On Library Correctness under Weak Memory Consistency

Specifying and Verifying Concurrent Libraries under Declarative Consistency Models

plv.mpi-sws.org/yacovet/

Azalea Raad Marko Doko Lovro Rožić Ori Lahav Viktor Vafeiadis

Max Planck Institute for Software Systems (MPI-SWS)
Tel Aviv University

azalea@mpi-sws.org SoundAndComplete.org @azalearaad

Concurrent Library Specification

- Sequential Consistency (SC) -- well-explored
  - semantic-based: linearisability
  - program logics: Hoare logic, separation logic, etc.
  - ***large*** body of case studies

Concurrent Library Specification

- Sequential Consistency (SC) -- well-explored
  - semantic-based: linearisability
  - program logics: Hoare logic, separation logic, etc.
  - ***large*** body of case studies
- Weak Memory Concurrency (WMC) -- under-explored
  - linearisability variants
  - program logic adaptations
  - ***small*** body of case studies

Concurrent Library Specification

- Sequential Consistency (SC) -- well-explored
  - semantic-based: linearisability
  - program logics: Hoare logic, separation logic, etc.
  - ***large*** body of case studies
- Weak Memory Concurrency (WMC) -- under-explored
  - linearisability variants
  - program logic adaptations
  - ***small*** body of case studies

tied to a
particular WMC
memory model (MM) **!**
E.g. C11, TSO, ...

Concurrent Library Specification

- Sequential Consistency (SC) -- well-explored
  - semantic-based: linearisability
  - program logics: Hoare logic, separation logic, etc.
  - ***large*** b
- Weak Me plored
  - linearis ed to a
  - progra ***cular*** WMC
  - ***small*** b ***model (MM)*** **!**
    C11, TSO, ...

<u>wanted</u>

general

MM-agnostic

declarative

specification & verification

framework

Declarative Framework Desiderata

- ***Agnostic*** to memory model
  - support *both SC* and *WMC* specs

Declarative Framework Desiderata

- ***Agnostic*** to memory model
  - support *both SC* and *WMC* specs
- ***General***
  - port *existing SC (linearisability) specs*
  - port *existing WMC specs* (e.g. C11, TSO)
  - built from the *ground up*: assume *no pre-existing* libraries or specs

Declarative Framework Desiderata

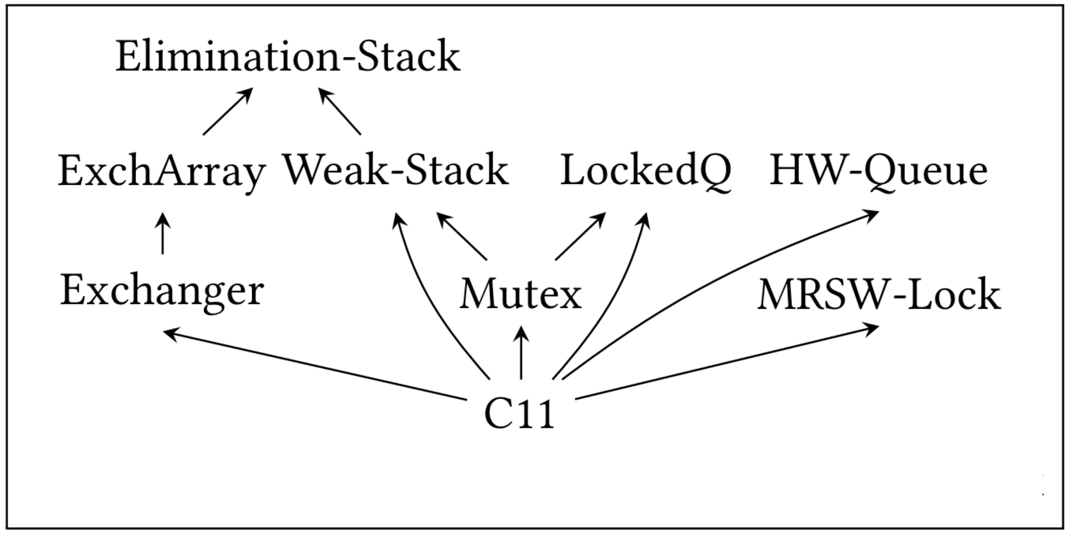
- ***Agnostic*** to memory model
  - support <u>*both SC*</u> and <u>*WMC*</u> specs
- ***General***
  - port <u>*existing SC (linearisability) specs*</u>
  - port <u>*existing WMC specs*</u> (e.g. C11, TSO)
  - built from the <u>*ground up*</u>: assume <u>*no pre-existing*</u> libraries or specs
- ***Compositional***
  - verify <u>*client programs*</u>

```
    q:=new-queue();
    s:=new-stack();
enq(q,1);  || a:=pop(s);
push(s,2)  || if(a==2)
           ||   b:=deq(q)  // returns 1
```

Declarative Framework Desiderata

- ***Agnostic*** to memory model
  - support *both SC* and *WMC* specs
- ***General***
  - port *existing SC (linearisability) specs*
  - port *existing WMC specs* (e.g. C11, TSO)
  - built from the *ground up*: assume *no pre-existing* libraries or specs
- ***Compositional***
  - verify *client programs*
  - verify *library implementations* ⇒ *towers of abstraction*

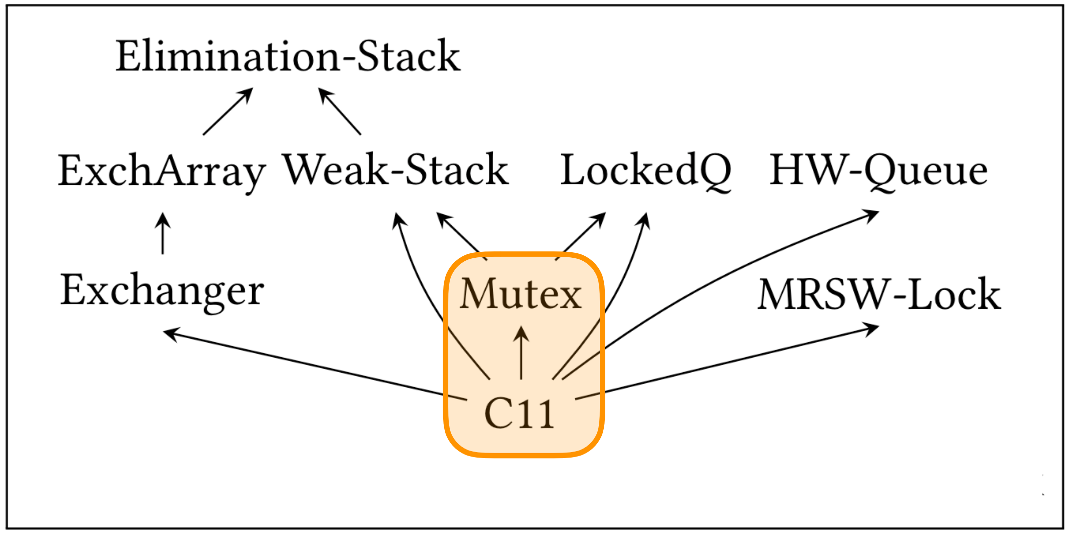
```
q:=new-queue();
s:=new-stack();

enq(q,1);    || a:=pop(s);
push(s,2)    || if(a==2)
             ||    b:=deq(q)   // returns 1
```



Declarative Framework Desiderata

- ***Agnostic*** to memory model
  - support *both SC* and *WMC* specs
- ***General***
  - port *existing SC (linearisability) specs*
  - port *existing WMC specs* (e.g. C11, TSO)
  - built from the *ground up*: assume *no pre-existing* libraries or specs
- ***Compositional***
  - verify *client programs*
  - verify *library implementations* ⇒ *towers of abstraction*

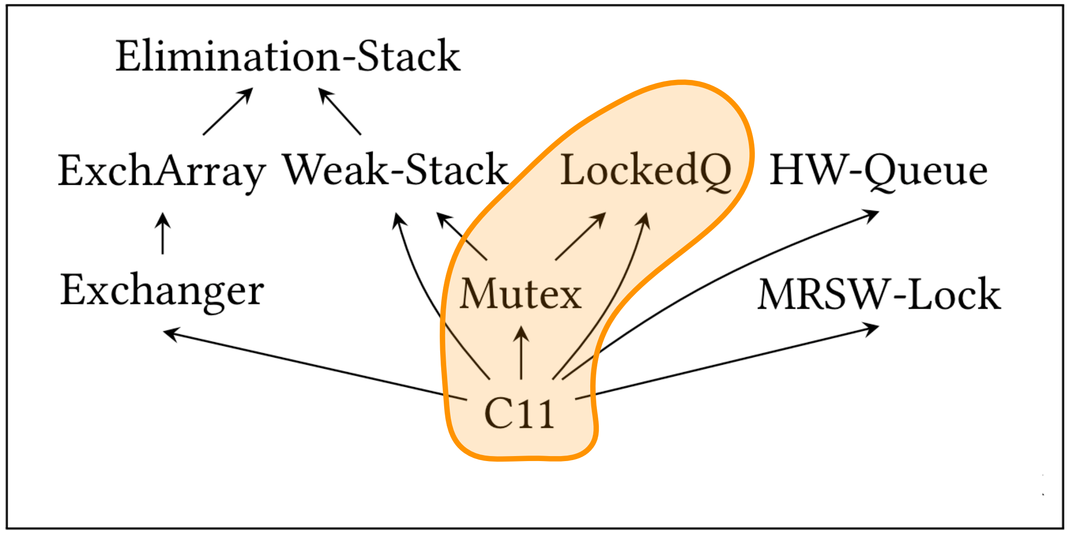
```
q:=new-queue();
s:=new-stack();

enq(q,1);   ||  a:=pop(s);
push(s,2)   ||  if(a==2)
            ||    b:=deq(q)   // returns 1
```



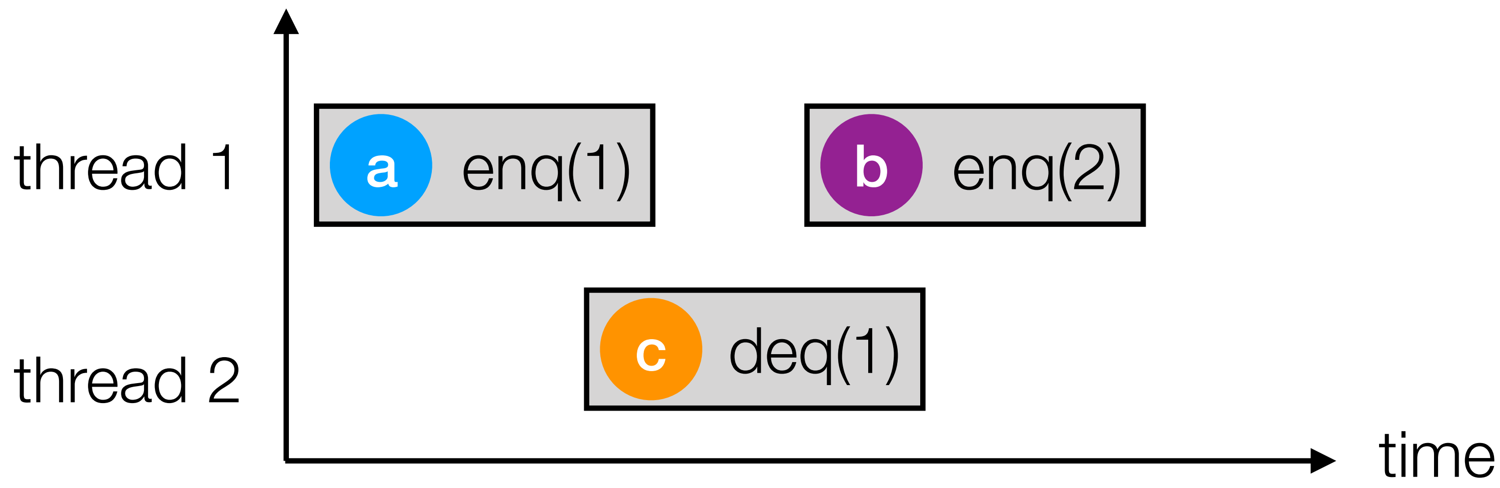
Declarative Framework Desiderata

- ***Agnostic*** to memory model
  - support *both SC* and *WMC* specs
- ***General***
  - port *existing SC (linearisability) specs*
  - port *existing WMC specs* (e.g. C11, TSO)
  - built from the *ground up*: assume *no pre-existing* libraries or specs
- ***Compositional***
  - verify *client programs*
  - verify *library implementations* ⇒ *towers of abstraction*

```
        q:=new-queue();
        s:=new-stack();

                    || a:=pop(s);
  enq(q,1);         || if(a==2)
  push(s,2)         ||    b:=deq(q)   // returns 1
```



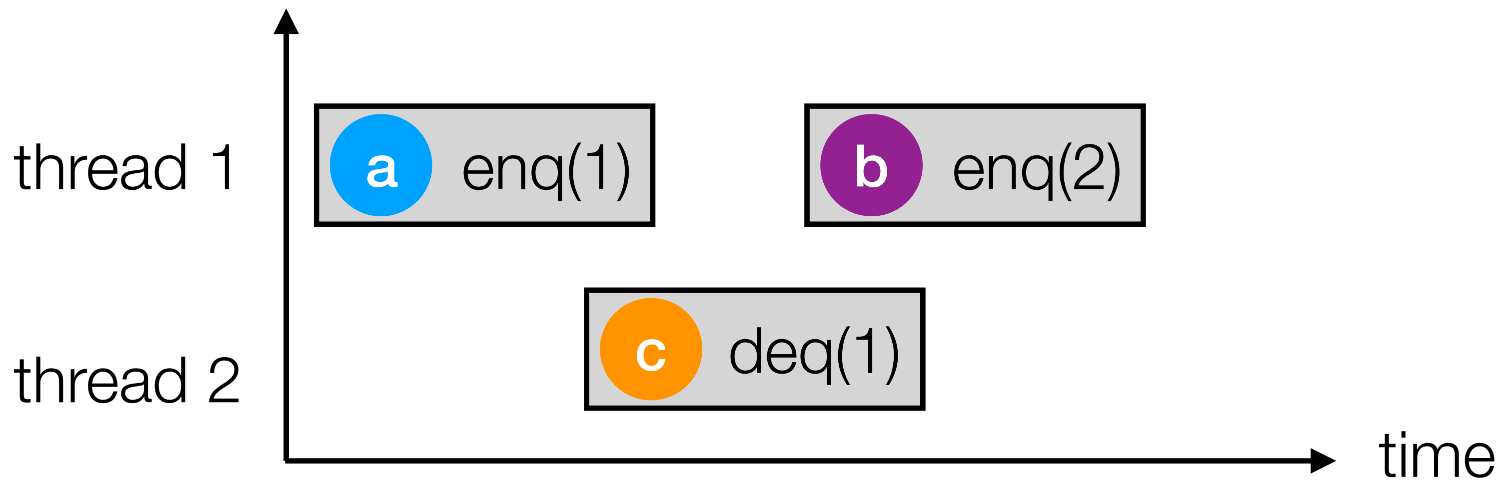
Linearisability



- Define a (partial) happens-before relation *hb* on events
  - $(e_1, e_2) \in hb \iff e_1.\mathrm{end} <_{\mathbf{time}} e_2.\mathrm{begin}$

    -- e.g. (a, b) ∈ *hb* (a, c) ∉ *hb*

Linearisability



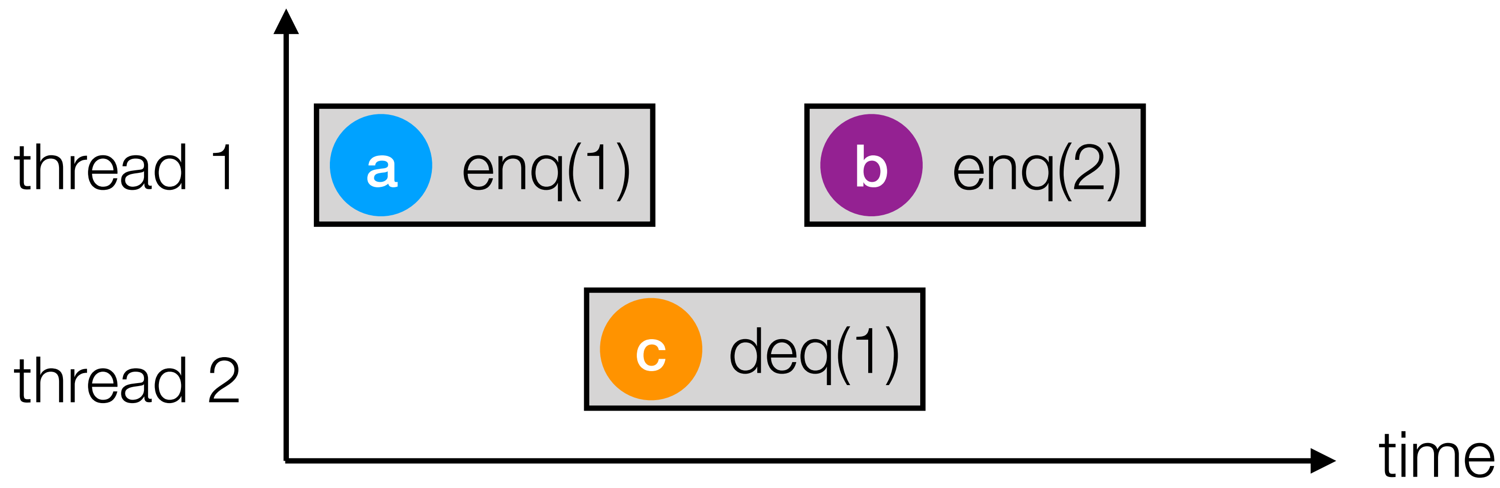
- Define a (partial) happens-before relation *hb* on events
  - $(e_1, e_2) \in hb \iff e_1.\text{end} <_{\textbf{time}} e_2.\text{begin}$

    -- e.g. (a, b) ∈ *hb*   (a, c) ∉ *hb*

- ***Linearisable*** ⟺ ∃ *to*. *to* ***totally*** orders events
  - *hb* ⊆ *to*
  - *to* is a ***legal*** sequence (library-specific)

    -- e.g. *to* is a ***FIFO*** sequence

Linearisability



- Define a (partial) happens-before relation *hb* on events
  - $(e_1, e_2) \in hb \iff e_1.\text{end} <_{\text{time}} e_2.\text{begin}$

    -- e.g. (a, b) ∈ *hb* (a, c) ∉ *hb*

- ***Linearisable*** ⟺ ∃ *to*. *to* ***totally*** orders events
  - *hb* ⊆ *to*
  - *to* is a ***legal*** sequence (library-specific)

    -- e.g. *to* is a ***FIFO*** sequence

a b c ✔ linearisable

Why Not Linearisability?

- ✗ assumes $<_{\mathbf{time}}$ order -- not present under WMC
- ✗ requires ***total*** order *to* on ***all*** events -- not always possible under WMC

Why Not Linearisability?

- ✘ assumes $<_{\textbf{time}}$ order -- not present under WMC
- ✘ requires ***total*** order *to* on ***all*** events -- not always possible under WMC

a	`x:=1;`	||	c	`y:=1;`	(SB)
b	`b:=y` // 0		d	`d:=x` // 0	

Why Not Linearisability?

- ✘ assumes **$<_{time}$** order -- not present under WMC
- ✘ requires ***total*** order *to* on ***all*** events -- not always possible under WMC

a	`x:=1;`	||	c	`y:=1;`
b	`b:=y` // 0	||	d	`d:=x` // 0

(SB)

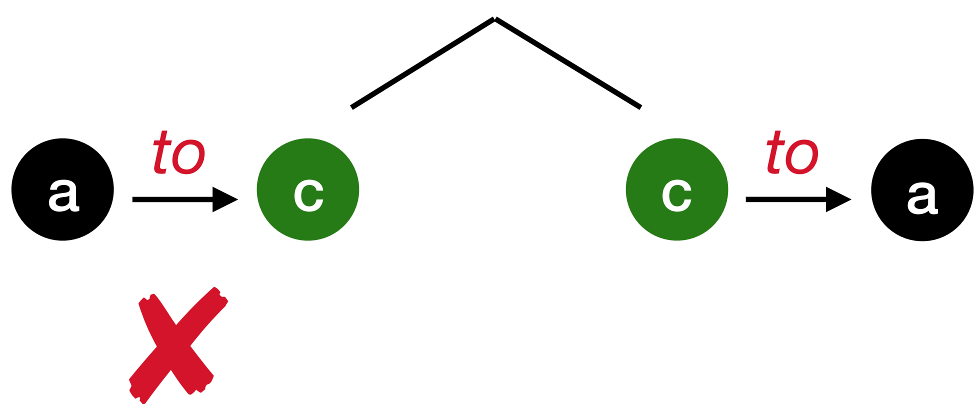
$a \xrightarrow{\textit{to}} c$ $\qquad$ $c \xrightarrow{\textit{to}} a$

Why Not Linearisability?

- ✗ assumes $<_{\mathbf{time}}$ order -- not present under WMC
- ✗ requires ***total*** order *to* on ***all*** events -- not always possible under WMC

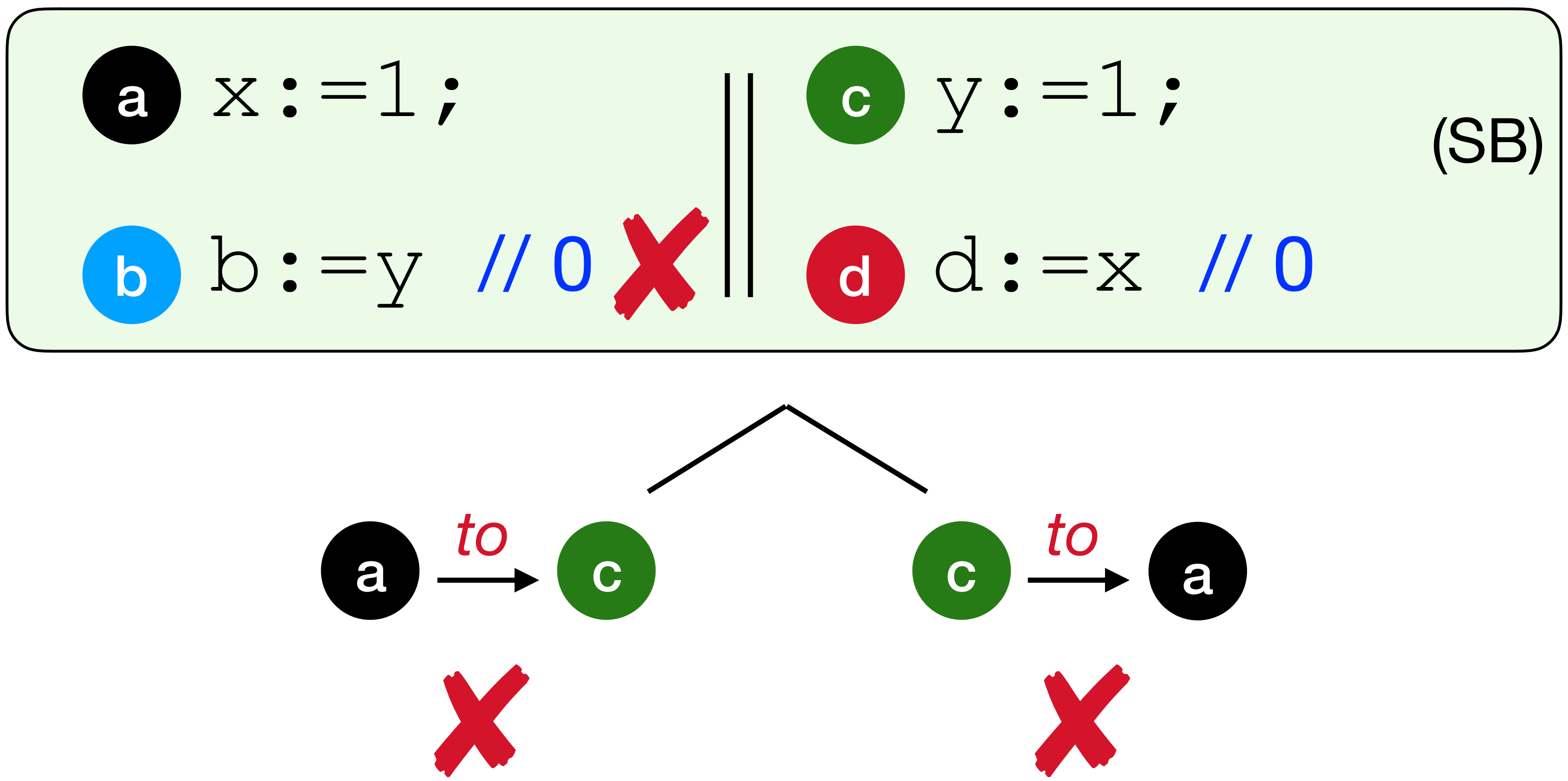
a `x:=1;`	||	c `y:=1;`
b `b:=y` // 0		d `d:=x` // 0 ✗

(SB)



Why Not Linearisability?

- ✗ assumes $<_{\textbf{time}}$ order -- not present under WMC
- ✗ requires ***total*** order *to* on ***all*** events -- not always possible under WMC



Why Not Linearisability?

- ✗ assumes **$<_{time}$** order -- not present under WMC
- ✗ requires ***total*** order *to* on ***all*** events -- not always possible under WMC

```
a  x:=1;        ||   c  y:=1;        (SB)
b  b:=y  // 0        d  d:=x  // 0
```

✗
not linearisable

Why Not Linearisability?

- ✗ assumes $\mathbf{<_{time}}$ order -- not present under WMC
- ✗ requires ***total*** order *to* on ***all*** events -- not always possible under WMC
- ? ***per-location*** linearisability?
  - ✗ cannot model ***weak specs***, e.g. C11, TSO

```
a  x:=1;         ||  c  y:=1;         (SB)
b  b:=y  // 0    ||  d  d:=x  // 0
```

✗ not linearisable

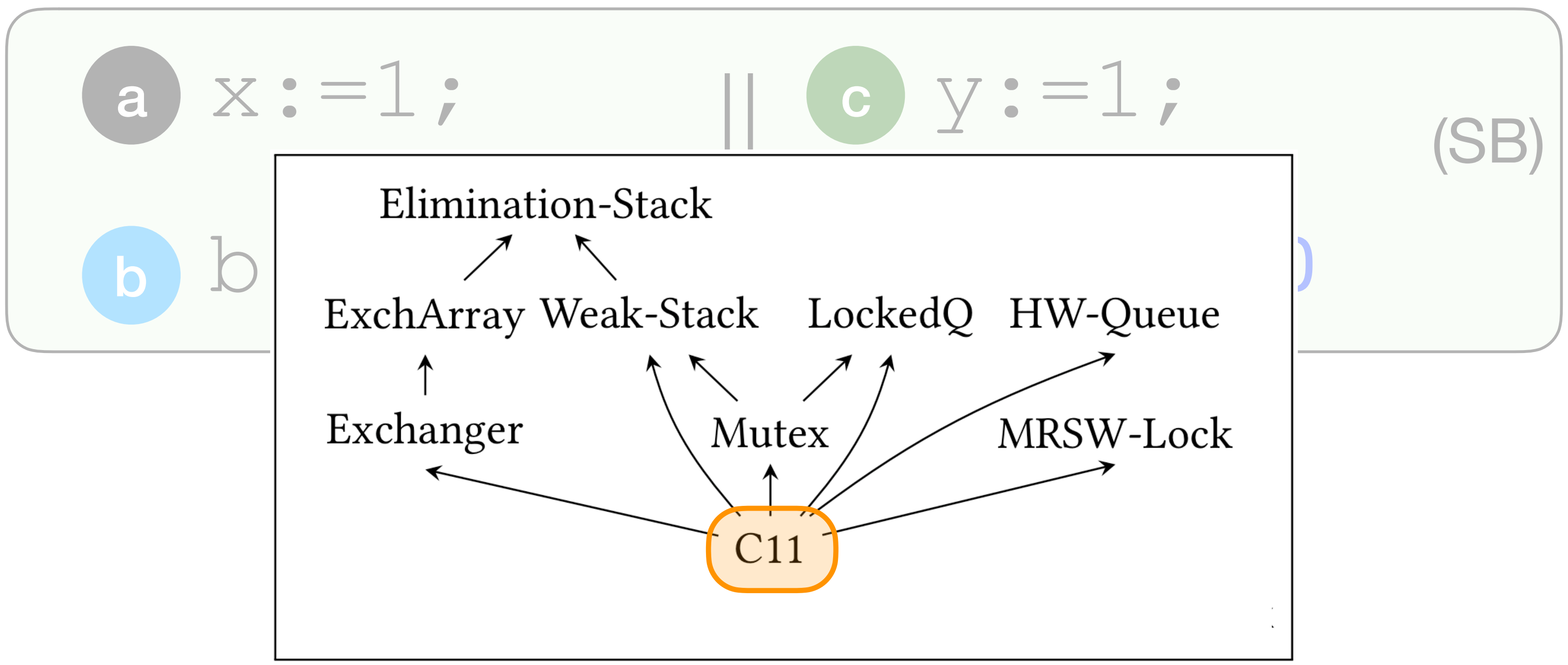
Why Not Linearisability?

✗ assumes $<_{time}$ order -- not present under WMC

✗ requires ***total*** order *to* on ***all*** events -- not always possible under WMC

? ***per-location*** linearisability?

- ✗ cannot model ***weak specs***, e.g. C11, TSO



✗ not linearisable

Our Solution

✔ ***no*** particular ***memory model***

✔ ***no*** $<_{\textbf{time}}$ order

✔ ***no total*** order on events

✔ ***per-library*** specification

Our Solution

- ✔ ***no*** particular ***memory model***
- ✔ ***no*** $<_{time}$ order
- ✔ ***no total*** order on events
- ✔ ***per-library*** specification
  - ➡ set of ***library executions***

$$\{G \mid G \text{ satisfies certain } \textbf{\textit{axioms}}\}$$

library execution

Library Executions

Example: ***Queue*** Library

```
q:=new-queue();
s:=new-stack();

enq(q,1);    ||  a:=pop(s);
push(s,2)    ||  if(a==2)
             ||     b:=deq(q) // may read 1 or empty (⊥)
```

$$G_{queue} = < E , po , so , hb >$$

Library Executions

Example: ***Queue*** Library

```
q:=new-queue();
s:=new-stack();

enq(q,1);     || a:=pop(s);
push(s,2)     || if(a==2)
              ||    b:=deq(q) // may read 1 or empty (⊥)
```

$$G_{queue} = < E, po\ ,\ so\ ,\ hb >$$

events

n new-queue(q)

e enq(q, 1)

d deq(q, 1)

Library Executions

Example: ***Queue*** Library

```
q:=new-queue();
s:=new-stack();

enq(q,1);    ||  a:=pop(s);
push(s,2)    ||  if(a==2)
             ||    b:=deq(q) // may read 1 or empty (⊥)
```

$$G_{queue} = \langle E, po , so , hb \rangle$$

events

n new-queue(q)

e enq(q, 1)

d deq(q, 1)

n new-queue(q)

e enq(q, 1)

d deq(q, ⊥)

Library Executions

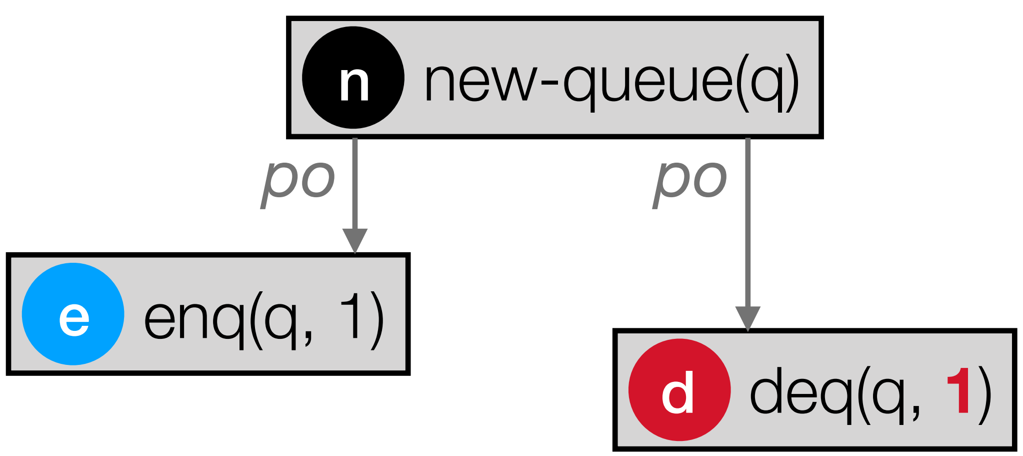
Example: ***Queue*** Library

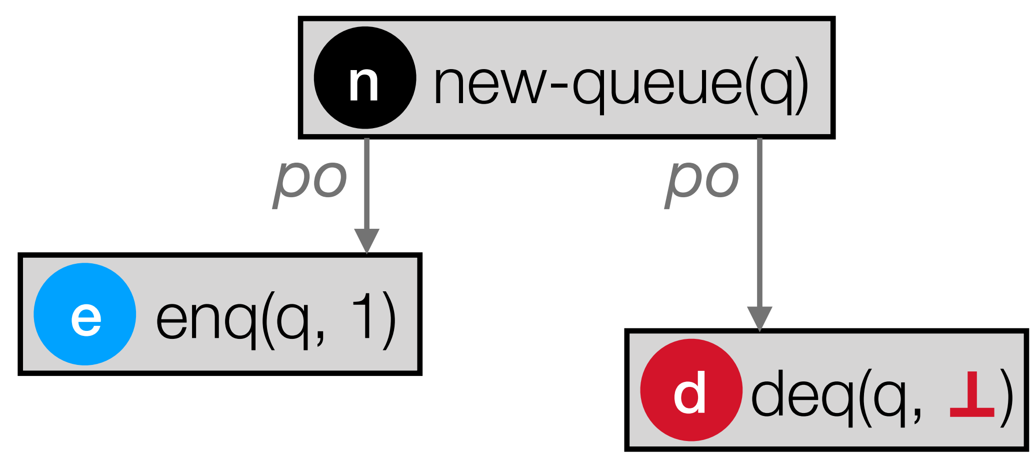
```
q:=new-queue();
s:=new-stack();

enq(q,1);   || a:=pop(s);
push(s,2)   || if(a==2)
            ||   b:=deq(q) // may read 1 or empty (⊥)
```

$$G_{queue} = < E, po, so, hb >$$

events

program-order





Library Executions

Example: ***Queue*** Library

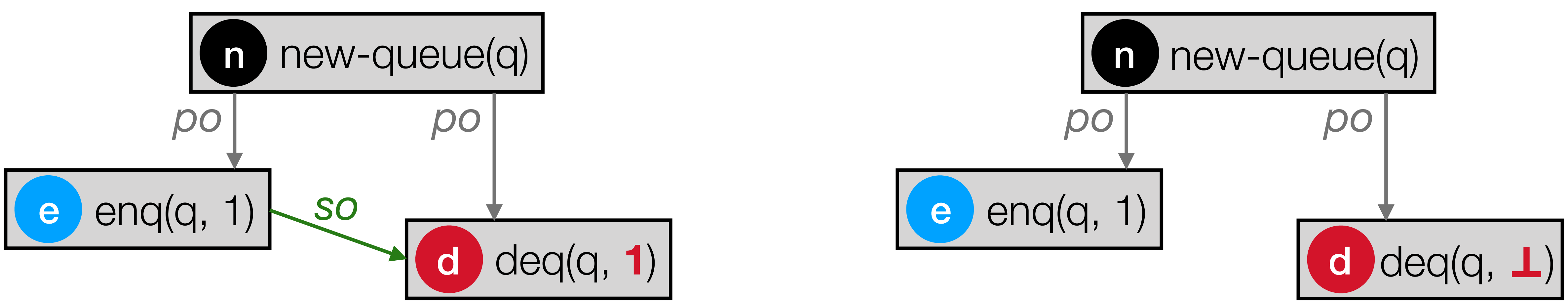
```
q:=new-queue();
s:=new-stack();

enq(q,1);   || a:=pop(s);
push(s,2)   || if(a==2)
            ||    b:=deq(q) // may read 1 or empty (⊥)
```

$$G_{queue} = < E, po, so, hb >$$

events

program-order

synchronisation-order



Library Executions

Example: ***Queue*** Library

```
                q:=new-queue();
                s:=new-stack();
enq(q,1);   ||  a:=pop(s);
push(s,2)   ||  if(a==2)
            ||     b:=deq(q) // may read 1 or empty (⊥)
```

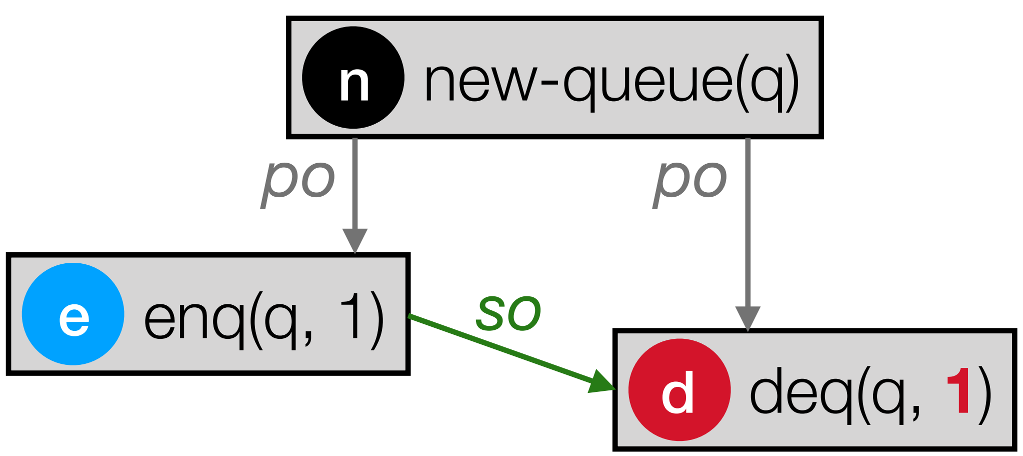
$$G_{queue} = < E, po, so, hb >$$

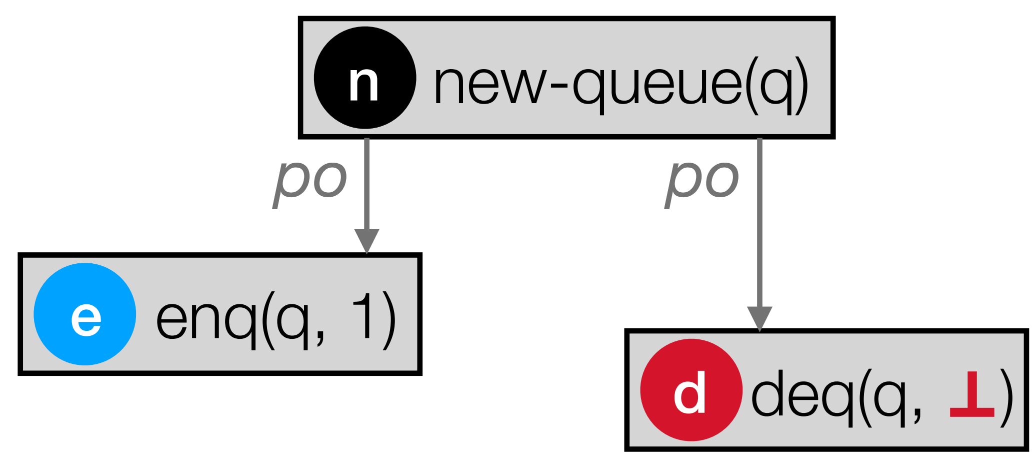
events

program-order

synchronisation-order

happens-before order
(coming soon!)





Library Executions

Example: ***Queue*** Library

```
q:=new-queue();
s:=new-stack();

enq(q,1);   ||  a:=pop(s);
push(s,2)   ||  if(a==2)
            ||     b:=deq(q) // may read 1 ~~or empty (⊥)~~
```

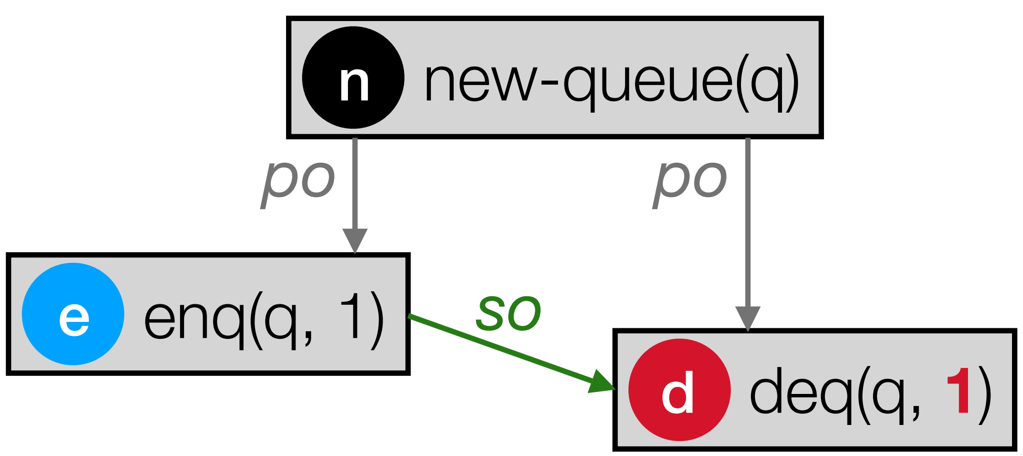
$$G_{queue} = \langle E, po, so, hb \rangle$$

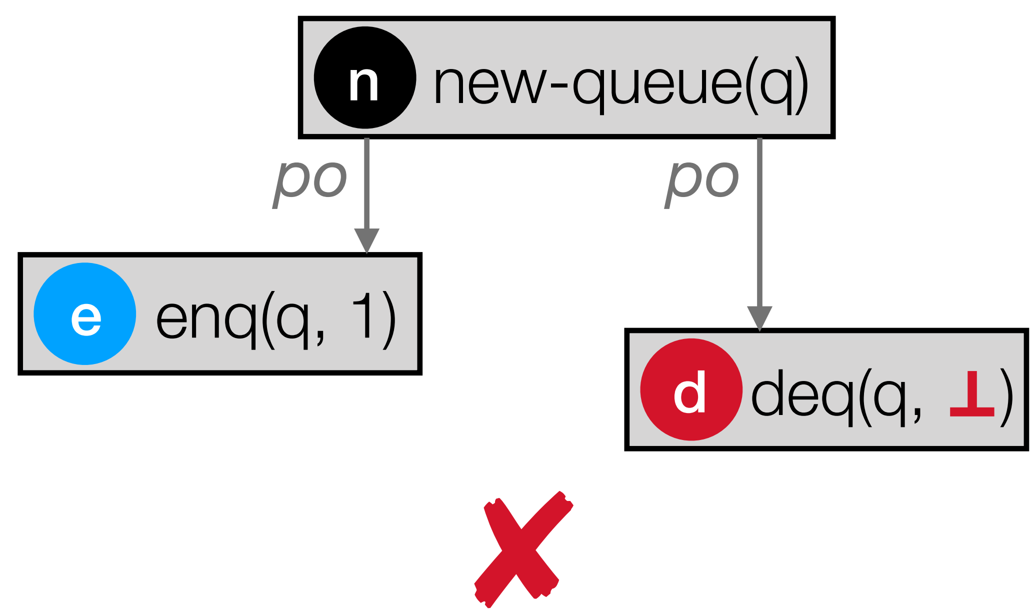
events

program-order

synchronisation-order

happens-before order (coming soon!)



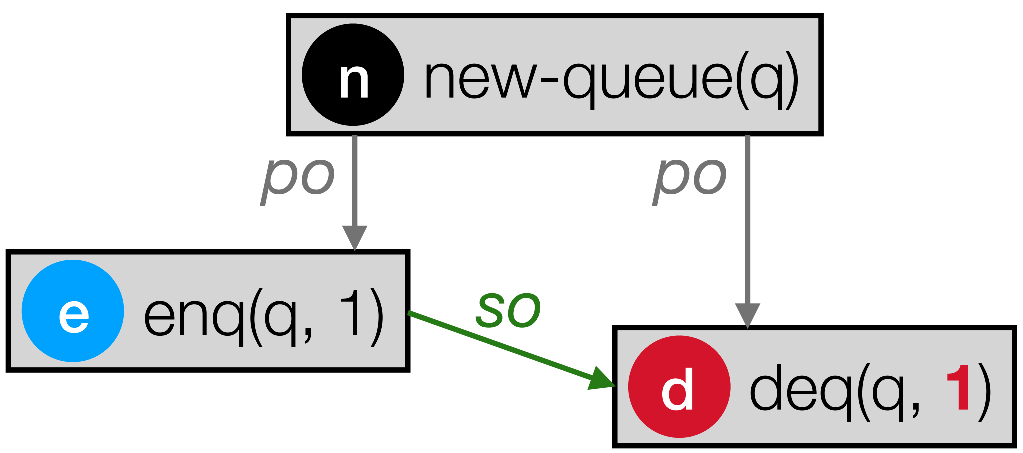


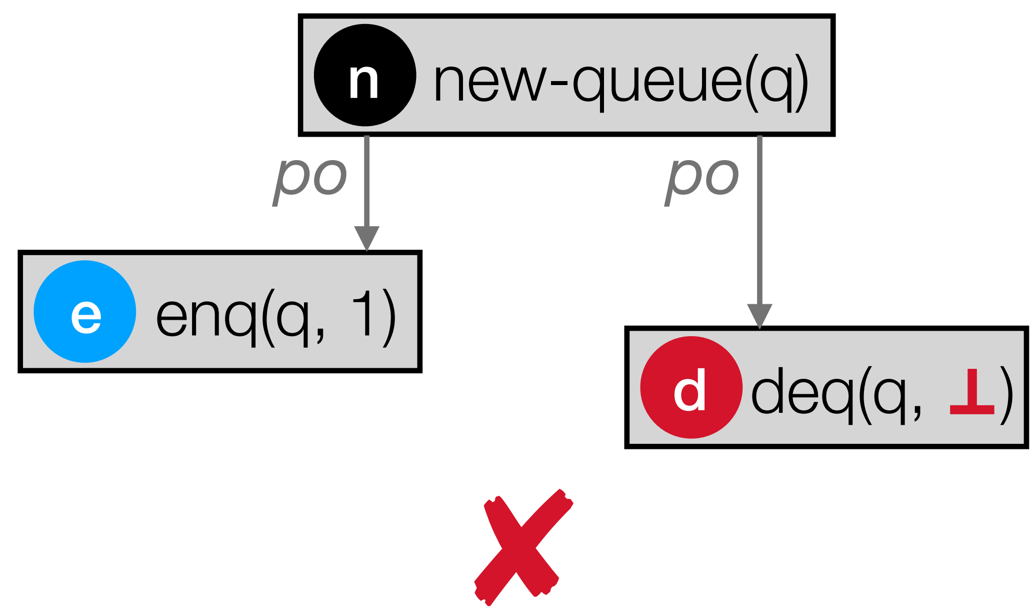
Library Executions

Example: ***Queue*** Library

```
q:=new-queue();
s:=new-stack();

enq(q,1);   ||  a:=pop(s);
push(s,2)   ||  if(a==2)
            ||    b:=deq(q) // may read 1 ~~or empty (⊥)~~
```

How to eliminate this "***incorrect***" behaviour?





Program Executions

$$[\![P]\!] = \{ G_P = \langle E, po, so \rangle \mid \cdots \}$$

semantics of P

execution of P

events

program-order

synchronisation-order
(library-specific)

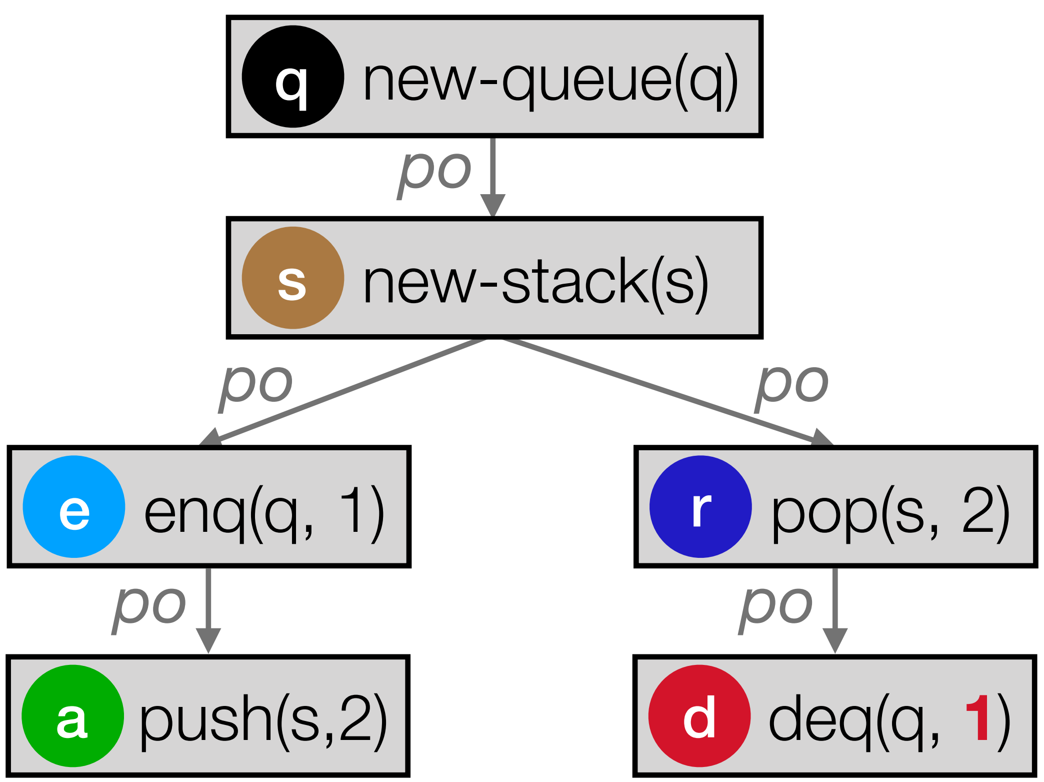
Program Executions

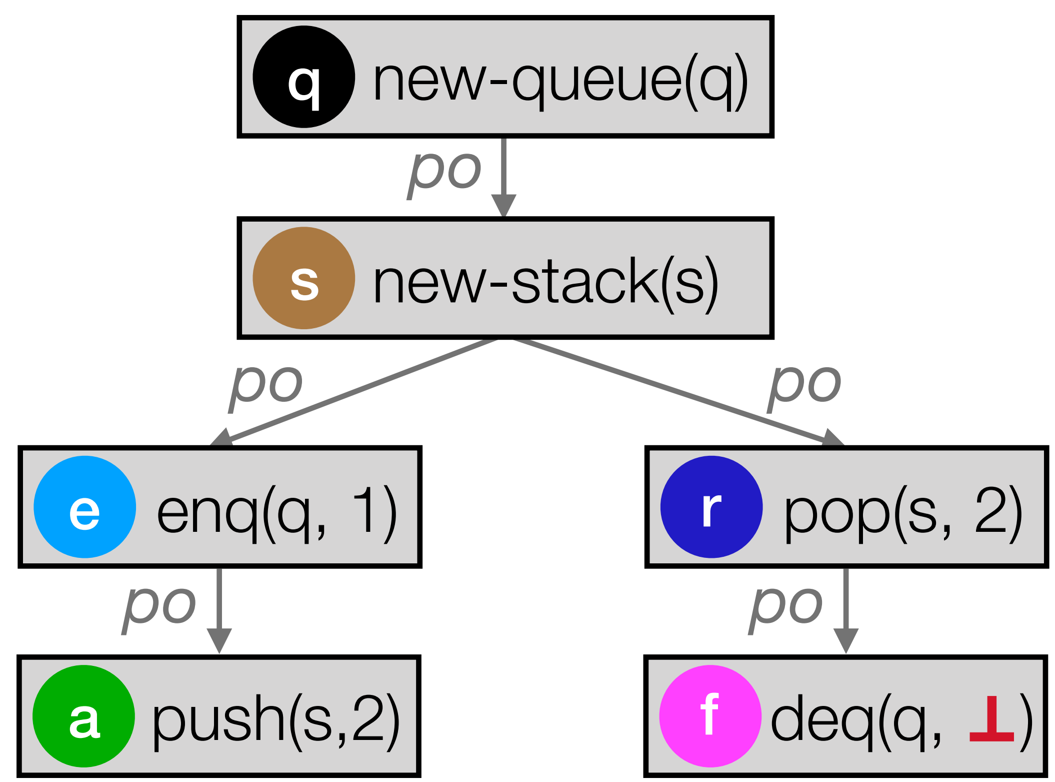
$$[\![P]\!] = \left\{ G_P = < E, po, so > \;\middle|\; \cdots \right\}$$

semantics of P

execution of P

```
        q:=new-queue();
        s:=new-stack();

enq(q,1);   ||  a:=pop(s);
push(s,2)   ||  if(a==2)
            ||     b:=deq(q) // should return 1
```





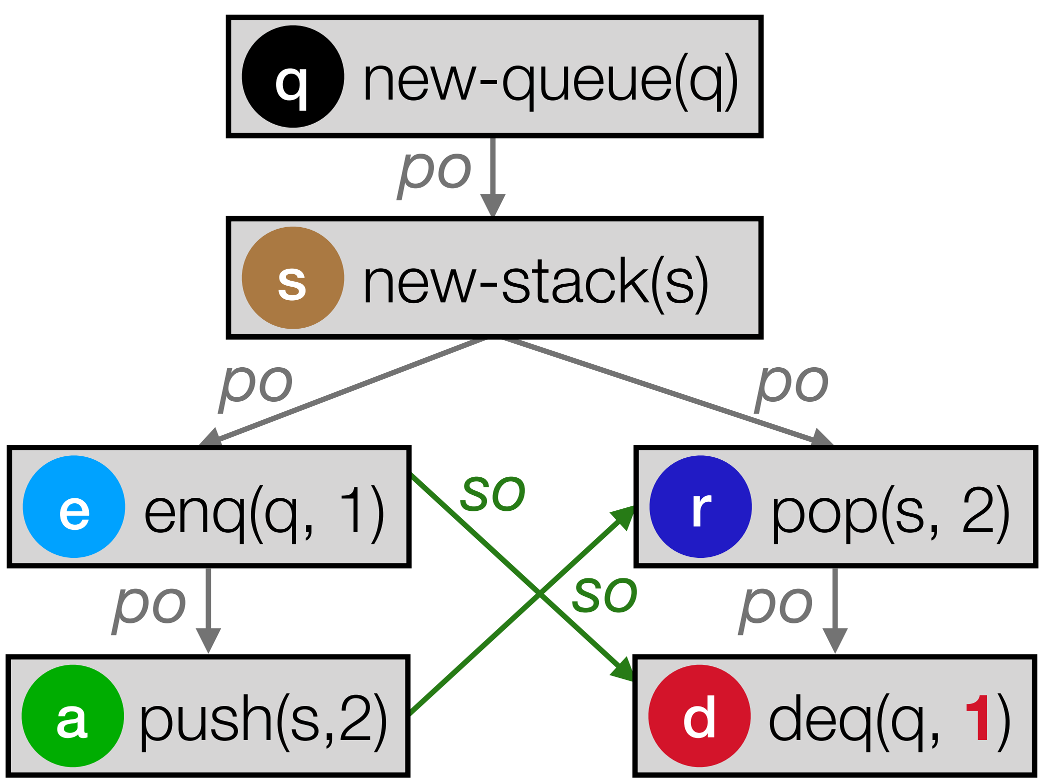
Program Executions

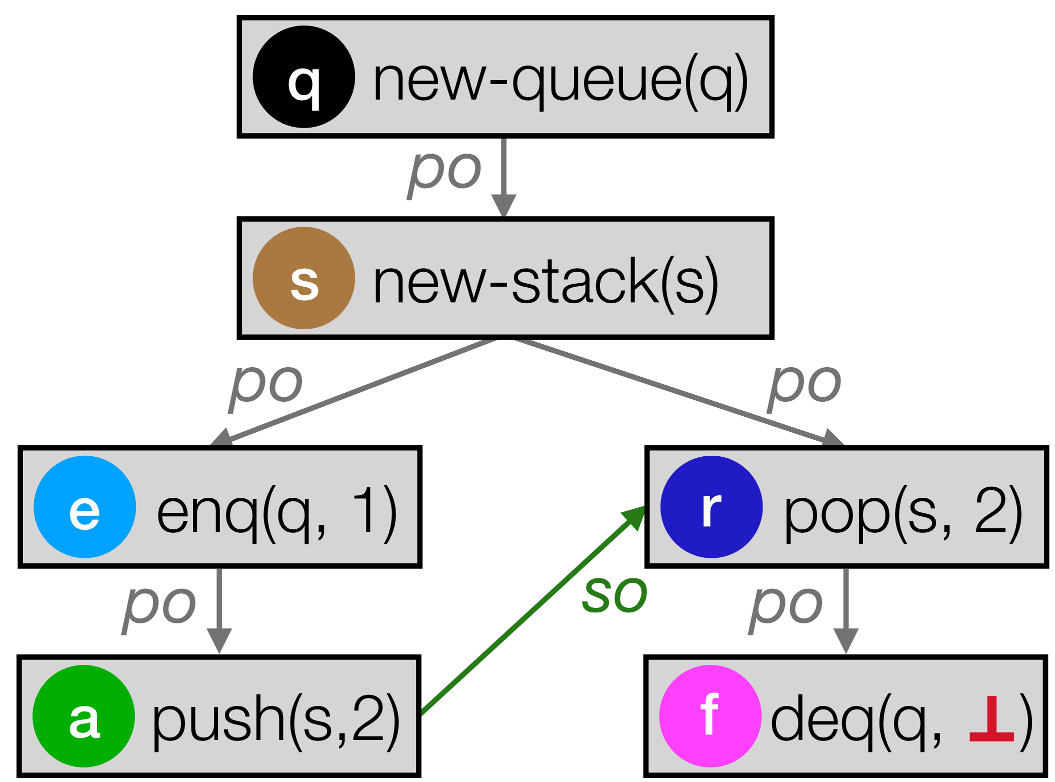
$$[\![P]\!] = \{ G_P = < E , po , so > \mid \cdots \}$$

semantics of P

execution of P

```
    q:=new-queue();
    s:=new-stack();
enq(q,1);   ||  a:=pop(s);
push(s,2)   ||  if(a==2)
            ||    b:=deq(q) // should return 1
```





Program Executions

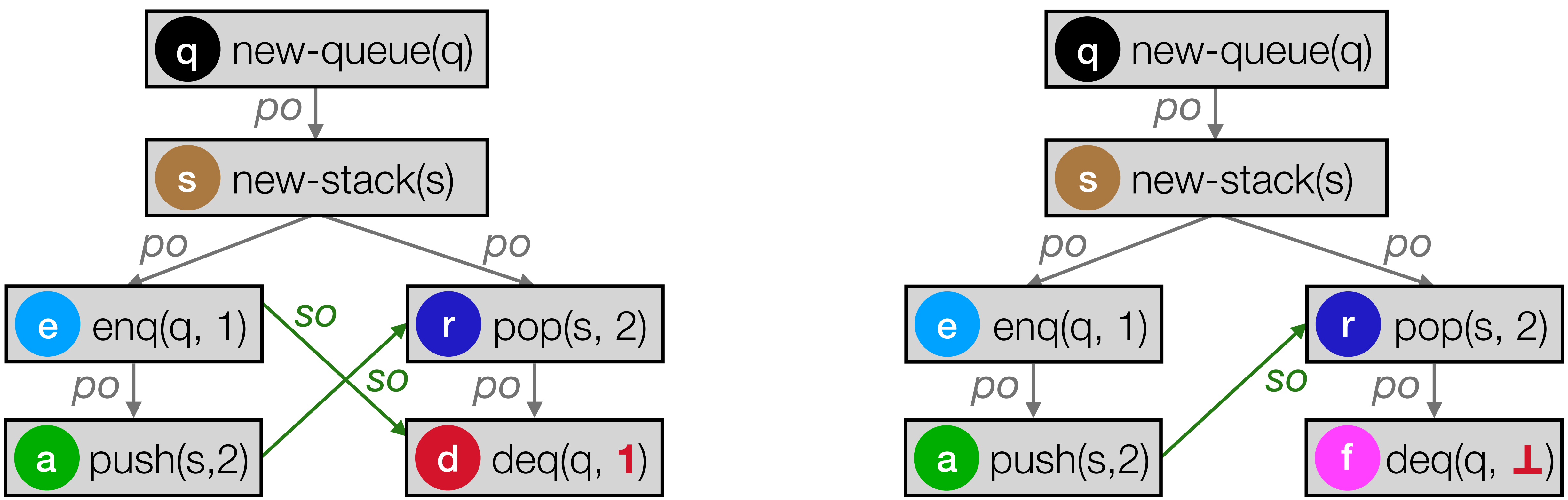
$$[\![P]\!] = \left\{ G_P = < E , po , so > \;\middle|\; \cdots \right\}$$

semantics of *P*

execution of *P*

$hb = (po \cup so)^+$

```
    q:=new-queue();
    s:=new-stack();

enq(q,1);  || a:=pop(s);
push(s,2)  || if(a==2)
           ||    b:=deq(q) // should return 1
```



Program Executions

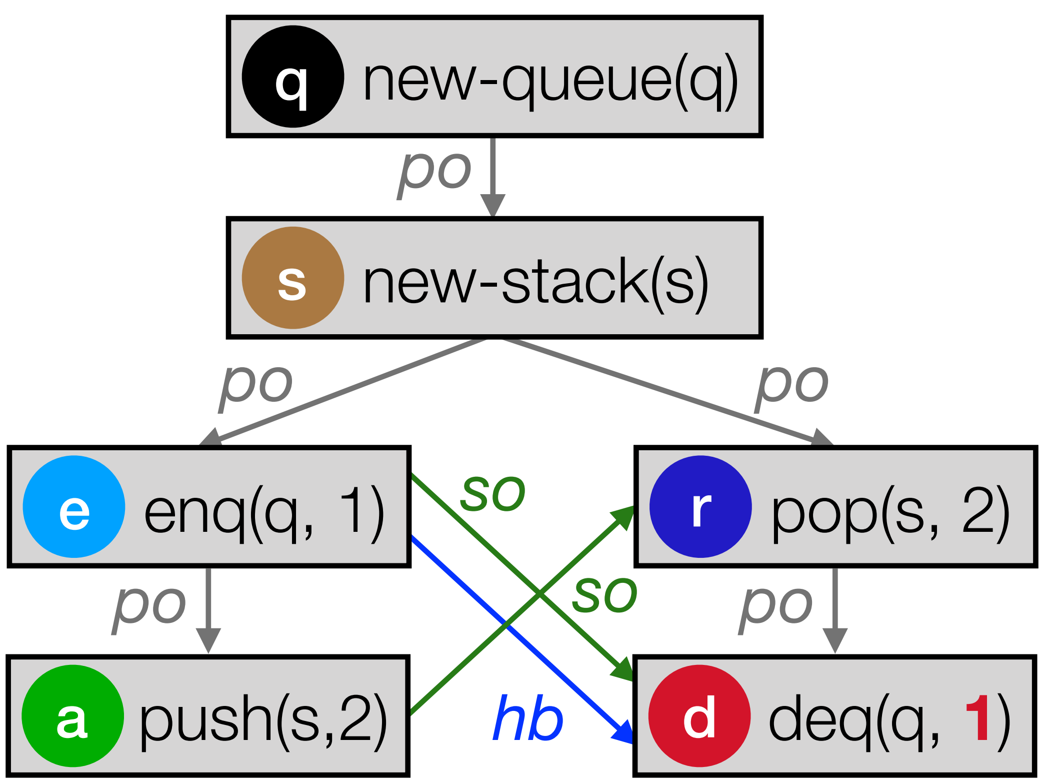
$$[\![P]\!] = \left\{ G_P = < E\,,\, po\,,\, so > \;\middle|\; \cdots \right\}$$

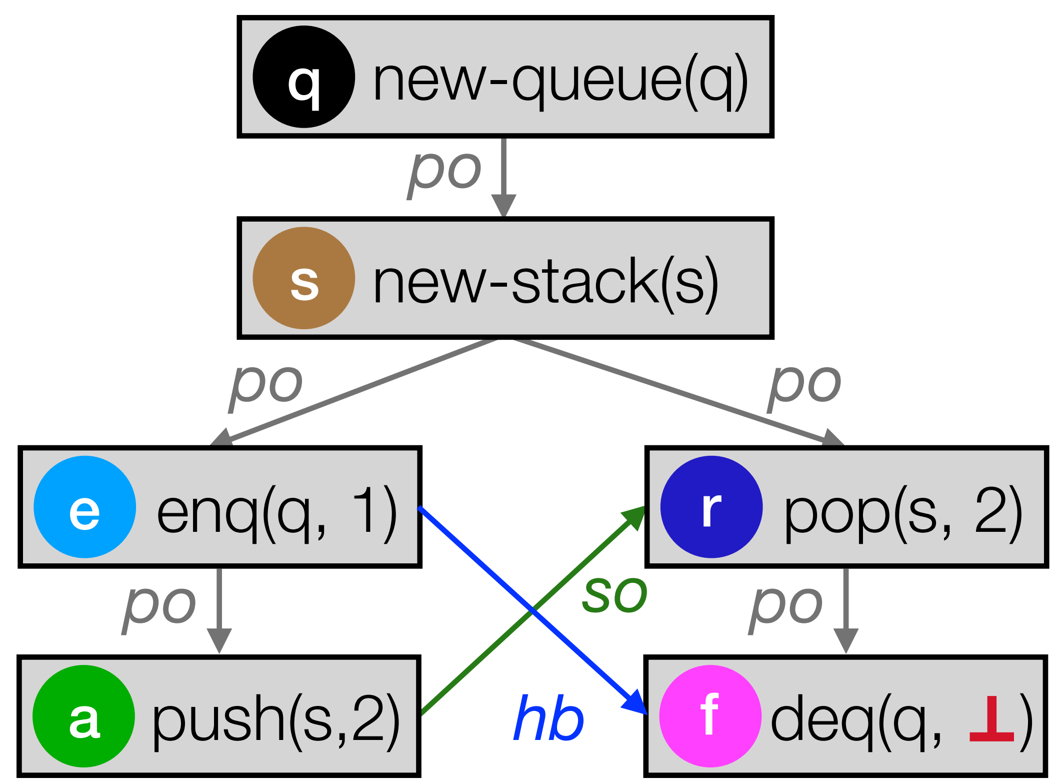
semantics of P

execution of P

$hb = (po \cup so)^+$

```
    q:=new-queue();
    s:=new-stack();

enq(q,1);  ||  a:=pop(s);
push(s,2)  ||  if(a==2)
           ||     b:=deq(q) // should return 1
```





Program Executions

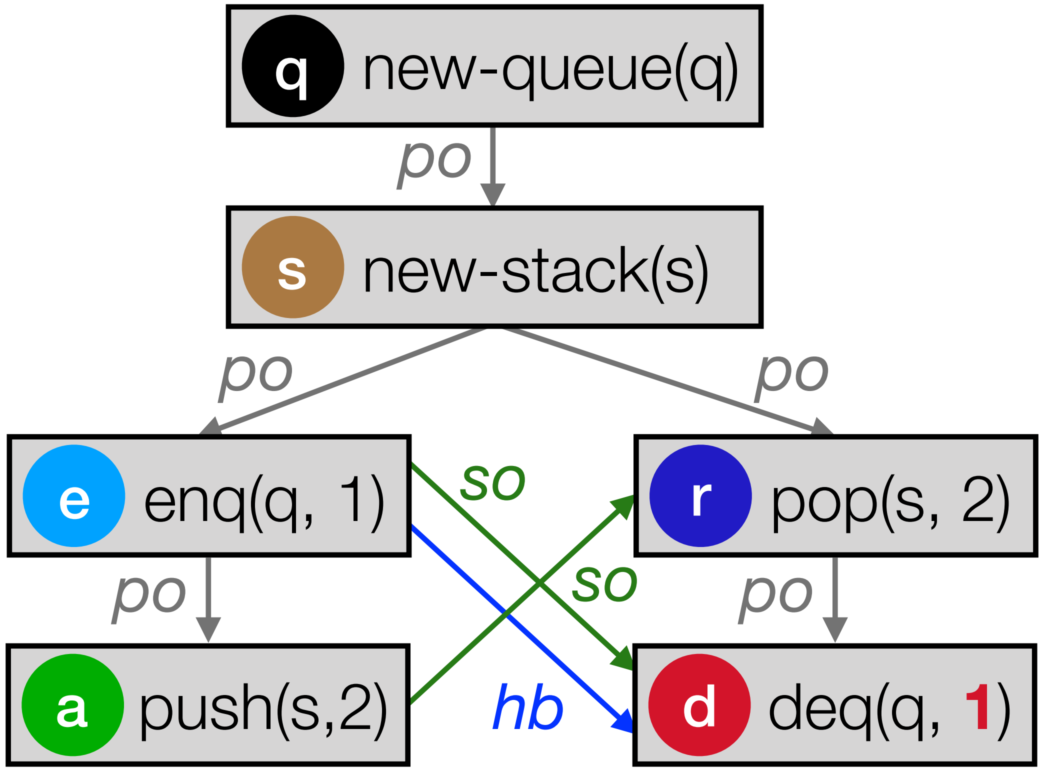
$$[\![P]\!] = \left\{ G_P = < \ E\ ,\ po\ ,\ so\ > \ \middle|\ \cdots \right\}$$

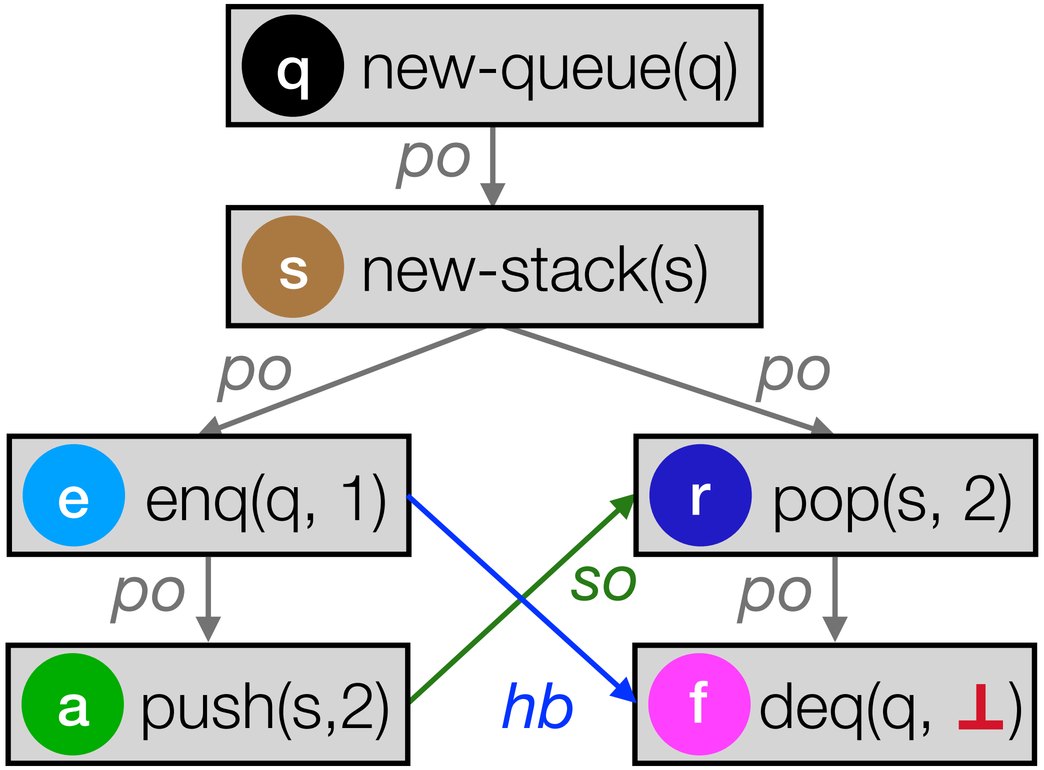
semantics of P

execution of P

$hb = (\ po \cup so\ )^{+}$

allow ***libraries*** to ***constrain each other*** via *hb*!





Program Executions

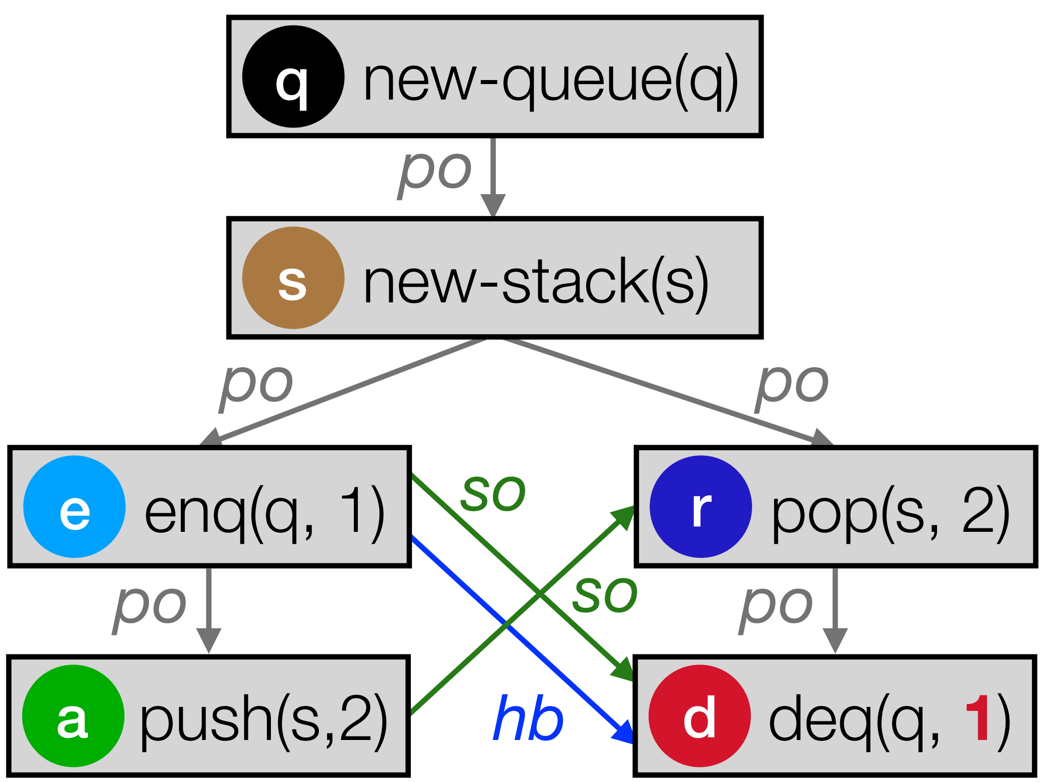
$$[\![P]\!] = \left\{ G_P = < E\,, po\,, so > \;\middle|\; \cdots \right\}$$

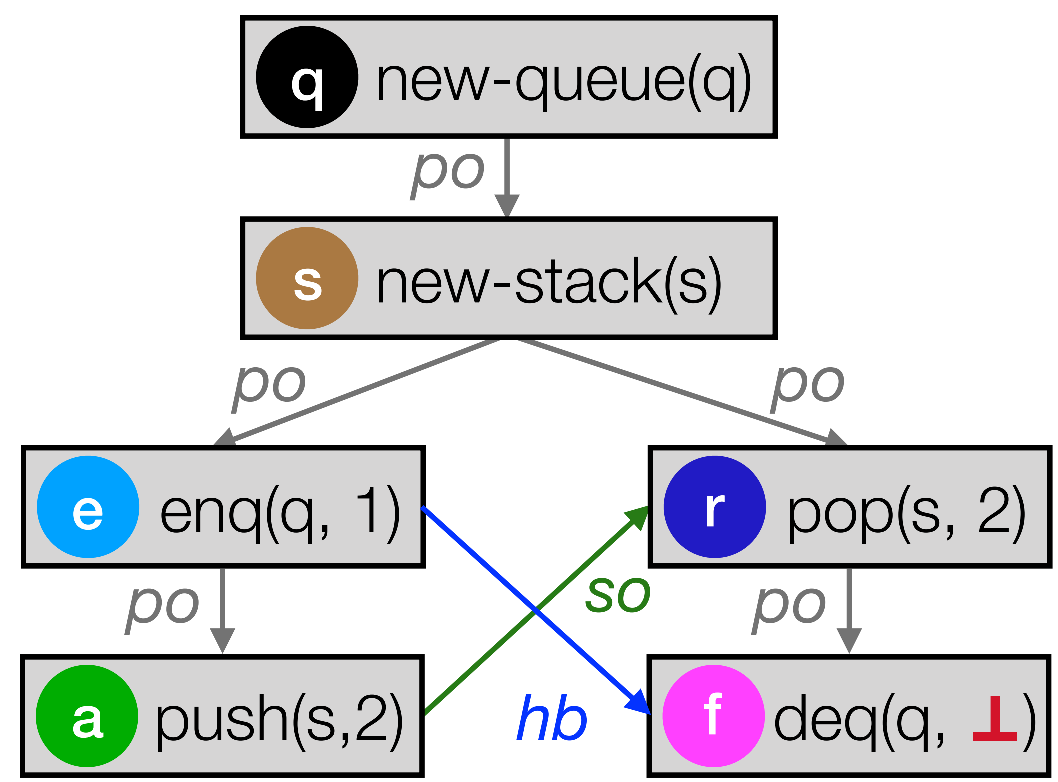
semantics of *P*

execution of *P*

hb = (*po* ∪ *so*)$^+$

allow ***libraries*** to ***constrain each other*** via *hb*!
How? *hb* defined on program executions

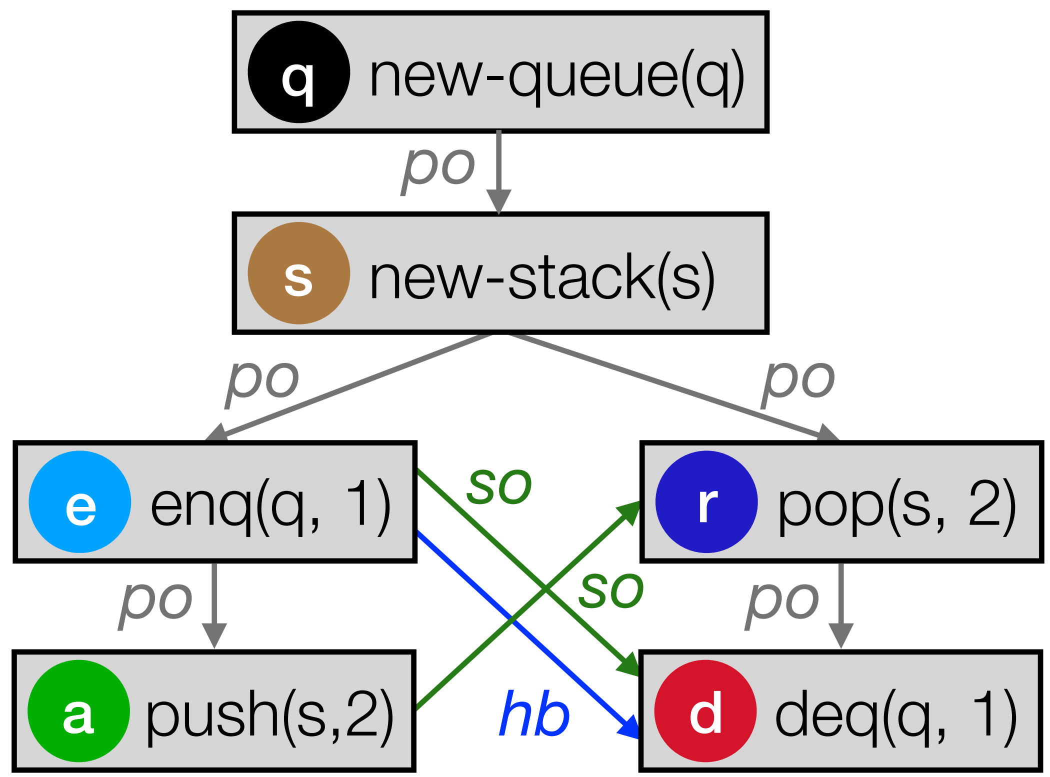




From ***Program*** to ***Library*** Executions

$G_P = <\ E\ ,\ po\ ,\ so\ >$

$hb = (\ po \cup so\ )^+$

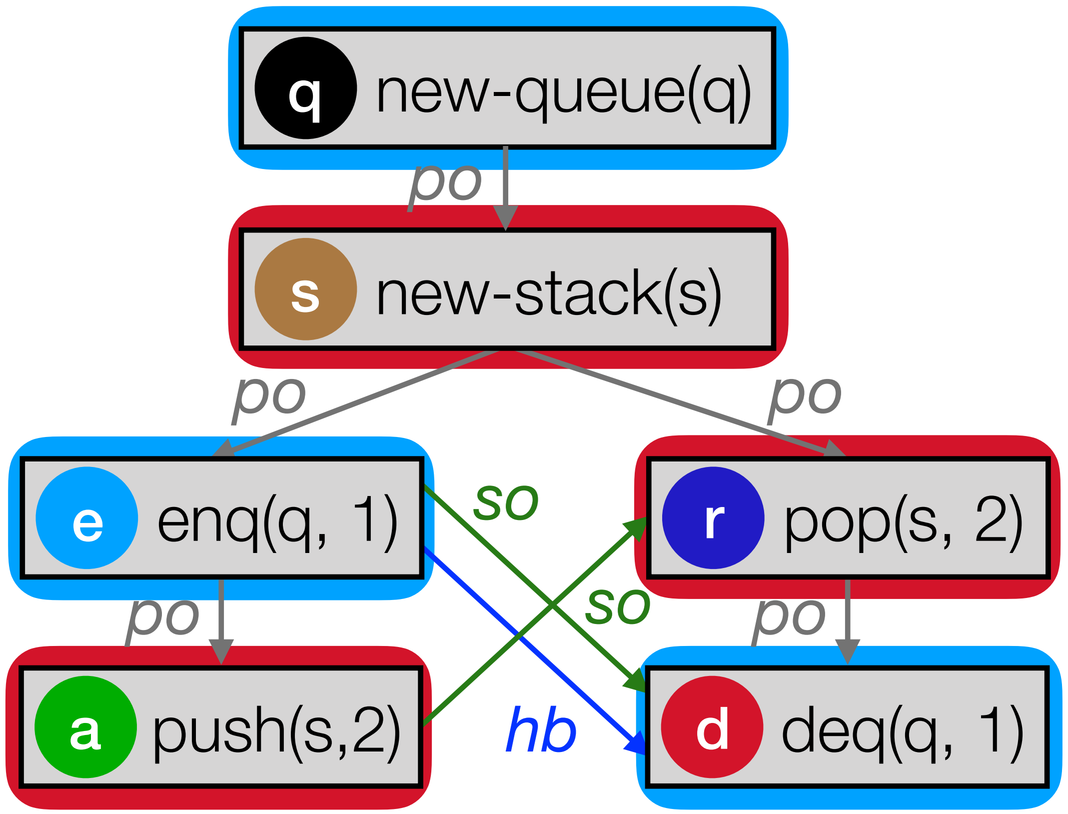


From ***Program*** to ***Library*** Executions

$$G_P = < E, po, so >$$

$$E = E_{queue} \cup E_{stack}$$

$$hb = (po \cup so)^+$$

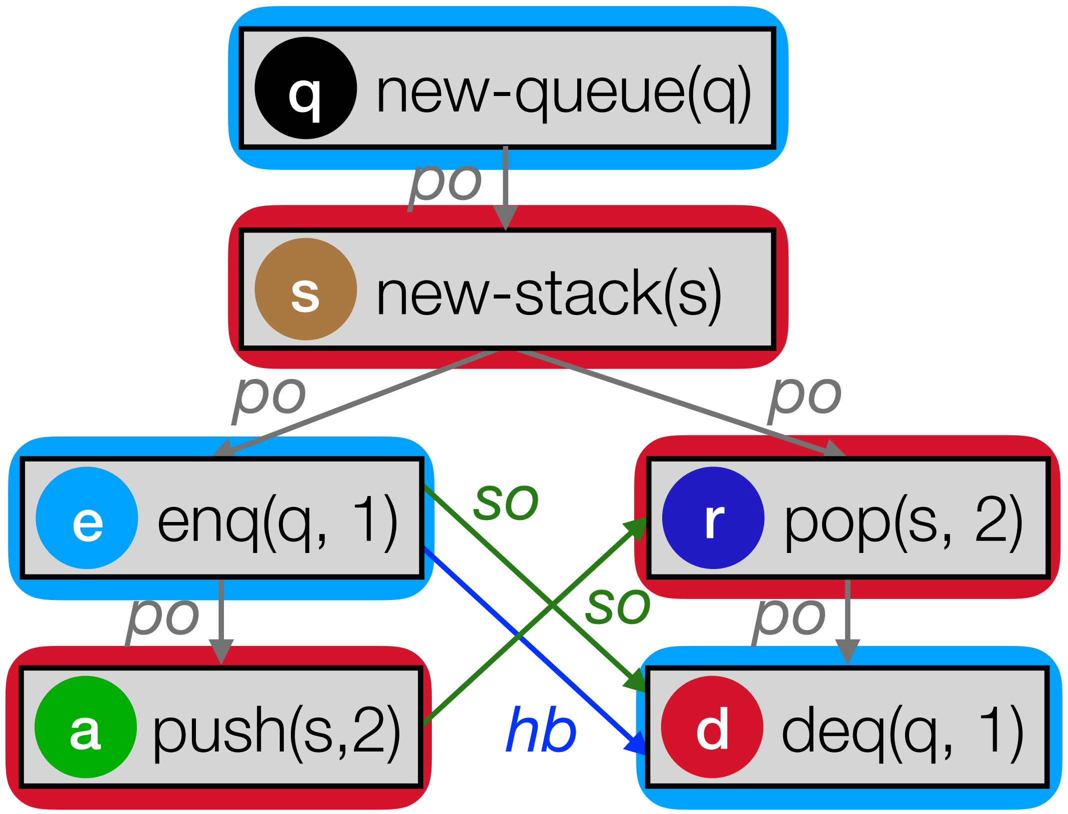


From ***Program*** to ***Library*** Executions

$$G_P = \langle\ E\ ,\ po\ ,\ so\ \rangle$$

$$G_{queue} \oplus G_{stack} \qquad E_{queue} \cup E_{stack}$$

$$hb = (\ po \cup so\ )^+$$

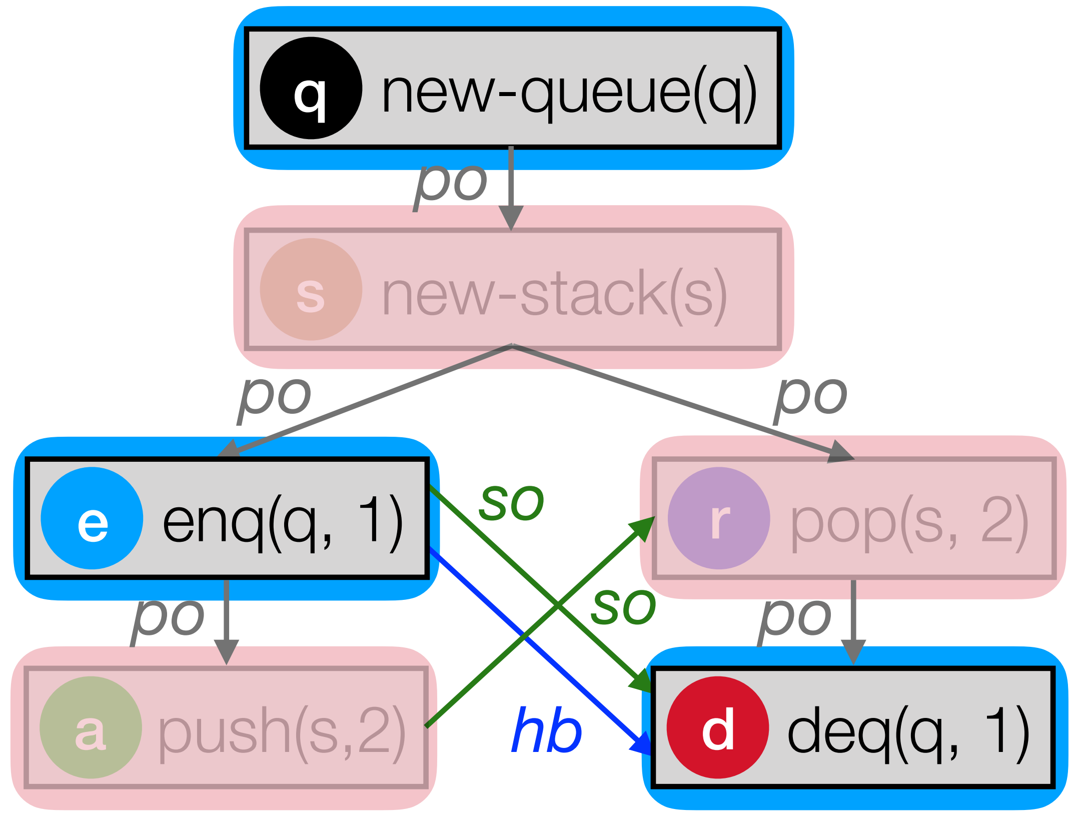


From ***Program*** to ***Library*** Executions

$$G_P = \langle\ E\ ,\ po\ ,\ so\ \rangle$$

$G_{queue} \oplus G_{stack}$ $\qquad$ $E_{queue} \cup E_{stack}$

$hb = (\ po \cup so\ )^+$



$$G_{queue} = \langle\ E_{queue}\ , \qquad\qquad\qquad\qquad \rangle$$

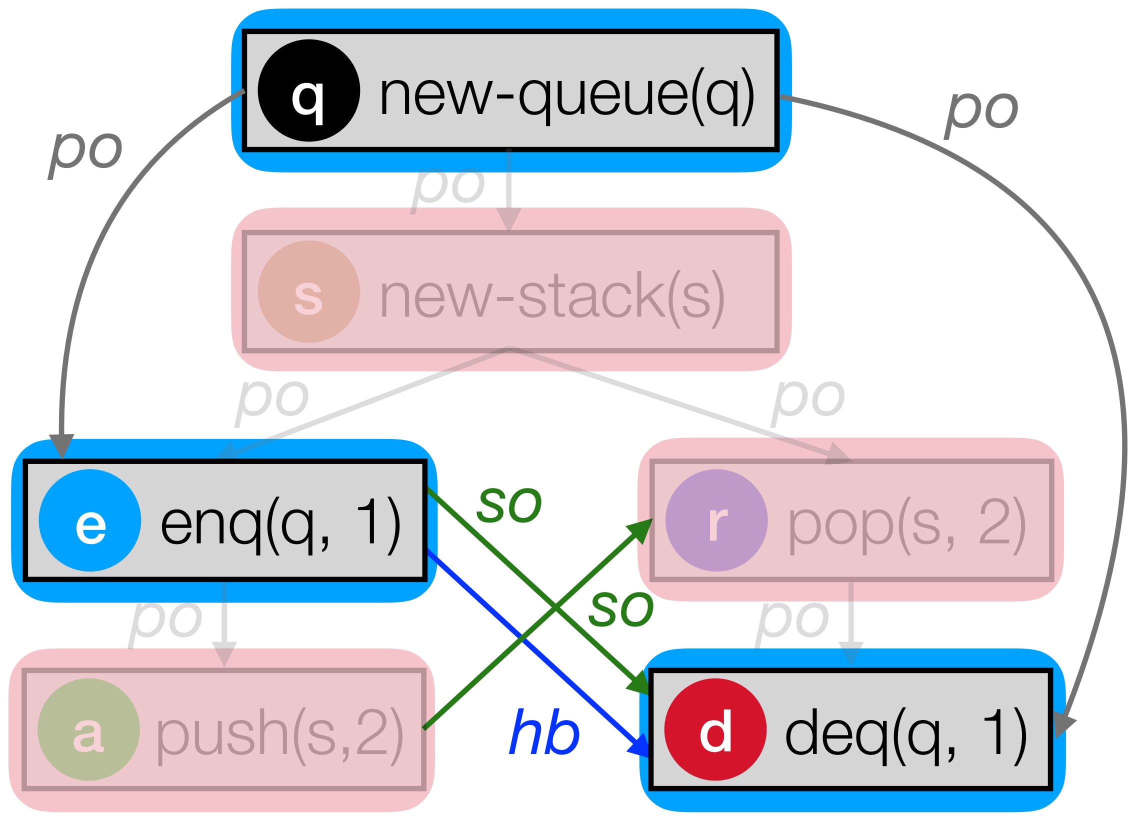
From ***Program*** to ***Library*** Executions

$$G_P = \langle\ E\ ,\ po\ ,\ so\ \rangle$$

$$G_P = G_{queue} \oplus G_{stack}$$

$$E = E_{queue} \cup E_{stack}$$

$$hb = (\ po \cup so\ )^+$$



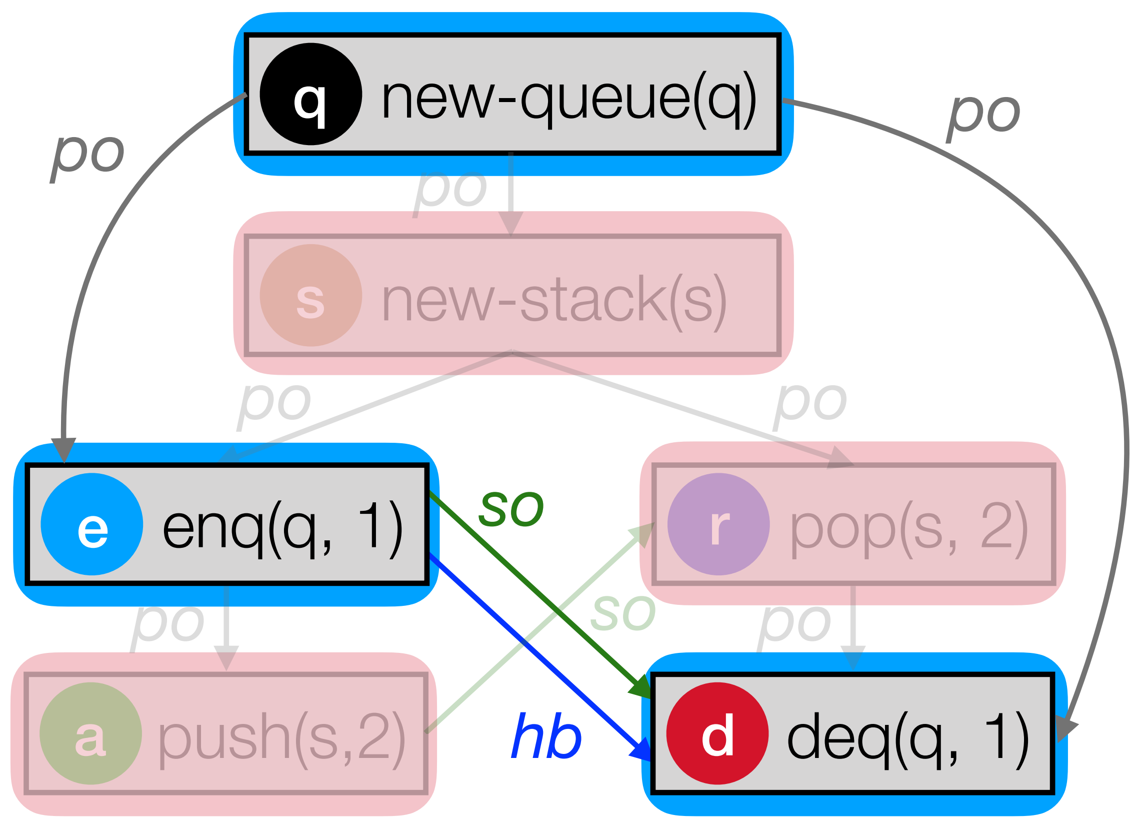
$$G_{queue} = \langle\ E_{queue}\ ,\ po_{queue}\ ,\ \qquad\qquad \rangle$$

From ***Program*** to ***Library*** Executions

$$G_P = \langle\ E\ ,\ po\ ,\ so\ \rangle$$

$$G_{queue} \oplus G_{stack} \qquad E_{queue} \cup E_{stack}$$

$hb = (\ po \cup so\ )^+$



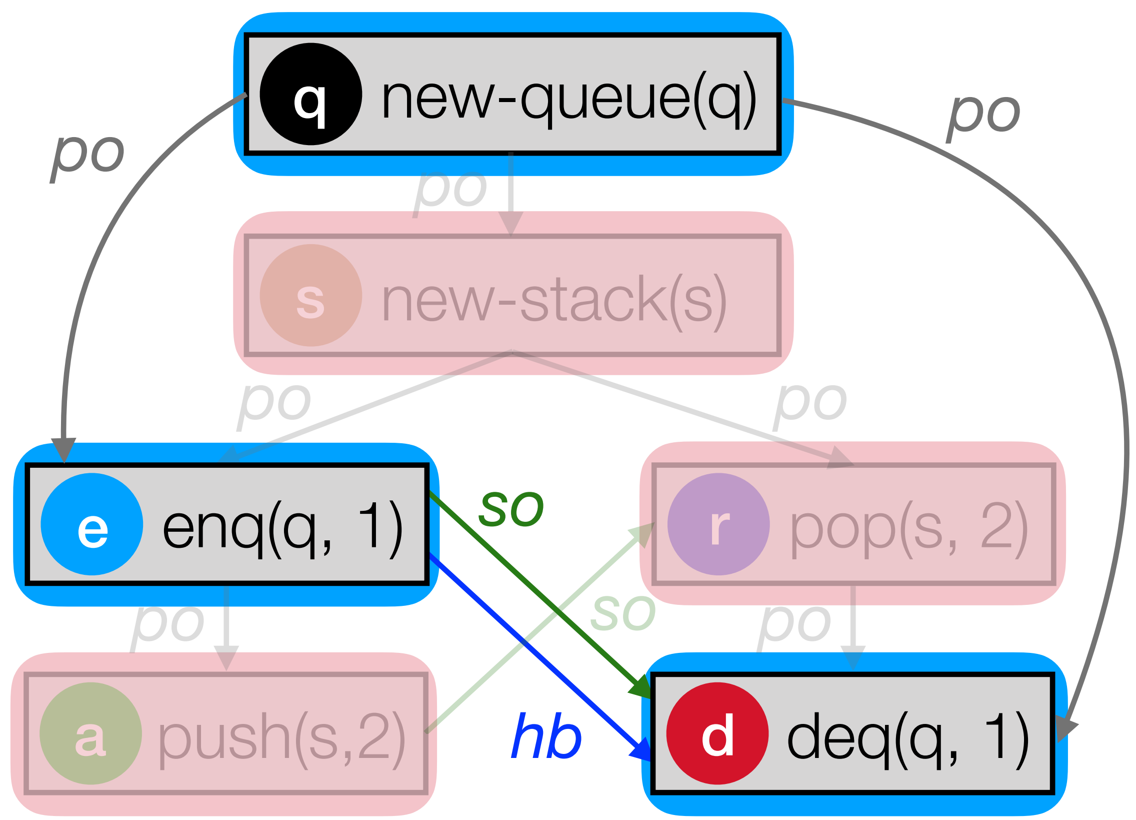
$$G_{queue} = \langle\ E_{queue}\ ,\ po_{queue}\ ,\ so_{queue}\ ,\qquad\qquad \rangle$$

From *Program* to *Library* Executions

$$G_P = \langle\ E\ ,\ po\ ,\ so\ \rangle$$

$$G_{queue} \oplus G_{stack} \qquad\qquad E_{queue} \cup E_{stack}$$

$$hb = (\ po \cup so\ )^+$$



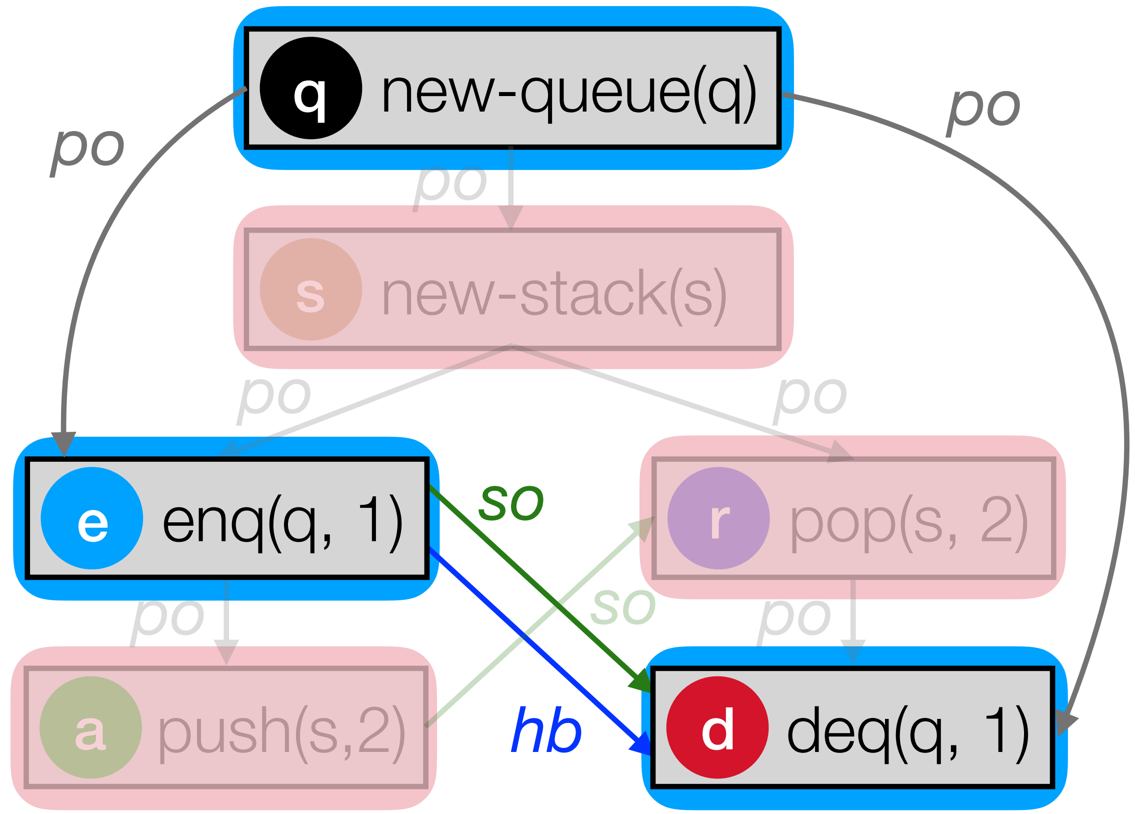
$$G_{queue} = \langle\ E_{queue}\ ,\ po_{queue}\ ,\ so_{queue}\ ,\ hb?\ \rangle$$

From ***Program*** to ***Library*** Executions

$$G_P = \langle\ E, po, so\ \rangle$$

$$G_{queue} \oplus G_{stack} \qquad E_{queue} \cup E_{stack}$$

$hb = (po \cup so)^+$

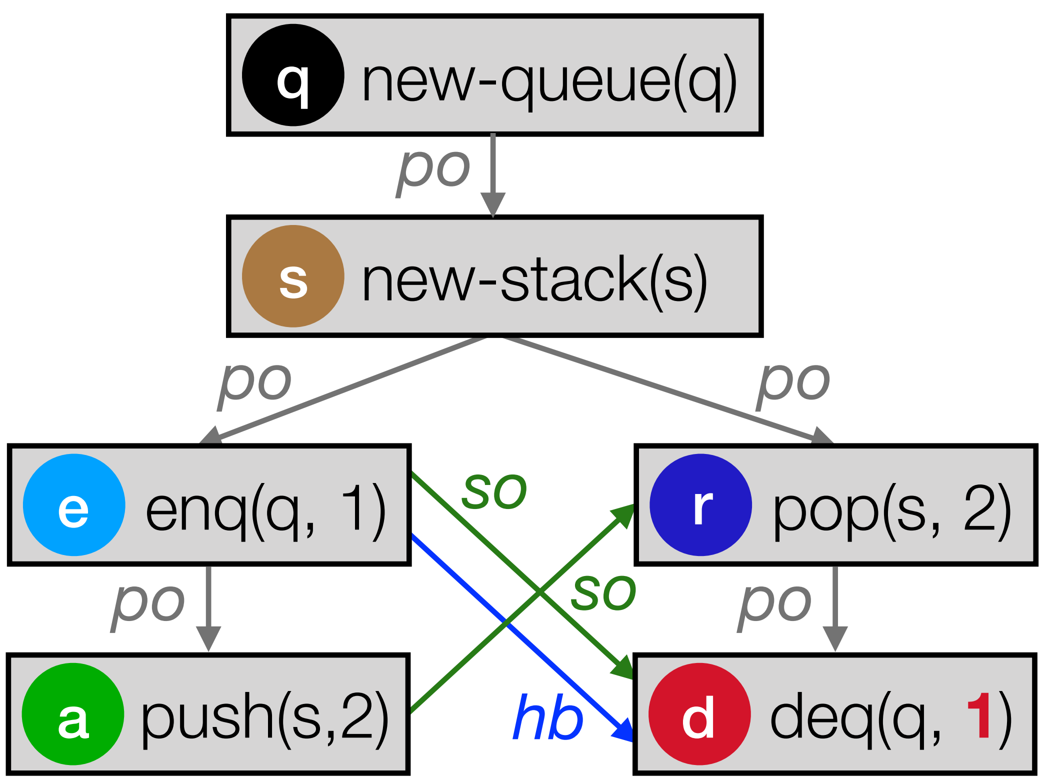


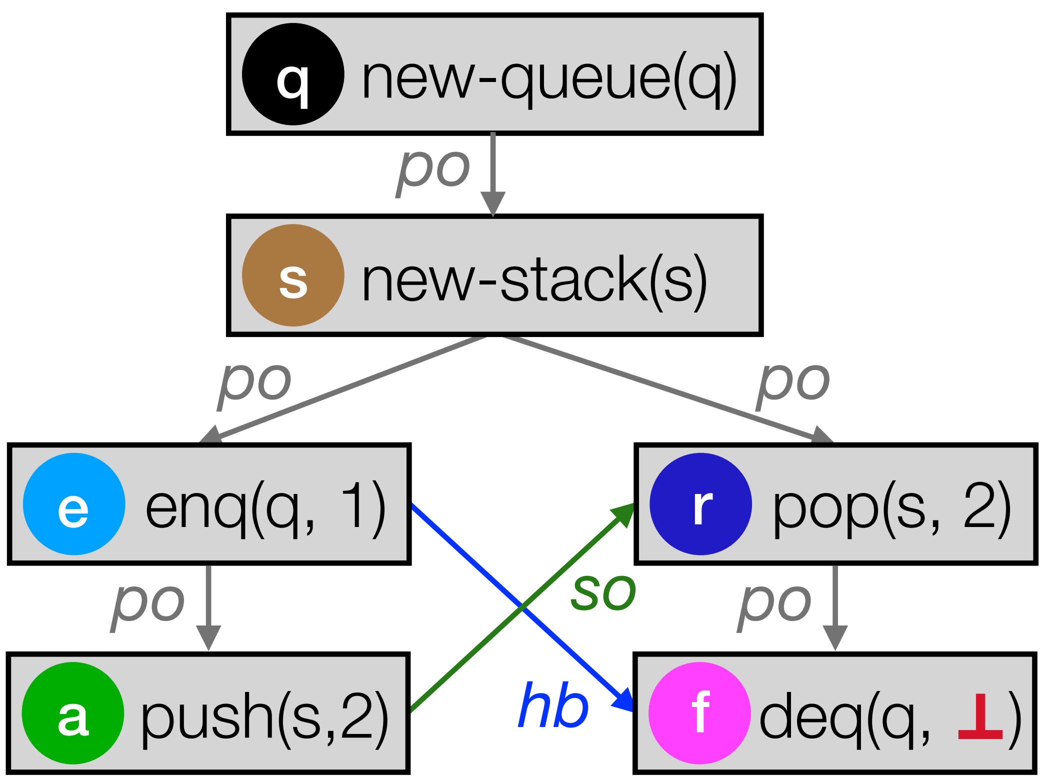
$$G_{queue} = \langle\ E_{queue}, po_{queue}, so_{queue}, hb?\ \rangle$$

$$hb_{queue}$$

Example Revisited

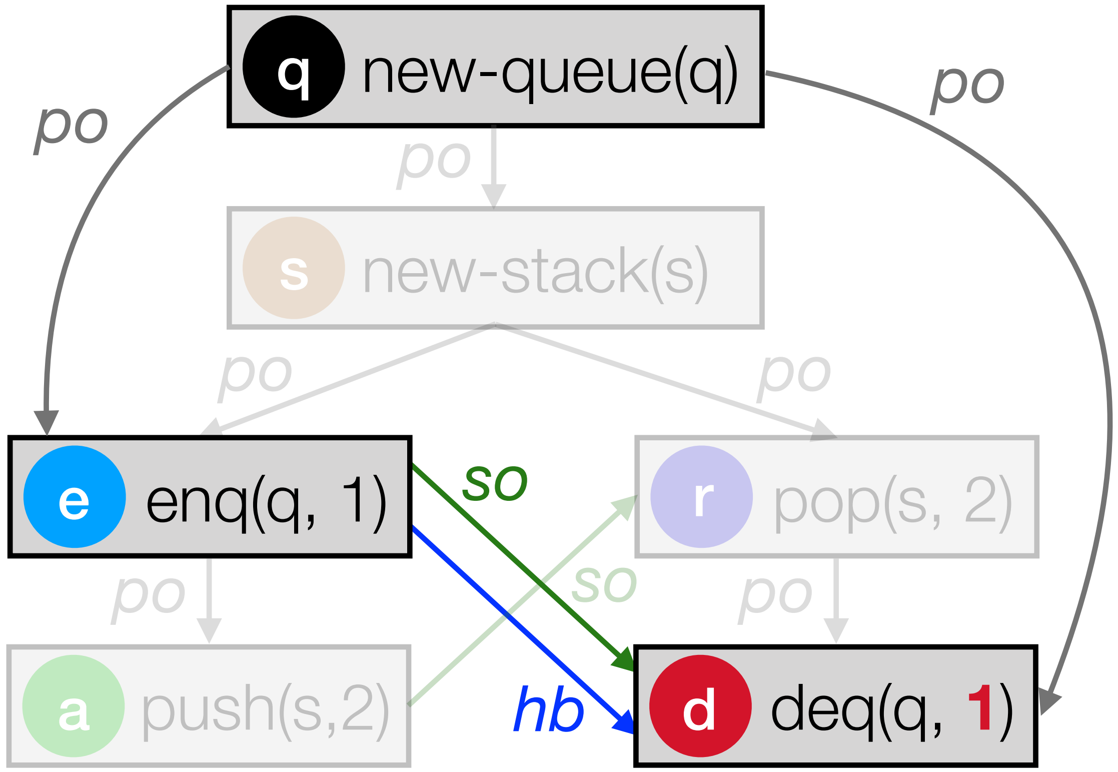
```
q:=new-queue();
s:=new-stack();

enq(q,1);     ||  a:=pop(s);
push(s,2)     ||  if(a==2)
              ||    b:=deq(q)  // should return 1
```

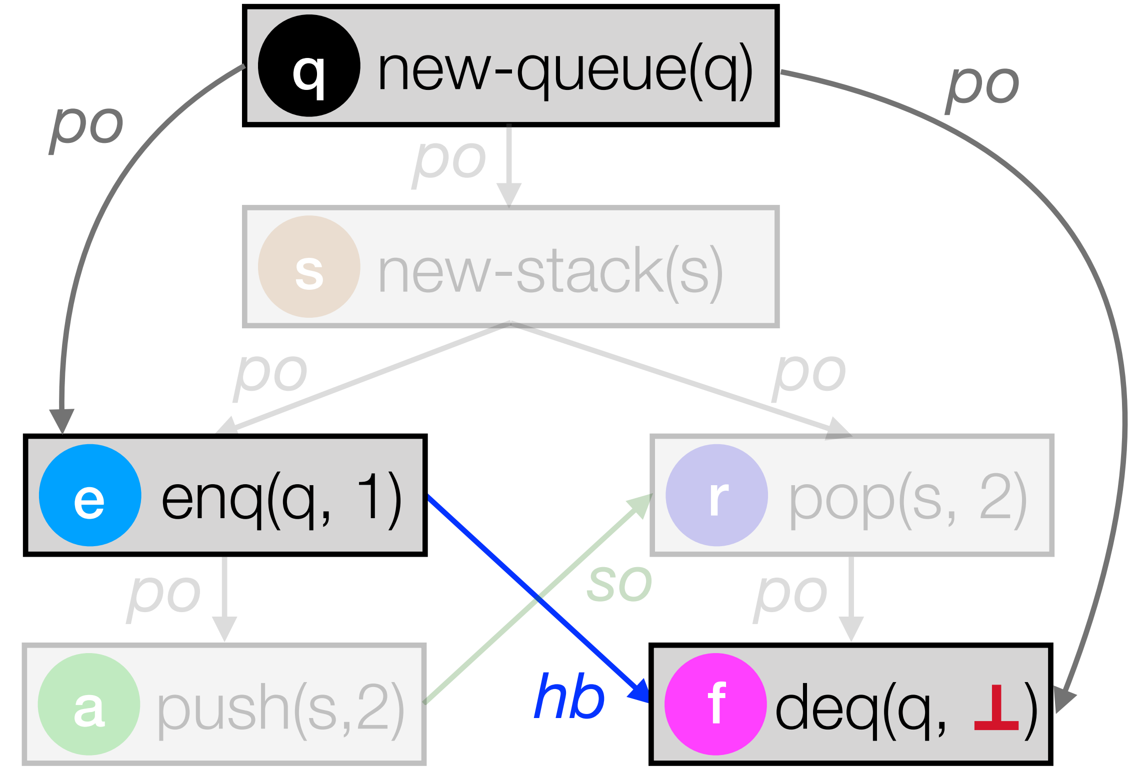




Example Revisited

```
q:=new-queue();
s:=new-stack();

enq(q,1);   || a:=pop(s);
push(s,2)   || if(a==2)
            ||   b:=deq(q)  // should return 1
```



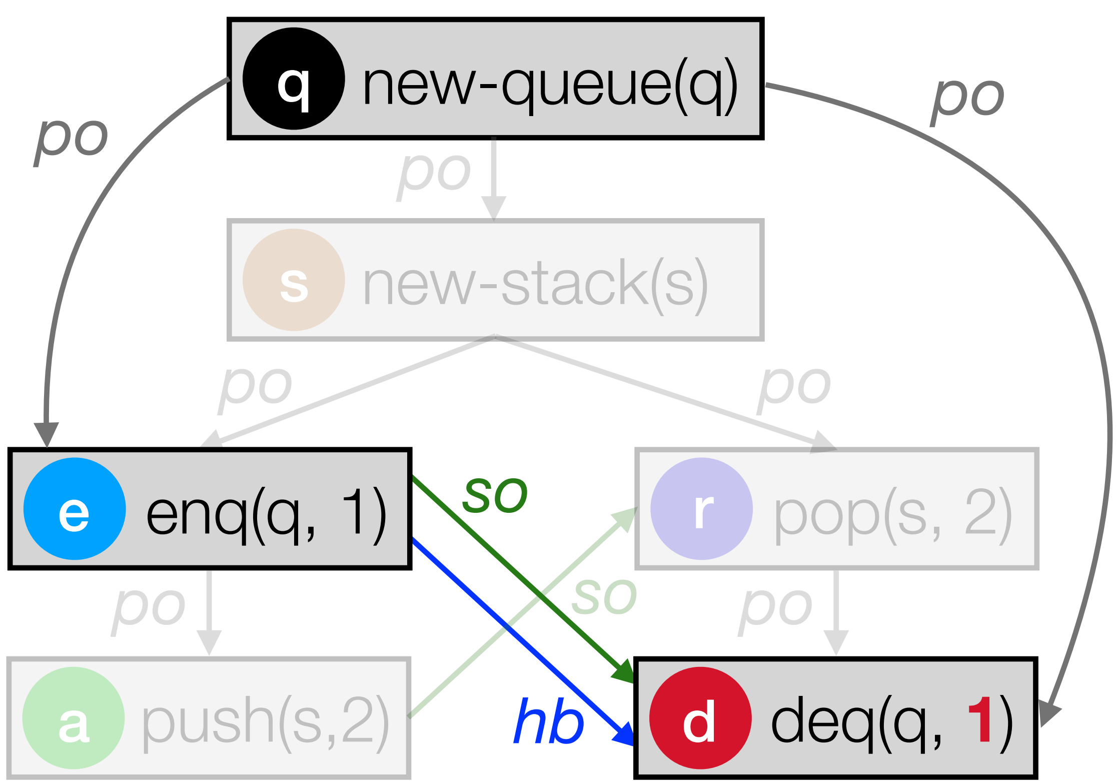


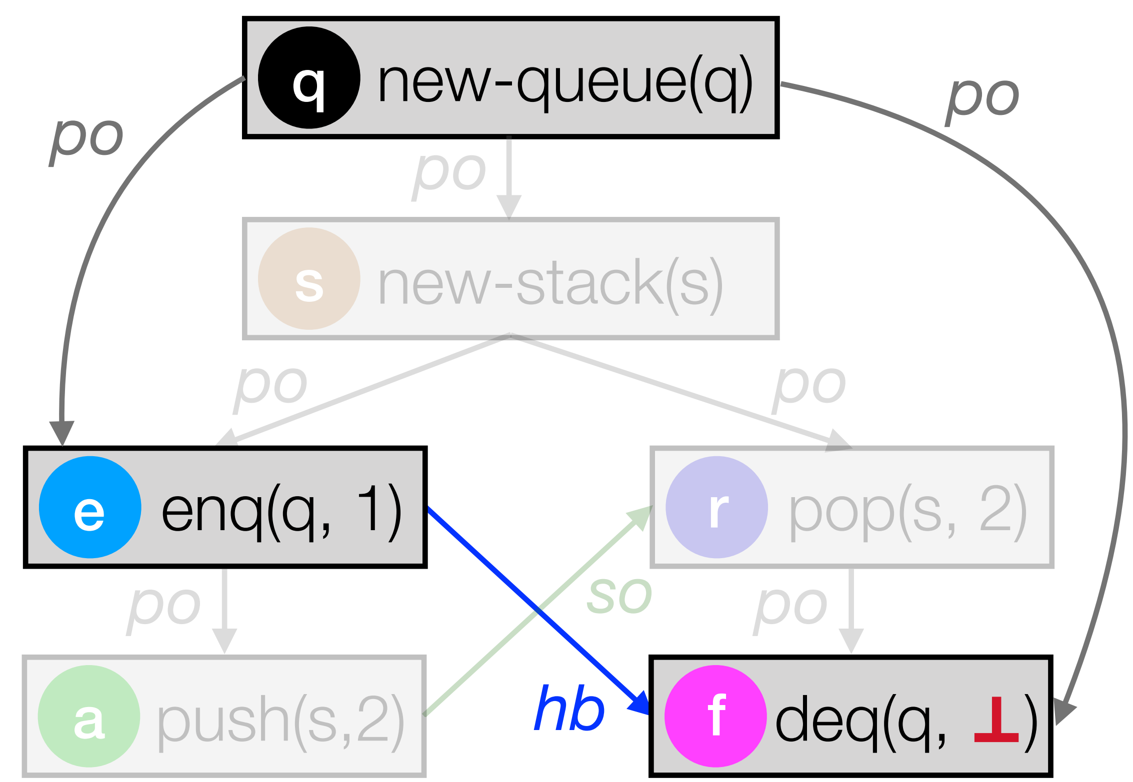
Example Revisited

```
q:=new-queue();
s:=new-stack();
enq(q,1);     ||  a:=pop(s);
push(s,2)     ||  if(a==2)
              ||    b:=deq(q)  // should return 1
```

How does *hb* ***exclude*** the ***RHS*** execution?

☛ library ***axioms***!





Queue Axioms

G = < *E* , *po* , *so* , *hb* > is a consistent ***queue*** execution iff*:*

1. *E* contains queue events

q new-queue(q)	e enq(q, v)	d deq(q, w)	f deq(q, ⊥)	v,w ∈ Val

Queue Axioms

G = < *E* , *po* , *so* , *hb* > is a consistent ***queue*** execution iff*:*

1. *E* contains queue events

   q new-queue(q) | e enq(q, v) | d deq(q, w) | f deq(q, ⊥) — v,w ∈ Val

2. *so* is ***1-to-1***
   so relates ***matching*** `enq`/`deq` events;

   if e enq(q, v) $\xrightarrow{so}$ d deq(q, w) *then* v = w

Queue Axioms

G = < *E* , *po* , *so* , *hb* > is a consistent ***queue*** execution iff***:***

1. *E* contains queue events

   q new-queue(q) e enq(q, v) d deq(q, w) f deq(q, ⊥) v,w ∈ Val

2. *so* is ***1-to-1***

   so relates ***matching*** `enq`/`deq` events;

if e enq(q, v) $\xrightarrow{so}$ d deq(q, w) *then* v = w

e enq(q, v) $\xrightarrow{\cancel{so}}$ f deq(q, ⊥) *for all* v

Queue Axioms

$G = \langle E, po, so, hb \rangle$ is a consistent ***queue*** execution iff:

1. E contains queue events

   q new-queue(q) | e enq(q, v) | d deq(q, w) | f deq(q, ⊥) | $v,w \in Val$

2. so is ***1-to-1***

   so relates ***matching*** `enq/deq` events;

   if e enq(q, v) $\xrightarrow{so}$ d deq(q, w) *then* $v = w$

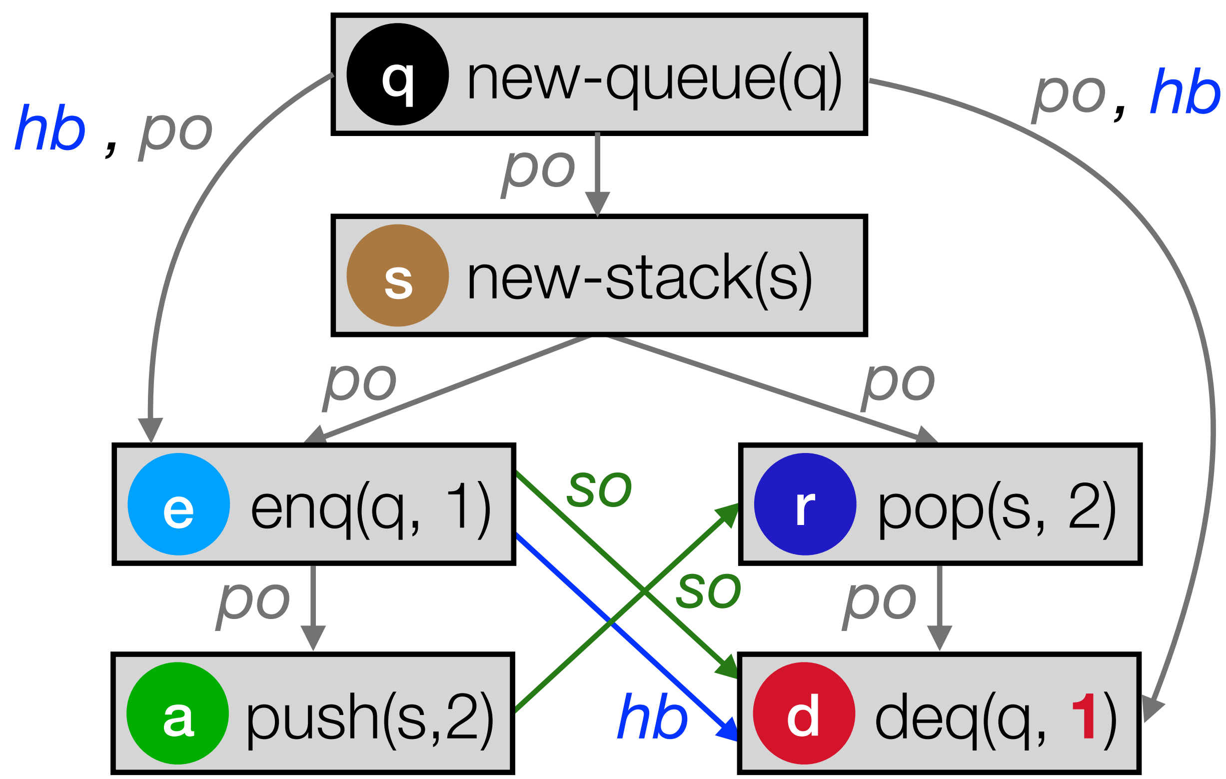
   e enq(q, v) $\xrightarrow{\cancel{so}}$ f deq(q, ⊥) *for all* v

3. $\exists$ to. to ***totally*** orders E,

   $hb \subseteq to$ and to is a FIFO sequence

Example Revisited

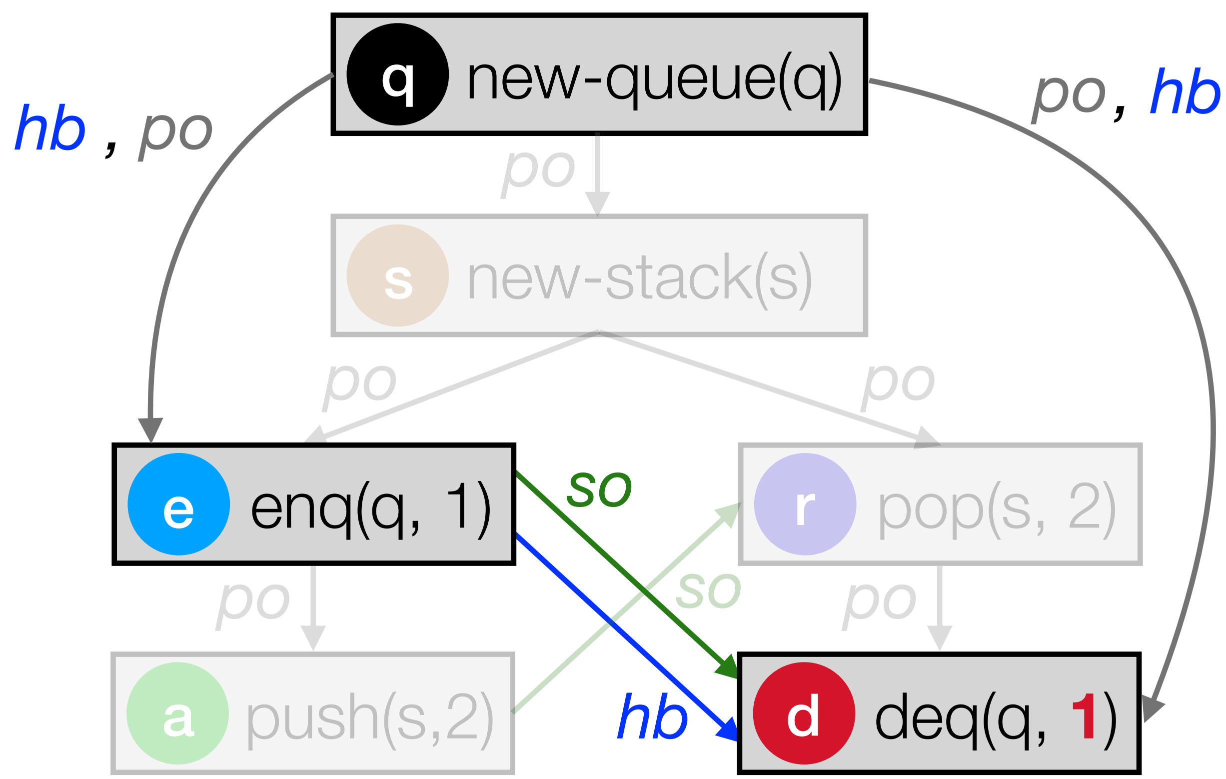
```
     q:=new-queue();
     s:=new-stack();

enq(q,1);     ||  a:=pop(s);
push(s,2)     ||  if(a==2)
              ||    b:=deq(q)  // should return 1
```



$hb = (po \cup so)^{+}$

Example Revisited

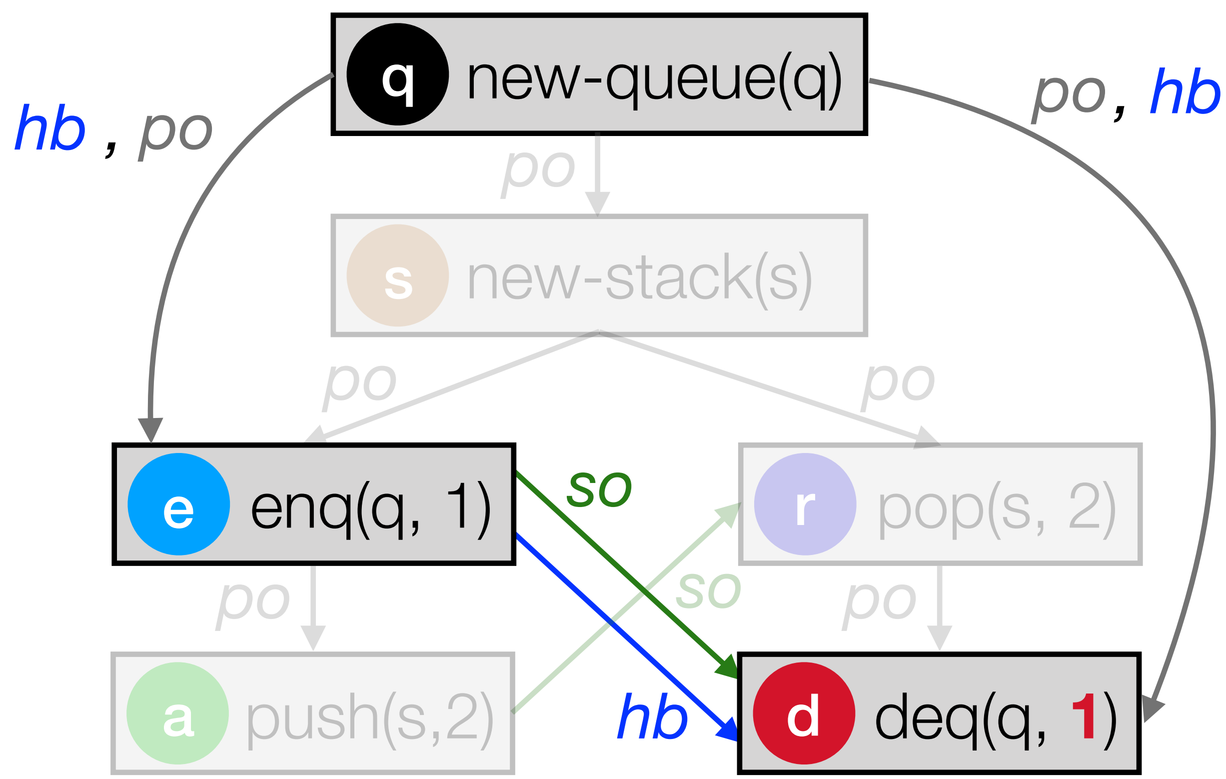
```
    q:=new-queue();
    s:=new-stack();

enq(q,1);    ||  a:=pop(s);
push(s,2)    ||  if(a==2)
             ||    b:=deq(q)   // should return 1
```

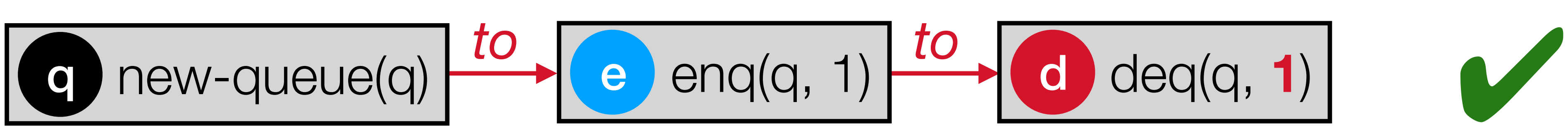


$hb = (po \cup so)^{+}$

Example Revisited

```
     q:=new-queue();
     s:=new-stack();
enq(q,1);  ||  a:=pop(s);
push(s,2)  ||  if(a==2)
                  b:=deq(q)  // should return 1
```



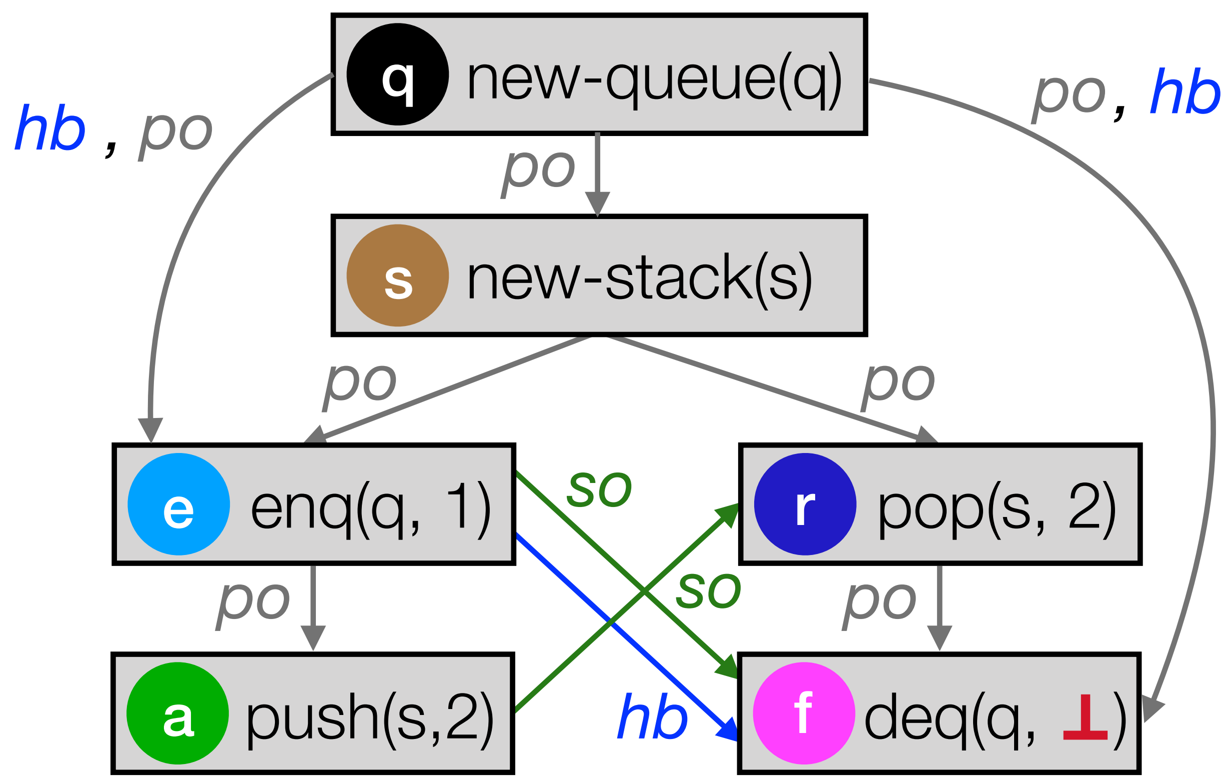


$hb = (\, po \cup so \,)^{+}$

$hb \subseteq to$

Example Revisited

```
      q:=new-queue();
      s:=new-stack();

enq(q,1);  ||  a:=pop(s);
push(s,2)  ||  if(a==2)
           ||    b:=deq(q)  // should return 1
```

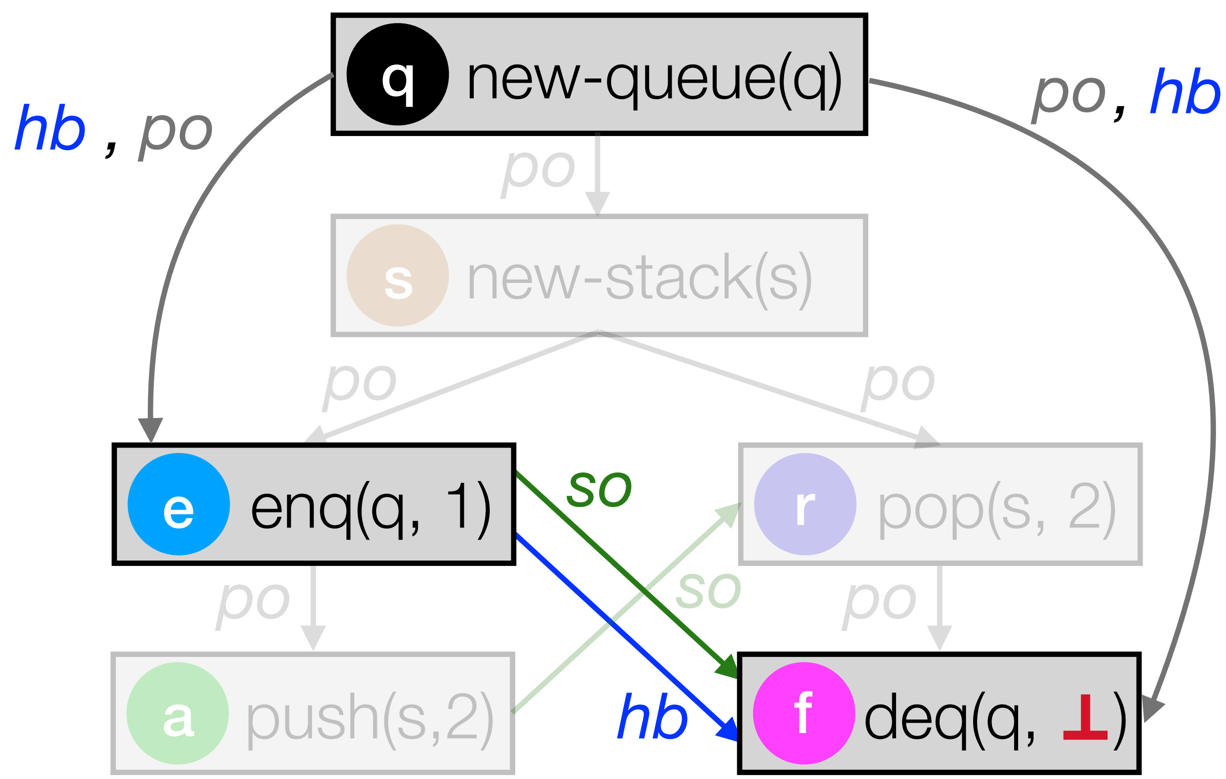


$hb = (po \cup so)^+$

$hb \subseteq to$

Example Revisited

```
      q:=new-queue();
      s:=new-stack();
enq(q,1);   || a:=pop(s);
push(s,2)   || if(a==2)
            ||    b:=deq(q)  // should return 1
```

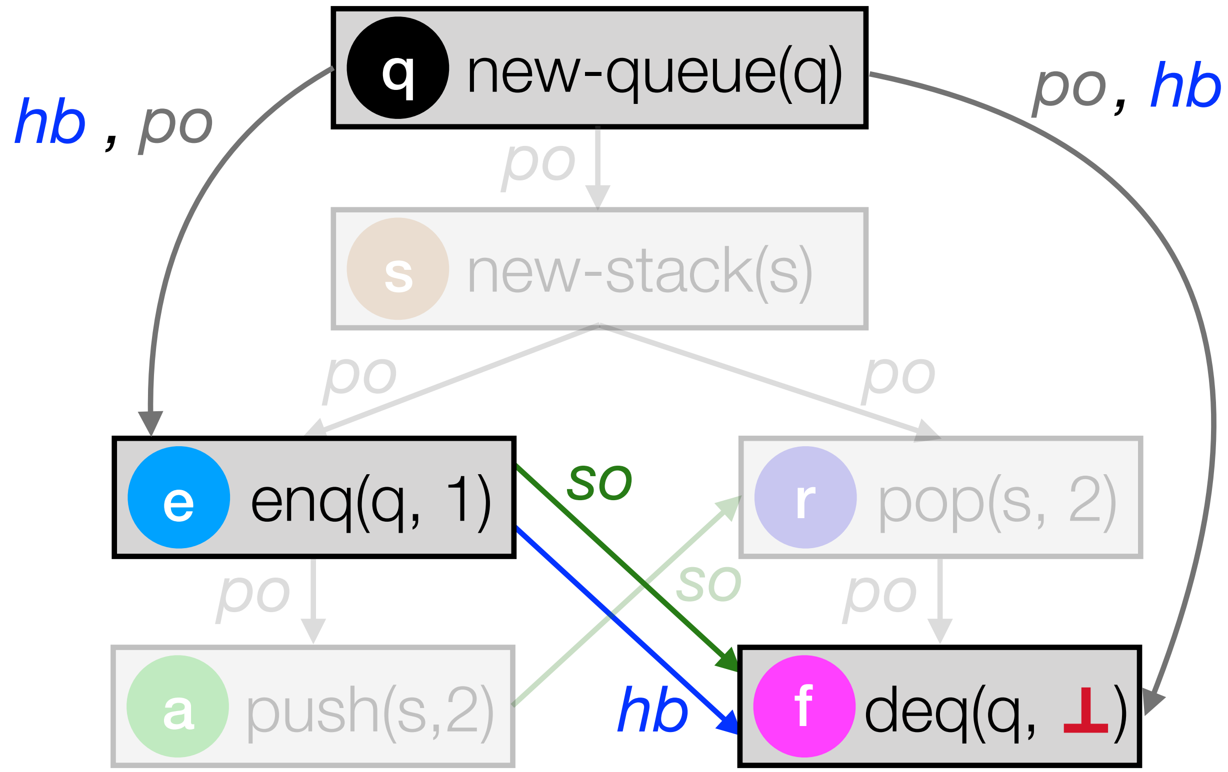


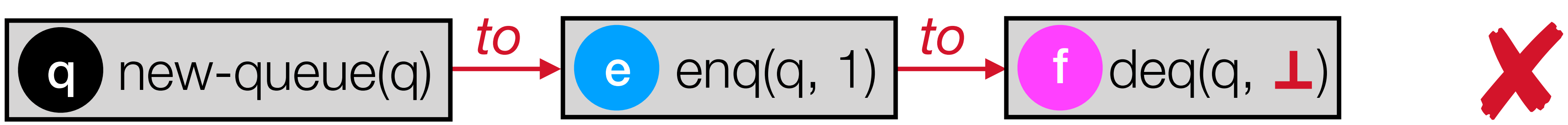
$hb = (po \cup so)^{+}$

$hb \subseteq to$

Example Revisited

```
     q:=new-queue();
     s:=new-stack();
enq(q,1);  || a:=pop(s);
push(s,2)  || if(a==2)
           ||   b:=deq(q)  // should return 1
```





$hb = (po \cup so)^{+}$

$hb \subseteq to$

Queue Axioms

G = < *E* , *po* , *so* , *hb* > is a consistent ***queue*** execution iff***:***

1. *E* contains queue events
2. *so* is ***1-to-1***; *so* relates ***matching*** `enq`/`deq` events
3. ∃ *to*. *to* ***totally*** orders *E*,

   hb ⊆ *to* and *to* is a FIFO sequence

✘ too strong

✘ difficult to find *to* witness

Why Not Strong Queue Axioms?

C11 Herlihy-Wing Queue Implementation

$new\text{-}queue() \triangleq$
  $\texttt{let}\ q = alloc(+\infty)\ \texttt{in}\ q$

$enq(q, v) \triangleq$
  $\texttt{let}\ i = fetch\text{-}add(q, 1, \texttt{rel})\ \texttt{in}$
  $store(q + i + 1, v, \texttt{rel});$

$deq(q) \triangleq$
  $\texttt{loop}$
    $\texttt{let}\ range = load(q, \texttt{acq})\ \texttt{in}$
    $\texttt{for}\ i = 1\ \texttt{to}\ range\ \texttt{do}$
      $\texttt{let}\ x = atomic\text{-}xchg(q + i, 0, \texttt{acq})\ \texttt{in}$
      $\texttt{if}\ x \neq 0\ \texttt{then}\ \texttt{break}_2\ x$

Why Not Strong Queue Axioms?

C11 Herlihy-Wing Queue Implementation

$\mathit{new\text{-}queue}() \triangleq$
  $\texttt{let}\ q = \mathit{alloc}(+\infty)\ \texttt{in}\ q$

$\mathit{enq}(q, v) \triangleq$
  $\texttt{let}\ i = \mathit{fetch\text{-}add}(q, 1, \texttt{rel})\ \texttt{in}$
  $\mathit{store}(q + i + 1, v, \texttt{rel});$

$\mathit{deq}(q) \triangleq$
  $\texttt{loop}$
    $\texttt{let}\ \mathit{range} = \mathit{load}(q, \texttt{acq})\ \texttt{in}$
    $\texttt{for}\ i = 1\ \texttt{to}\ \mathit{range}\ \texttt{do}$
      $\texttt{let}\ x = \mathit{atomic\text{-}xchg}(q + i, 0, \texttt{acq})\ \texttt{in}$
      $\texttt{if}\ x \neq 0\ \texttt{then}\ \texttt{break}_2\ x$

Why Not Strong Queue Axioms?

C11 Herlihy-Wing Queue Implementation

$\mathit{new\text{-}queue}() \triangleq$
  $\mathtt{let}\ q = \mathit{alloc}(+\infty)\ \mathtt{in}\ q$

$\mathit{enq}(q, v) \triangleq$
  $\mathtt{let}\ i = \mathit{fetch\text{-}add}(q, 1, \mathtt{rel})\ \mathtt{in}$
  $\mathit{store}(q + i + 1, v, \mathtt{rel});$

$\mathit{deq}(q) \triangleq$
  $\mathtt{loop}$
    $\mathtt{let}\ \mathit{range} = \mathit{load}(q, \mathtt{acq})\ \mathtt{in}$
    $\mathtt{for}\ i = 1\ \mathtt{to}\ \mathit{range}\ \mathtt{do}$
      $\mathtt{let}\ x = \mathit{atomic\text{-}xchg}(q + i, 0, \mathtt{acq})\ \mathtt{in}$
      $\mathtt{if}\ x \neq 0\ \mathtt{then}\ \mathtt{break}_2\ x$

✘ does ***not*** satisfy ***strong*** queue axioms

Why Not Strong Queue Axioms?

C11 Herlihy-Wing Queue Implementation

$$
\begin{aligned}
&\mathit{new\text{-}queue}() \triangleq \\
&\quad \texttt{let } q = \mathit{alloc}(+\infty) \texttt{ in } q \\
\\
&\mathit{enq}(q, v) \triangleq \\
&\quad \texttt{let } i = \mathit{fetch\text{-}add}(q, 1, \texttt{rel}) \texttt{ in} \\
&\quad \mathit{store}(q + i + 1, v, \texttt{rel});
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{deq}(q) \triangleq \\
&\quad \texttt{loop} \\
&\qquad \texttt{let } \mathit{range} = \mathit{load}(q, \texttt{acq}) \texttt{ in} \\
&\qquad \texttt{for } i = 1 \texttt{ to } \mathit{range} \texttt{ do} \\
&\qquad\quad \texttt{let } x = \mathit{atomic\text{-}xchg}(q + i, 0, \texttt{acq}) \texttt{ in} \\
&\qquad\quad \texttt{if } x \neq 0 \texttt{ then break}_2\ x
\end{aligned}
$$

wanted

Weaker *queue axioms*

Why weak axioms?

strong enough for certain uses: ***single-producer-single-consumer***

Queue Axioms

$G = \langle\ E, po, so, hb\ \rangle$ is a consistent ***queue*** execution iff:

1. E contains queue events
2. so is ***1-to-1***; so relates ***matching*** `enq`/`deq` events
3. $\exists$ to. to ***totally*** orders E,

   $hb \subseteq to$ and to is a FIFO sequence

✘ too strong ☛ ***weak axioms*** (see our paper)

✘ difficult to find to witness

Queue Axioms

$G = \langle E, po, so, hb \rangle$ is a consistent ***queue*** execution iff*:*

1. E contains queue events
2. *so* is ***1-to-1***; *so* relates ***matching*** `enq`/`deq` events
3. $\exists$ *to*. *to* ***totally*** orders E,

   $hb \subseteq to$ and *to* is a FIFO sequence

✘ too strong ☛ ***weak axioms*** (see our paper)

✘ difficult to find *to* witness

equivalent acyclicity axiom

(see our paper)

Summary

A declarative ***specification*** and ***verification*** framework:

✓ ***Agnostic*** to memory model

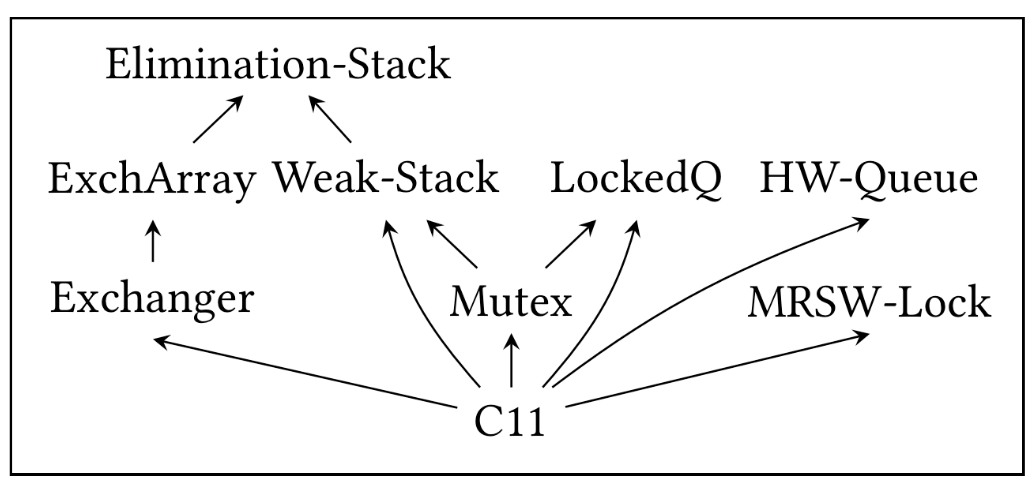
- ➡ support *both SC* and *WMC* specs

✓ ***General***

- ➡ port *existing SC (linearisability) specs*
- ➡ port *existing WMC specs* (e.g. C11, TSO)
- ➡ built from the *ground up*: assume *no pre-existing* libraries or specs

✓ ***Compositional***

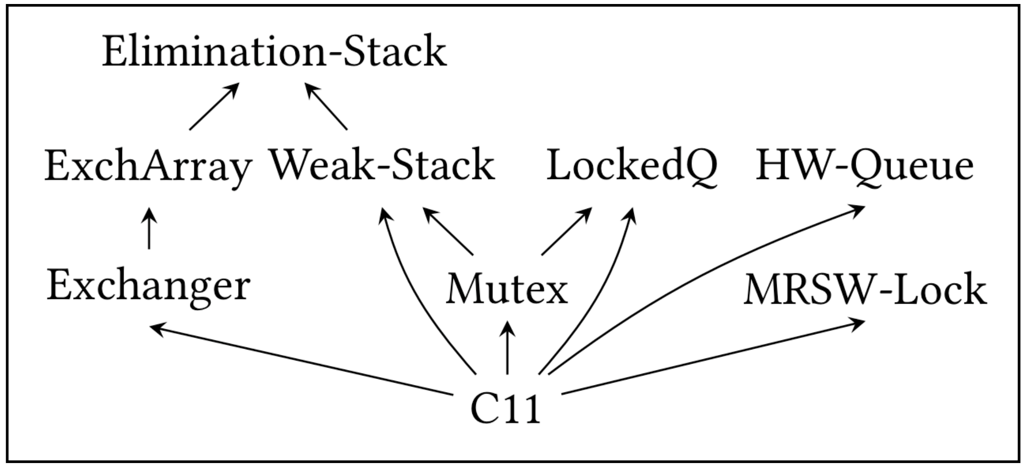
- ➡ *vertical* composition to verify *library implementations*
- ➡ *horizontal* composition to verify *client programs*



Thank You for Listening!

A declarative ***specification*** and ***verification*** framework:

- ✓ ***Agnostic*** to memory model
  - ➡ support *both SC* and *WMC* specs
- ✓ ***General***
  - ➡ port *existing SC (linearisability) specs*
  - ➡ port *existing WMC specs* (e.g. C11, TSO)
  - ➡ built from the *ground up*: assume *no pre-existing* libraries or specs
- ✓ ***Compositional***
  - ➡ *vertical* composition to verify *library implementations*
  - ➡ *horizontal* composition to verify *client programs*



azalea@mpi-sws.org
SoundAndComplete.org
@azalearaad