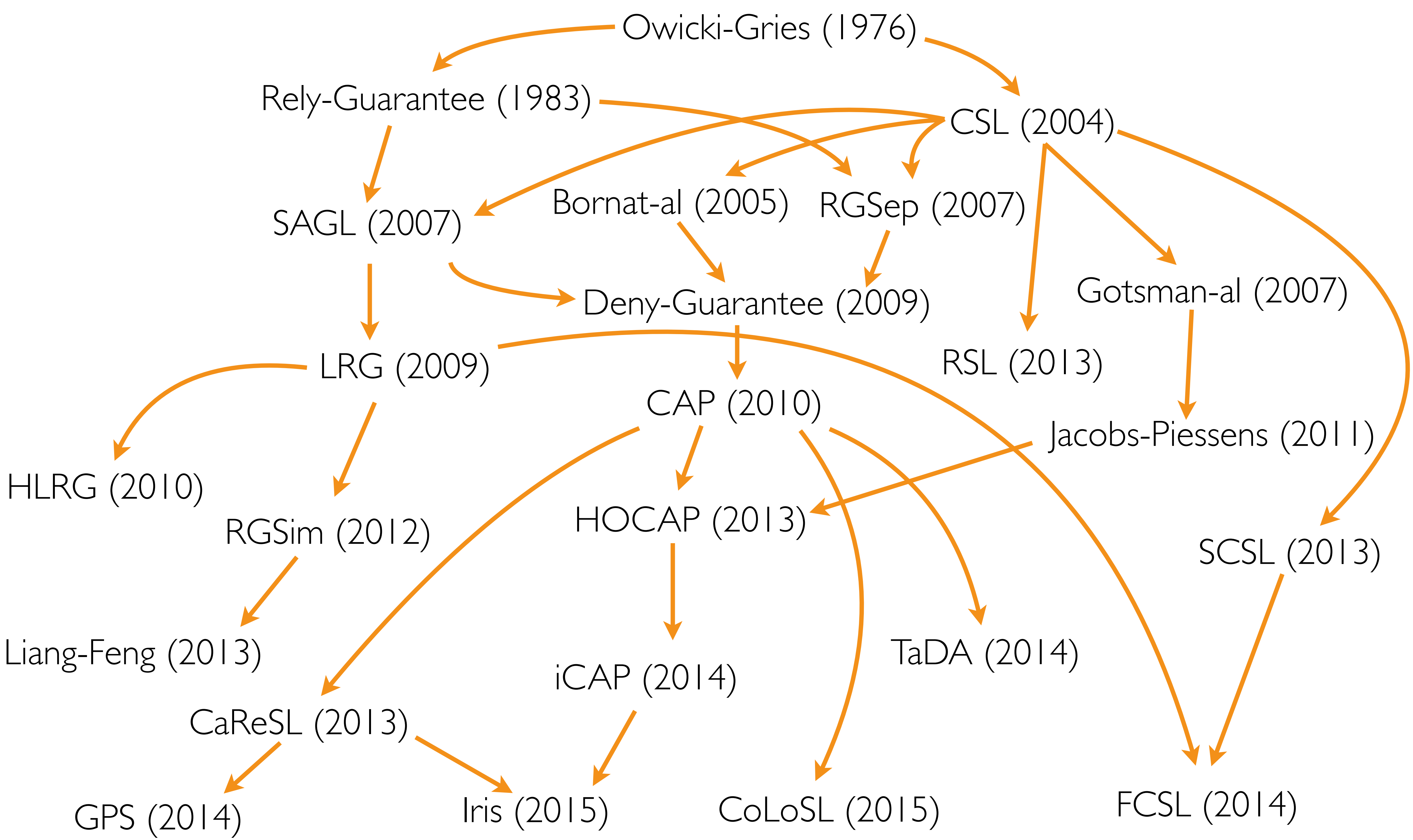
Specifications in program logics

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

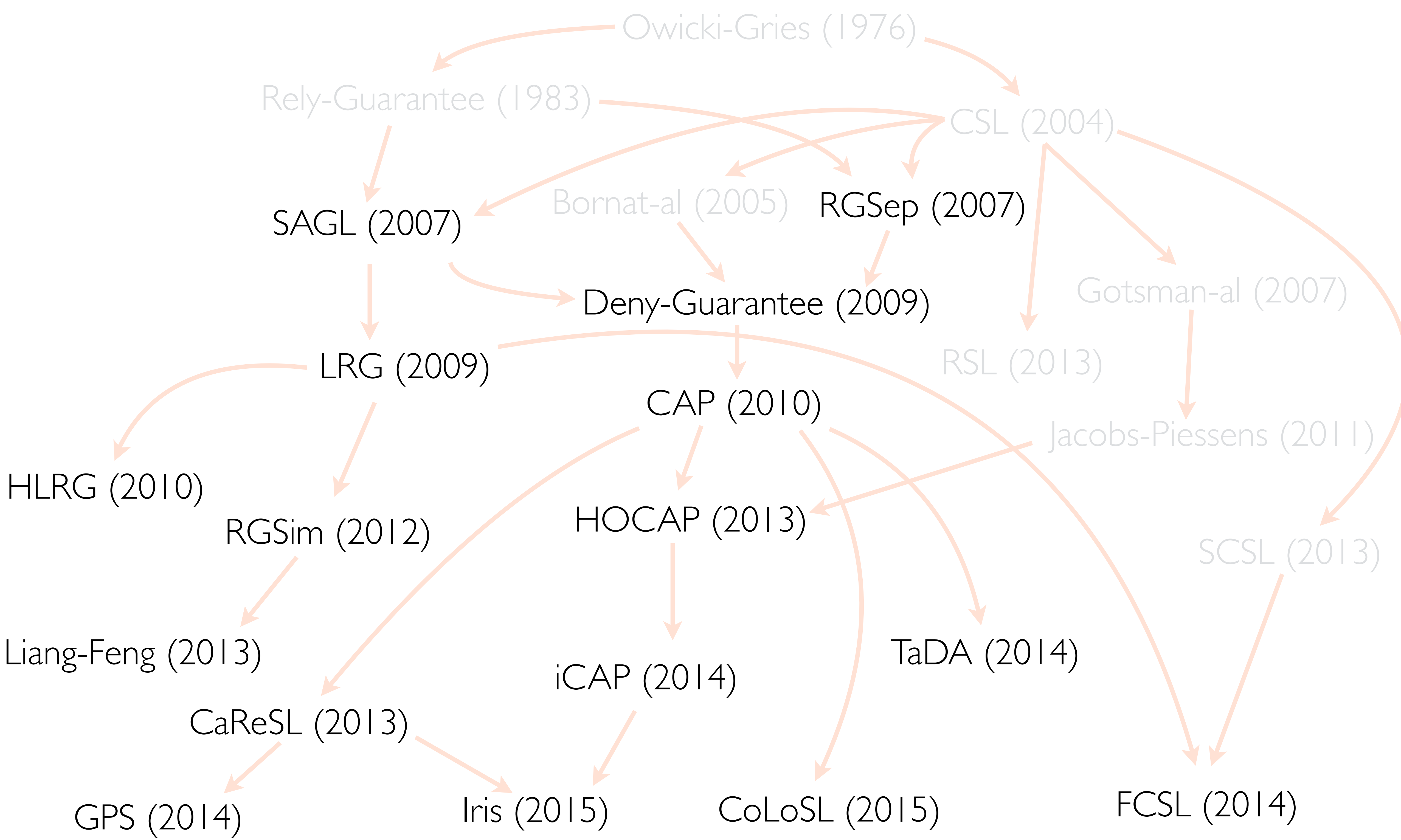
precondition   postcondition

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

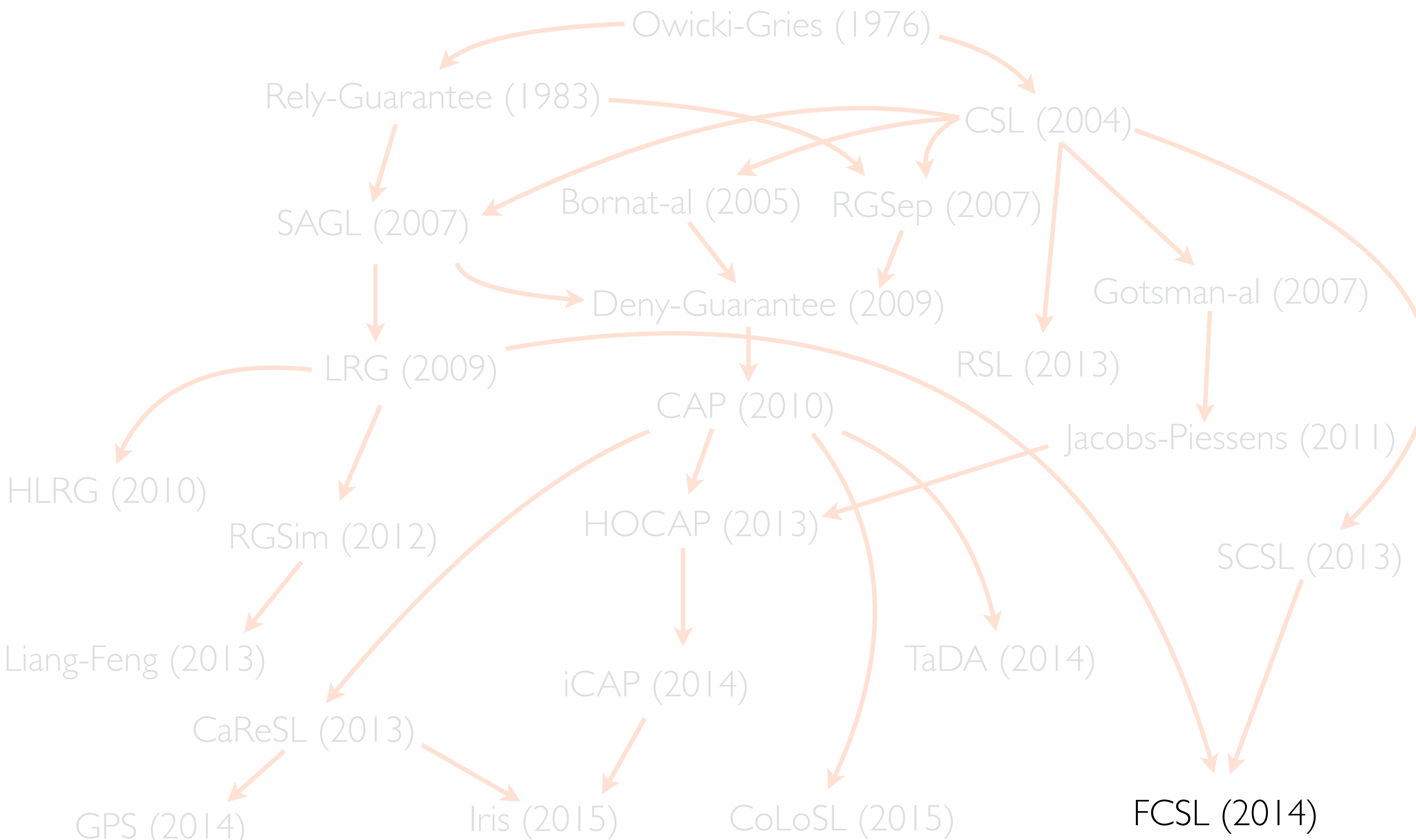
Program logics for concurrency



Program logics for concurrency



Program logics for concurrency



FCSL: Fine-grained Concurrent Separation Logic

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

a *logic* for specifying and verifying
FG concurrent programs

and also

a *verification tool*,
implemented as a DSL in Coq

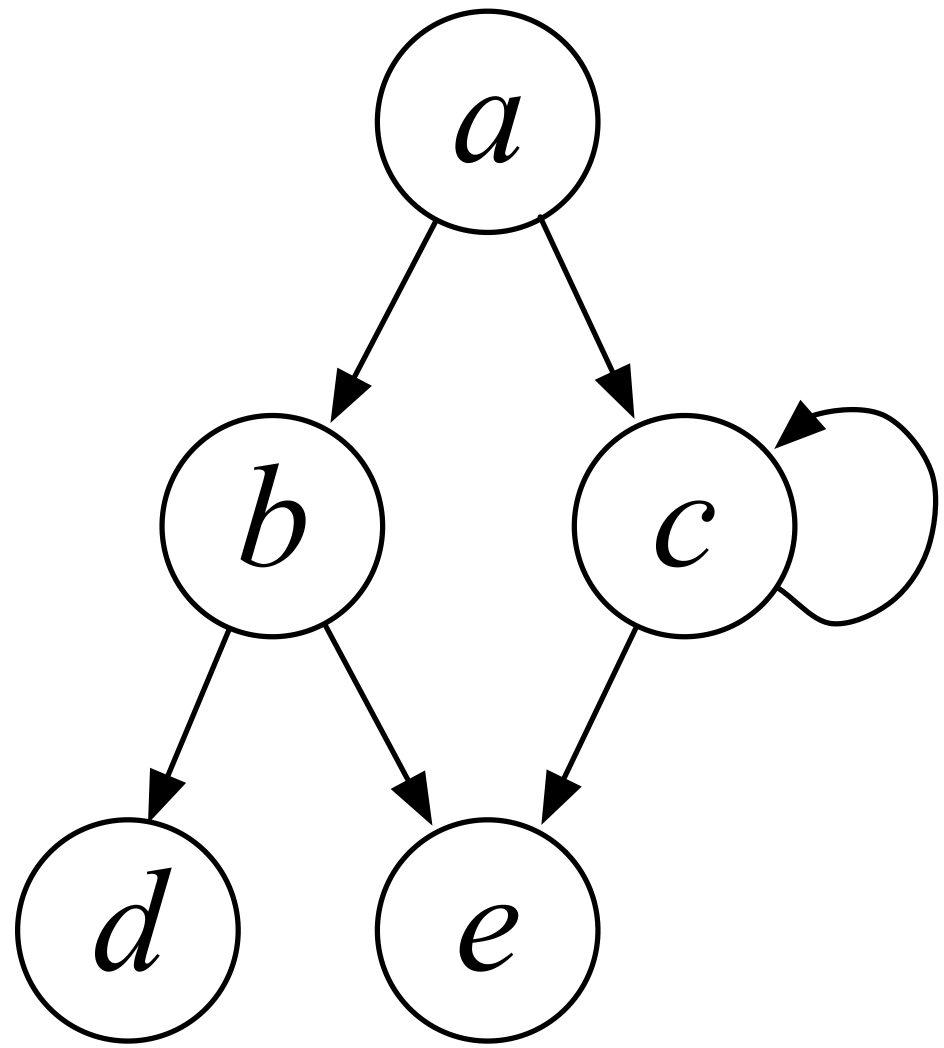
(this talk)

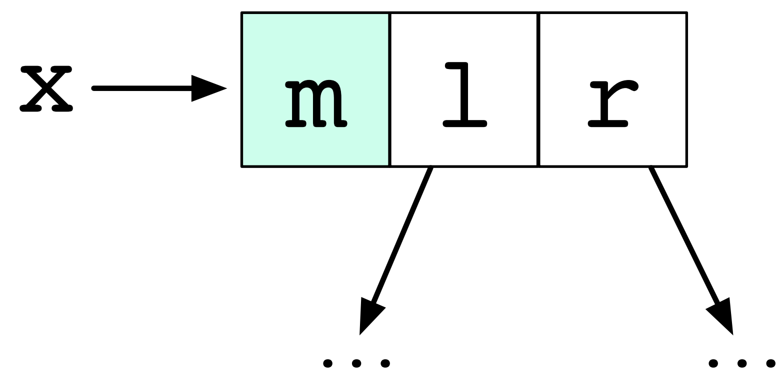
Key Ingredients

- Subjective Auxiliary State
- State-Transition Systems
- Types

Running example

Concurrent construction
of a *spanning tree*
of a binary graph


a
b
c
d
e



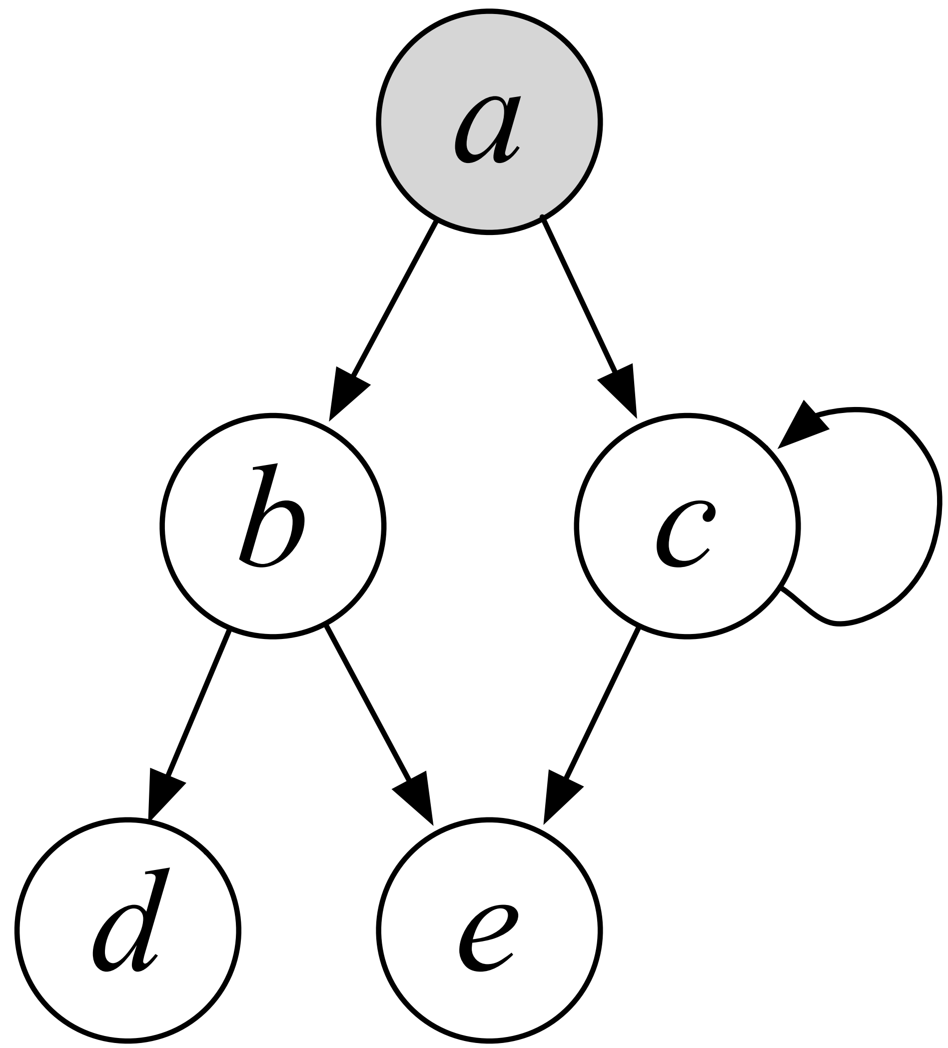
check the node **x**

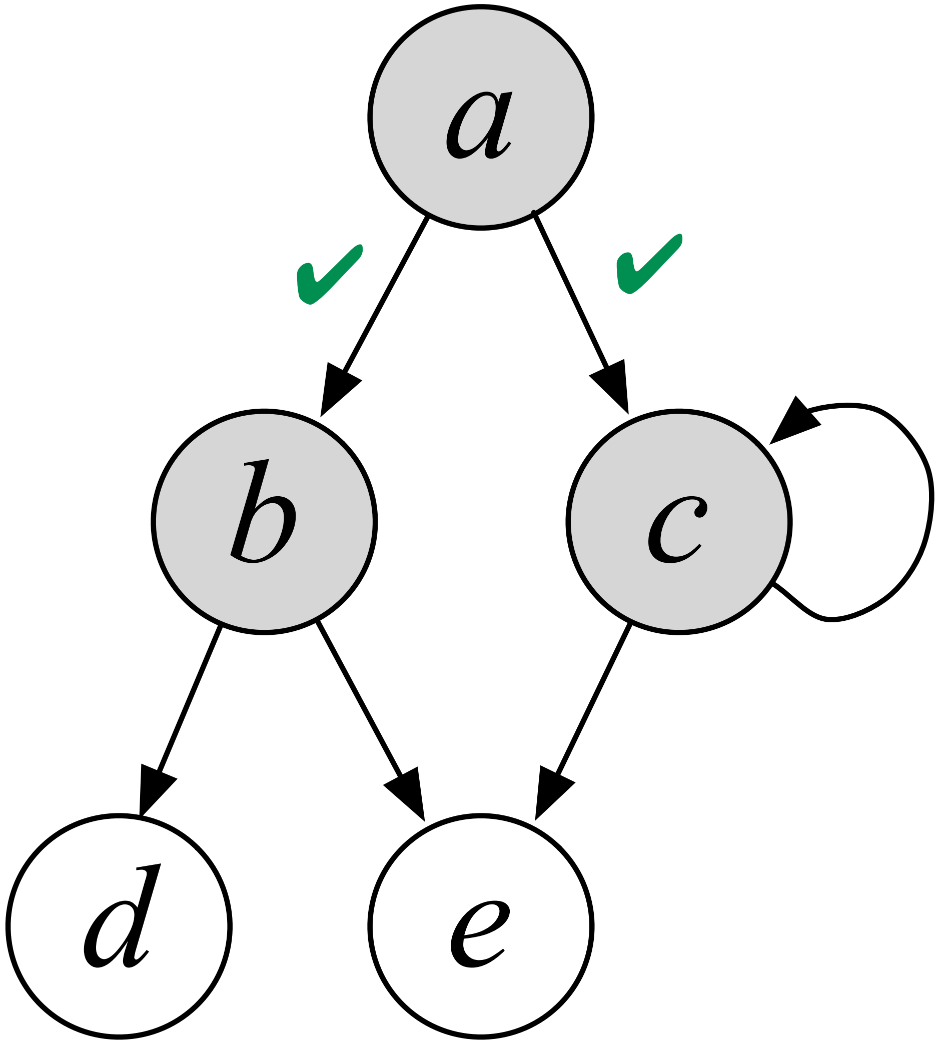
mark the node **x**

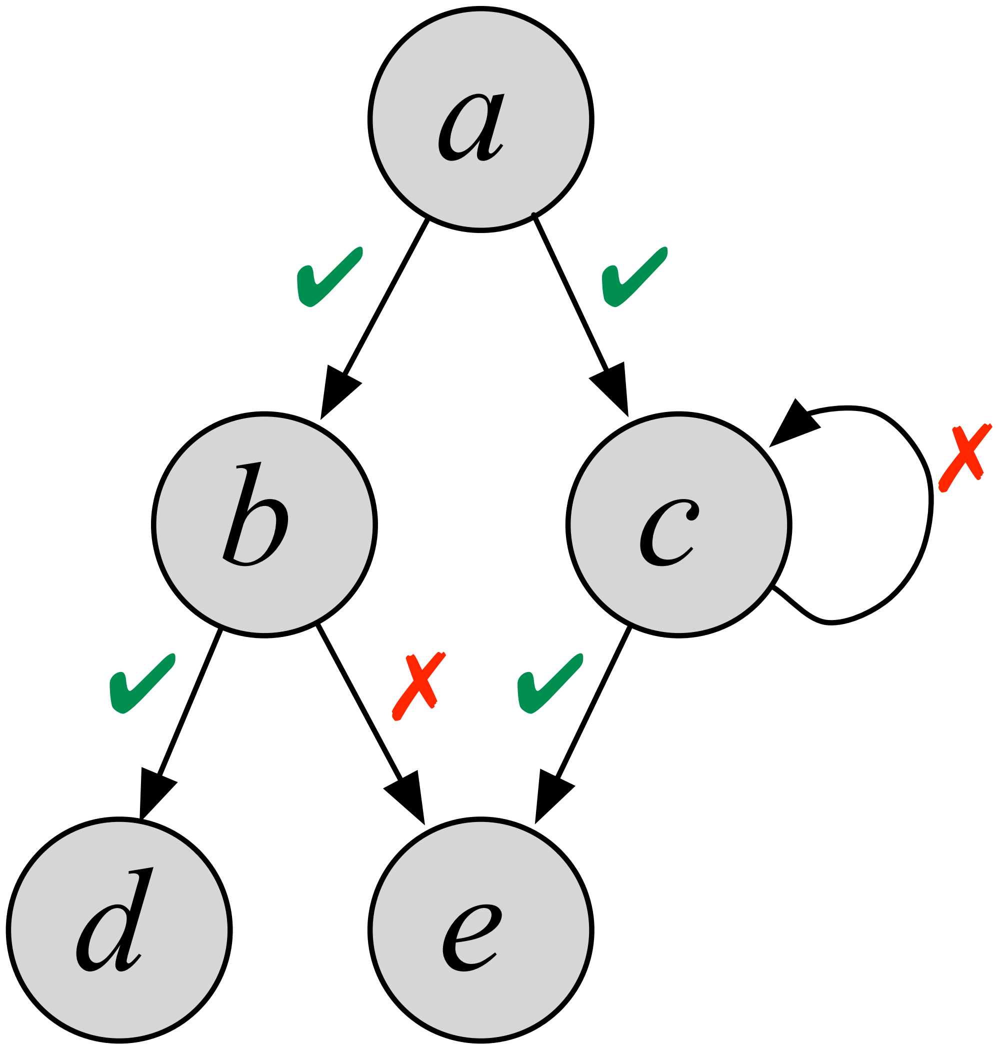
```
letrec span (x : ptr) : bool = {
  if x == null then return false;
  else
    b ← CAS(x->m, 0, 1);
    if b then
      (rₗ,rᵣ) ← (span(x->l) || span(x->r));
      if ¬rₗ then x->l := null;
      if ¬rᵣ then x->r := null;
      return true;
    else return false;
}
```

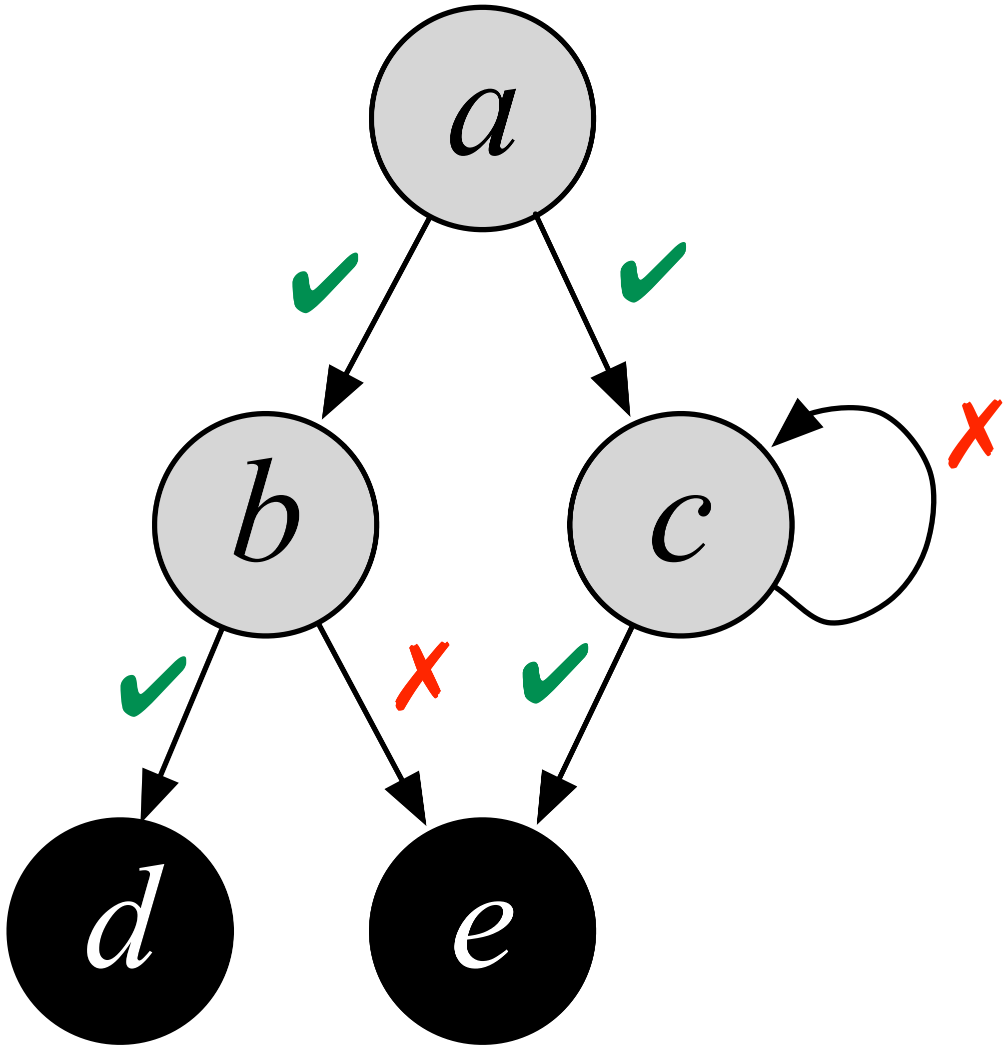
run in parallel for successors

prune redundant edges

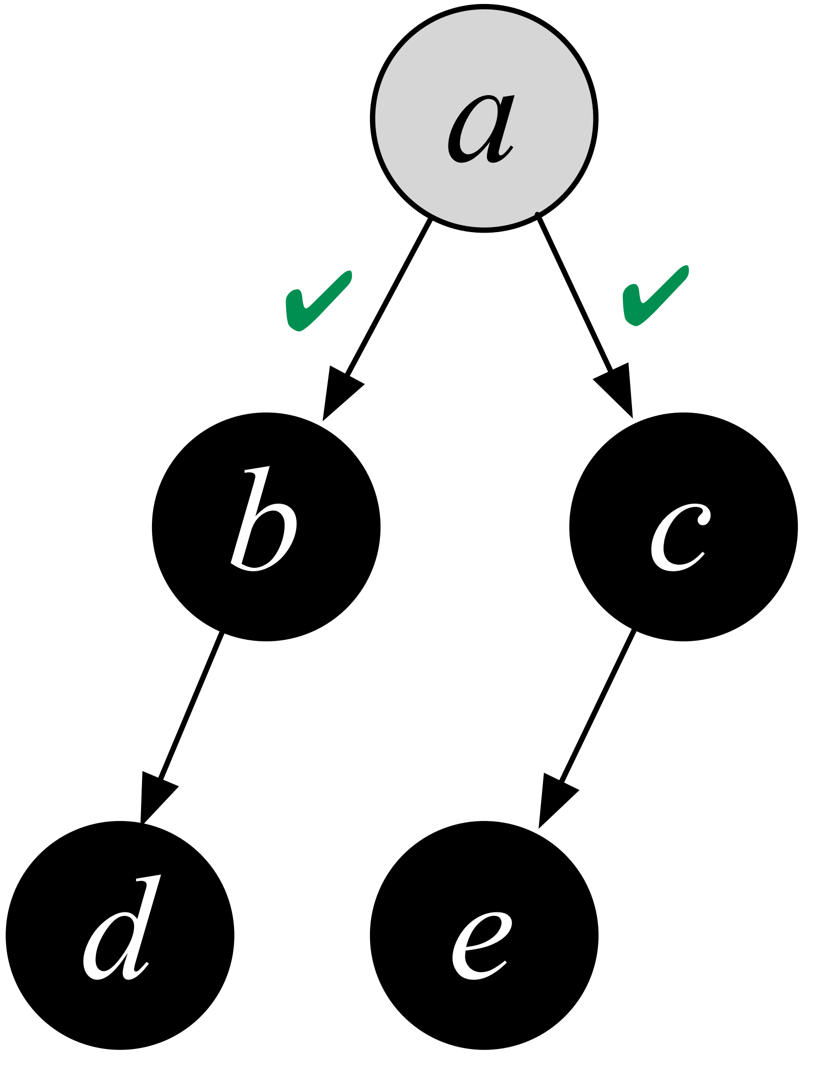

a
b
c
d
e


a
b
c
d
e

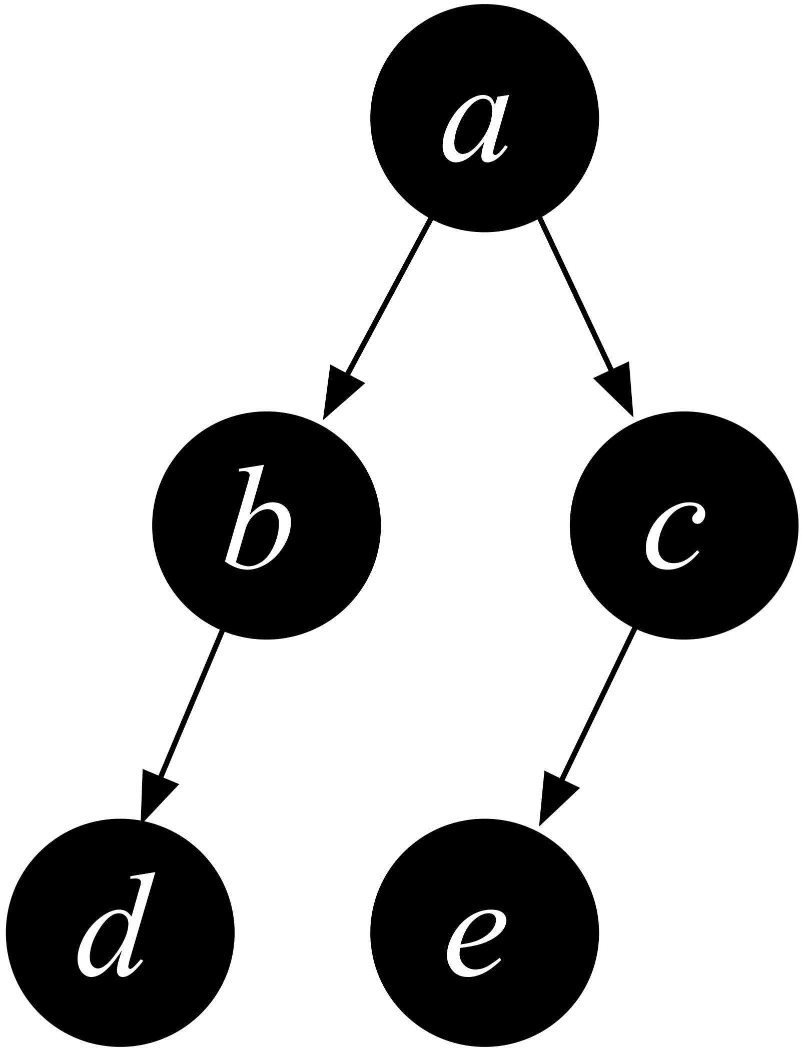

a
b
c
d
e
✔
✔
✔
✗
✔
✗



a
b
c
d
e



a
b
c
d
e



a
b
c
d
e

The verification goal

```
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;
}
```

Prove the resulting heap to represent a spanning tree of the initial one

Establishing correctness of `span`

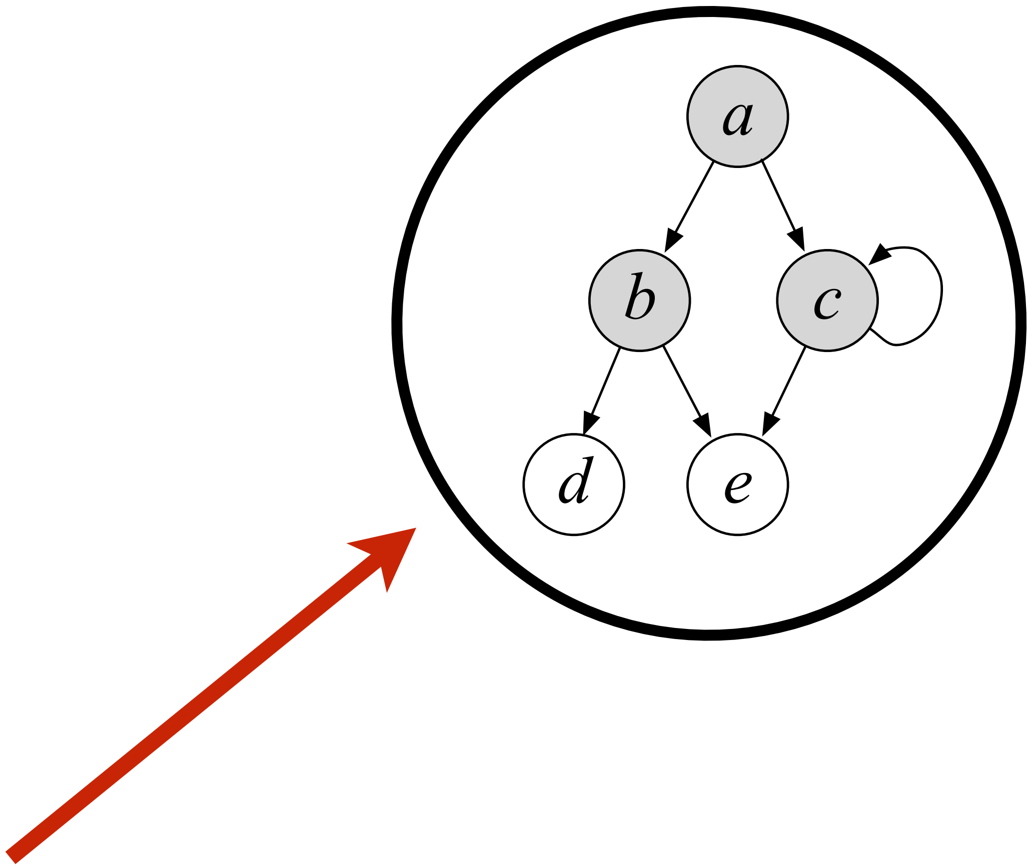
```
letrec span (x : ptr) : bool = {
  if x == null then return false;
  else
    b ← CAS(x->m, 0, 1);
    if b then
      (r_l,r_r) ← (span(x->l) || span(x->r));
      if ¬r_l then x->l := null;
      if ¬r_r then x->r := null;
      return true;
    else return false;
}
```

- All reachable nodes are *marked* by the end
- The graph modified *only* by the commands of span
- The initial call is done from a root node without interference

Key Ingredients

- Subjective Auxiliary State
- State-Transition Systems
- Types

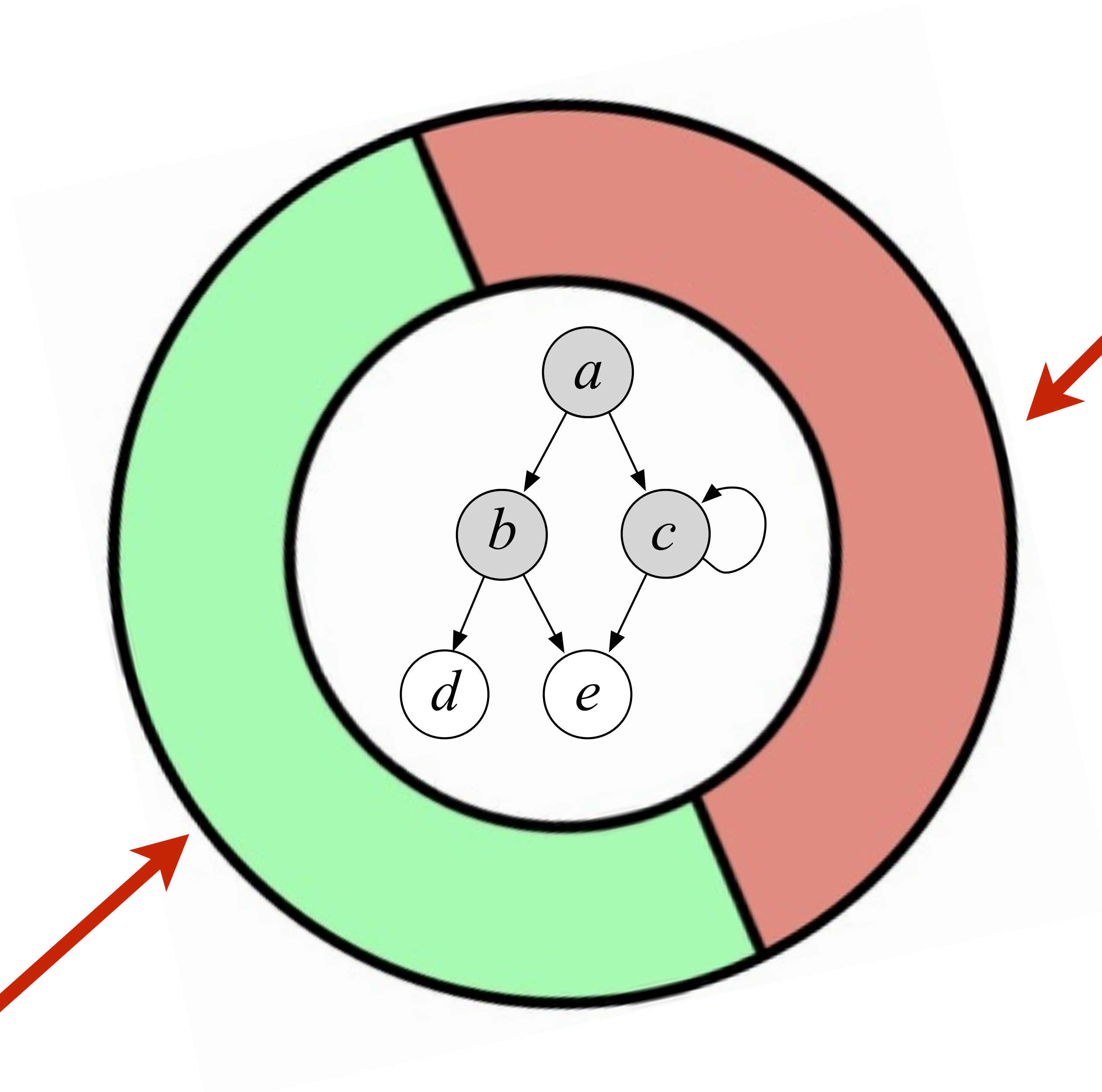
Capturing thread contributions



shared state (heap)

Capturing thread contributions

Auxiliary state
of all *other* threads

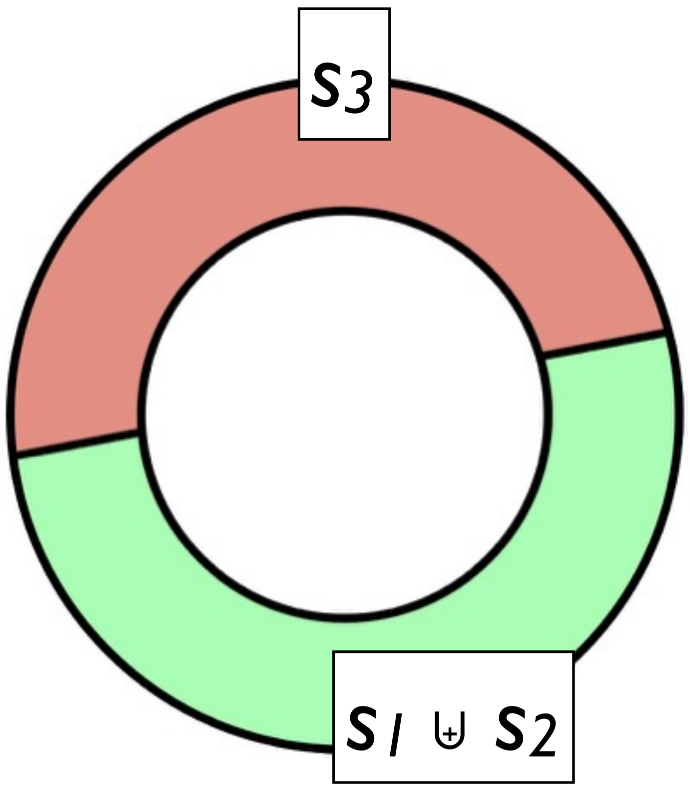


Auxiliary state
of *this* thread

Accounting for dynamic forking

```
span(x)

span(x->l) || span(x->r)
```

Accounting for dynamic forking

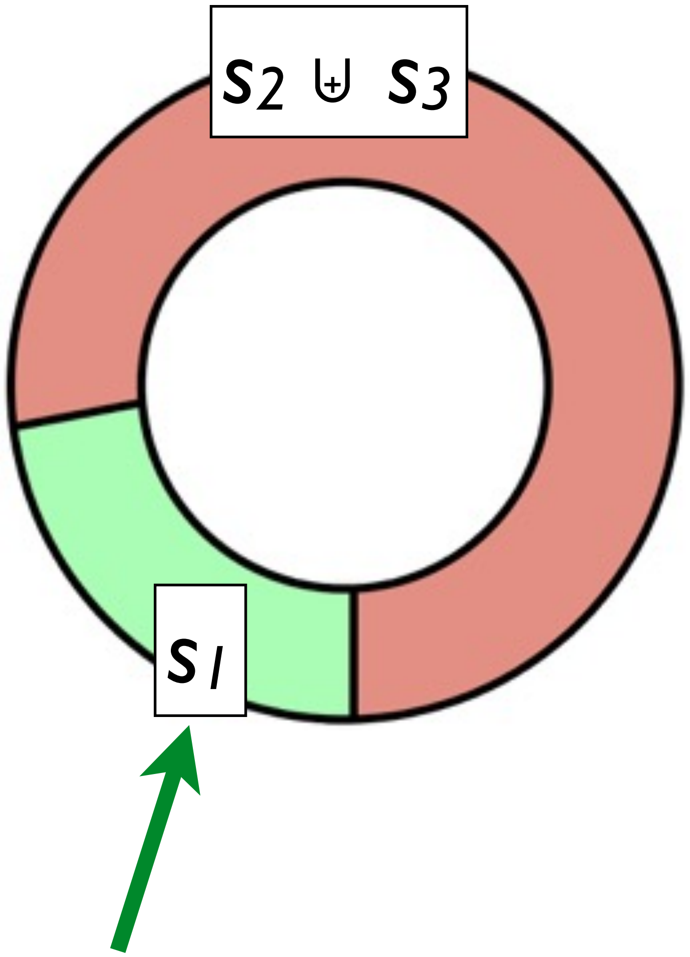


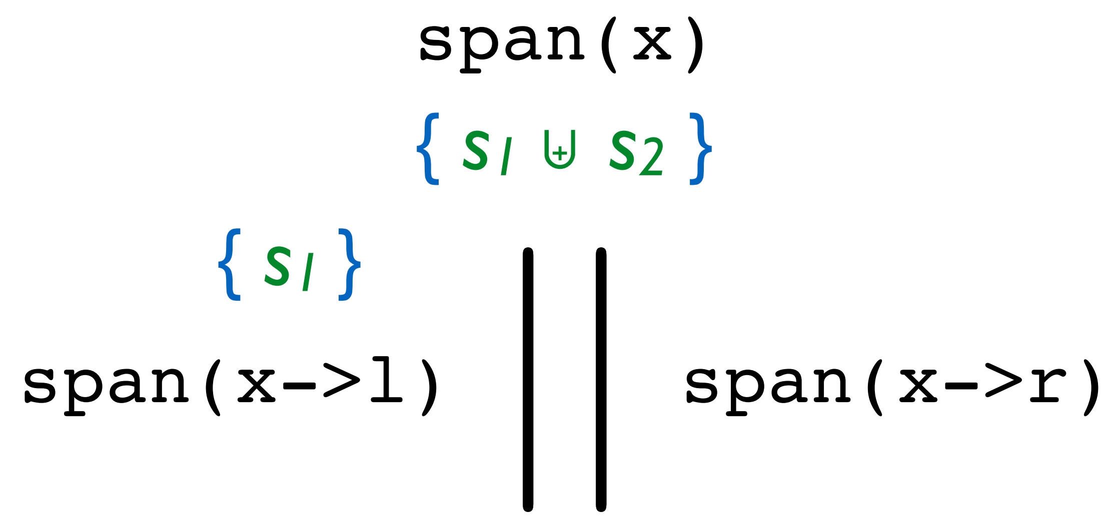
`span(x)`

$\{ s_1 \uplus s_2 \}$

`span(x->l)` || `span(x->r)`

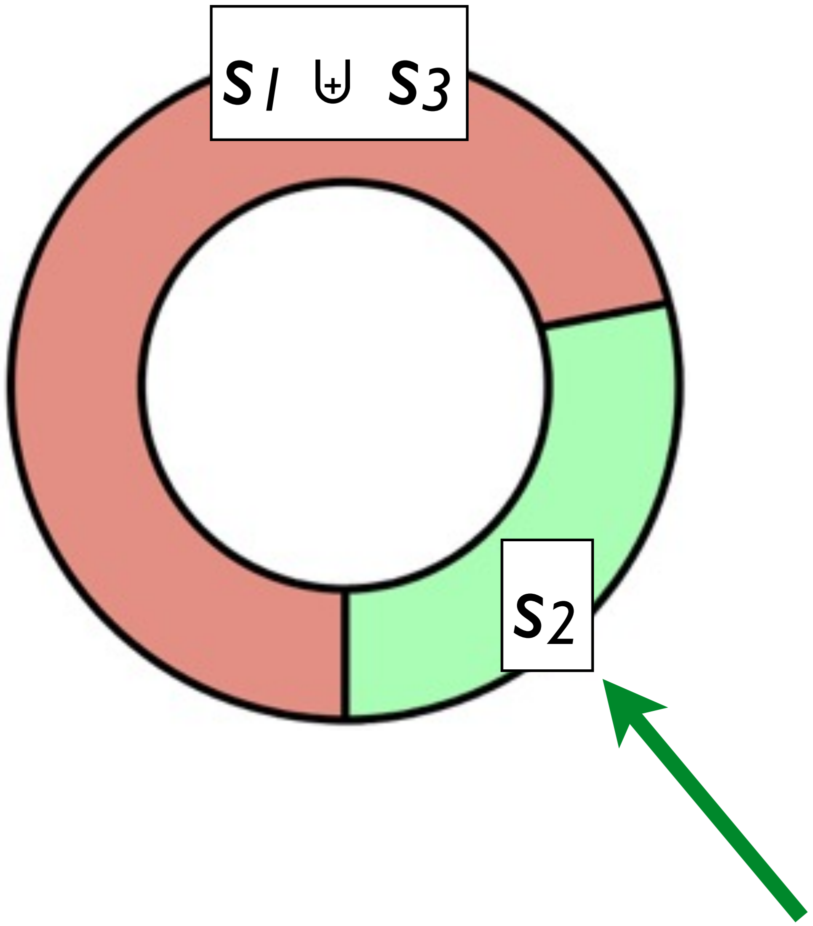
Accounting for dynamic forking





Nodes that belong to `span(x->l)`

Accounting for dynamic forking



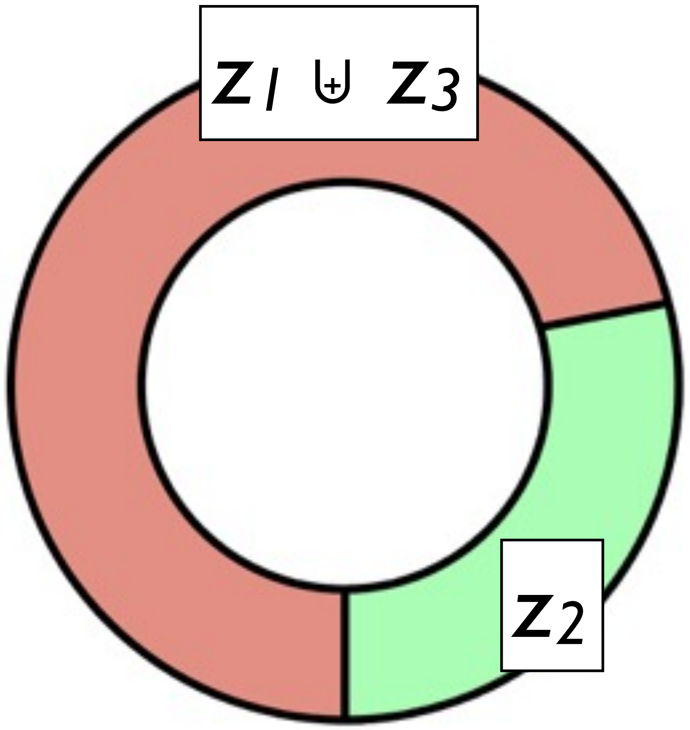
`span(x)`

$\{\, s_1 \uplus s_2 \,\}$

$\{\, s_1 \,\}$ `span(x->l)` || $\{\, s_2 \,\}$ `span(x->r)`

Nodes that belong to **`span(x->r)`**

Accounting for dynamic forking

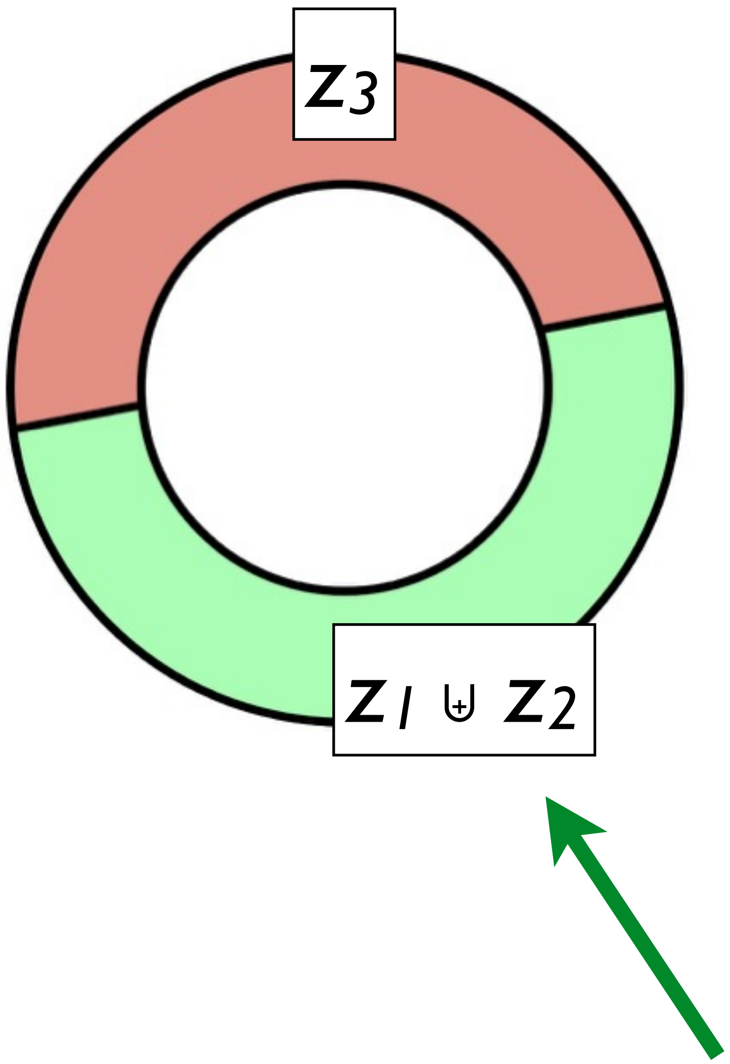


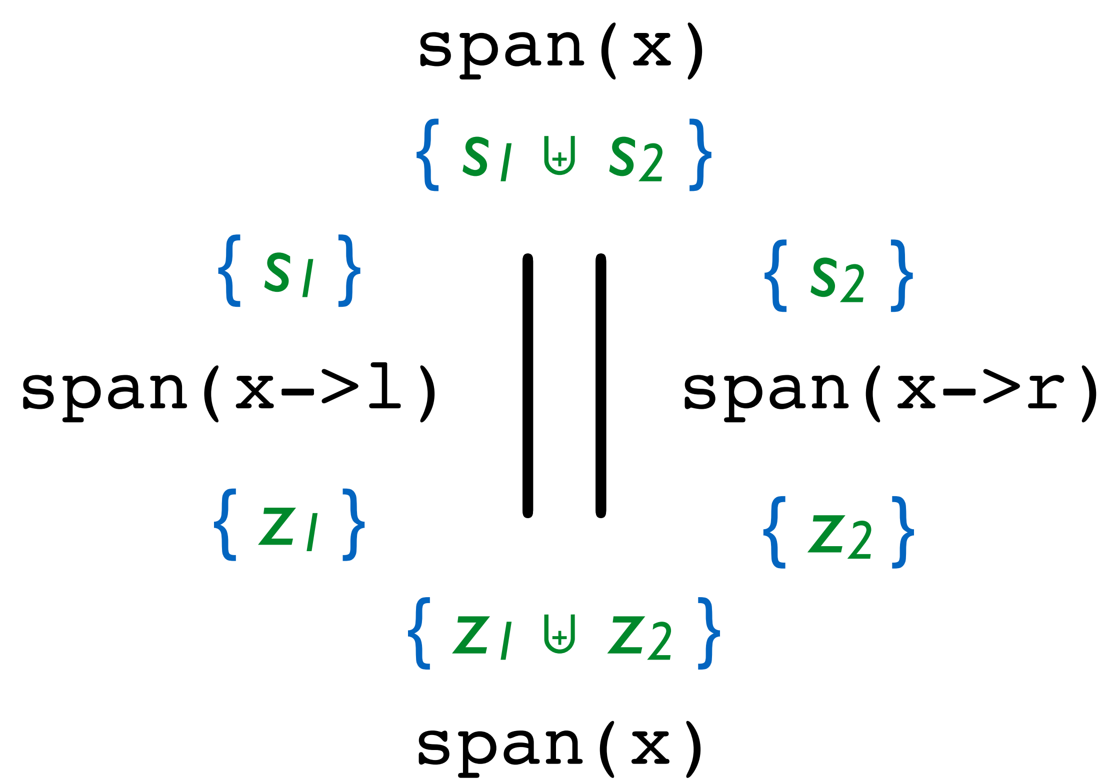
`span(x)`

$\{ s_1 \uplus s_2 \}$

$\{ s_1 \}$		$\{ s_2 \}$
`span(x->l)`	‖	`span(x->r)`
$\{ z_1 \}$		$\{ z_2 \}$

Accounting for dynamic forking





Nodes that belong to **`span(x)`** at the end

Key Ingredients

- Subjective Auxiliary State
- State-Transition Systems
- Types

Key Ingredients

- Subjective Auxiliary State —
  capturing thread-specific contributions
- State-Transition Systems
- Types

Key Ingredients

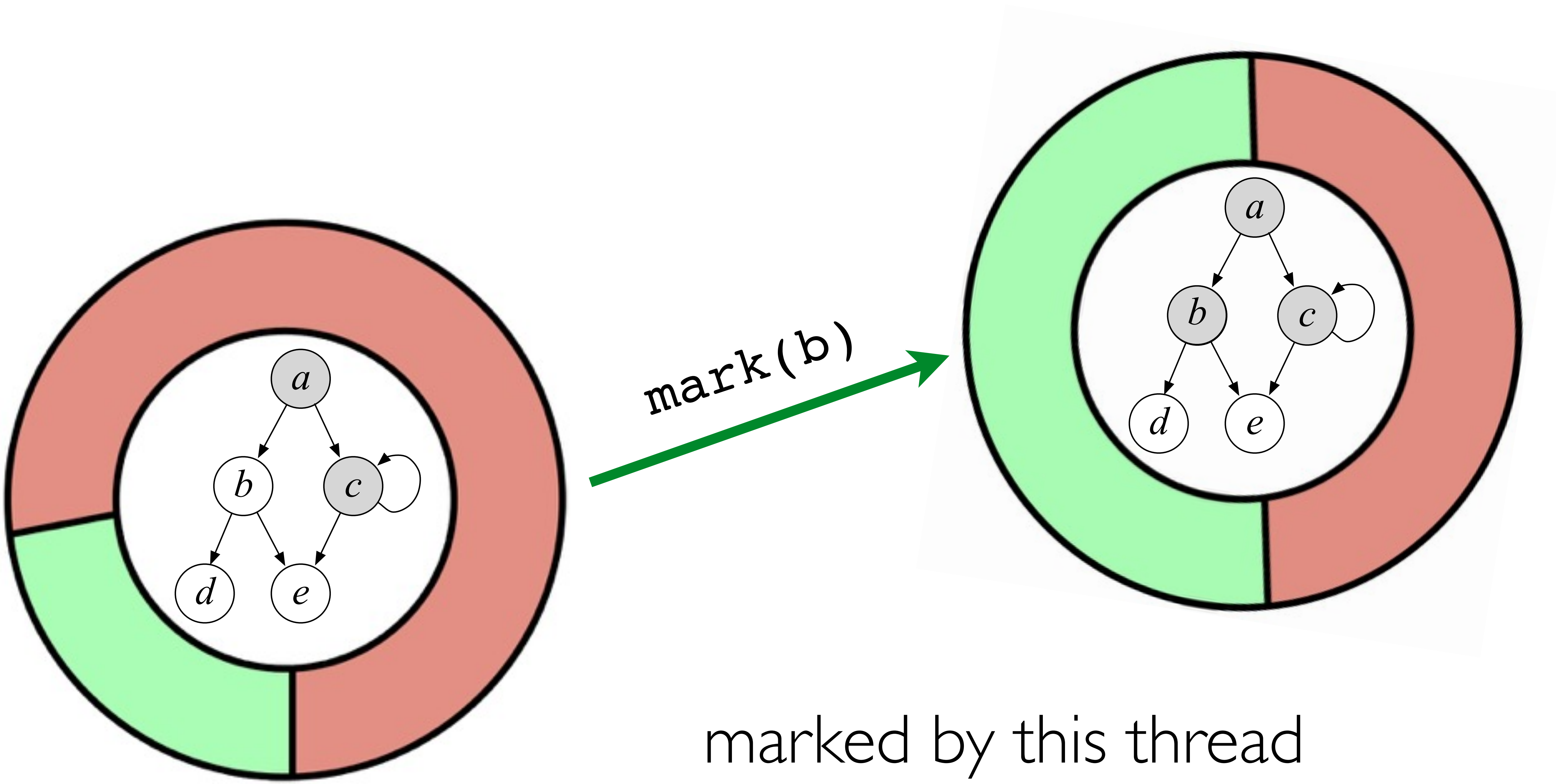
- Subjective Auxiliary State —
  capturing thread-specific contributions
- State-Transition Systems
- Types

Establishing correctness of `span`

```
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;
}
```

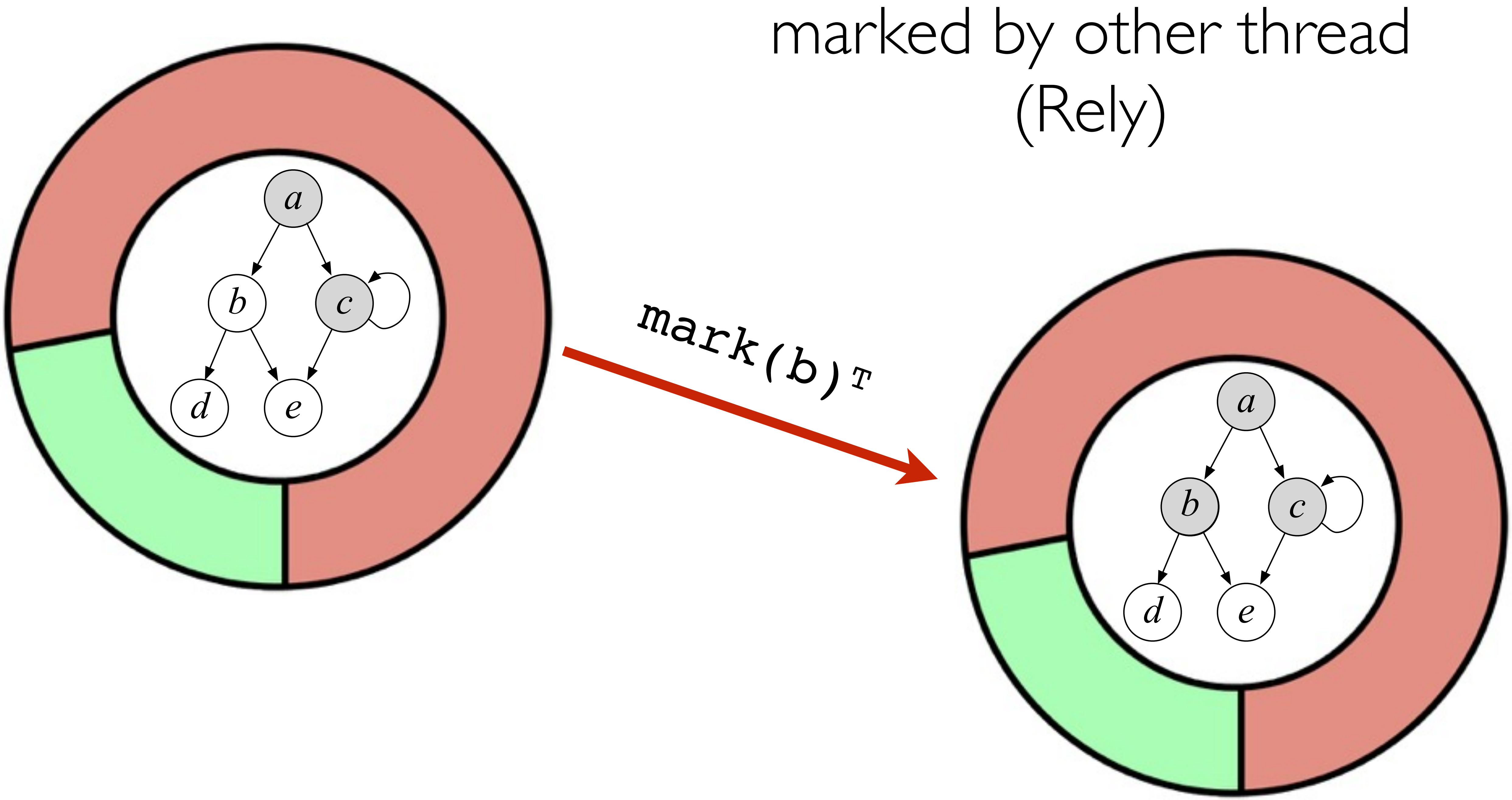
- All reachable nodes are marked by the end
- The graph modified only by the commands of span
- The initial call is done from a root node without interference

Transition 1: marking a node

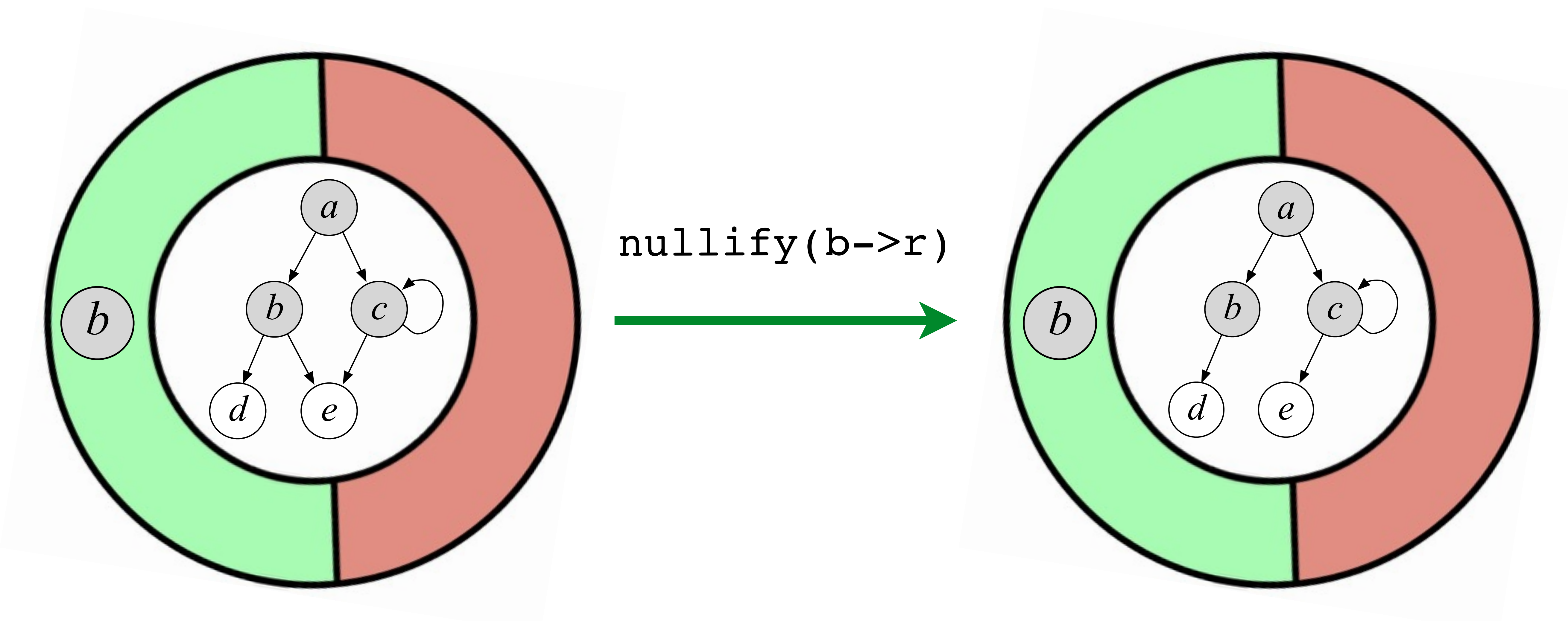


marked by this thread
(Guarantee)

Transition 1: marking a node



Transition 2: pruning an edge



No other thread can do it!

Pseudocode implementation

```
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;
}
```

FCSL/Coq implementation

```
Program Definition span : span_tp :=
 ffix (fun (loop : span_tp) (x : ptr) =>
   Do (if x == null then ret false else
       b <-- trymark x;
       if b then
         xl <-- read_child x Left;
         xr <-- read_child x Right;
         rs <-- par (loop xl) (loop xr);
         (if ~~rs.1 then nullify x Left else ret tt);;
         (if ~~rs.2 then nullify x Right else ret tt);;
         ret true
       else ret false)).
```

Transition-aware commands
(equivalent to **CAS**, **write**, etc.)

Key Ingredients

- Subjective Auxiliary State —
  capturing thread-specific contributions
- State-Transition Systems
- Types

Key Ingredients

- Subjective Auxiliary State —
  capturing thread-specific contributions
- State-Transition Systems —
  specification of concurrent protocols
- Types

Key Ingredients

- Subjective Auxiliary State —
  capturing thread-specific contributions
- State-Transition Systems —
  specification of concurrent protocols
- Types

Key Ingredients

- Subjective Auxiliary State —
  capturing thread-specific contributions
- State-Transition Systems —
  specification of concurrent protocols
- Dependent Types

FCSL/Coq implementation

Specification
(loop invariant)

```
Program Definition span : span_tp :=
 ffix (fun (loop : span_tp) (x : ptr) =>
   Do (if x == null then ret false else
      b <-- trymark x;
      if b then
        xl <-- read_child x Left;
        xr <-- read_child x Right;
        rs <-- par (loop xl) (loop xr);
        (if ~~rs.1 then nullify x Left else ret tt);;
        (if ~~rs.2 then nullify x Right else ret tt);;
        ret true
      else ret false)).
```

Next Obligation. *(about 200 LOC)* **Qed.**

Specification for `span`

starting node

```
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 protocol

```
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).
```

Specification for `span`

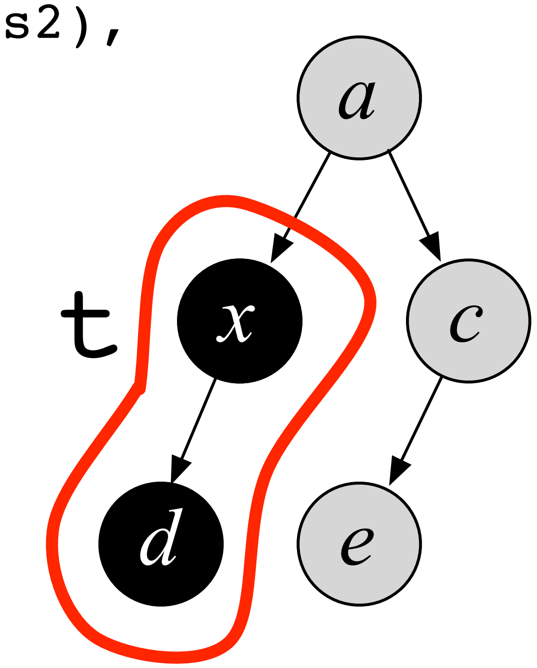
```
Definition span_tp (x : ptr) :=
  {i (g1 : graph (joint i))}, STsep [SpanTree]

    (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).
```

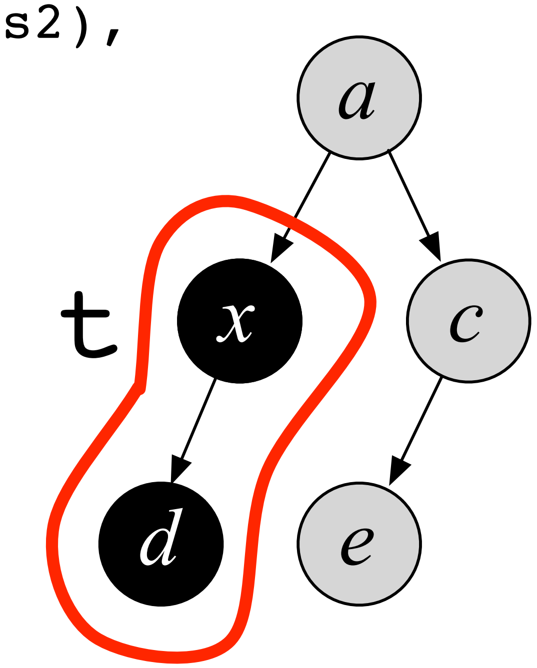
Specification for `span`

```
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).
```



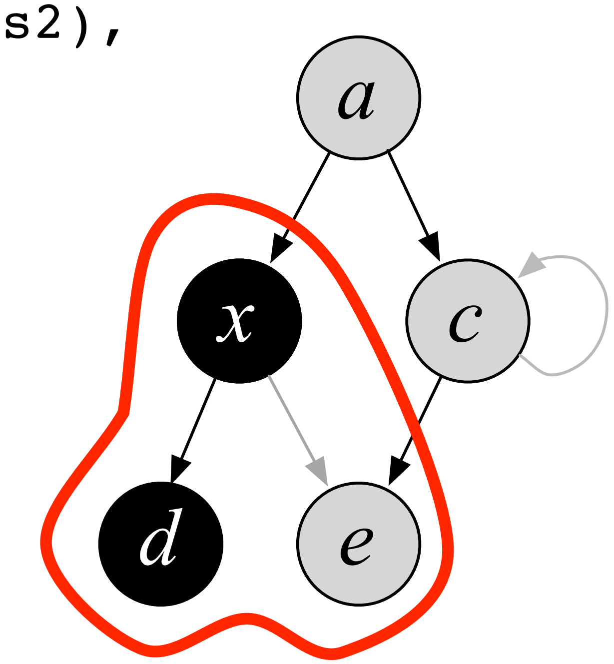
Specification for `span`

```
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).
```



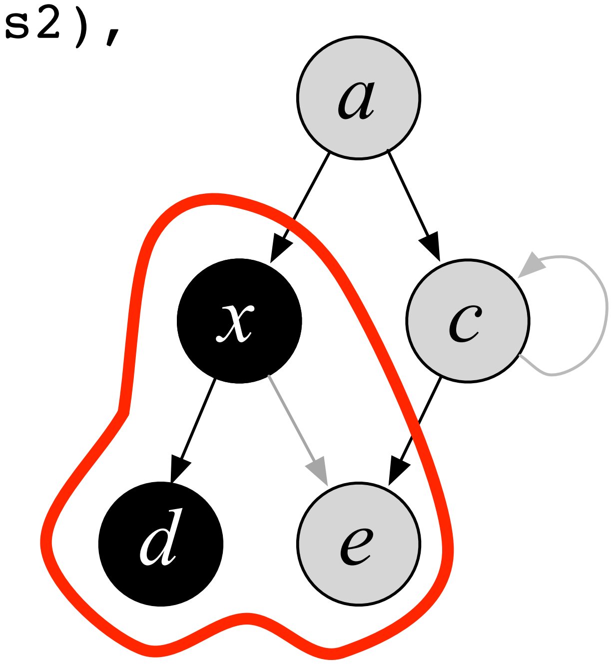
Establishing correctness of `span`

```
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;
}
```

- All reachable nodes are marked by the end
- The graph modified only by the commands of span
- The initial call is done from a root node without interference

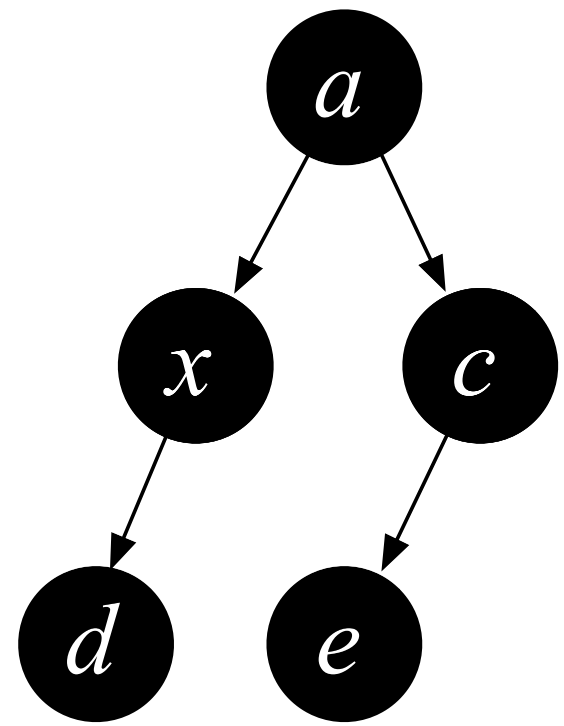
Specification for `span`

```
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)

No interference for the top call



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
```

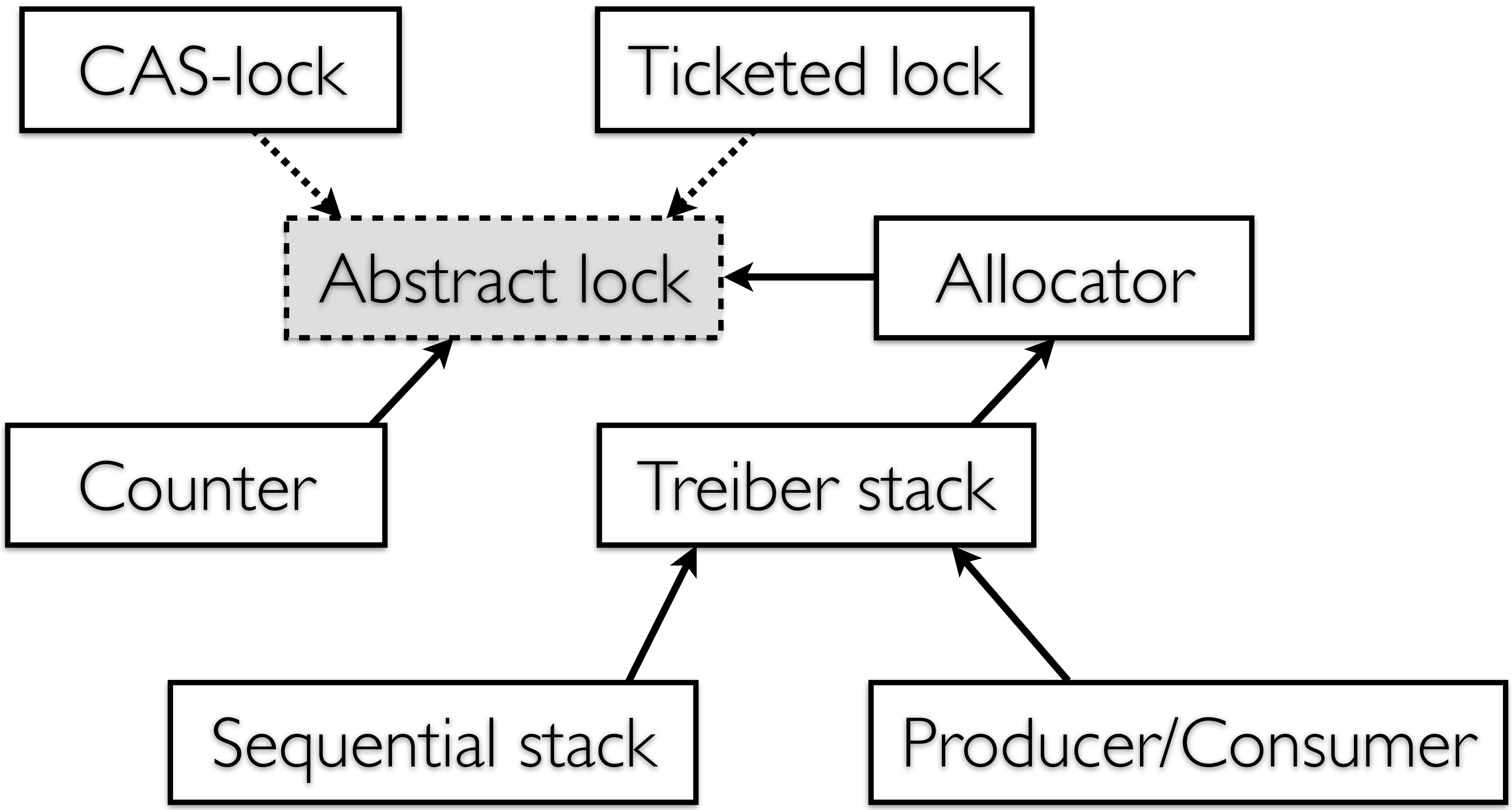
Key Ingredients

- Subjective Auxiliary State —
  capturing thread-specific contributions
- State-Transition Systems —
  specification of concurrent protocols
- Types

Key Ingredients

- Subjective Auxiliary State —
  capturing thread-specific contributions
- State-Transition Systems —
  specification of concurrent protocols
- Types — mechanization

Composing programs and proofs



Key Ingredients

- Subjective Auxiliary State —
  capturing thread-specific contributions

- State-Transition Systems —
  specification of concurrent protocols

- Types — mechanization

Key Ingredients

- **Subjective Auxiliary State** — capturing thread-specific contributions
- **State-Transition Systems** — specification of concurrent protocols
- **Types** — mechanization and compositionality

More in the paper and TR

- Specifying and verifying locks, stacks, snapshots, allocators, higher-order universal constructions and their clients
- Composing concurrent protocols
- Proof layout and reasoning about stability
- Semantic model and embedding into Coq
- Evaluation and proof sizes

To take away

FCSL — an expressive *logic* for FG concurrency, implemented as an *interactive verification tool*.

- **Subjective Auxiliary State** — recording thread-specific contributions;
- **State-Transition Systems** — specification of concurrent protocols;
- **Types** — mechanization and compositionality.

`software.imdea.org/fcsl`

Thanks!

Q&A slides

Some statistics

- Semantics, metatheory, lemmas (~17 KLOC)
- Examples

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
CG increment	26	-	-	-	44	70	8s
CG 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

Don’t require implementing new protocols

Encoding VC in FCSL

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

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

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

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

Proof of `span : span_tp`

```
Next Obligation.
apply: gh=>_ [s1 g1][<- Dx] C1; case: ifP Dx=>/= [/eqP -> _|_ Dx].
- apply: val_ret=>s2 M; case: (menvs_coh M)=>_ /sp_cohG g2; exists g2.
  by split; [apply: subgr_steps M | rewrite (menvs_loc M)].
apply: step; apply: (gh_ex s1); apply: (gh_ex g1); apply: val_do=>//.
case; last first.
- move=>i1 [gi1][Sgi Si Mxi _] Ci1.
  apply: val_ret=>i2 M; case: (menvs_coh M)=>_ /sp_cohG g2; exists g2.
  split; first by apply: subgr_trans Sgi (subgr_steps _ _ M).
  by rewrite -(menvs_loc M) (mark_steps g2 M Mxi).
move=>i1 [gi1][Sgi Si Mxi /(_ (erefl _)) Cti] Ci1.
have Dxi : x \in dom (self i1).
- by move/validL: (cohVSO Ci1); rewrite Si um_domPtUn inE eq_refl => ->.
apply: step; apply: (gh_ex i1); apply: (gh_ex gi1); apply: val_do=>//.
move=>_ i2 [gi2][Sgi2 Si2 ->] Ci2.
apply: step; apply: (gh_ex i2); apply: (gh_ex gi2); apply: val_do.
- by rewrite Si2.
move=>_ i3 [gi3][/(subgr_transT Sgi2) Sgi3 Si3 ->] Ci3.
rewrite (subgrM Sgi2 Dxi); rewrite {Sgi2 gi2 i2 Ci2}Si2 in Si3 *.
apply: step.
have Spl : sself [:: sp_getcoh sp] i3 = self i3 \+ Unit by rewrite unitR.
set i3r := sp ->> [Unit, joint i3, self i3 \+ other i3].
have gi3r : graph (joint i3r) by rewrite getE.
apply: (par_do (r1:=span_post (edgl gi1 x) i3 gi3)
               (r2:=span_post (edgr gi1 x) i3r gi3r) _ Spl)=>//=.
- apply: (gh_ex i3); apply: (gh_ex gi3); apply: val_do=>//.
  - rewrite unitL -(cohE Ci3) -(subgrD Sgi3); split=>//.
    by apply: (@edgeG _ _ x); rewrite inE eq_refl.
- apply: (gh_ex i3r); apply: (gh_ex gi3r); apply: val_do=>// Ci3r.
  rewrite getE -(subgrD Sgi3); split=>//.
  by apply: (@edgeG _ _ x); rewrite !inE eq_refl orbT.
case=>{Spl} [rl rr] i4 gsl gsr Ci4 _ _ Si'
  [gi4][Sg X1][gi4'][Sg'] /=; move: X1.
rewrite /subgraph !getE in gi4 gi4' Sg Sg' *.
rewrite {}/i3r !getE in gi3r Sg' *.
rewrite -{gi3r}(proof_irrelevance gi3 gi3r) in Sg' *.
rewrite -{gi4'}(proof_irrelevance gi4 gi4') in Sg' *.
rewrite -(subgrM Sgi3 Dxi) in Mxi Cti *; rewrite -{}Si3 in Si Dxi.
move: (subgr_transT Sgi Sgi3)=>{Sgi3 i1 gi1 Ci1 Sgi} Sgi.
have Fxr tr u : {subset dom tr <= dom gsr} ->
  front (edge gi3) tr u -> front (edge g1) tr u.
- move=>S; apply: front_mono; first by move=>z; rewrite (subgrD Sgi).
  move=>y /S Dsr; rewrite (subgrN Sgi) // -(sp_markE gi3 y Ci3).
  apply/negP; case: Sg'=>_ _ S' _ _ _ /S'.
  move: (cohVSO Ci4); rewrite Si' -joinA joinCA.
  by case: validUn=>// _ _ /(_ _ Dsr) /negbTE ->.
have Fxl tl u : valid (#x \+ self s1 \+ tl) ->
    {subset dom tl <= dom gsl} ->
    front (edge gi3) tl u -> front (edge g1) tl u.
- move=>V S; apply: front_mono; first by move=>z; rewrite (subgrD Sgi).
  move=>y Dy; rewrite /= (subgrN Sgi) // -(sp_markE gi3 y Ci3) Si.
  rewrite domUn inE -Si (cohVSO Ci3) /= negb_or Si.
  rewrite joinC in V; case: validUn V=>// _ _ /(_ _ Dy) -> _.
  apply/negP; case: Sg=>_ _ O _ _ _ /O.
  move: (cohVSO Ci4); rewrite Si' -joinA.
  by case: validUn (S _ Dy)=>// _ _ N /N /negbTE ->.
have {Sg Sg'} Sgi' : subgraphT gi3 gi4.
- case: Sg Sg'=>D S O M N Ed [_ S' O' _ _ _]; split=>//.
  - by move=>z /S X; rewrite Si' domUn inE -Si'
    (validL (cohVSO Ci4)) X.
  move=>z Dz; have: z \in dom (self i3 \+ other i3).
  - by rewrite domUn inE (cohVSO Ci3) Dz orbT.
  move/(O' z); rewrite domUn inE; case/andP=>_ /orP [|//].
  move/(O z): Dz; rewrite domUn inE; case/andP=>_ /orP [L R|//].
  move: (validL (cohVSO Ci4)); rewrite Si'.
  by case: validUn L=>//_ _ /(_ _ R) /negbTE ->.
case: (Sgi')=>_ S _ E _ _; rewrite -{}E // in Mxi Cti *.
move/S: Dxi=>{S} Dxi /=; rewrite {}Si.
move: (subgr_transT Sgi Sgi')=>{Sgi Sgi'} Sgi.
case: rl; last first.
- case=>Sl Ml X; rewrite {Fxl gsl}Sl -joinA in Si' X *.
  apply: step; apply: (gh_ex i4); apply: (gh_ex gi4).
  apply: (gh_ex (self s1 \+ gsr)).
  apply: val_do=>//; case=>i5 [gi5][Sgi5 Si5 Cti5] Ci5.
  rewrite -Si5 in Si' Dxi.
  case: rr X; last first.
  - case=>Sr Mr; rewrite {gsr}Sr unitR in Si' Fxr Sgi5.
    apply: step; apply: (gh_ex i5); apply: (gh_ex gi5).
    apply: (gh_ex (self s1)); apply: val_do=>//; case=>i6 [gi6][Sgi6 Si6].
    rewrite {}Cti5 => /= Cti' Ci6.
    move/(subgr_trans (meetpp _) Sgi5): Sgi6=>{Sgi5} Sgi6.
    rewrite -{}Si6 {gi5 Ci5} in Si' Si5 Dxi.
    apply: val_ret=>i7 M; case: (menvs_coh M)=>_ Ci7;
    move: (sp_cohG Ci7)=>gi7.
    move: (subgr_trans (meetpT _) Sgi6 (subgr_steps _ gi7 M))=>{Sgi6} Sgi7.
    rewrite -(marked_steps gi6 gi7 M Dxi) in Cti'.
    rewrite (menvs_loc M) in Si5 Si' Dxi.
    exists gi7; split=>//.
    - by apply/subgrX; apply: subgr_trans Sgi Sgi7.
    exists (#x); rewrite joinC.
    have X : edge gi7 x =1 pred0.
    - by move=>z; rewrite inE Cti' inE andbC; case: eqP.
    split=>//; first by [apply: tree0]; first by apply: max0.
    apply: frontPt; last by rewrite domUn inE (cohVSO Ci7) Dxi.
    move=>z; rewrite inE Cti inE; case/and3P=>_ Nz D.
    rewrite (sp_markE _ _ Ci7); apply: subgr_marked Sgi7 _.
    by case/orP: D Nz Ml Mr => /eqP -> /negbTE ->.
  case=>tr [Sr Nr Tr Mr Fr]; rewrite {gsr}Sr unitL in Fxr Fr Sgi5 Si' *.
  rewrite joinCA joinA -(joinA (#x)) -Si' Si5 in Fr.
  move/Fxr: Fr => /(_ (fun x k => k)) {i3 gi3 Ci3 Fxr} Fr.
  apply: step; apply: val_ret=>i6 M;
  apply: val_ret=>i7 /(menvs_trans M)=>{M} M.
  case: (menvs_coh M)=>_ Ci7; move: (sp_cohG Ci7)=>gi7.
  rewrite -(marked_steps gi5 gi7 M Dxi) in Cti5.
  rewrite (menvs_loc M) in Dxi Si' Si5.
  move/validL: (cohVSO Ci7)=>/= V; rewrite Si' in V.
  move: (subgr_trans (meetpT _) Sgi5 (subgr_steps _ gi7 M))=>{Sgi5} Sgi5.
  exists gi7; split=>{i5 gi5 Ci5 M}.
  - apply/subgrX; move/subgrX: Sgi Sgi5; apply: subgr_trans.
    by move=>z; rewrite inE /= domUn inE (validR V) orbC orKb.
  exists (#x \+ tr); rewrite joinCA; move: (subgrD Sgi5) => Di.
  have Ci : {in dom tr, forall y : ptr, contents gi4 y = contents gi7 y}.
  - move=>z Dz /=; rewrite (subgrM Sgi5) // -Si5 Si' !domUn !inE.
    by rewrite domUn inE Dz V (validR V) /= !orbT.
  have E: edge gi7 x =1 pred1 (edgr gi4 x).
  - move=>z /=; rewrite Cti5 inE -Di -(subgrD Sgi).
    by rewrite Dx !(eq_sym z); case: eqP=>//= <-; case: eqP Nr.
  split=>//.
  - by apply: tree1 E (tree_mono Di Ci Tr) (max_mono Di Ci Mr).
  - by apply: max1 E (proj1 Tr) (max_mono Di Ci Mr).
  apply: frontUn; last first.
  - apply: front_leq Fr=>z; rewrite !domUn !inE (cohVSO Ci7) /= Si5.
    by case/andP=>_  /orP [->//|] /(subgrO Sgi5) ->; rewrite orbT.
  apply: frontPt; last by rewrite domUn inE (cohVSO Ci7) Dxi.
  move=>z; rewrite inE Cti inE; case/and3P=>_ Nz D.
  case/orP: D Nz Ml Nr=>/eqP -> /negbTE -> /=.
  - by move/(subgr_marked Sgi5); rewrite (sp_markE _ _ Ci7).
  rewrite domUn inE (cohVSO Ci7) Si' joinA domUn inE -joinA V.
  by rewrite (proj1 Tr) orbT.
case=>tl [Sl Nl Tl Ml Fl]; rewrite {gsl}Sl in Si' Fl Fxl *.
have V : valid (#x \+ self s1 \+ tl \+ gsr).
- by move/validL: (cohVSO Ci4); rewrite Si'.
have S: {subset dom tl <= dom (#x \+ self s1 \+ tl)}.
- by move=>z; rewrite domUn inE (validL V) orbC => ->.
move/(Fxl _ _ (validL V) S): Fl=>{Fxl} Fl X.
apply: step; apply: val_ret=>i5 M.
case: (menvs_coh M)=>_ Ci5; move: (sp_cohG Ci5)=>gi5.
rewrite -!(joinA (#x)) in Si' V Fl.
have Si5: self i4 = self i5 by rewrite (menvs_loc M).
move: (Dxi)=>Dxi'; rewrite Si5 in Si' Dxi'.
move: (subgr_steps gi4 gi5 M)=>{M} Sgi5.
case: rr X; last first.
- case=>Sr Mr; rewrite {gsr Fxr}Sr unitL unitR in V Si' Si5 Fl.
  apply: step; apply: (gh_ex i5); apply: (gh_ex gi5).
  apply: (gh_ex (self s1 \+ tl)); apply: val_do=>//.
  case=>i6 [gi6][Sgi6 Si6 Cti6] Ci6.
  rewrite (subgrM Sgi5) // in Cti6; rewrite -{}Si6 in Si' Si5 Dxi'.
  move/(subgr_trans (meetTp _) Sgi5): Sgi6=>{Sgi5 i5 gi5 Ci5} Sgi5.
  apply: val_ret=>i7 M; case: (menvs_coh M)=>_ Ci7; move: (sp_cohG Ci7)=>gi7.
  rewrite -(marked_steps gi6 gi7 M Dxi') in Cti6.
  rewrite (menvs_loc M) in Si' Si5 Dxi'.
  move: (subgr_trans (meetpT _) Sgi5 (subgr_steps _ gi7 M))=>{Sgi5} Sgi5.
  exists gi7; split.
  - apply/subgrX; move/subgrX: Sgi Sgi5; apply: subgr_trans.
    by move=>z; rewrite inE /= domUn inE (validR V) /= orbC orKb.
  exists (#x \+ tl); rewrite joinCA; move: (subgrD Sgi5)=>Di.
  have Ci : {in dom tl, forall y, contents gi4 y = contents gi7 y}.
  - move=>z Dz; rewrite /= (subgrM Sgi5) // Si5 Si' !domUn !inE.
    by rewrite domUn inE Dz V (validR V) /= !orbT.
  have E : edge gi7 x =i pred1 (edgl gi4 x).
  - move=>z; rewrite /= inE /= -Di Cti6 inE -(subgrD Sgi) Dx /= inE.
    by rewrite !(eq_sym z) orbC; case: eqP=>//= <-; case: eqP Nl.
  split=>//.
  - by apply: tree1 E (tree_mono Di Ci Tl) (max_mono Di Ci Ml).
  - by apply: max1 E (proj1 Tl) (max_mono Di Ci Ml).
  apply: frontUn; last first.
  - apply: front_leq Fl=>z; rewrite joinA !domUn !inE (cohVSO Ci7) /= -Si'.
    by case/andP=>_  /orP [->//|] /(subgrO Sgi5) ->; rewrite orbT.
  apply: frontPt; last by rewrite domUn inE (cohVSO Ci7) Dxi'.
  move=>z; rewrite inE Cti inE; case/and3P=>_ Nz D.
  case/orP: D Nz Mr Nl=>/eqP -> /negbTE -> /=; last first.
  - by move/(subgr_marked Sgi5); rewrite (sp_markE _ _ Ci7).
  rewrite domUn inE (cohVSO Ci7) Si' joinA domUn inE -joinA V.
  by rewrite (proj1 Tl) orbT.
case=>tr [Sr Nr Tr Mr Fr]; rewrite {gsr}Sr unitL in V Fl Fxr Si' Fr.
move/Fxr: Fr=>/(_ (fun x k => k)) {Fxr} Fr.
rewrite -(joinA _ tl) in Si' V.
rewrite (joinA (_ \+ tl)) joinA -(joinA _ tl) in Fl.
rewrite joinCA joinA -(joinA _ tl) -(joinA _ (self _)) in Fr.
have W : valid (tl \+ tr).
- by move: (cohVSO Ci5); rewrite Si'; move/validL/validR/validR.
apply: step; apply: val_ret=>i6 M.
apply: val_ret=>i7 /(menvs_trans M)=>{i6 M} M.
case: (menvs_coh M)=>_ Ci7; move: (sp_cohG Ci7)=>gi7.
move: (subgr_transT Sgi5 (subgr_steps _ gi7 M))=>{Sgi5} Sgi5.
rewrite (menvs_loc M) {i5 gi5 Ci5 M} in Si5 Si' Dxi'.
exists gi7; split.
- by apply/subgrX; apply/subgrX; apply: subgr_trans Sgi Sgi5.
exists (#x \+ (tl \+ tr)); rewrite joinCA; move: (subgrD Sgi5)=>Di.
have [Cil Cir] : {in dom tl, forall y, contents gi4 y = contents gi7 y} /\
                 {in dom tr, forall y, contents gi4 y = contents gi7 y}.
- split=>z Dz /=; rewrite (subgrM Sgi5) //= Si5;
  move/validL: (cohVSO Ci7); rewrite Si' (joinA (#x)) joinC;
  by rewrite domUn inE (domUn tl) inE W Dz => -> //=; rewrite orbT.
have E: edge gi7 x =i pred2 (edgl gi4 x) (edgr gi4 x).
- move=>z /=; rewrite inE /= -Di (subgrM Sgi5) //.
  case: edgeP Nl Nr=>//= _ xl xr _ _ _ _ /negbTE Nl /negbTE Nr.
  by rewrite inE !(eq_sym z); case: eqP=>// <-; rewrite Nl Nr.
split=>//.
- by apply: tree2 E (tree_mono Di Cil Tl) (max_mono Di Cil Ml)
                    (tree_mono Di Cir Tr) (max_mono Di Cir Mr) W.
- by apply: max2 E (proj1 Tl) (max_mono Di Cil Ml)
                   (proj1 Tr) (max_mono Di Cir Mr).
apply: frontUn; last first.
- apply: frontUn; [apply: front_leq Fl | apply: front_leq Fr]=>z;
  rewrite -Si' !domUn !inE (cohVSO Ci7);
  by case/andP=>_  /orP [->//|] /(subgrO Sgi5) ->; rewrite orbT.
apply: frontPt; last by rewrite domUn inE (cohVSO Ci7) Dxi'.
move=>z; rewrite inE Cti inE; case/and3P=>_ _ X.
move: (cohVSO Ci7); rewrite Si' (joinA (#x)) -(joinC (tl \+ tr)).
rewrite -(joinA (tl \+ tr)) domUn inE domUn inE W => -> /=.
by case/orP: X=>/eqP ->; rewrite ?(proj1 Tl) ?(proj1 Tr) ?orbT.
Qed.
```

Proof of `span : span_tp`

```
Next Ob val_ret <- Dx] C1; case: ifP Dx=>/= [/eqP -> _|_ Dx].
- apply val_ret M; case: (menvs_coh M)=>_ /sp_cohG g2; exists g2.
  by s step ply: subgr_steps M | rewrite (menvs_loc val_do
apply: step ly: (gh_ex s1); apply: (gh_ex g1); app val_do .
case; step .
- move step [Sgi Si Mxi _] Ci1.
  apply: val_ret=>i2 M; case: (menvs_coh M)=>_ /sp_cohG g2; exists g2.
  split; first by apply: subgr_trans Sgi (subgr_steps _ _ M).
  by rewrite -(menvs_loc M) (mark_steps g2 M Mxi).
move=>i1 [gi1][Sgi Si Mxi /(_ (erefl _)) Cti] Ci1.
have D step dom (self i1).
- by m step : (cohVSO Ci1); rewrite Si um_domPtUn in val_do ->.
apply: step ly: (gh_ex i1); apply: (gh_ex gi1); appl val_do
move=> step [Sgi2 Si2 ->] Ci2.
apply: step ly: (gh_ex i2); apply: (gh_ex gi2); appl val_do
- by r step .
move=> step [/(subgr_transT Sgi2) Sgi3 Si3 ->] Ci3.
rewrit step Sgi2 Dxi); rewrite {Sgi2 gi2 i2 Ci2}Si2 in Si3 *.
apply: step
have S step [:: sp_getcoh sp] i3 = self i3 \+ Unit by rewrite unitR.
set i3r := sp ->> [Unit, joint i3, self i3 \+ other i3].
have par_do (joint i3r) by rewrite getE.
apply par_do :=span_post (edgl gi1 x) i3
         :=span_post (edgr gi1 x) i3r val_do =>//=.
- apply: par_do ); apply: (gh_ex gi3); apply val_do
  - rewrite unitL -(cohE Ci3) -(subgrD Sgi3);
    by apply: (@edgeG _ _ x); rewrite inE eq_re val_do
- apply: (gh_ex i3r); apply: (gh_ex gi3r); apply val_do Ci3r.
  rewrite getE -(subgrD Sgi3); split=>//.
  by apply: (@edgeG _ _ x); rewrite !inE eq_refl orbT.
case=>{Spl} [rl rr] i4 gsl gsr Ci4 _ _ Si'
  [gi4][Sg X1][gi4'][Sg'] /=; move: X1.
rewrite /subgraph !getE in gi4 gi4' Sg Sg' *.
rewrite {}/i3r !getE in gi3r Sg' *.
rewrite -{gi3r}(proof_irrelevance gi3 gi3r) in Sg' *.
rewrite -{gi4'}(proof_irrelevance gi4 gi4') in Sg' *.
rewrite -(subgrM Sgi3 Dxi) in Mxi Cti *; rewrite -{}Si3 in Si Dxi.
move: (subgr_transT Sgi Sgi3)=>{Sgi3 i1 gi1 Ci1 Sgi} Sgi.
have Fxr tr u : {subset dom tr <= dom gsr} ->
  front (edge gi3) tr u -> front (edge g1) tr u.
- move=>S; apply: front_mono; first by move=>z; rewrite (subgrD Sgi).
  move=>y /S Dsr; rewrite (subgrN Sgi) // -(sp_markE gi3 y Ci3).
  apply/negP; case: Sg'=>_ _ S' _ _ _ /S'.
  move: (cohVSO Ci4); rewrite Si' -joinA joinCA.
  by case: validUn=>// _ _ /(_ _ Dsr) /negbTE ->.
have Fxl tl u : valid (#x \+ self s1 \+ tl) ->
    {subset dom tl <= dom gsl} ->
    front (edge gi3) tl u -> front (edge g1) tl u.
- move=>V S; apply: front_mono; first by move=>z; rewrite (subgrD Sgi).
  move=>y Dy; rewrite /= (subgrN Sgi) // -(sp_markE gi3 y Ci3) Si.
  rewrite domUn inE -Si (cohVSO Ci3) /= negb_or Si.
  rewrite joinC in V; case: validUn V=>// _ _ /(_ _ Dy) -> _.
  apply/negP; case: Sg=>_ _ O _ _ _ /O.
  move: (cohVSO Ci4); rewrite Si' -joinA.
  by case: validUn (S _ Dy)=>// _ _ N /N /negbTE ->.
have {Sg Sg'} Sgi' : subgraphT gi3 gi4.
- case: Sg Sg'=>D S O M N Ed [_ S' O' _ _ _]; split=>//.
  - by move=>z /S X; rewrite Si' domUn inE -Si'
    (validL (cohVSO Ci4)) X.
  move=>z Dz; have: z \in dom (self i3 \+ other i3).
  - by rewrite domUn inE (cohVSO Ci3) Dz orbT.
  move/(O' z); rewrite domUn inE; case/andP=>_ /orP [|//].
  move/(O z): Dz; rewrite domUn inE; case/andP=>_ /orP [L R|//].
  move: (validL (cohVSO Ci4)); rewrite Si'.
  by case: validUn L=>//_ _ /(_ _ R) /negbTE ->.
case: (Sgi')=>_ S _ E _ _; rewrite -{}E // in Mxi Cti *.
move/S: Dxi=>{S} Dxi /=; rewrite {}Si.
move: (subgr_transT Sgi Sgi')=>{Sgi Sgi'} Sgi.
case: rl; last first.
- c step X; rewrite {Fxl gsl}Sl -joinA in Si' X *.
  a step ; apply: (gh_ex i4); apply: (gh_ex gi4).
```

```
  apply val_do f s1 \+ gsr)).
  apply val_do case=>i5 [gi5][Sgi5 Si5 Cti5] Ci5.
  rewri val_do ' Dxi.
  case: rr X; last first.
  - case=> step write {gsr}Sr u val_do xr Sgi5.
    apply: step ply: (gh_ex i5) val_do ex gi5).
    apply: step self s1)); appl val_do ; case=>i6 [gi6][Sgi6 Si6].
    rewrite {}Cti5 => /= Cti' Ci6.
    move/(subgr_trans (meetpp _) Sgi5): Sgi6=>{Sgi5} Sgi6.
    rewrit val_ret i5} in Si' Si5 Dxi.
    apply: val_ret case: (menvs_coh M)=>_ Ci7;
    move: val_ret >gi7.
    move: (subgr_trans (meetpT _) Sgi6 (subgr_steps _ gi7 M))=>{Sgi6} Sgi7.
    rewrite -(marked_steps gi6 gi7 M Dxi) in Cti'.
    rewrite (menvs_loc M) in Si5 Si' Dxi.
    exists gi7; split=>//.
    - by apply/subgrX; apply: subgr_trans Sgi Sgi7.
    exists (#x); rewrite joinC.
    have X : edge gi7 x =1 pred0.
    - by move=>z; rewrite inE Cti' inE andbC; case: eqP.
    split=>//; first by [apply: tree0]; first by apply: max0.
    apply: frontPt; last by rewrite domUn inE (cohVSO Ci7) Dxi.
    move=>z; rewrite inE Cti inE; case/and3P=>_ Nz D.
    rewrite (sp_markE _ _ Ci7); apply: subgr_marked Sgi7 _.
    by case/orP: D Nz Ml Mr => /eqP -> /negbTE ->.
  case=>tr [Sr Nr Tr Mr Fr]; rewrite {gsr}Sr unitL in Fxr Fr Sgi5 Si' *.
  rewrite joinCA joinA -(joinA (#x)) -Si' Si5 in Fr.
  move/F step /( val_ret )) {i3 gi3 Ci3 Fxr} Fr.
  apply: step ply val_ret ;
  apply: step >i7 val_ret M)=>{M} M.
  case: (menvs_coh M)=>_ Ci7; move: (sp_cohG Ci7)=>gi7.
  rewrite -(marked_steps gi5 gi7 M Dxi) in Cti5.
  rewrite (menvs_loc M) in Dxi Si' Si5.
  move/validL: (cohVSO Ci7)=>/= V; rewrite Si' in V.
  move: (subgr_trans (meetpT _) Sgi5 (subgr_steps _ gi7 M))=>{Sgi5} Sgi5.
  exists gi7; split=>{i5 gi5 Ci5 M}.
  - apply/subgrX; move/subgrX: Sgi Sgi5; apply: subgr_trans.
    by move=>z; rewrite inE /= domUn inE (validR V) orbC orKb.
  exists (#x \+ tr); rewrite joinCA; move: (subgrD Sgi5) => Di.
  have Ci : {in dom tr, forall y : ptr, contents gi4 y = contents gi7 y}.
  - move=>z Dz /=; rewrite (subgrM Sgi5) // -Si5 Si' !domUn !inE.
    by rewrite domUn inE Dz V (validR V) /= !orbT.
  have E: edge gi7 x =1 pred1 (edgr gi4 x).
  - move=>z /=; rewrite Cti5 inE -Di -(subgrD Sgi).
    by rewrite Dx !(eq_sym z); case: eqP=>//= <-; case: eqP Nr.
  split=>//.
  - by apply: tree1 E (tree_mono Di Ci Tr) (max_mono Di Ci Mr).
  - by apply: max1 E (proj1 Tr) (max_mono Di Ci Mr).
  apply: frontUn; last first.
  - apply: front_leq Fr=>z; rewrite !domUn !inE (cohVSO Ci7) /= Si5.
    by case/andP=>_  /orP [->//|] /(subgrO Sgi5) ->; rewrite orbT.
  apply: frontPt; last by rewrite domUn inE (cohVSO Ci7) Dxi.
  move=>z; rewrite inE Cti inE; case/and3P=>_ Nz D.
  case/orP: D Nz Ml Nr=>/eqP -> /negbTE -> /=.
  - by move/(subgr_marked Sgi5); rewrite (sp_markE _ _ Ci7).
  rewrite domUn inE (cohVSO Ci7) Si' joinA domUn inE -joinA V.
  by rewrite (proj1 Tr) orbT.
case=>tl [Sl Nl Tl Ml Fl]; rewrite {gsl}Sl in Si' Fl Fxl *.
have V : valid (#x \+ self s1 \+ tl \+ gsr).
- by move/validL: (cohVSO Ci4); rewrite Si'.
have S: {subset dom tl <= dom (#x \+ self s1 \+ tl)}.
- by move=>z; rewrite domUn inE (validL V) orbC => ->.
move/ step vali val_ret Fxl} Fl X.
apply step pply: val_ret
case: step h M) val_ret (sp_cohG Ci5)=>gi5.
rewrite -!(joinA (#x)) in Si' V Fl.
have Si5: self i4 = self i5 by rewrite (menvs_loc M).
move: (Dxi)=>Dxi'; rewrite Si5 in Si' Dxi'.
move: (subgr_steps gi4 gi5 M)=>{M} Sgi5.
case: rr X; last first.
- case=>Sr Mr; rewrite {gsr Fxr}Sr unitL unitR in V Si' Si5 Fl.
```

```
  apply step ply: (gh_ex i5); apply val_do .
  apply step self s1 \+ tl)); apply
  case=>i6 [gi6][Sgi6 Si6 Cti6] Ci6.
  rewrite (subgrM Sgi5) // in Cti6; rewrite -{}Si6 in Si' Si5 Dxi'.
  move/ val_ret eetTp _) Sgi5): Sgi6=>{Sgi5 i5 gi5 Ci5} Sgi5.
  apply val_ret ; case: (menvs_coh M)=>_ Ci7; move: (sp_cohG Ci7)=>gi7.
  rewri val_ret ps gi6 gi7 M Dxi') in Cti6.
  rewrite (menvs_loc M) in Si' Si5 Dxi'.
  move: (subgr_trans (meetpT _) Sgi5 (subgr_steps _ gi7 M))=>{Sgi5} Sgi5.
  exists gi7; split.
  - apply/subgrX; move/subgrX: Sgi Sgi5; apply: subgr_trans.
    by move=>z; rewrite inE /= domUn inE (validR V) /= orbC orKb.
  exists (#x \+ tl); rewrite joinCA; move: (subgrD Sgi5)=>Di.
  have Ci : {in dom tl, forall y, contents gi4 y = contents gi7 y}.
  - move=>z Dz; rewrite /= (subgrM Sgi5) // Si5 Si' !domUn !inE.
    by rewrite domUn inE Dz V (validR V) /= !orbT.
  have E : edge gi7 x =i pred1 (edgl gi4 x).
  - move=>z; rewrite /= inE /= -Di Cti6 inE -(subgrD Sgi) Dx /= inE.
    by rewrite !(eq_sym z) orbC; case: eqP=>//= <-; case: eqP Nl.
  split=>//.
  - by apply: tree1 E (tree_mono Di Ci Tl) (max_mono Di Ci Ml).
  - by apply: max1 E (proj1 Tl) (max_mono Di Ci Ml).
  apply: frontUn; last first.
  - apply: front_leq Fl=>z; rewrite joinA !domUn !inE (cohVSO Ci7) /= -Si'.
    by case/andP=>_  /orP [->//|] /(subgrO Sgi5) ->; rewrite orbT.
  apply: frontPt; last by rewrite domUn inE (cohVSO Ci7) Dxi'.
  move=>z; rewrite inE Cti inE; case/and3P=>_ Nz D.
  case/orP: D Nz Ml Nr=>/eqP -> /negbTE -> /=; last first.
  - by move/(subgr_marked Sgi5); rewrite (sp_markE _ _ Ci7).
  rewrite domUn inE (cohVSO Ci7) Si' joinA domUn inE -joinA V.
  by rewrite (proj1 Tl) orbT.
case=>tr [Sr Nr Tr Mr Fr]; rewrite {gsr}Sr unitL in V Fl Fxr Si' Fr.
move/Fxr: Fr=>/(_ (fun x k => k)) {Fxr} Fr.
rewrite -(joinA _ tl) in Si' V.
rewrite (joinA (_ \+ tl)) joinA -(joinA _ tl) in Fl.
rewrite joinCA joinA -(joinA _ tl) -(joinA _ (self _)) in Fr.
have W : valid (tl \+ tr).
- by step hVSO val_ret Si'; move/validL/validR/validR.
apply step pply
apply step >i7 val_ret M)=>{i6 M} M.
case: (menvs_coh M)=>_ Ci7; move: (sp_cohG Ci7)=>gi7.
move: (subgr_transT Sgi5 (subgr_steps _ gi7 M))=>{Sgi5} Sgi5.
rewrite (menvs_loc M) {i5 gi5 Ci5 M} in Si5 Si' Dxi'.
exists gi7; split.
- by apply/subgrX; apply/subgrX; apply: subgr_trans Sgi Sgi5.
exists (#x \+ (tl \+ tr)); rewrite joinCA; move: (subgrD Sgi5)=>Di.
have [Cil Cir] : {in dom tl, forall y, contents gi4 y = contents gi7 y} /\
                 {in dom tr, forall y, contents gi4 y = contents gi7 y}.
- split=>z Dz /=; rewrite (subgrM Sgi5) //= Si5;
  move/validL: (cohVSO Ci7); rewrite Si' (joinA (#x)) joinC;
  by rewrite domUn inE (domUn tl) inE W Dz => -> //=; rewrite orbT.
have E: edge gi7 x =i pred2 (edgl gi4 x) (edgr gi4 x).
- move=>z /=; rewrite inE /= -Di (subgrM Sgi5) //.
  case: edgeP Nl Nr=>//= _ xl xr _ _ _ _ /negbTE Nl /negbTE Nr.
  by rewrite inE !(eq_sym z); case: eqP=>// <-; rewrite Nl Nr.
split=>//.
- by apply: tree2 E (tree_mono Di Cil Tl) (max_mono Di Cil Ml)
                  (tree_mono Di Cir Tr) (max_mono Di Cir Mr) W.
- by apply: max2 E (proj1 Tl) (max_mono Di Cil Ml)
                   (proj1 Tr) (max_mono Di Cir Mr).
apply: frontUn; last first.
- apply: frontUn; [apply: front_leq Fl | apply: front_leq Fr]=>z;
  rewrite -Si' !domUn !inE (cohVSO Ci7);
  by case/andP=>_  /orP [->//|] /(subgrO Sgi5) ->; rewrite orbT.
apply: frontPt; last by rewrite domUn inE (cohVSO Ci7) Dxi'.
move=>z; rewrite inE Cti inE; case/and3P=>_ _ X.
move: (cohVSO Ci7); rewrite Si' (joinA (#x)) -(joinC (tl \+ tr)).
rewrite -(joinA (tl \+ tr)) domUn inE domUn inE W => -> -> /=.
by case/orP: X=>/eqP ->; rewrite ?(proj1 Tl) ?(proj1 Tr) ?orbT.
Qed.
```

Proof of span : span_tp

```
Next Ob val_ret <- Dx] C1; case: ifP Dx=>/= [/eqP -> _|_ Dx].
apply:
- appl M; case: (menvs_coh M)=>_ /sp_cohG g2; exists g2.
  by s           ly: subgr_steps M | rewrite (menvs_loc  val_do
apply: step ly: (gh_ex s1); apply: (gh_ex g1); app       .
case;
- move       [Sgi Si Mxi _] Ci1.
  apply: val_ret=>i2 M; case: (menvs_coh M)=>_ /sp_cohG g2; exists g2.
  split; first by apply: subgr_trans Sgi (subgr_steps _ _ M).
  by rewrite -(menvs_loc M) (mark_steps g2 M Mxi).
move=>i1 [gi1][Sgi Si Mxi /(_ (erefl _)) Cti] Ci1.
have D       dom (self i1).
- by m step : (cohVSO Ci1); rewrite Si um_domPtUn in val_do >.
apply:       ly: (gh_ex i1); apply: (gh_ex gi1); appl
move=>       [Sgi2 Si2 ->] Ci2.
apply: step  ly: (gh_ex i2); apply: (gh_ex gi2); appl val_do
- by r
move=>       [/(subgr_transT Sgi2) Sgi3 Si3 ->] Ci3.
rewrit       Sgi2 Dxi); rewrite {Sgi2 gi2 i2 Ci2}Si2 in Si3 *.
apply: step
have S        [:: sp_getcoh sp] i3 = self i3 \+ Unit by rewrite unitR.
set i3r := sp ->> [Unit, joint i3, self i3 \+ other i3].
have          (joint i3r) by rewrite getE.
apply         :=span_post (edgl gi1 x) i3
  par_do      :=span_post (edgr gi1 x) i3r  val_do  =>//=.
- apply       ); apply: (gh_ex gi3); apply
  - rewrite unitL -(cohE Ci3) -(subgrD Sgi3);
    by apply: (@edgeG _ _ x); rewrite inE eq_re
- apply: (gh_ex i3r); apply: (gh_ex gi3r); appl val_do Ci3r.
  rewrite getE -(subgrD Sgi3); split=>//.
  by apply: (@edgeG _ _ x); rewrite !inE eq_refl orbT.
case=>{Spl} [rl rr] i4 gsl gsr Ci4 _ _ Si'
  [gi4][Sg X1][gi4'][Sg'] /=; move: X1.
rewrite /subgraph !getE in gi4 gi4' Sg Sg' *.
rewrite {}/i3r !getE in gi3r Sg' *.
rewrite -{gi3r}(proof_irrelevance gi3 gi3r) in Sg' *.
rewrite -{gi4'}(proof_irrelevance gi4 gi4') in Sg' *.
rewrite -(subgrM Sgi3 Dxi) in Mxi Cti *; rewrite -{}Si3 in Si Dxi.
move: (subgr_transT Sgi Sgi3)=>{Sgi3 i1 gi1 Ci1 Sgi} Sgi.
have Fxr tr u : {subset dom tr <= dom gsr} ->
  front (edge gi3) tr u -> front (edge g1) tr u.
- move=>S; apply: front_mono; first by move=>z; rewrite (subgrD Sgi).
  move=>y /S Dsr; rewrite (subgrN Sgi) // -(sp_markE gi3 y Ci3).
  apply/negP; case: Sg'=>_ _ S' _ _ _ /S'.
  move: (cohVSO Ci4); rewrite Si' -joinA joinCA.
  by case: validUn=>// _ _ _ /(_ _ Dsr) /negbTE ->.
have Fxl tl u : valid (#x \+ self s1 \+ tl) ->
    {subset dom tl <= dom gsl} ->
    front (edge gi3) tl u -> front (edge g1) tl u.
- move=>V S; apply: front_mono; first by move=>z; rewrite (subgrD Sgi).
  move=>y Dy; rewrite /= (subgrN Sgi) // -(sp_markE gi3 y Ci3) Si.
  rewrite domUn inE -Si (cohVSO Ci3) Dz orbT.
  rewrite joinC in V; case: validUn V=>// _ _ /(_ _ Dy) -> _.
  apply/negP; case: Sg=>_ _ O _ _ _ /O.
  move: (cohVSO Ci4); rewrite Si' -joinA.
  by case: validUn (S _ Dy)=>// _ _ N /N /negbTE ->.
have {Sg Sg'} Sgi' : subgraphT gi3 gi4.
- case: Sg Sg'=>D S O M N Ed [_ S' O' _ _ _]; split=>//.
  - by move=>z /S X; rewrite Si' domUn inE -Si'
    (validL (cohVSO Ci4)) X.
  move=>z Dz; have: z \in dom (self i3 \+ other i3).
  - by rewrite domUn inE (cohVSO Ci3) Dz orbT.
  move/(O' z); rewrite domUn inE; case/andP=>_ /orP [|//].
  move/(O z): Dz; rewrite domUn inE; case/andP=>_ /orP [L R|//].
  move: (validL (cohVSO Ci4)); rewrite Si'.
  by case: validUn L=>//_ _ /(_ _ R) /negbTE ->.
case: (Sgi')=>_ S _ E _ _; rewrite -{}E // in Mxi Cti *.
move/S: Dxi=>{S} Dxi /=; rewrite {}Si.
move: (subgr_transT Sgi Sgi')=>{Sgi Sgi'} Sgi.
case: rl; last first.
-      X; rewrite {Fxl gsl}Sl -joinA in Si' X *.
  step ; apply: (gh_ex i4); apply: (gh_ex gi4).
```

```
apply     val_do  f s1 \+ gsr)).
apply             case=>i5 [gi5][Sgi5 Si5 Cti5] Ci5.
rewri             ' Dxi.
case: rr X; last first.
- case=>          write {gsr}Sr u  val_do  xr Sgi5.
  apply step ply: (gh_ex i5)              ex gi5).
  apply           self s1)); appl         ; case=>i6 [gi6][Sgi6 Si6].
  rewrite {}Cti5 => /= Cti' Ci6.
  move/(subgr_trans (meetpp _) Sgi
  rewrit             i5} in Si'
  apply val_ret    case: (me
  move:              gi7.
  move: (subgr_trans (meetpT _) Sg
  rewrite -(marked_steps gi6 gi7 M Dxi) in Cti'.
  rewrite (menvs_loc M) in Si5 Si' Dxi.
  exists gi7; split=>//.
  - by apply/subgrX; apply: subgr_trans Sgi Sgi7.
  exists (#x); rewrite joinC.
  have X : edge gi7 x =1 pred0.
  - by move=>z; rewrite inE Cti' inE andbC; case: eqP.
  split=>//; first by [apply: tree0]; first by apply: max0.
  apply: frontPt; last by rewrite domUn inE (cohVSO Ci7) Dxi.
  move=>z; rewrite inE Cti inE; case/and3P=>_ Nz D.
  rewrite (sp_markE _ _ Ci7); apply: subgr_marked Sgi7 _.
  by case/orP: D Nz Ml Mr => /eqP -> /negbTE ->.
case=>tr [Sr Nr Tr Mr Fr]; rewrite {gsr}Sr unitL in Fxr Fr Sgi5 Si' *.
rewrite joinCA joinA -(joinA (#x)
move/F               /(
apply step ply val_ret
apply             i7
case: (menvs_coh M)=>_ Ci7; move:
rewrite -(marked_steps gi5 gi7 M Dxi) in Cti5.
rewrite (menvs_loc M) in Dxi Si' Si5.
move/validL: (cohVSO Ci7)=>/= V; rewrite Si' in V.
move: (subgr_trans (meetpT _) Sgi5 (subgr_steps _ gi7 M))=>{Sgi5} Sgi5.
exists gi7; split=>{i5 gi5 Ci5 M}.
- apply/subgrX; move/subgrX: Sgi Sgi5; apply: subgr_trans.
  by move=>z; rewrite inE /= domUn inE (validR V) orbC orKb.
exists (#x \+ tr); rewrite joinCA; move: (subgrD Sgi5) => Di.
have Ci : {in dom tr, forall y : ptr, contents gi4 y = contents gi7 y}.
- move=>z Dz /=; rewrite (subgrM Sgi5) // -Si5 Si' !domUn !inE.
  by rewrite domUn inE Dz V (validR V) /= !orbT.
have E: edge gi7 x =1 pred1 (edgr gi4 x).
- move=>z /=; rewrite Cti5 inE -Di -(subgrD Sgi).
  by rewrite Dx !(eq_sym z); case: eqP=>//= <-; case: eqP Nr.
split=>//.
- by apply: tree1 E (tree_mono Di Ci Tr) (max_mono Di Ci Mr).
- by apply: max1 E (proj1 Tr) (max_mono Di Ci Mr).
apply: frontUn; last first.
- apply: front_leq Fr=>z; rewrite !domUn !inE (cohVSO Ci7) /= Si5.
  by case/andP=>_ /orP [->//|] /(subgrO Sgi5) ->; rewrite orbT.
apply: frontPt; last by rewrite domUn inE (cohVSO Ci7) Dxi.
move=>z; rewrite inE Cti inE; case/and3P=>_ Nz D.
case/orP: D Nz Ml Nr=>/eqP -> /negbTE -> /=.
- by move/(subgr_marked Sgi5); rewrite (sp_markE _ _ Ci7).
rewrite domUn inE (cohVSO Ci7) Si' joinA domUn inE -joinA V.
by rewrite (proj1 Tr) orbT.
case=>tl [Sl Nl Tl Ml Fl]; rewrite {gsl}Sl in Si' Fl Fxl *.
have V : valid (#x \+ self s1 \+ tl \+ gsr).
- by move/validL: (cohVSO Ci4); rewrite Si'.
have S: {subset dom tl <= dom (#x \+ self s1 \+ tl)}.
- by move=>z; rewrite domUn inE (validL V) orbC => ->.
move/          vali                    Fxl} Fl X.
apply step    ply: val_ret
case:          h M)                    (sp_cohG Ci5)=>gi5.
rewrite -!(joinA (#x)) in Si' V Fl.
have Si5: self i4 = self i5 by rewrite (menvs_loc M).
move: (Dxi)=>Dxi'; rewrite Si5 in Si' Dxi'.
move: (subgr_steps gi4 gi5 M)=>{M} Sgi5.
case: rr X; last first.
- case=>Sr Mr; rewrite {gsr Fxr}Sr unitL unitR in V Si' Si5 Fl.
```

```
apply step ply: (gh_ex i5); apply val_do .
apply          self s1 \+ tl)); appl
case=>i6 [gi6][Sgi6 Si6 Cti6] Ci6.
rewrite (subgrM Sgi5) // in Cti6; r
move/                                etTp _) Sgi5):
apply val_ret ; case: (menvs_
rewri                                 s gi6 gi7 M Dx
rewrite (menvs_loc M) in Si' Si5 Dx
move: (subgr_trans (meetpT _) Sgi5 (subgr_steps _ gi7 M))=>{Sgi5} Sgi5.
exists gi7; split.
- apply/subgrX; move/subgrX: Sgi Sgi5; apply: subgr_trans.
  by move=>z; rewrite inE /= domUn inE (validR V) /= orbC orKb.
exists (#x \+ tl); rewrite joinCA; move: (subgrD Sgi5)=>Di.
have Ci : {in dom tl, forall y, contents gi4 y = contents gi7 y}.
- move=>z Dz; rewrite /= (subgrM Sgi5) // Si5 Si' !domUn !inE.
  by rewrite domUn inE Dz V (validR V) /= !orbT.
have E : edge gi7 x =i pred1 (edgl gi4 x).
- move=>z; rewrite /= inE /= -Di Cti6 inE -(subgrD Sgi) Dx /= inE.
  by rewrite !(eq_sym z) orbC; case: eqP=>//= <-; case: eqP Nl.
split=>//.
- by apply: tree1 E (tree_mono Di Ci Tl) (max_mono Di Ci Ml).
- by apply: max1 E (proj1 Tl) (max_mono Di Ci Ml).
apply: frontUn; last first.
- apply: front_leq Fl=>z; rewrite joinA !domUn !inE (cohVSO Ci7) /= -Si'.
  by case/andP=>_  /orP [->//|] /(subgrO Sgi5) ->; rewrite orbT.
apply: frontPt; last by rewrite domUn inE (cohVSO Ci7) Dxi'.
move=>z; rewrite inE Cti inE; case/and3P=>_ Nz D.
case/orP: D Nz Mr Nl=>/eqP -> /negbTE -> /=; last first.
- by move/(subgr_marked Sgi5); rewrite (sp_markE _ _ Ci7).
rewrite domUn inE (cohVSO Ci7) Si' joinA domUn inE -joinA V.
by rewrite (proj1 Tl) orbT.
case=>tr [Sr Nr Tr Mr Fr]; rewrite {gsr}Sr unitL in V Fl Fxr Si' Fr.
move/Fxr: Fr=>/(_ (fun x k => k)) {Fxr} Fr.
rewrite -(joinA _ tl) in Si' V.
rewrite (joinA (_ \+ tl)) joinA -(joinA _ tl) in Fl.
rewrite joinCA joinA -(joinA _ tl) -(joinA _ (self _)) in Fr.
have W : valid (tl \+ tr).
- by step hVSO val_ret Si'
appl      pply
appl      =>i7               M)=>
case: (menvs_coh M)=>_ Ci7; move: (sp
move: (subgr_transT Sgi5 (subgr_steps _ gi7 M))=>{Sgi5} Sgi5.
rewrite (menvs_loc M) {i5 gi5 Ci5 M} in Si5 Si' Dxi'.
exists gi7; split.
- by apply/subgrX; apply/subgrX; apply: subgr_trans Sgi Sgi5.
exists (#x \+ (tl \+ tr)); rewrite joinCA; move: (subgrD Sgi5)=>Di.
have [Cil Cir] : {in dom tl, forall y, contents gi4 y = contents gi7 y} /\
                 {in dom tr, forall y, contents gi4 y = contents gi7 y}.
- split=>z Dz /=; rewrite (subgrM Sgi5) //= Si5;
  move/validL: (cohVSO Ci7); rewrite Si' (joinA (#x)) joinC;
  by rewrite domUn inE (domUn tl) inE W Dz => -> //=; rewrite orbT.
have E: edge gi7 x =i pred2 (edgl gi4 x) (edgr gi4 x).
- move=>z /=; rewrite inE /= -Di (subgrM Sgi5) //.
  case: edgeP Nl Nr=>//= _ xl xr _ _ _ _ _ /negbTE Nl /negbTE Nr.
  by rewrite inE !(eq_sym z); case: eqP=>// <-; rewrite Nl Nr.
split=>//.
- by apply: tree2 E (tree_mono Di Cil Tl) (max_mono Di Cil Ml)
                    (tree_mono Di Cir Tr) (max_mono Di Cir Mr) W.
- by apply: max2 E (proj1 Tl) (max_mono Di Cil Ml)
                   (proj1 Tr) (max_mono Di Cir Mr).
apply: frontUn; last first.
- apply: frontUn; [apply: front_leq Fl | apply: front_leq Fr]=>z;
  rewrite -Si' !domUn !inE (cohVSO Ci7);
  by case/andP=>_  /orP [->//|] /(subgrO Sgi5) ->; rewrite orbT.
apply: frontPt; last by rewrite domUn inE (cohVSO Ci7) Dxi'.
move=>z; rewrite inE Cti inE; case/and3P=>_ _ X.
move: (cohVSO Ci7); rewrite Si' (joinA (#x)) -(joinC (tl \+ tr)).
rewrite -(joinA (tl \+ tr)) domUn inE domUn inE W => -> /=.
by case/orP: X=>/eqP ->; rewrite ?(proj1 Tl) ?(proj1 Tr) ?orbT.
Qed.
```

case (false, false)

case (true, false)

case (false, true)

case (true, true)

Proof of `span : span_tp`

```
Next Ol val_ret
apply: val_ret <- Dx] C1; case: ifP Dx=>/= [/eqP -> _|_ Dx].
- apply M; case: (menvs_coh M)=>_ /sp_cohG g2; exists g2.
  by s ... ply: subgr_steps M | rewrite (menvs_loc val_do
apply: step ly: (gh_ex s1); apply: (gh_ex g1); app val_do .
case; step .
- move [Sgi Si Mxi _] Ci1.
  apply: val_ret=>i2 M; case: (menvs_coh M)=>_ /sp_cohG g2; exists g2.
  split; first by apply: subgr_trans Sgi (subgr_steps _ _ M).
  by rewrite -(menvs_loc M) (mark_steps g2 M Mxi).
move=>i1 [gi1][Sgi Si Mxi /(_ (erefl _)) Cti] Ci1.
have D step dom (self i1).
- by m step : (cohVSO Ci1); rewrite Si um_domPtUn in val_do ->.
apply: step ly: (gh_ex i1); apply: (gh_ex gi1); appl val_do
move=> [Sgi2 Si2 ->] Ci2.
apply: step ly: (gh_ex i2); apply: (gh_ex gi2); appl val_do
- by r step .
move=> [/(subgr_transT Sgi2) Sgi3 Si3 ->] Ci3.
rewrit Sgi2 Dxi); rewrite {Sgi2 gi2 i2 Ci2}Si2 in Si3 *.
apply: step
have S [:: sp_getcoh sp] i3 = self i3 \+ Unit by rewrite unitR.
set i3r := sp ->> [Unit, joint i3, self i3 \+ other i3].
have (joint i3r) by rewrite getE.
apply par_do :=span_post (edgl gi1 x) i3
             :=span_post (edgr gi1 x) i3r val_do =>//=.
- apply ); apply: (gh_ex gi3); apply val_do
  - rewrite unitL -(cohE Ci3) -(subgrD Sgi3);
    by apply: (@edgeG _ _ x); rewrite inE eq_re
- apply: (gh_ex i3r); apply: (gh_ex gi3r); apply val_do Ci3r.
  rewrite getE -(subgrD Sgi3); split=>//.
  by apply: (@edgeG _ _ x); rewrite !inE eq_refl orbT.
case=>{Spl} [rl rr] i4 gsl gsr Ci4 _ _ Si'
  [gi4][Sg X1][gi4'][Sg'] /=; move: X1.
rewrite /subgraph !getE in gi4 gi4' Sg Sg' *.
rewrite {}/i3r !getE in gi3r Sg' *.
rewrite -{gi3r}(proof_irrelevance gi3 gi3r) in Sg' *.
rewrite -{gi4'}(proof_irrelevance gi4 gi4') in Sg' *.
rewrite -(subgrM Sgi3 Dxi) in Mxi Cti *; rewrite -{}Si3 in Si Dxi.
move: (subgr_transT Sgi Sgi3)=>{Sgi3 i1 gi1 Ci1 Sgi} Sgi.
have Fxr tr u : {subset dom tr <= dom gsr} ->
  front (edge gi3) tr u -> front (edge g1) tr u.
- move=>S; apply: front_mono; first by move=>z; rewrite (subgrD Sgi).
  move=>y /S Dsr; rewrite (subgrN Sgi) // -(sp_markE gi3 y Ci3).
```

graph-related stuff

```
                                                          Sgi).
  move=>y Dy; rewrite /= (subgrN Sgi) // -(sp_markE gi3 y Ci3) Si.
  rewrite domUn inE -Si (cohVSO Ci3) /= negb_or Si.
  rewrite joinC in V; case: validUn V=>// _ _ /(_ _ Dy) -> _.
  apply/negP; case: Sg=>_ _ O _ _ _ /O.
  move: (cohVSO Ci4); rewrite Si' -joinA.
  by case: validUn (S _ Dy)=>// _ _ N /N /negbTE ->.
have {Sg Sg'} Sgi' : subgraphT gi3 gi4.
- case: Sg Sg'=>D S O M N Ed [_ S' O' _ _ _]; split=>//.
  - by move=>z /S X; rewrite Si' domUn inE -Si'
    (validL (cohVSO Ci4)) X.
  move=>z Dz; have: z \in dom (self i3 \+ other i3).
  - by rewrite domUn inE (cohVSO Ci3) Dz orbT.
  move/(O' z); rewrite domUn inE; case/andP=>_ /orP [|//].
  move/(O z): Dz; rewrite domUn inE; case/andP=>_ /orP [L R|//].
  move: (validL (cohVSO Ci4)); rewrite Si'.
  by case: validUn L=>//_ _ /(_ _ R) /negbTE ->.
case: (Sgi')=>_ S _ E _ _; rewrite -{}E // in Mxi Cti *.
move/S: Dxi=>{S} Dxi /=; rewrite {}Si.
move: (subgr_transT Sgi Sgi')=>{Sgi Sgi'} Sgi.
case: rl; last first.
- c X; rewrite {Fxl gsl}Sl -joinA in Si' X *.
  a step ; apply: (gh_ex i4); apply: (gh_ex gi4).
```

```
apply val_do f s1 \+ gsr)).
apply val_do case=>i5 [gi5][Sgi5 Si5 Cti5] Ci5.
rewri ' Dxi.
case: rr X; last first.
- case=> step write {gsr}Sr u val_do xr Sgi5.
  apply: step ply: (gh_ex i5) val_do ex gi5).
  apply: self s1)); appl ; case=>i6 [gi6][Sgi6 Si6].
  rewrite {}Cti5 => /= Cti' Ci6.
  move/(subgr_trans (meetpp _) Sgi
  rewrit val_ret i5} in Si'
  apply: val_ret case: (men
  move: gi7.
  move: (subgr_trans (meetpT _) Sg
  rewrite -(marked_steps gi6 gi7 M Dxi) in Cti'.
  rewrite (menvs_loc M) in Si5 Si' Dxi.
  exists gi7; split=>//.
```

case (false, false)

graph-related stuff

```
  rewrite (sp_markE _ _ Ci7); apply: subgr_marked Sgi7 _.
  by case/orP: D Nz Ml Mr => /eqP -> /negbTE ->.
case=>tr [Sr Nr Tr Mr Fr]; rewrite {gsr}Sr unitL in Fxr Fr Sgi5 Si' *.
rewrite joinCA joinA -(joinA (#x))
move/F /(
apply: step ply val_ret
apply: i7
case: (menvs_coh M)=>_ Ci7; move:
rewrite -(marked_steps gi5 gi7 M Dxi) in Cti5.
rewrite (menvs_loc M) in Dxi Si' Si5.
move/validL: (cohVSO Ci7)=>/= V; rewrite Si' in V.
move: (subgr_trans (meetpT _) Sgi5 (subgr_steps _ gi7 M))=>{Sgi5} Sgi5.
exists gi7; split=>{i5 gi5 Ci5 M}.
- apply/subgrX; move/subgrX: Sgi Sgi5; apply: subgr_trans.
  by move=>z; rewrite inE /= domUn inE (validR V) orbC orKb.
exists (#x \+ tr); rewrite joinCA; move: (subgrD Sgi5) => Di.
                                                        i7 y}.
```

case (true, false)

graph-related stuff

```
- by apply: tree1 E (tree_mono Di Ci Tr) (max_mono Di Ci Mr).
- by apply: max1 E (proj1 Tr) (max_mono Di Ci Mr).
apply: frontUn; last first.
- apply: front_leq Fr=>z; rewrite !domUn !inE (cohVSO Ci7) /= Si5.
  by case/andP=>_ /orP [->//|] /(subgrO Sgi5) ->; rewrite orbT.
apply: frontPt; last by rewrite domUn inE (cohVSO Ci7) Dxi.
move=>z; rewrite inE Cti inE; case/and3P=>_ Nz D.
case/orP: D Nz Ml Nr=>/eqP -> /negbTE -> /=.
- by move/(subgr_marked Sgi5); rewrite (sp_markE _ _ Ci7).
rewrite domUn inE (cohVSO Ci7) Si' joinA domUn inE -joinA V.
by rewrite (proj1 Tr) orbT.
case=>tl [Sl Nl Tl Ml Fl]; rewrite {gsl}Sl in Si' Fl Fxl *.
have V : valid (#x \+ self s1 \+ tl \+ gsr).
- by move/validL: (cohVSO Ci4); rewrite Si'.
have S: {subset dom tl <= dom (#x \+ self s1 \+ tl)}.
- by move=>z; rewrite domUn inE (validL V) orbC => ->.
move/ step vali val_ret Fxl} Fl X.
apply pply:
case: h M) (sp_cohG Ci5)=>gi5.
rewrite -!(joinA (#x)) in Si' V Fl.
have Si5: self i4 = self i5 by rewrite (menvs_loc M).
move: (Dxi)=>Dxi'; rewrite Si5 in Si' Dxi'.
move: (subgr_steps gi4 gi5 M)=>{M} Sgi5.
case: rr X; last first.
- case=>Sr Mr; rewrite {gsr Fxr}Sr unitL unitR in V Si' Si5 Fl.
```

```
apply step ply: (gh_ex i5); apply val_do .
apply  self s1 \+ tl)); apply
case=>i6 [gi6][Sgi6 Si6 Cti6] Ci6.
rewrite (subgrM Sgi5) // in Cti6; r
move/ eetTp _) Sgi5):
apply val_ret ; case: (menvs_
rewri os gi6 gi7 M Dx
rewrite (menvs_loc M) in Si' Si5 Dx
move: (subgr_trans (meetpT _) Sgi5 (subgr_steps _ gi7 M))=>{Sgi5} Sgi5.
exists gi7; split.
- apply/subgrX; move/subgrX: Sgi Sgi5; apply: subgr_trans.
  by move=>z; rewrite inE /= domUn inE (validR V) /= orbC orKb.
exists (#x \+ tl); rewrite joinCA; move: (subgrD Sgi5)=>Di.
have Ci : {in dom tl, forall y, contents gi4 y = contents gi7 y}.
- move=>z Dz; rewrite /= (subgrM Sgi5) // Si5 Si' !domUn !inE.
  by rewrite domUn inE Dz V (validR V) /= !orbT.
```

case (false, true)

graph-related stuff

```
- apply: front_leq Fl=>z; rewrite joinA !domUn !inE (cohVSO Ci7) /= -Si'.
  by case/andP=>_ /orP [->//|] /(subgrO Sgi5) ->; rewrite orbT.
apply: frontPt; last by rewrite domUn inE (cohVSO Ci7) Dxi'.
move=>z; rewrite inE Cti inE; case/and3P=>_ Nz D.
case/orP: D Nz Ml Mr=> /eqP -> /negbTE -> /=; last first.
- by move/(subgr_marked Sgi5); rewrite (sp_markE _ _ Ci7).
rewrite domUn inE (cohVSO Ci7) Si' joinA domUn inE -joinA V.
by rewrite (proj1 Tl) orbT.
case=>tr [Sr Nr Tr Mr Fr]; rewrite {gsr}Sr unitL in V Fl Fxr Si' Fr.
move/Fxr: Fr=>/(_ (fun x k => k)) {Fxr} Fr.
rewrite -(joinA _ tl) in Si' V.
rewrite (joinA (_ \+ tl)) joinA -(joinA _ tl) in Fl.
rewrite joinCA joinA -(joinA _ tl) -(joinA _ (self _)) in Fr.
have W : valid (tl \+ tr).
- by step nVSO val_ret Si'.
apply pply
apply =>i7 M)=>
case: (menvs_coh M)=>_ Ci7; move: (sp
move: (subgr_transT Sgi5 (subgr_steps _ gi7 M))=>{Sgi5} Sgi5.
rewrite (menvs_loc M) {i5 gi5 Ci5 M} in Si5 Si' Dxi'.
exists gi7; split.
- by apply/subgrX; apply/subgrX; apply: subgr_trans Sgi Sgi5.
exists (#x \+ (tl \+ tr)); rewrite joinCA; move: (subgrD Sgi5)=>Di.
have [Cil Cir] : {in dom tl, forall y, contents gi4 y = contents gi7 y} /\
                 {in dom tr, forall y, contents gi4 y = contents gi7 y}.
- split=>z Dz; rewrite (subgrM Sgi5) //= Si5;
  move/validL: (cohVSO Ci7); rewrite Si' (joinA (#x)) joinC;
  by rewrite domUn inE (domUn tl) inE W Dz => -> //=; rewrite orbT.
have E: edge gi7 x =i pred2 (edgl gi4 x) (edgr gi4 x).
```

case (true, true)

graph-related stuff

```
            (proj1 Tr) (max_mono Di Cir Mr).
apply: frontUn; last first.
- apply: frontUn; [apply: front_leq Fl | apply: front_leq Fr]=>z;
  rewrite -Si' !domUn !inE (cohVSO Ci7);
  by case/andP=>_ /orP [->//|] /(subgrO Sgi5) ->; rewrite orbT.
apply: frontPt; last by rewrite domUn inE (cohVSO Ci7) Dxi'.
move=>z; rewrite inE Cti inE; case/and3P=>_ _ X.
move: (cohVSO Ci7); rewrite Si' (joinA (#x)) -(joinC (tl \+ tr)).
rewrite -(joinA (tl \+ tr)) domUn inE domUn inE W => -> /=.
by case/orP: X=>/eqP ->; rewrite ?(proj1 Tl) ?(proj1 Tr) ?orbT.
Qed.
```

Future work

- Implement program extraction
  (will require to have proofs of actions' "operationality");
- Adopt Coq 8.5 universe polymorphism to support higher-order heaps;
- Work out better abstractions for proving stability.