FCSL

something more

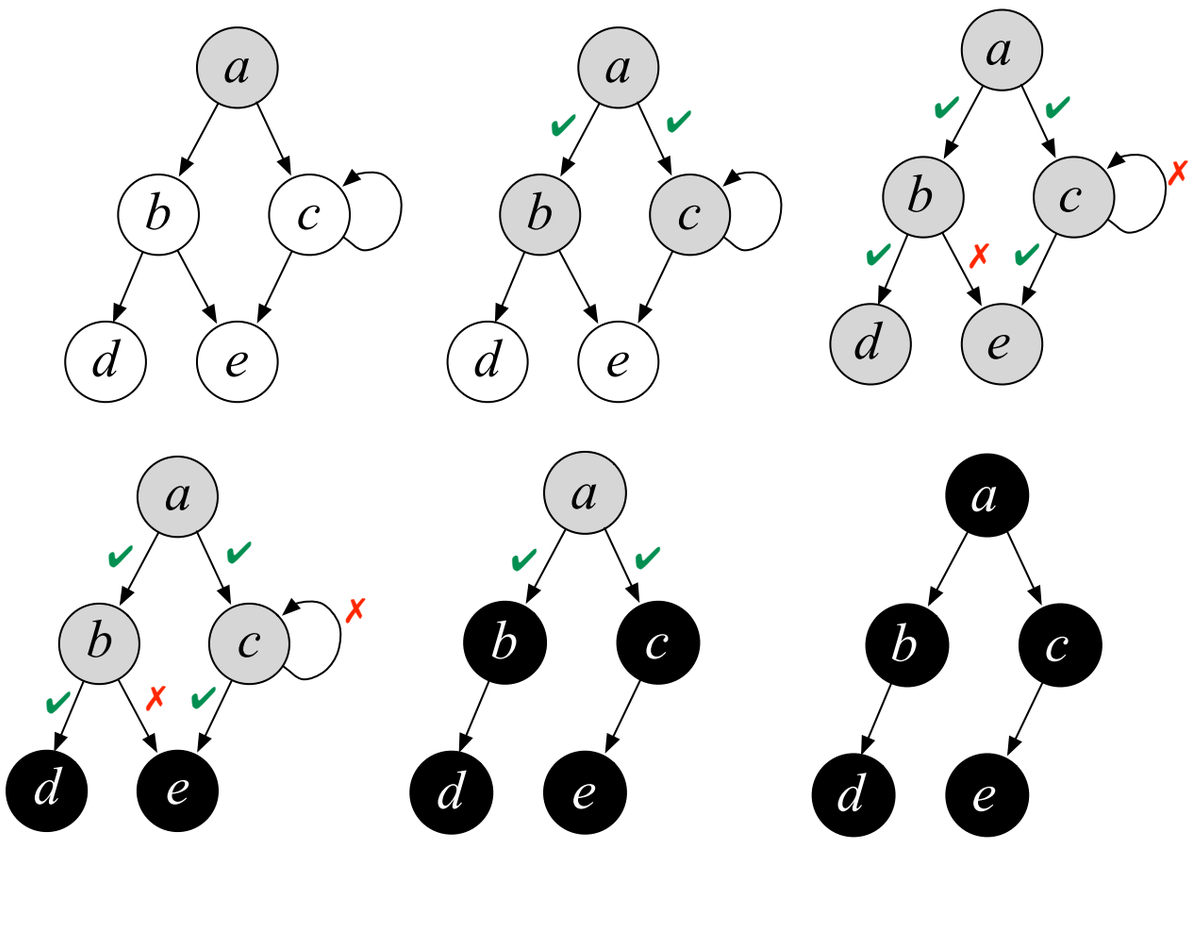
Previously on this channel…

Previously on this channel…

```
letrec span (x : ptr) : bool = {
  if x == null then val_ret 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;
      val_ret true;
    else val_ret false;
}
```

Previously on this channel…

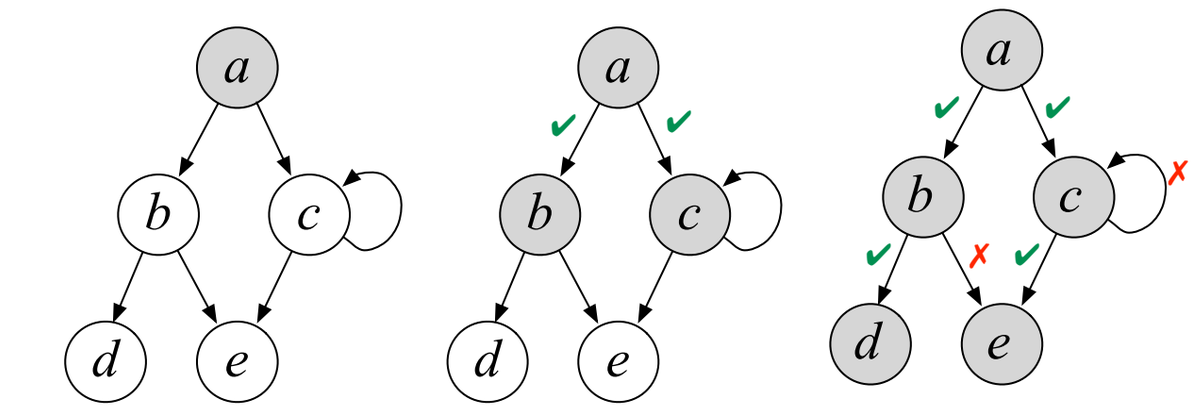
```
letrec span (x : ptr) : bool = {
  if x == null then val_ret false;
  else
    b ← CAS(x->m, 0, 1);
    if b then
      (r_l,r_r) ← (span(x->l) || span(x->r));
      if ¬r_l then x->l := null;
      if ¬r_r then x->r := null;
      val_ret true;
    else val_ret false;
}
```

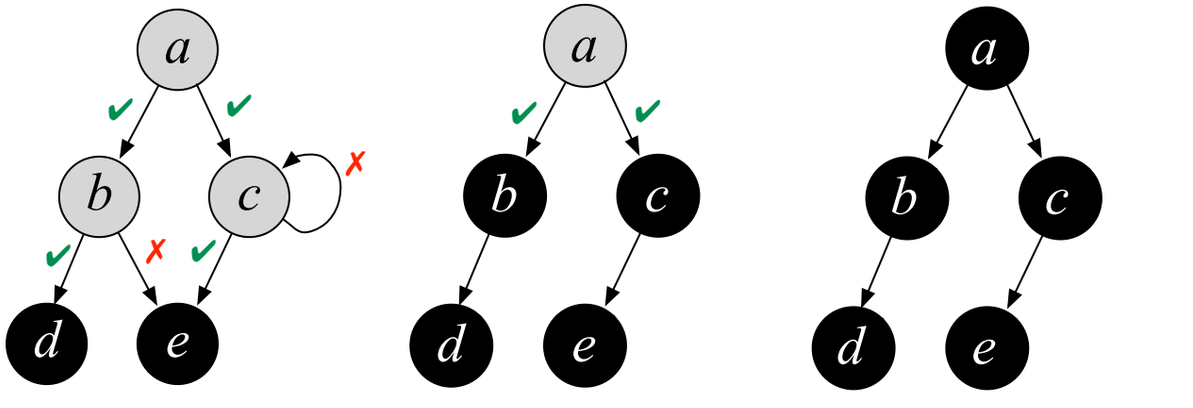


Previously on this channel…

```
letrec span (x : ptr) : bool = {
  if x == null then val_ret 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;
      val_ret true;
    else val_ret false;
}
```

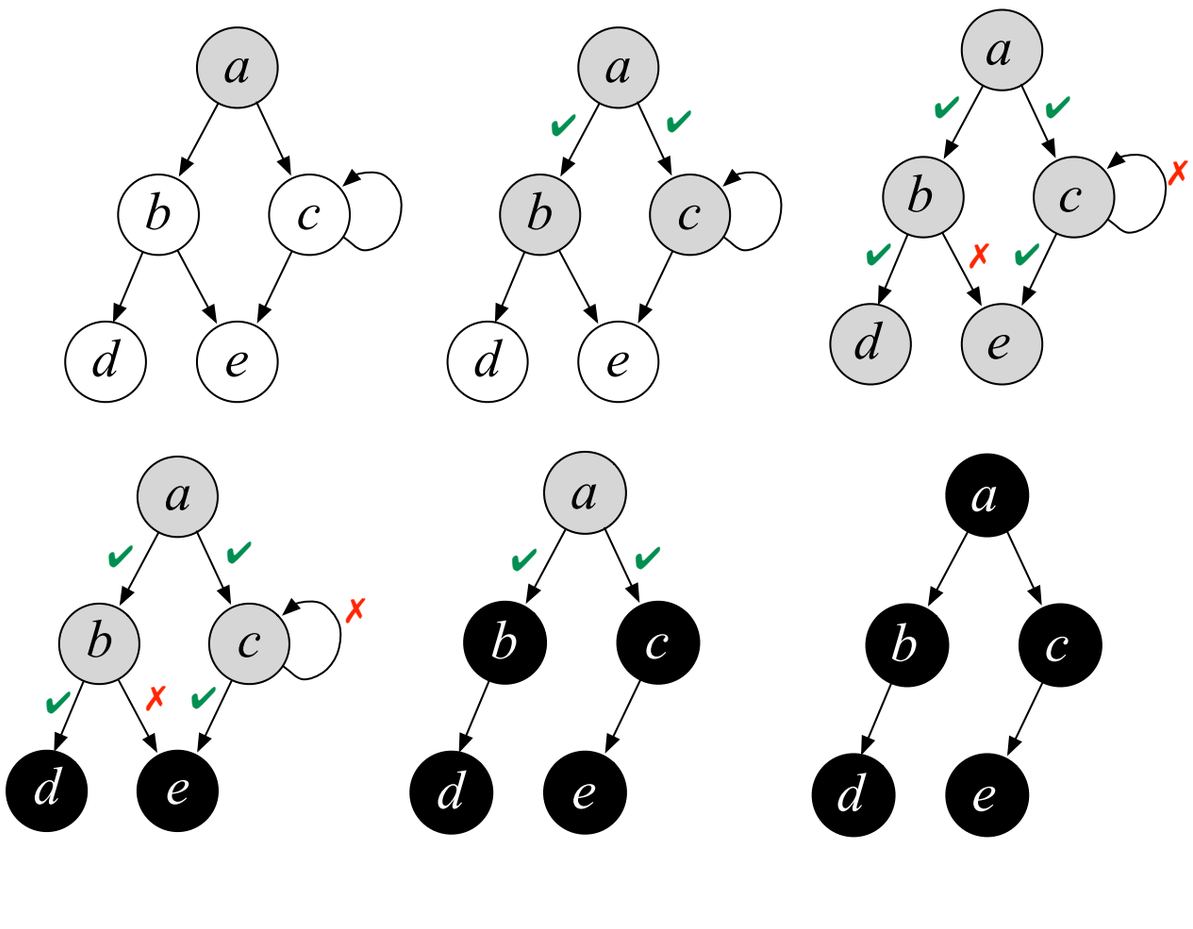
```
Definition span_tp (x : ptr) :=
  {i (g1 : graph (joint i))}, STsep [SpanTree]

    (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).
```





Previously on this channel…

```
letrec span (x : ptr) : bool = {
  if x == null then val_ret 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;
      val_ret true;
    else val_ret false;
}
```



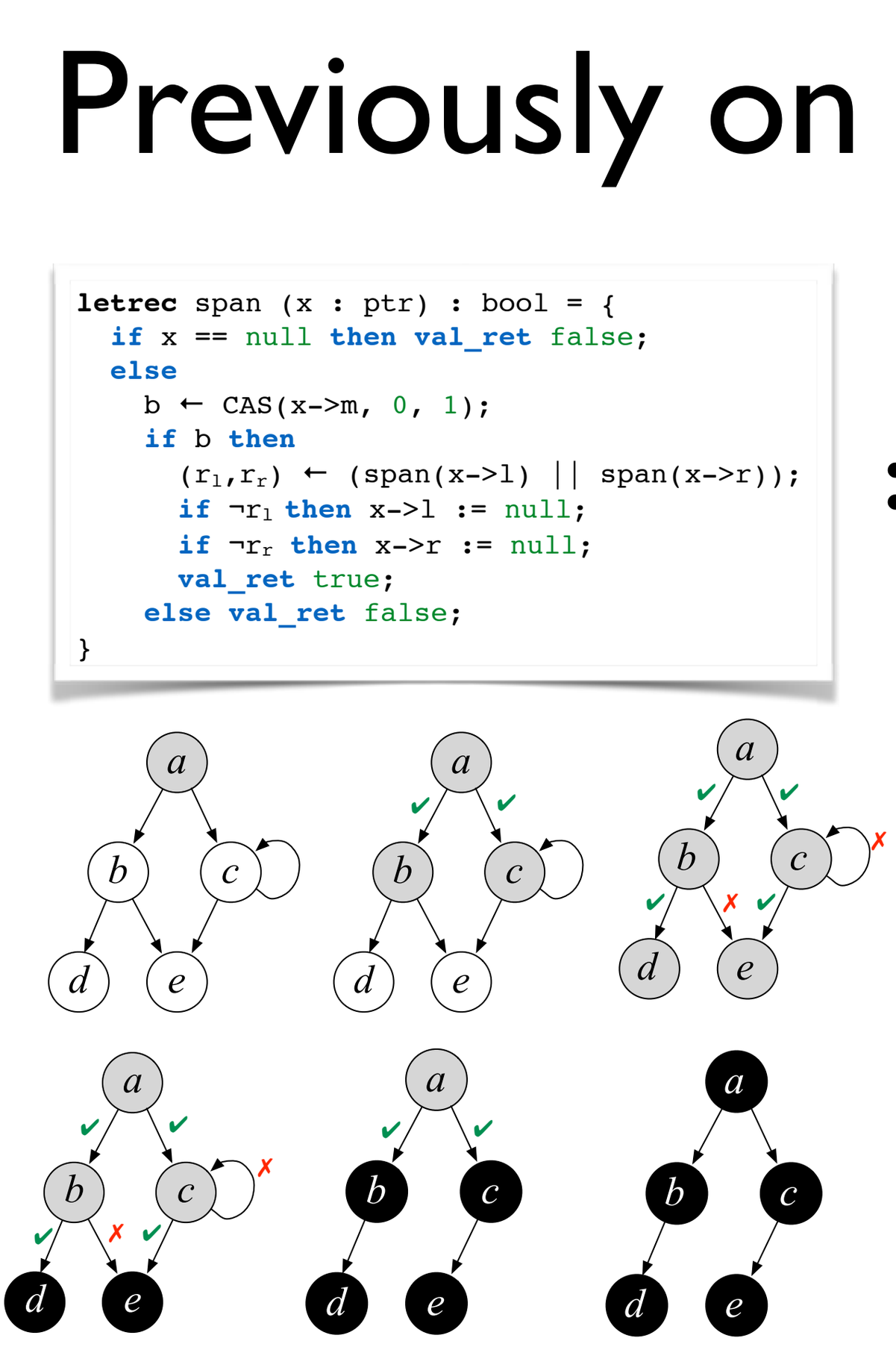

```
Definition span_tp (x : ptr) :=
  {i (g1 : graph (joint i))}, STsep [SpanTree]

    (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).
```

+ with {}

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

Previously on this channel…

```
letrec span (x : ptr) : bool = {
  if x == null then val_ret 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;
      val_ret true;
    else val_ret false;
}
```




```
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).
```

+ with {}

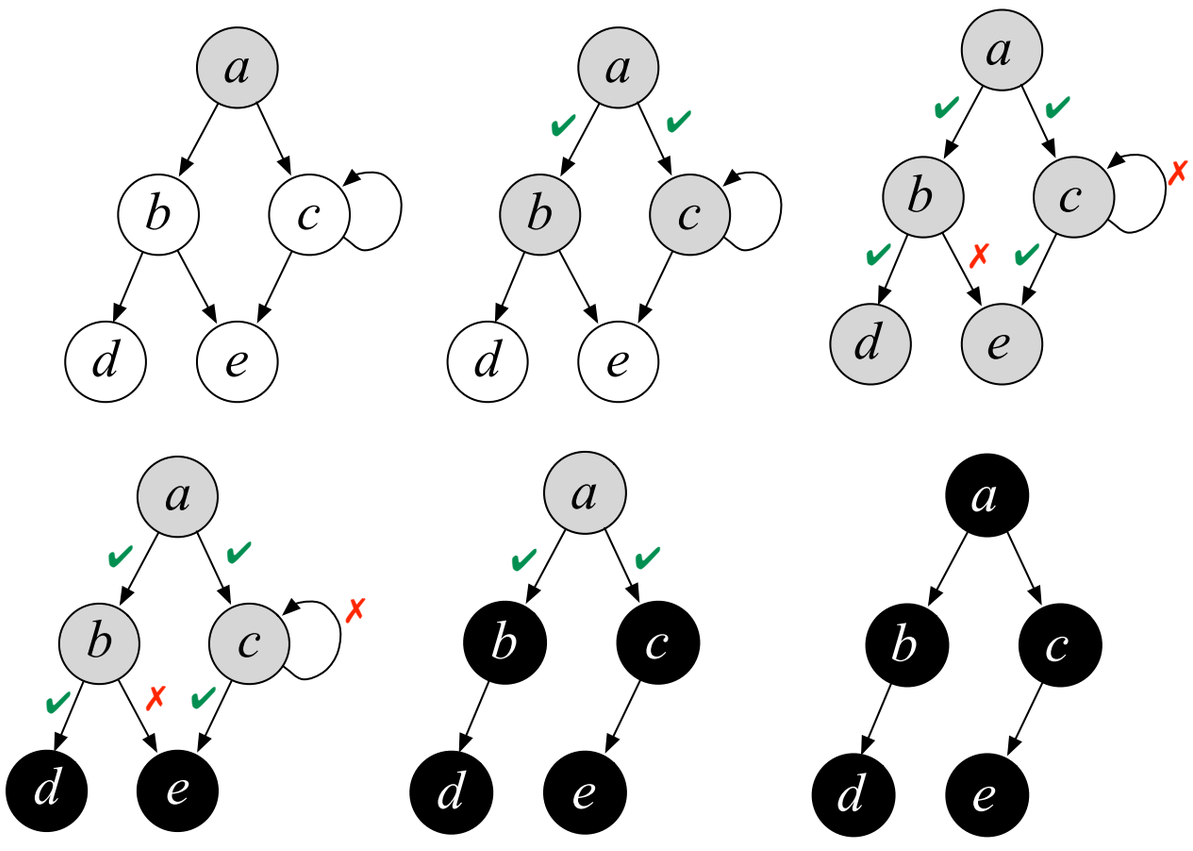
```
tree g2 a t              ∧ maximal g2 t ∧
front g1 t (self s2)     ∧ t = self s2  ∧
is_root a g1             ∧ subgraph g1 g2
```

⇓

spanning t g1

Previously on this channel…

```
letrec span (x : ptr) : bool = {
  if x == null then val_ret 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;
      val_ret true;
    else val_ret false;
}
```




```
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).
```

+ with {}

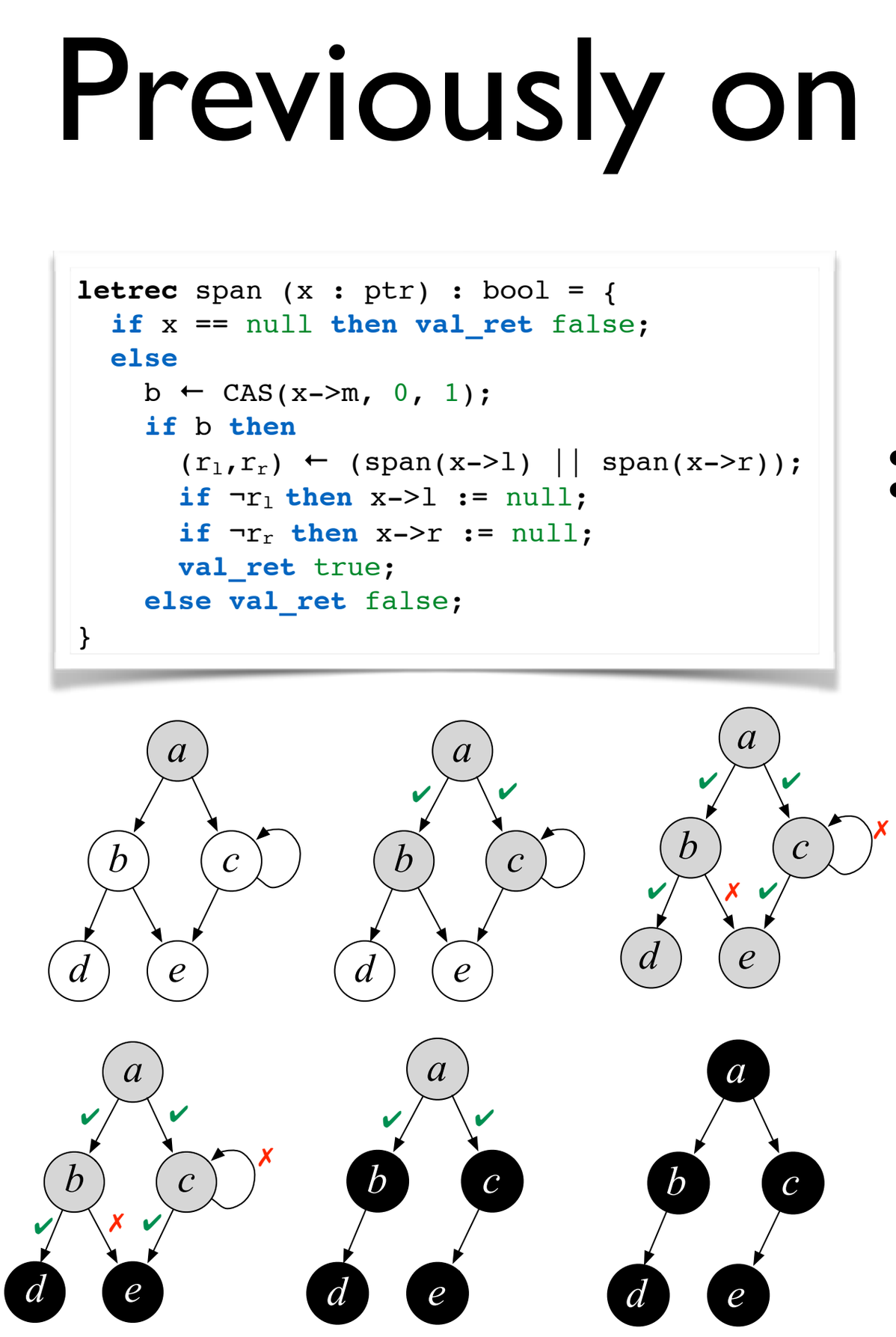
```
tree g2 a t              ∧ maximal g2 t ∧
front g1 t (self s2)     ∧ t = self s2  ∧
is_root a g1             ∧ subgraph g1 g2
```

⇓

spanning t g1

Previously on this channel…

```
letrec span (x : ptr) : bool = {
  if x == null then val_ret 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;
      val_ret true;
    else val_ret false;
}
```




```
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).
```

+ with {}

```
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

Anatomy of mechanized reasoning about fine-grained concurrency

Ilya Sergey

joint work with

Aleks Nanevski
Anindya Banerjee
Ruy Ley-Wild
Germán Delbianco

Concurrent Hoare-style specifications

Concurrent Hoare-style specifications

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

Concurrent Hoare-style specifications

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

Concurrent Hoare-style specifications

Context that
specifies expected
thread interference

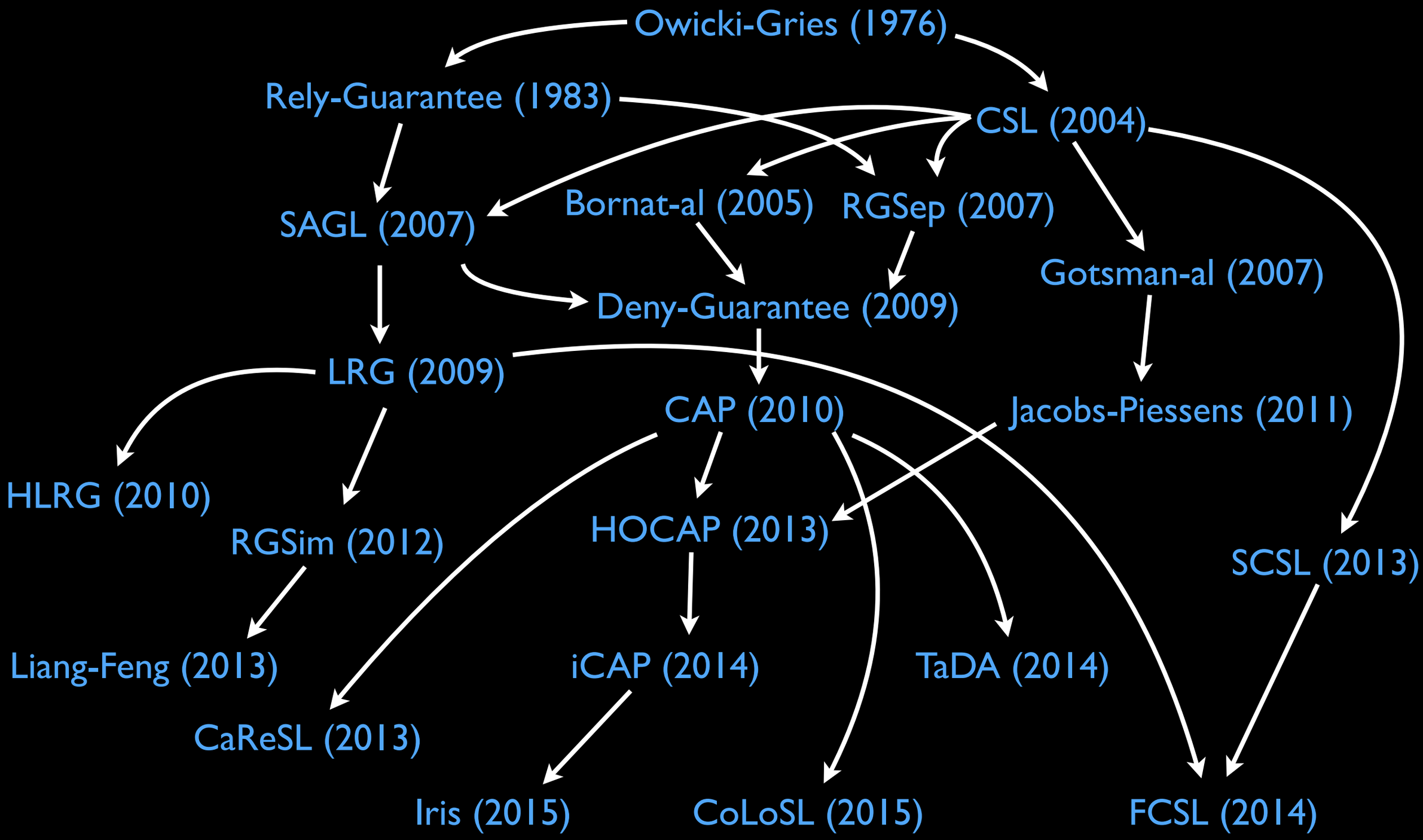
$$C \vdash \{ P \}\ c\ \{ Q \}$$

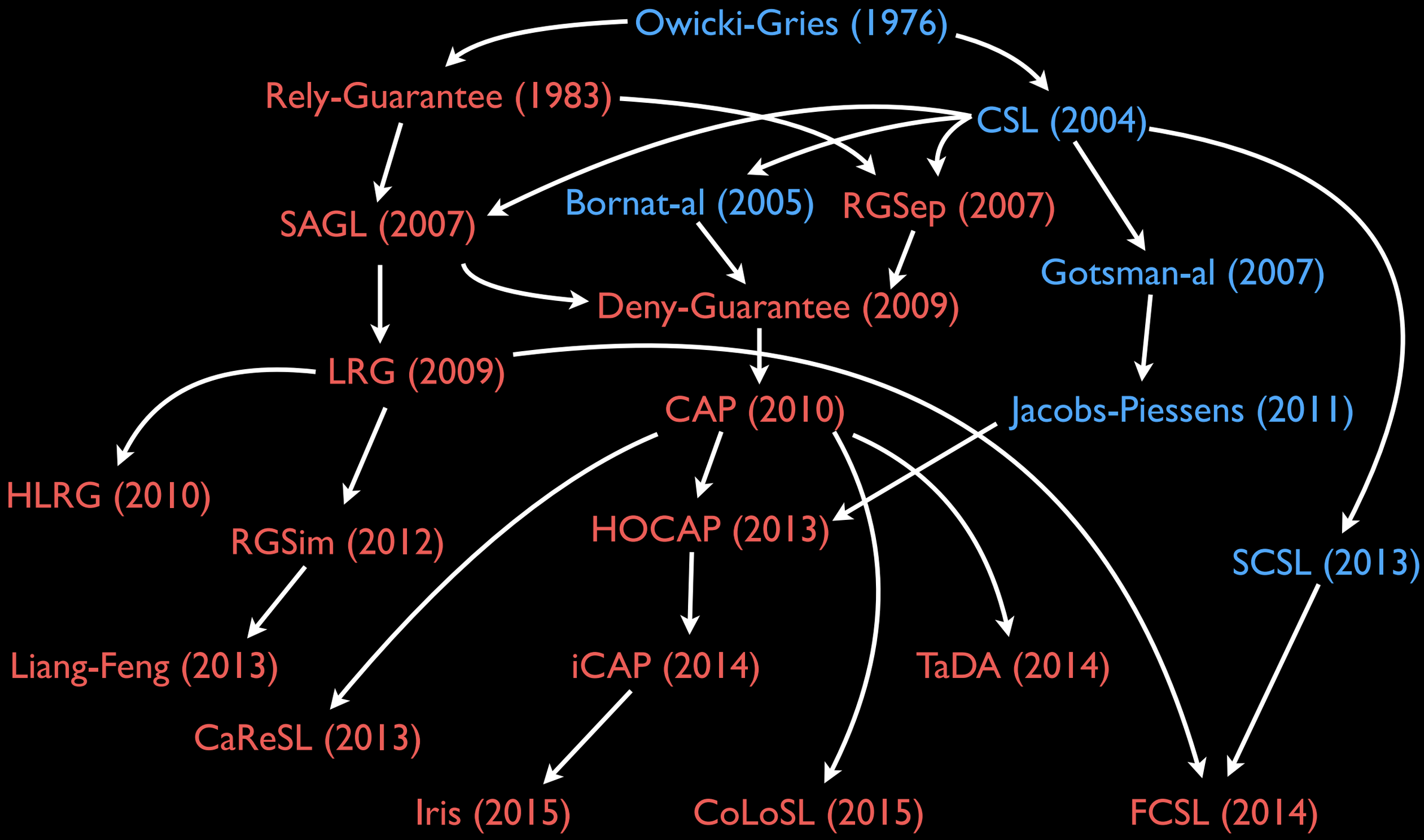
Concurrent Hoare-style specifications

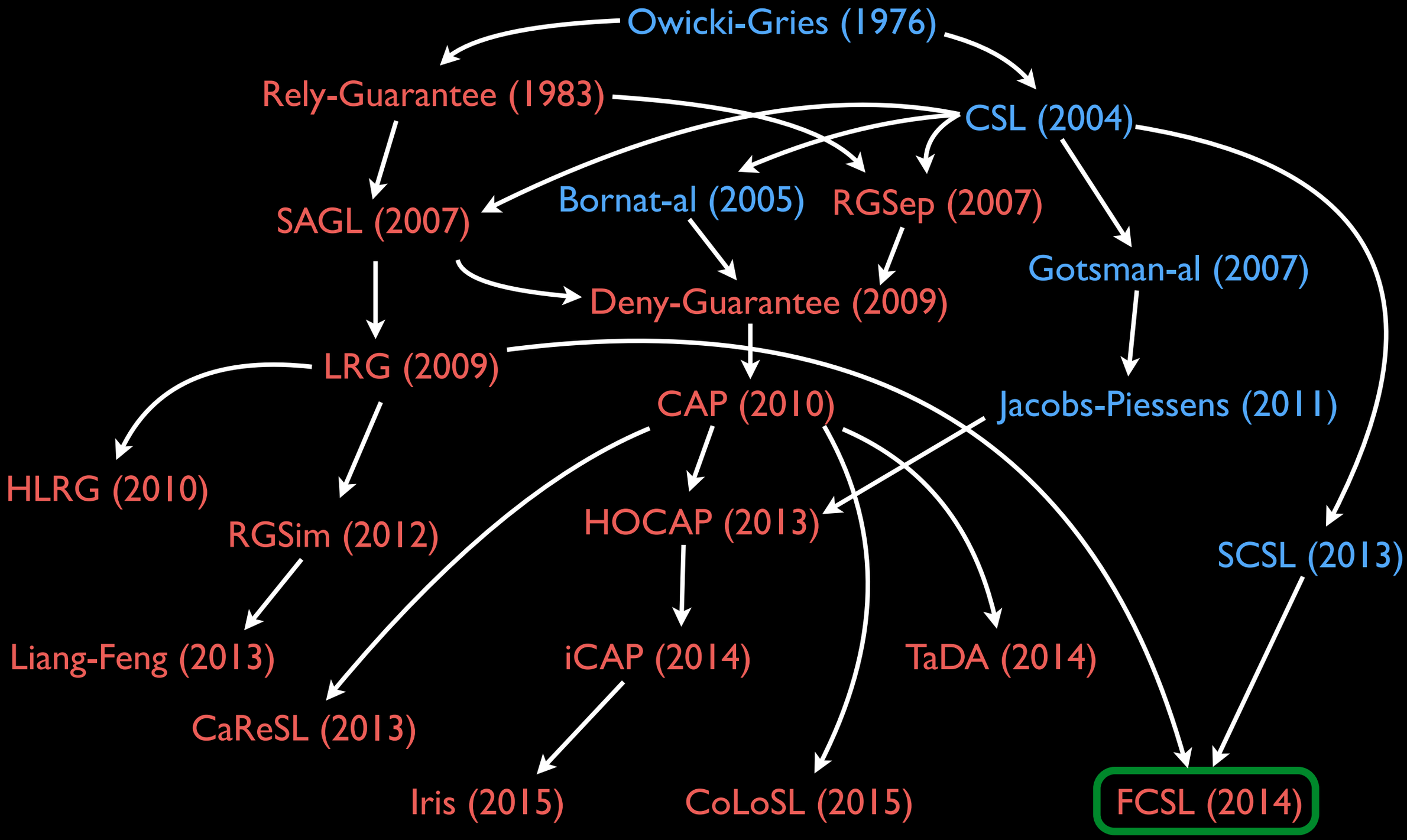
Context that
specifies expected
thread interference

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

aka.
rely/guarantee,
concurrent resources,
regions,
protocols,
islands,
invariants,
concurroids,
monitors


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


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


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

FCSL: Fine-grained Concurrent Separation Logic

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

FCSL: Fine-grained Concurrent Separation Logic

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

- Logic for reasoning with concurroids

FCSL: Fine-grained Concurrent Separation Logic

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

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

Agenda

- Defining concurroids
- Atomic actions and stable specifications
- Verification layout

Agenda

- Defining concurroids
- Atomic actions and stable specifications
- Verification layout

Concurroids

- State-transition systems with subjective state
- *Coherence predicate defines* resource state-space

Demo 1:

Definition of a concurroid's coherence predicate

Concurroids

Concurroids

- State-transition systems with subjective state
- *Coherence predicate defines* resource state-space

Concurroids

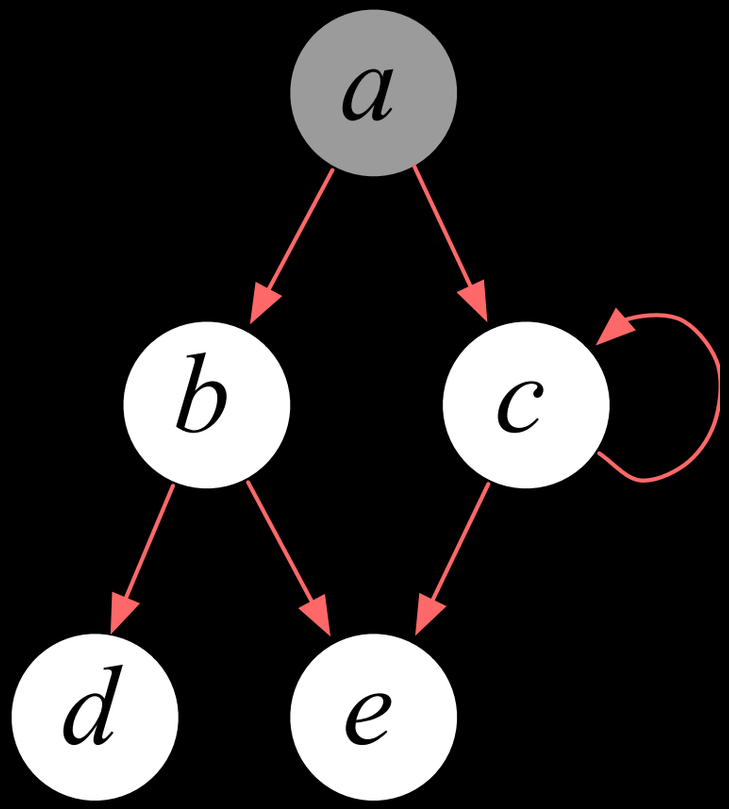
- State-transition systems with subjective state
- *Coherence predicate defines* resource state-space
- *Transitions* describe *guarantee* (and *rely* by transposition)

`mark-node` transition

A successful attempt to atomically mark a node and add it to *self*

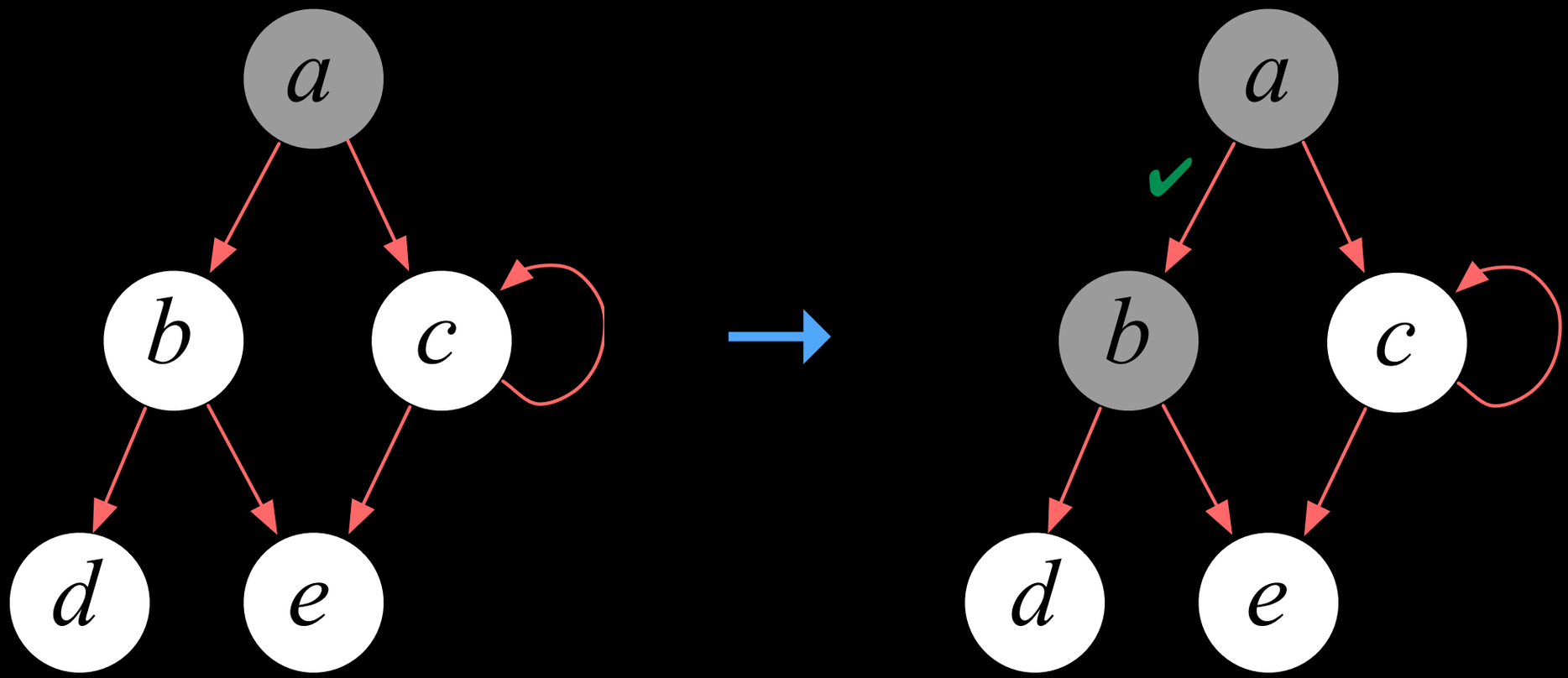
`mark-node` transition

A successful attempt to atomically mark a node and add it to *self*



`mark-node` transition

A successful attempt to atomically mark a node and add it to *self*

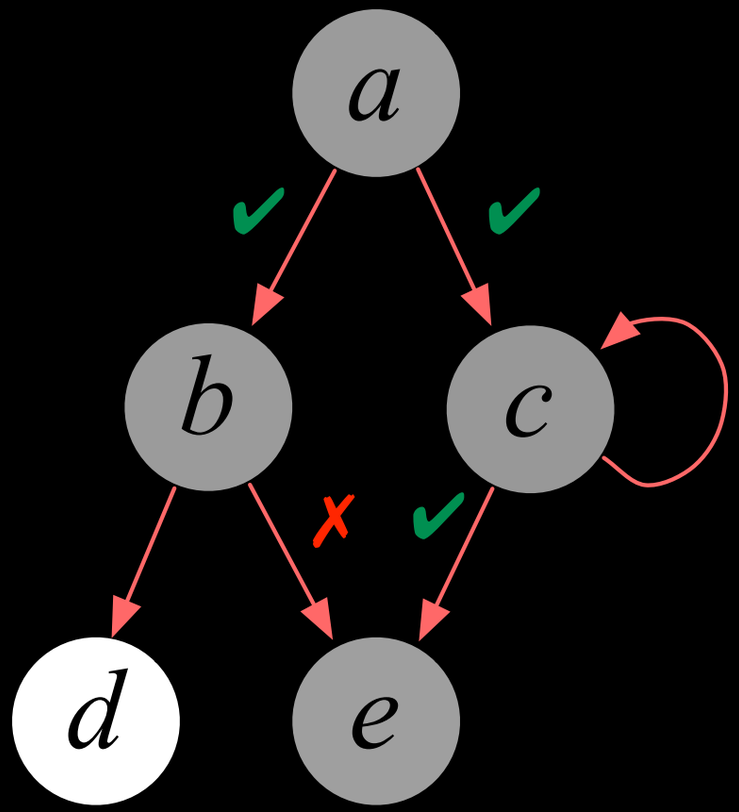


`nullify-edge` transition

Atomically pruning of an edge from a node, owned in *self*

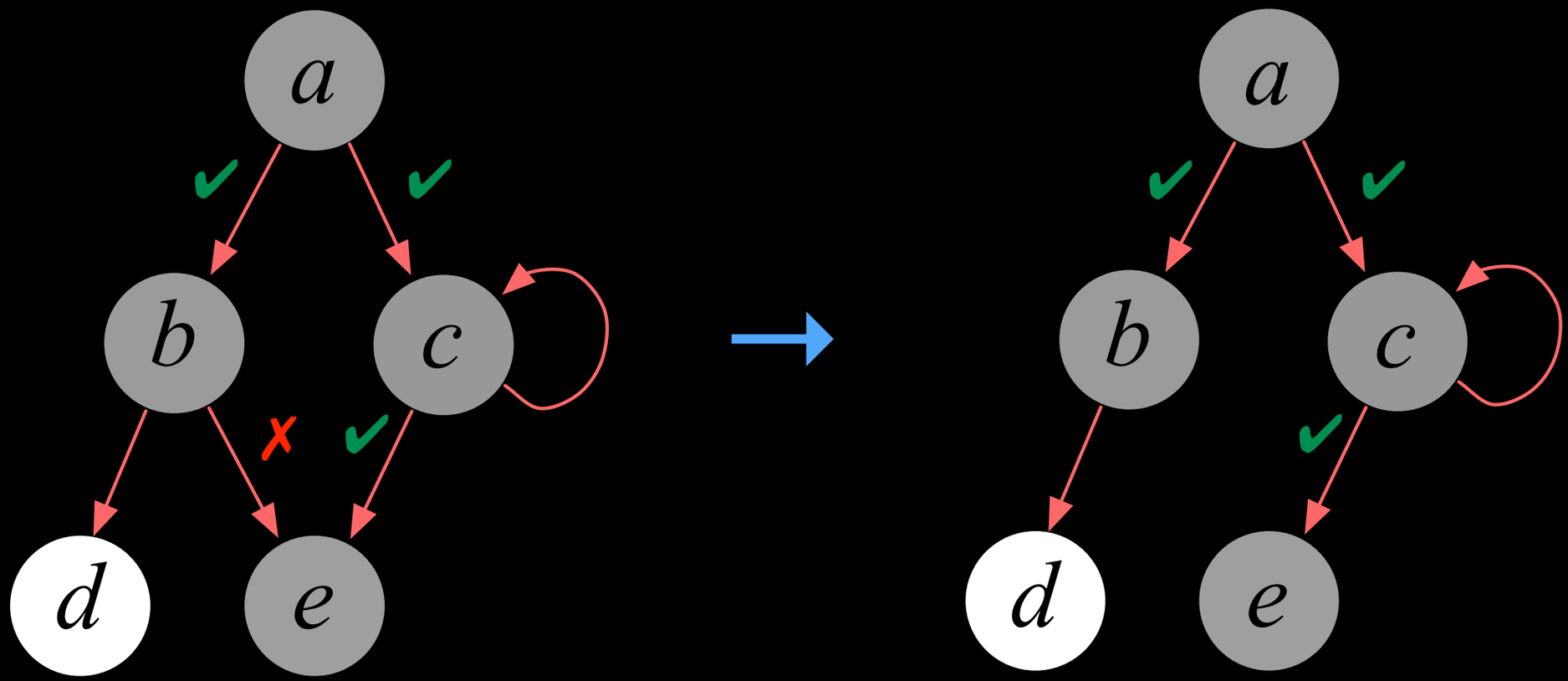
`nullify-edge` transition

Atomically pruning of an edge from a node, owned in *self*



`nullify-edge` transition

Atomically pruning of an edge from a node, owned in *self*



Demo 2:

Defining concurroid transitions

Agenda

- Defining concurroids
- Atomic actions and stable specifications
- Verification layout

Agenda

- Defining concurroids
- Atomic actions and stable specifications
- Verification layout

Atomic actions

Attaching concurroid transitions to machine commands (`CAS`, `write`, *etc*)

Atomic actions

Attaching concurroid transitions to machine commands (`CAS`, `write`, *etc*)

```
letrec span (x : ptr) : bool = {
  if x == null then val_ret 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;
      val_ret true
    else val_ret false
}
```

Atomic actions

Attaching concurroid transitions to machine commands (`CAS`, `write`, *etc*)

`mark-node` or `skip`

```
letrec span (x : ptr) : bool = {
  if x == null then val_ret 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;
      val_ret true
    else val_ret false
}
```

Atomic actions

Attaching concurroid transitions to machine commands (`CAS`, `write`, *etc*)

`mark-node` or `skip`

`skip`

```
letrec span (x : ptr) : bool = {
  if x == null then val_ret 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;
      val_ret true
    else val_ret false
}
```

Atomic actions

Attaching concurroid transitions to machine commands (`CAS`, `write`, *etc*)

```
letrec span (x : ptr) : bool = {
  if x == null then val_ret 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;
      val_ret true
    else val_ret false
}
```

mark-node or skip

skip

skip

nullify-edge

Demo 3:

Defining an action for *trying to mark* a node

Assigning Hoare specs to actions

Assigning Hoare specs to actions

Cannot use an action immediately: need a stable specification

Assigning Hoare specs to actions

Cannot use an action immediately: need a stable specification

a = (C, safe, step, …) is an action

$\forall s\ s'\ v$, P(s) ⇒ safe(s) ∧ step(s, v, s') ⇒ Q(v, s')

P, Q are stable under C's rely

{ P } act (a) { Q }@C

(Action)

Assigning Hoare specs to actions

Cannot use an action immediately: need a stable specification

a = (C, safe, step, …) is an action

$\forall s\ s'\ v,\ \mathsf{P}(s) \Rightarrow \mathsf{safe}(s) \land \mathsf{step}(s, v, s') \Rightarrow \mathsf{Q}(v, s')$

P, Q are stable under C's rely

$\{\ \mathsf{P}\ \}\ \mathsf{act}\ (a)\ \{\ \mathsf{Q}\ \}@\mathsf{C}$ (Action)

P, Q — are explicit stabilisation of an action's *atomic* pre's and post's

Demo 4:

Proving stable specs for atomic actions

Agenda

- Defining concurroids
- Atomic actions and stable specifications
- Verification layout

Agenda

- Defining concurroids
- Atomic actions and stable specifications
- Verification layout

Encoding verification in FCSL

Encoding verification in FCSL

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

Encoding verification in FCSL

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

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

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

Encoding verification in FCSL

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

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

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

Encoding verification in FCSL

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

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

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

Encoding verification in FCSL

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

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

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

Structural lemmas

$$\frac{\{\ P\ \}\ c_1\ \{\ Q\ \}@C \qquad \{\ [x/res]Q\ \}\ c_2\ \{\ R\ \}@C \qquad x \notin FV(R)}{\{\ P\ \}\ x \leftarrow c_1; c_2\ \{\ R\ \}@C}\ \text{(Seq)}$$

Structural lemmas

$$\frac{\{\ P\ \}\ c_1\ \{\ Q\ \}@C \qquad \{\ [x/res]Q\ \}\ c_2\ \{\ R\ \}@C \qquad x \notin FV(R)}{\{\ P\ \}\ x \leftarrow c_1; c_2\ \{\ R\ \}@C}\ \text{(Seq)}$$

```
Lemma step W A B (c1 : ST W A) (c2 : A -> ST W B) i (r : cont B):
   verify i c1 (fun x m => verify m (c2 x) r) ->
   verify i (x <-- c1; c2 x) r.
```

The lemma `step` “corresponds” to the FCSL rule for sequential composition.

Proof of `span : span_tp`

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_comp (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.
```

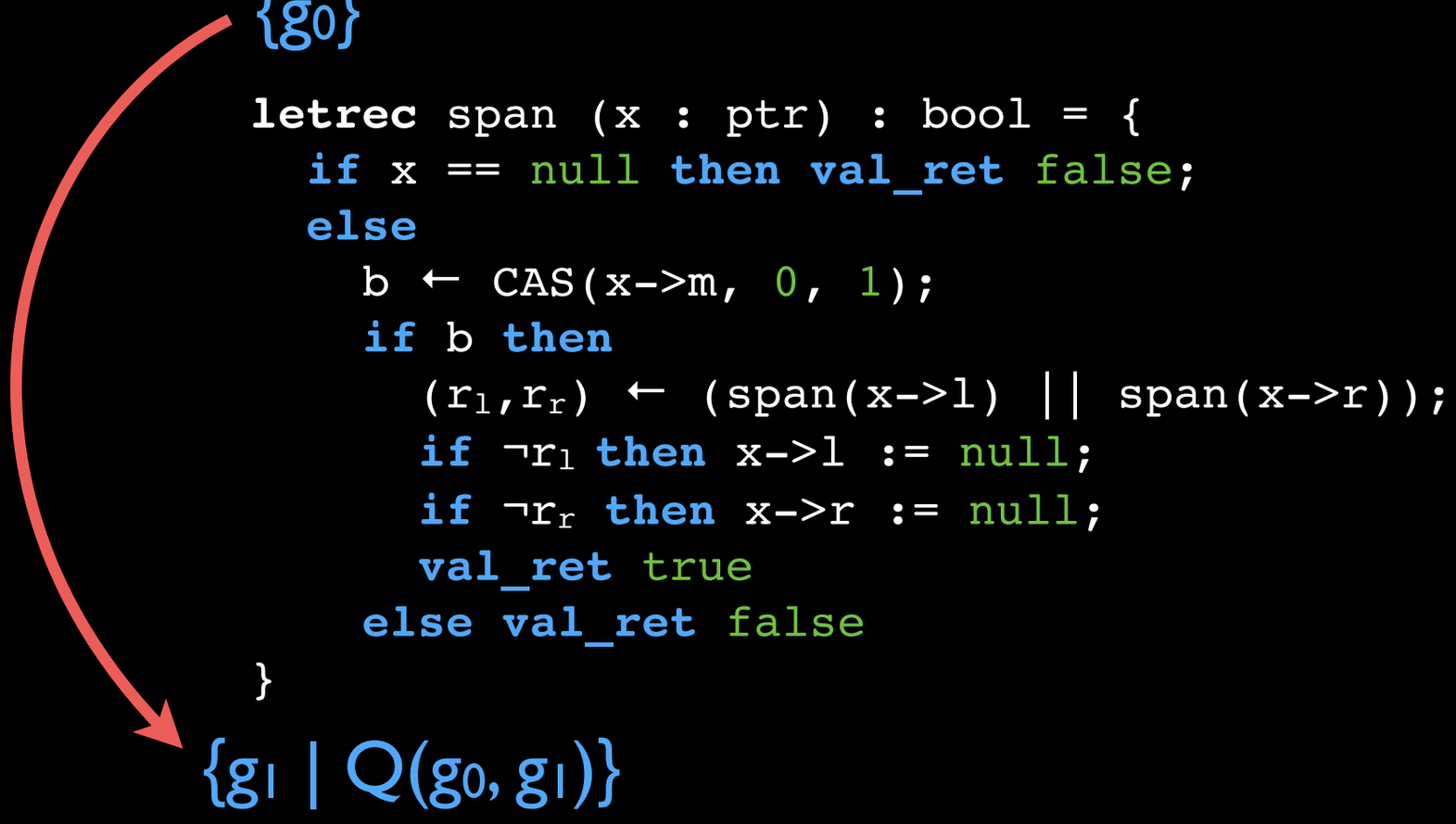
Proof of span : span_tp

A remark on backwards stability

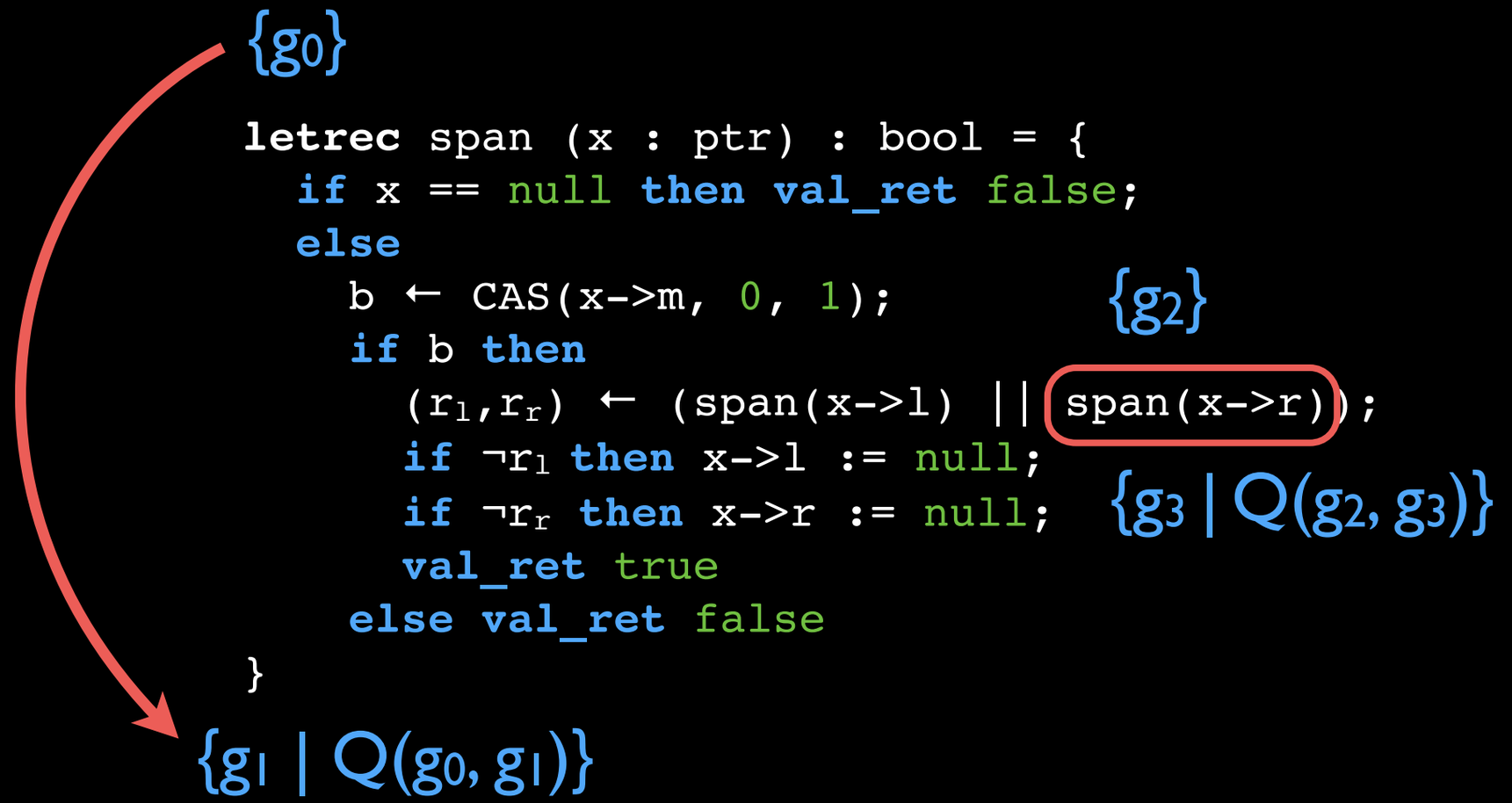
A remark on backwards stability

```
letrec span (x : ptr) : bool = {
  if x == null then val_ret 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;
      val_ret true
    else val_ret false
}
```

A remark on backwards stability



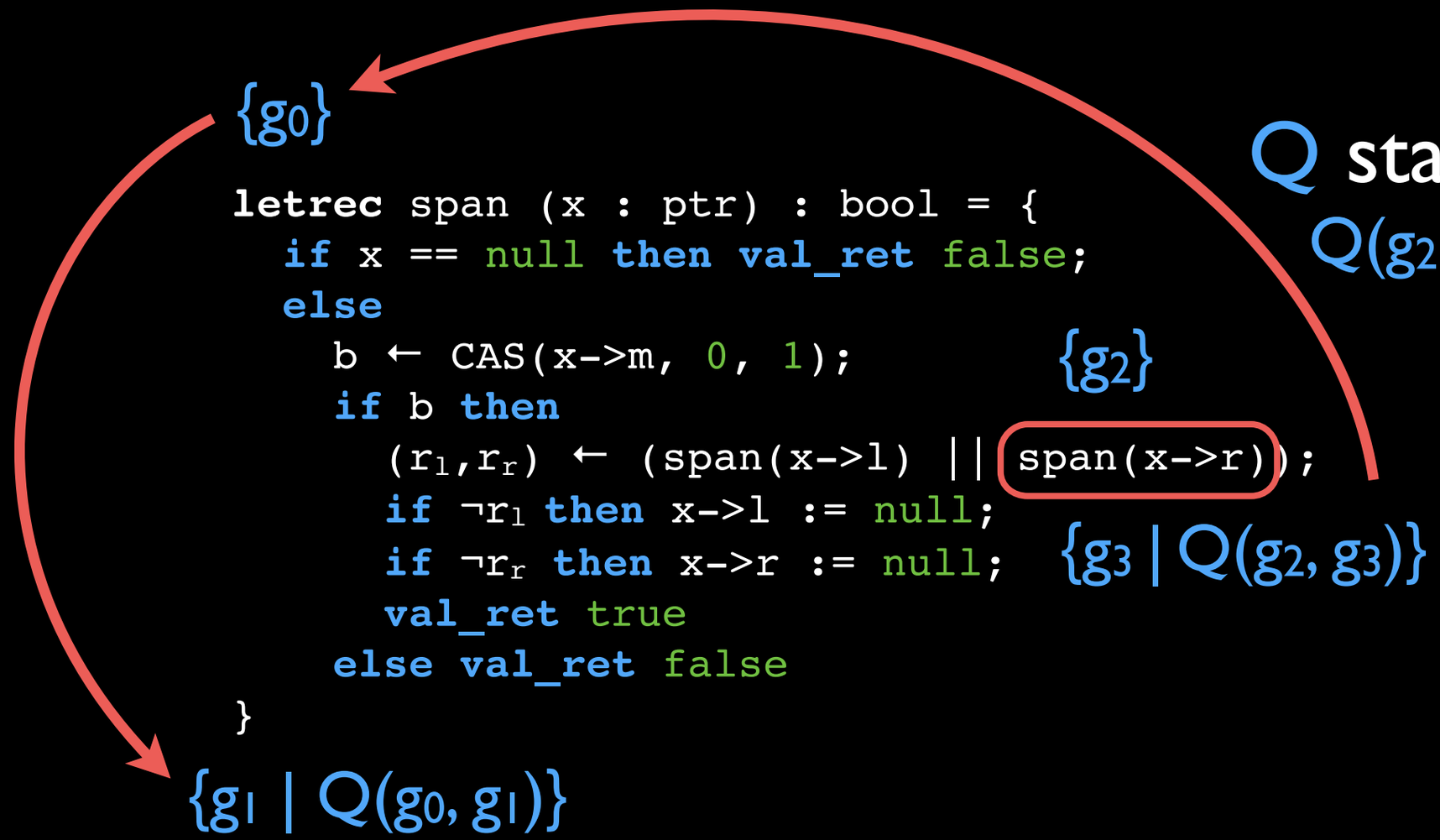
A remark on backwards stability



Result of the recursive call produced *knowledge* about g_2, which also holds for g_0:

$$Q(g_0, g_1) \;:=\; \ldots \wedge \text{front } g_0\ t\ (\text{self}(\text{state}(g_1)) \oplus \text{other}(\text{state}(g_1)) \wedge \ldots$$

A remark on backwards stability



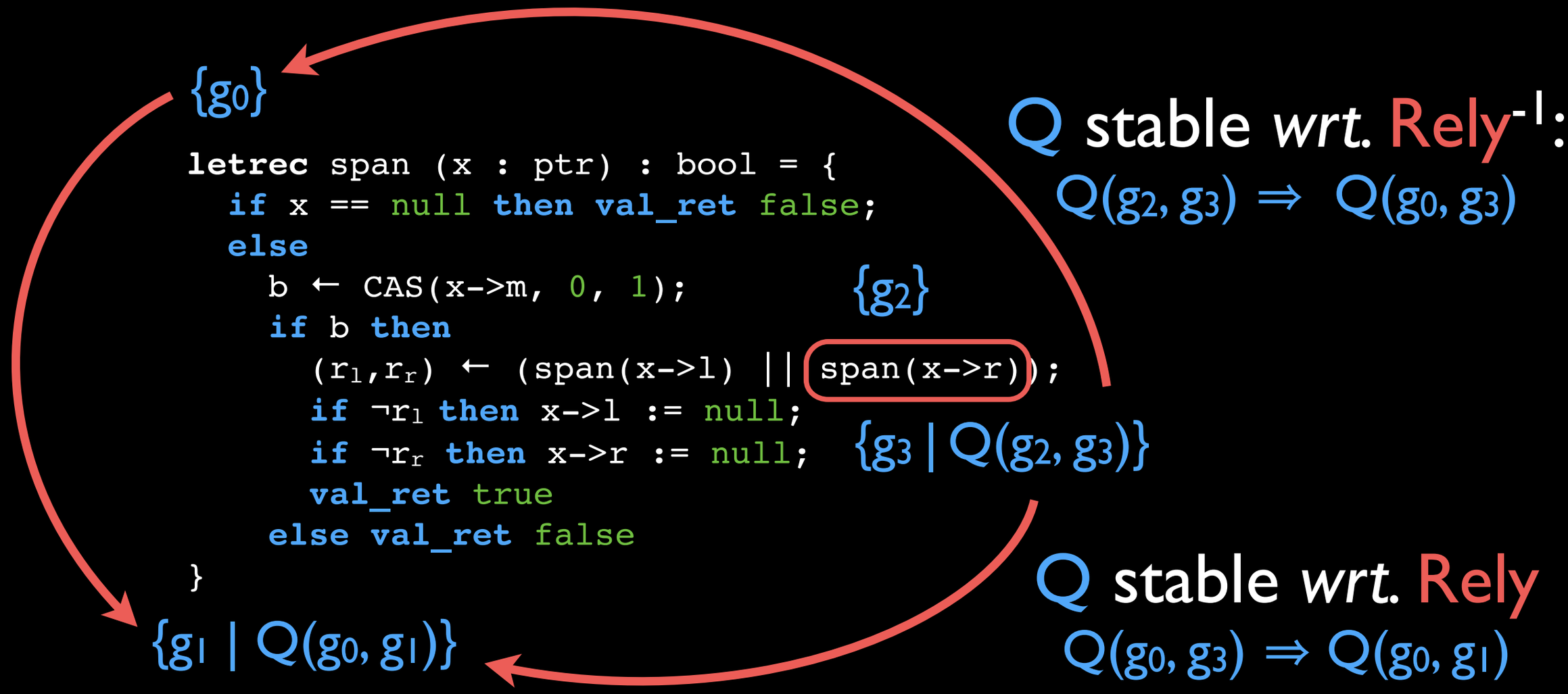
Q stable *wrt.* Rely^{-1}:

$Q(g_2, g_3) \Rightarrow Q(g_0, g_3)$

Result of the recursive call produced *knowledge* about g_2, which also holds for g_0:

$$Q(g_0, g_1) \;:=\; \ldots \wedge \text{front } g_0\ t\ (\text{self}(\text{state}(g_1)) \oplus \text{other}(\text{state}(g_1)) \wedge \ldots$$

A remark on backwards stability



Result of the recursive call produced *knowledge* about g_2, which also holds for g_0:

$$Q(g_0, g_1) \;:=\; \ldots \wedge \mathsf{front}\ g_0\ t\ (\mathsf{self}(\mathsf{state}(g_1)) \oplus \mathsf{other}(\mathsf{state}(g_1)) \wedge \ldots$$

To take away

To take away

- **Concurroids** — coherence and transitions

To take away

- **Concurroids** — coherence and transitions
- **Atomic actions** —
  tying context transitions to program commands

To take away

- **Concurroids** — coherence and transitions
- **Atomic actions** —
  tying context transitions to program commands
- **Verification layout** — applying structural lemmas
  to prove weakening of the strongest spec

To take away

- **Concurroids** — coherence and transitions
- **Atomic actions** —
  tying context transitions to program commands
- **Verification layout** — applying structural lemmas
  to prove weakening of the strongest spec

`software.imdea.org/fcsl`

To take away

- **Concurroids** — coherence and transitions
- **Atomic actions** —
  tying context transitions to program commands
- **Verification layout** — applying structural lemmas
  to prove weakening of the strongest spec

`software.imdea.org/fcsl`

Thanks!

Discussion

- How transitions/actions are different from *view-shifts*?
  How concurroids are related to *regions/invariants/monitors*?
- What is relation between our model and existing object-based verification tools (Chalice, VCC)?
- Can we find better abstractions to *automate* reasoning about stability (forward/backwards)?
- Can we extract FCSL programs to existing certified back-ends (e.g., Bedrock, VST)?

Q&A slides

Statistics for proofs

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

Statistics for proofs

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

Did not require describing new concurrent resources

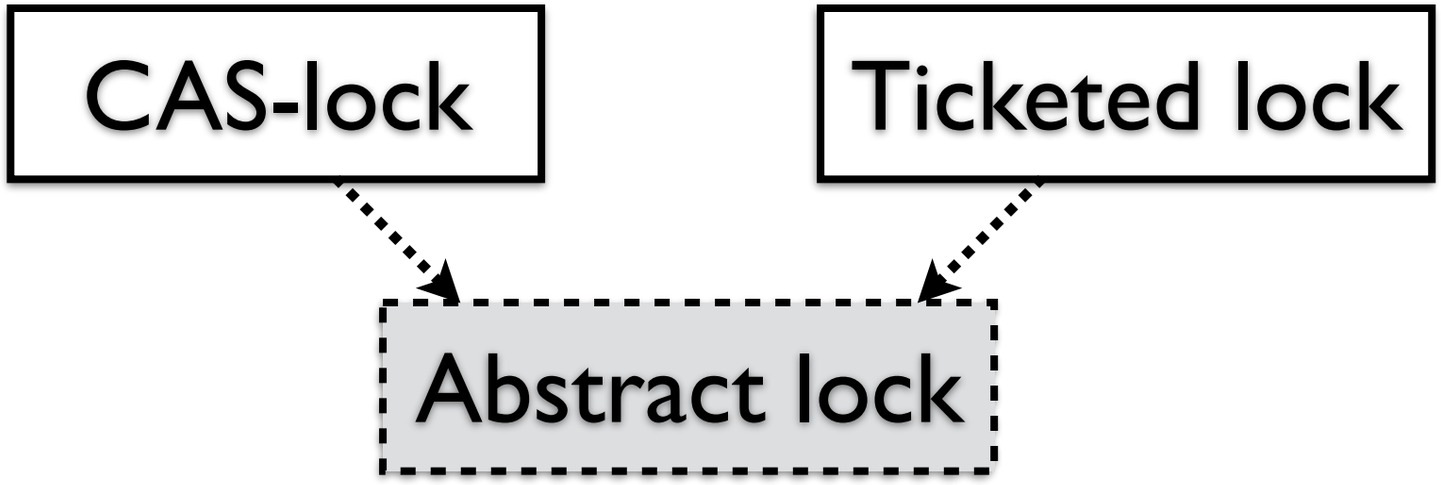
Composing programs and proofs

Composing programs and proofs

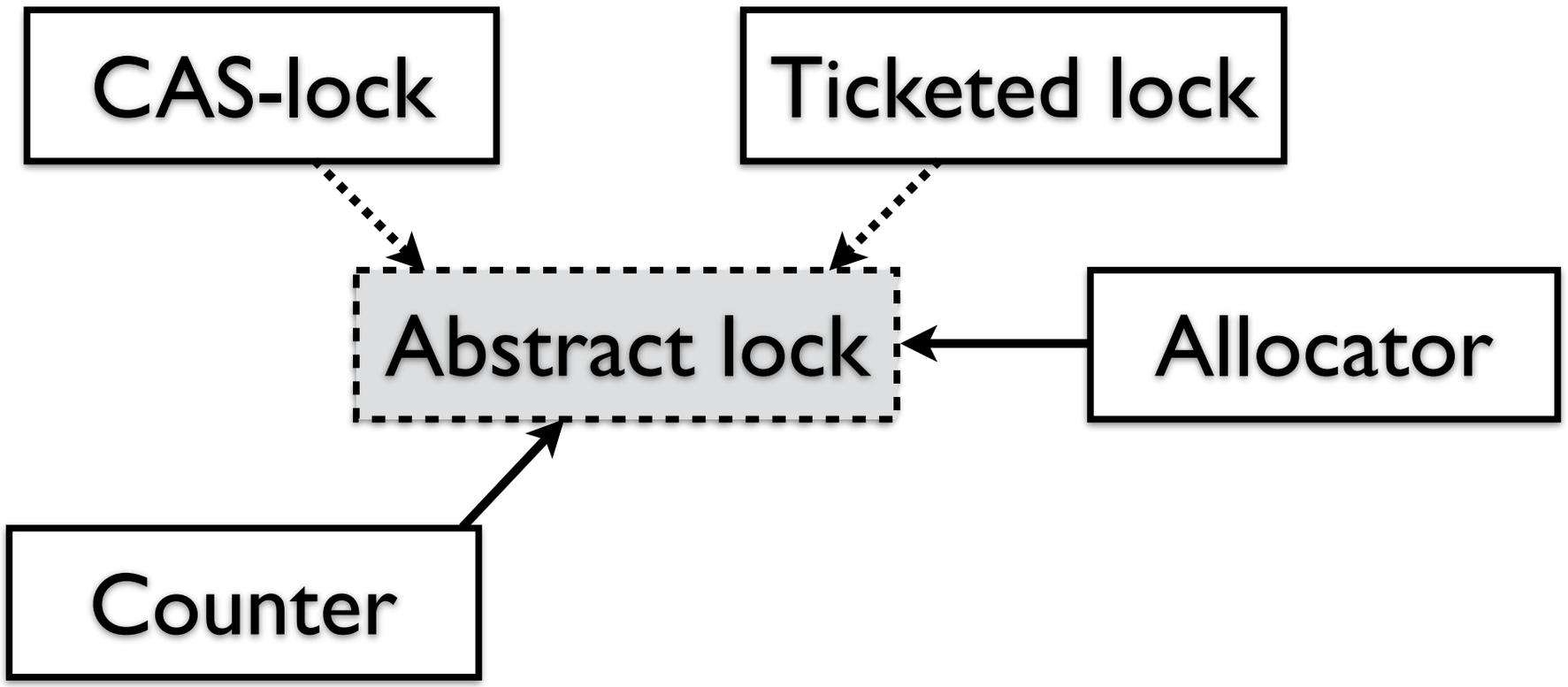
CAS-lock

Ticketed lock

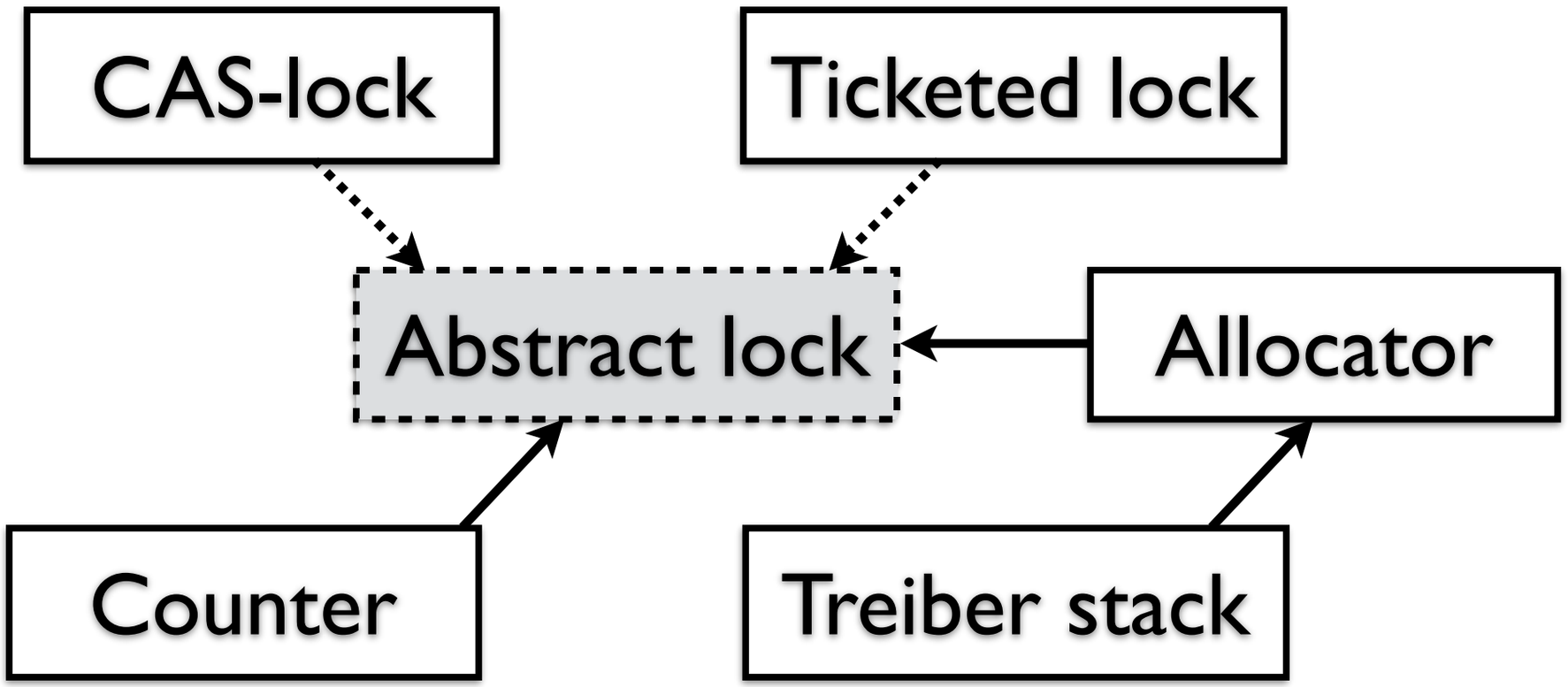
Composing programs and proofs



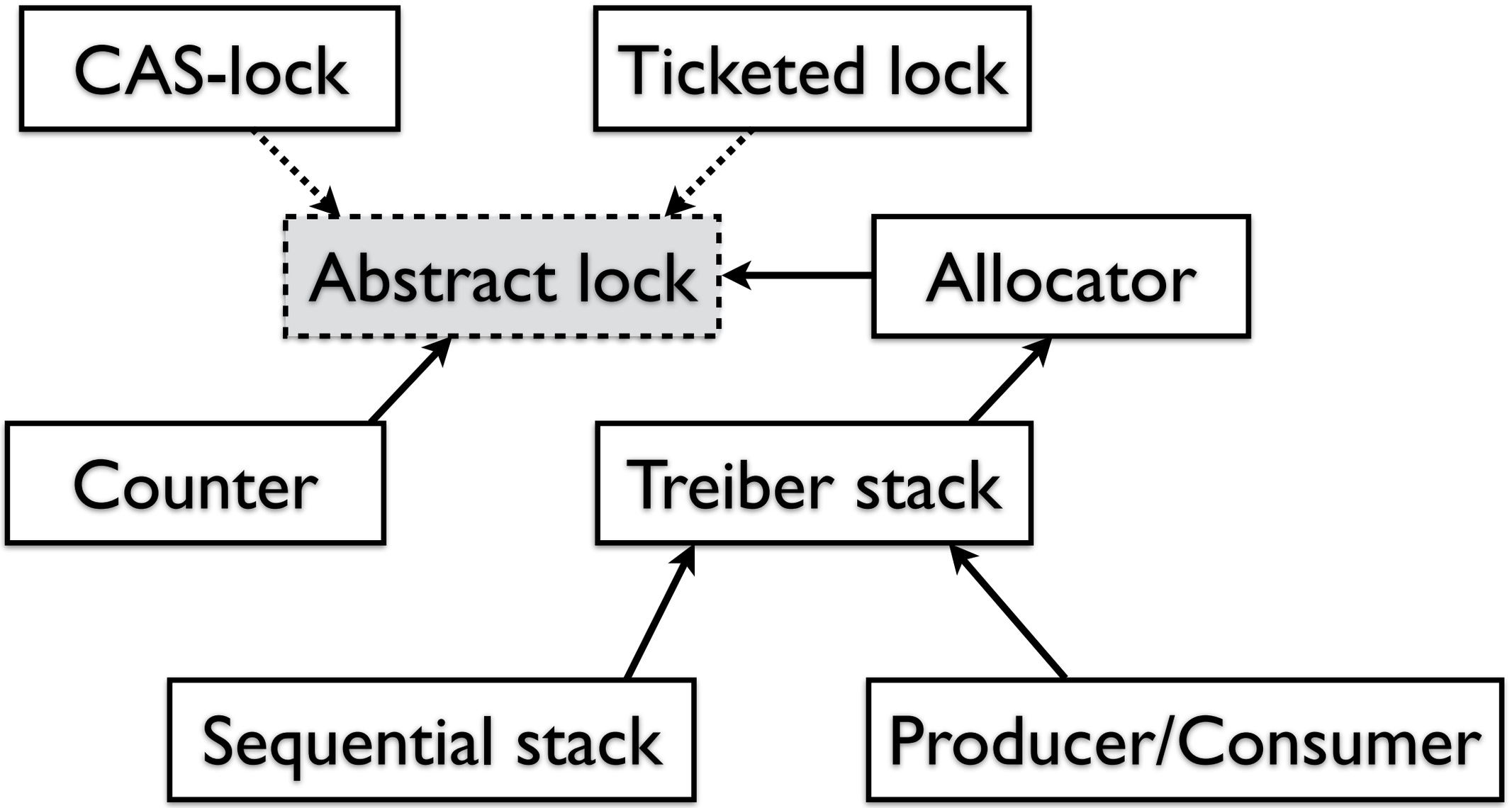
Composing programs and proofs



Composing programs and proofs



Composing programs and proofs



How is your stuff different from other existing concurrent logics?

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

How is your stuff different from Iris?

Jung-al [POPL'15]

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

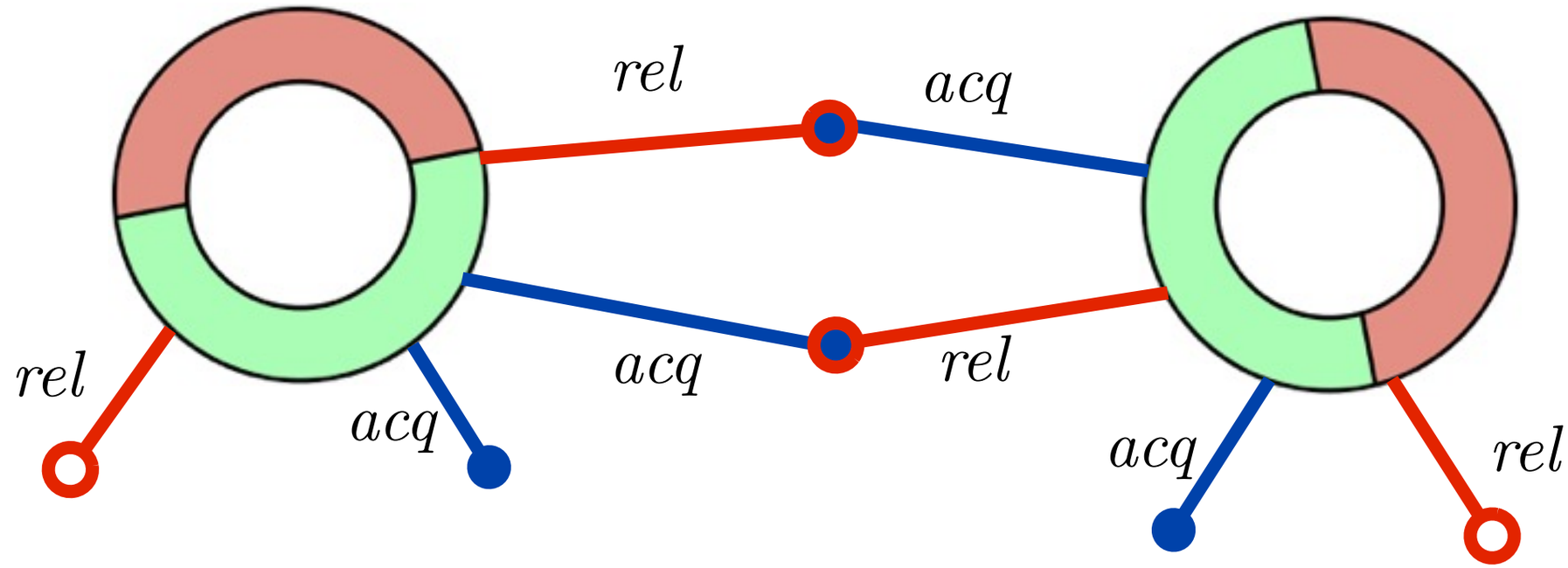
Composing concurrent resources

Composing concurrent resources

Connect ownership-transferring transitions with right polarity

Composing concurrent resources

Connect ownership-transferring transitions with right polarity



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