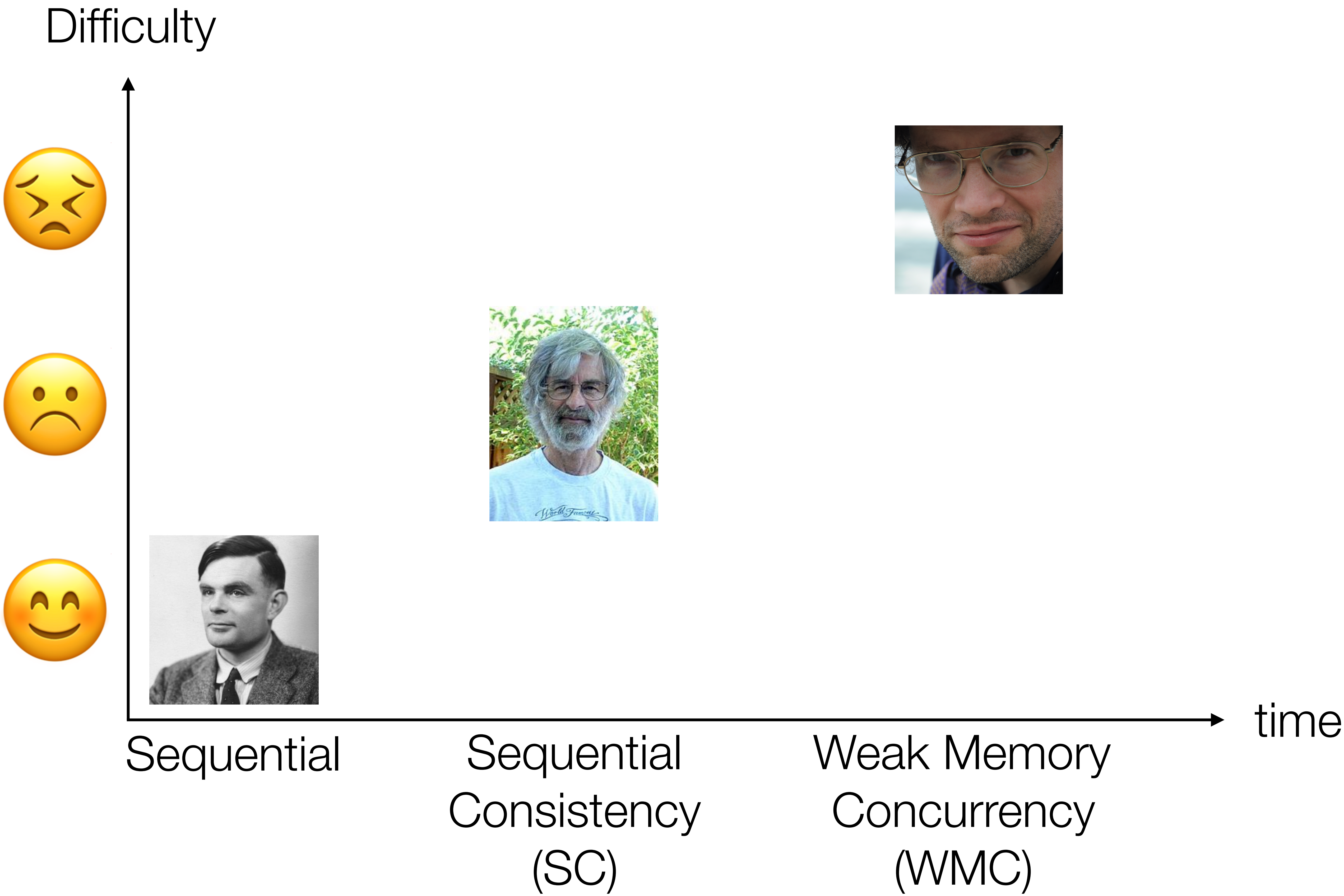
Correctness in a Weakly Consistent Setting

Azalea Raad

Max Planck Institute for Software Systems (MPI-SWS)

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

History


Difficulty
time
Sequential
Sequential Consistency (SC)
Weak Memory Concurrency (WMC)

Sequential Consistency (SC)

a	`x:=1;`	‖	c	`y:=1;`
b	`b:=y` // v_b	‖	d	`d:=x` // v_d

SC (a.k.a. interleaving concurrency):

total execution order (*to*) that respects program order (*po*)

Sequential Consistency (SC)

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

SC (a.k.a. interleaving concurrency):

total execution order (*to*) that respects program order (*po*)

Execution order (*to*)	v_b	v_d	
a b c d	**0**	**1**	a before d ; c after b

Sequential Consistency (SC)

a	`x:=1;`	c	`y:=1;`
b	`b:=y` // v_b	d	`d:=x` // v_d

SC (a.k.a. interleaving concurrency):

total execution order (*to*) that respects program order (*po*)

Execution order (*to*)	v_b	v_d	
a b c d	**0**	**1**	a before d ; c after b
c d a b	**1**	**0**	a after d ; c before b

Sequential Consistency (SC)

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

SC (a.k.a. interleaving concurrency):

total execution order (*to*) that respects program order (*po*)

Execution order (*to*)	v_b	v_d	
a b c d	0	1	a before d ; c after b
c d a b	1	0	a after d ; c before b
a c b d	1	1	
a c d b	1	1	a before d ; c before b
c a b d	1	1	
c a d b	1	1	

Sequential Consistency (SC)

a `x:=1;`	c `y:=1;`
b `b:=y` // v_b	d `d:=x` // v_d

SC (a.k.a. interleaving concurrency):

total execution order (*to*) that respects program order (*po*)

v_b	v_d	
0	**1**	a before d ; c after b
1	**0**	a after d ; c before b
1	**1**	a before d ; c before b
0	**0**	not possible!

Weak Memory Concurrency (WMC)

No total execution order (*to*) ⇒

anomalies (***litmus tests***): behaviour absent under SC;
caused by:

- instruction ***reordering;***
  — e.g. store buffering (SB)

Weak Memory Concurrency (WMC)

No total execution order (*to*) $\Rightarrow$

anomalies (***litmus tests***): behaviour absent under SC; caused by:

- instruction ***reordering;***
  — e.g. store buffering (SB)
- ***different write propagation*** across cache hierarchy;
  — e.g. Independent Reads of Independent Writes (IRIW)

WMC: Store Buffering

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

v_b	v_d
0	1
1	0
1	1
0	0

possible, due to reordering!

store buffering(SB)

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

Fences to Stop Reordering

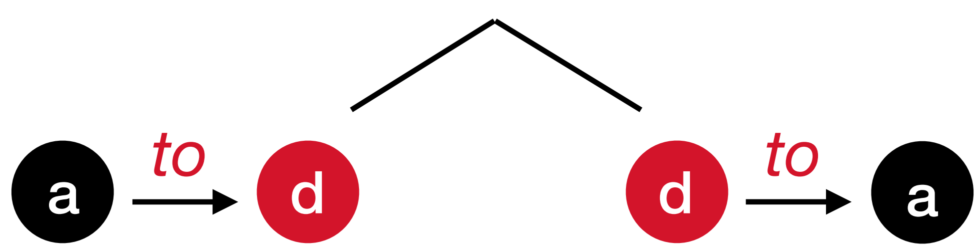
a `x:=1;`	‖	c `y:=1;`
b `b:=y` // v_b	‖	d `d:=x` // v_d

	v_b	v_d
	0	**1**
	1	**0**
	1	**1**
~~possible, due to reordering!~~	~~**0**~~	~~**0**~~

WMC: Independent Reads of Independent Writes

a `x:=1`	‖	b `b:=x;` // 1 c `c:=y` // 0	‖	d `y:=1`	‖	e `e:=y;` // 1 f `f:=x` // 0

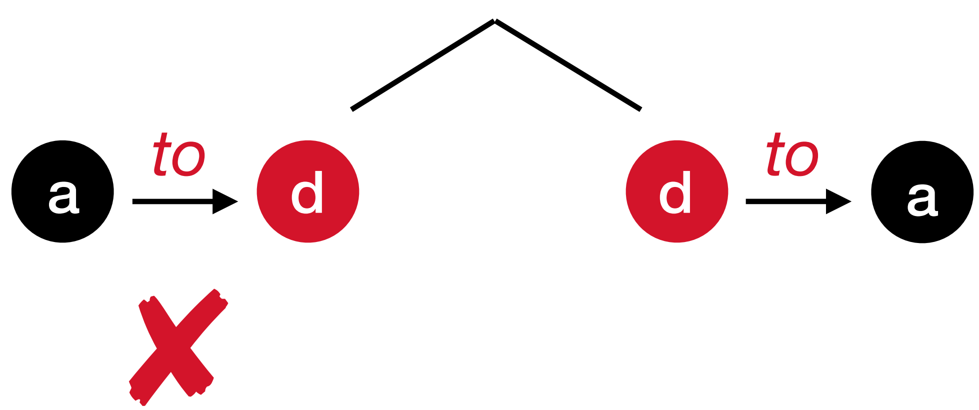
annotated behaviour ***not*** allowed ***under SC*** :



WMC: Independent Reads of Independent Writes

a `x:=1`	‖	b `b:=x;` // 1 c `c:=y` // 0	‖	d `y:=1`	‖	e `e:=y;` // 1 f `f:=x` // 0

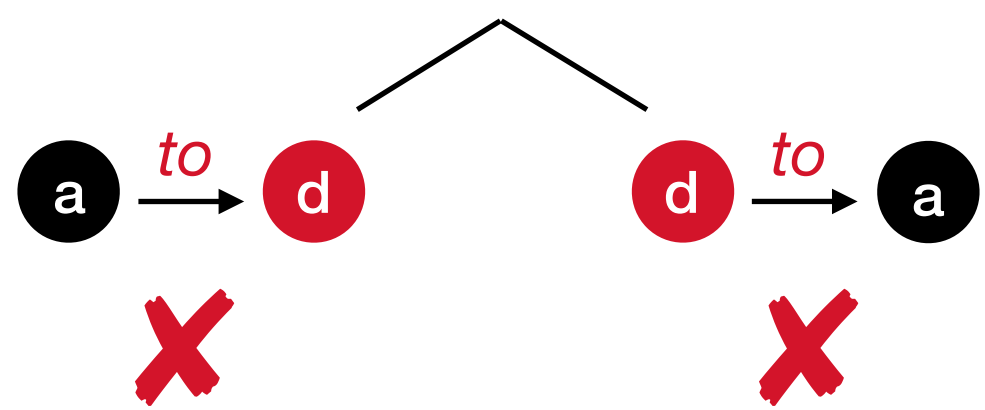
annotated behaviour ***not*** allowed ***under SC*** :



WMC: Independent Reads of Independent Writes

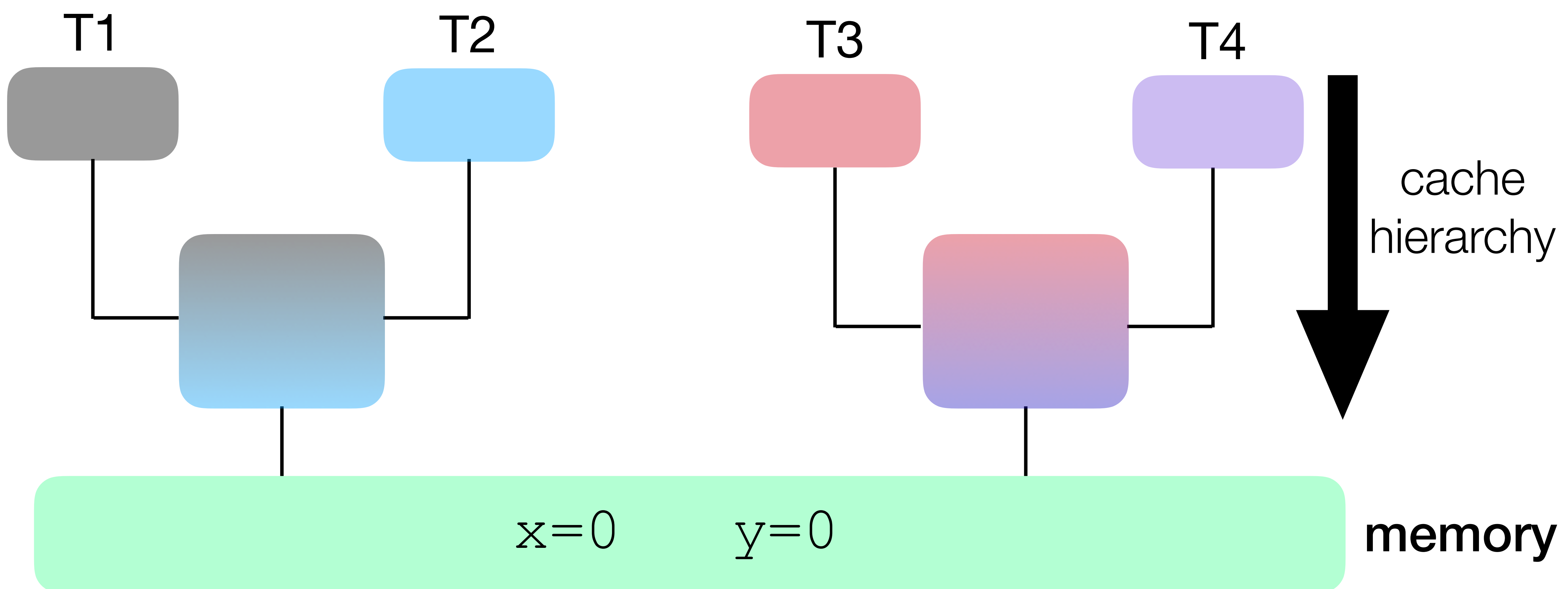
a `x:=1`	‖	b `b:=x;` // 1	‖	d `y:=1`	‖	e `e:=y;` // 1
		c `c:=y` // 0				f `f:=x` // 0

annotated behaviour ***not*** allowed ***under SC*** :



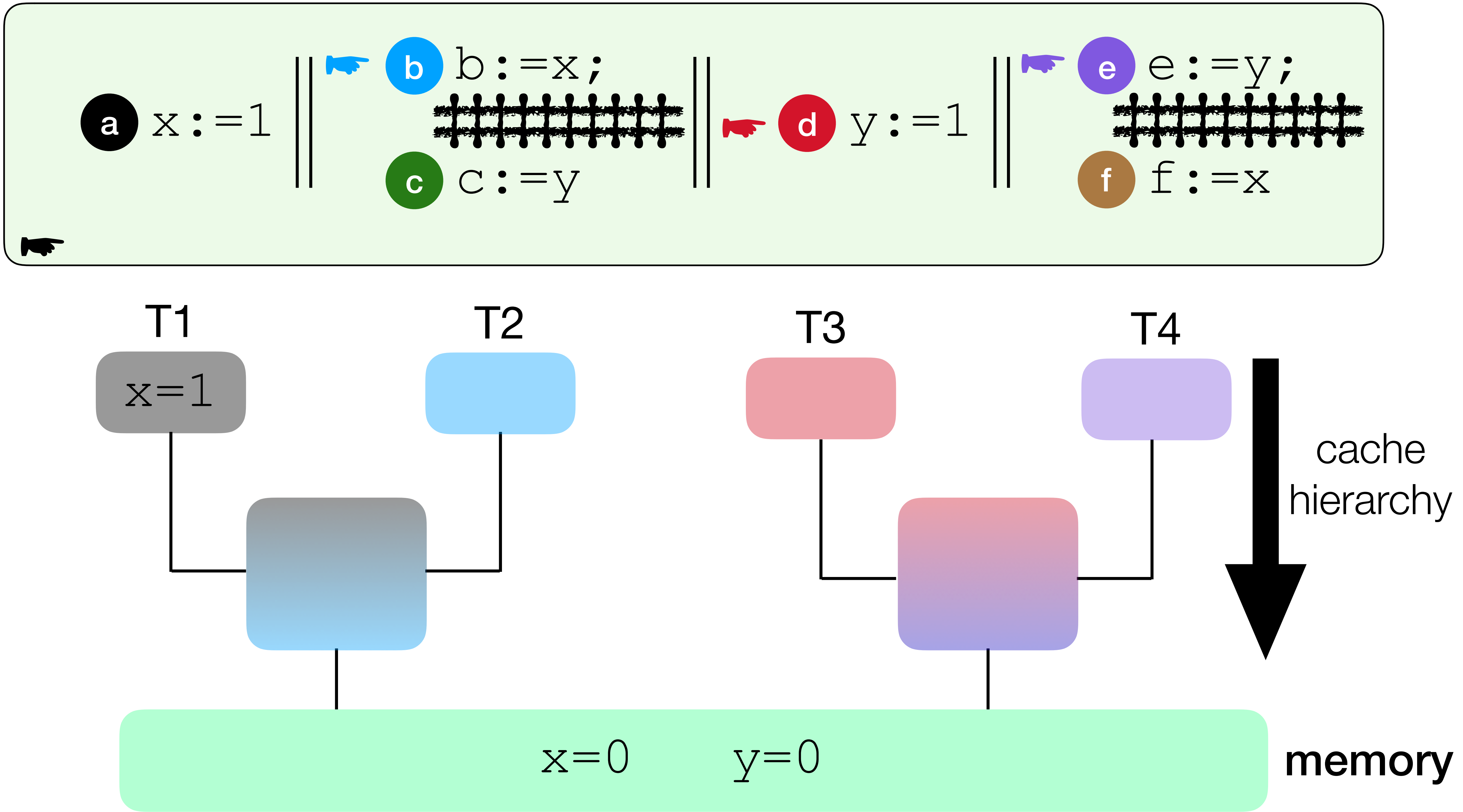
WMC: Independent Reads of Independent Writes

```
a x:=1 || b b:=x;     || d y:=1 || e e:=y;
           c c:=y                  f f:=x
```



non-multi-copy-atomic architecture

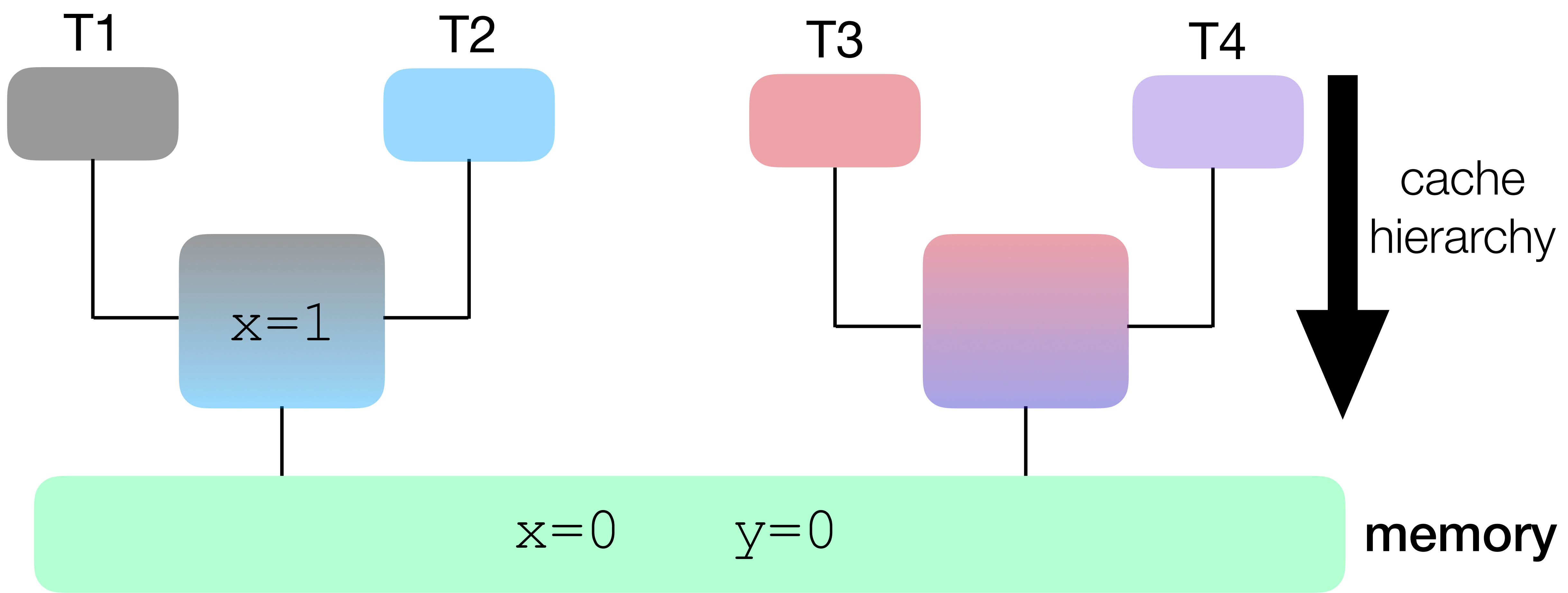
WMC: Independent Reads of Independent Writes



non-multi-copy-atomic architecture

WMC: Independent Reads of Independent Writes

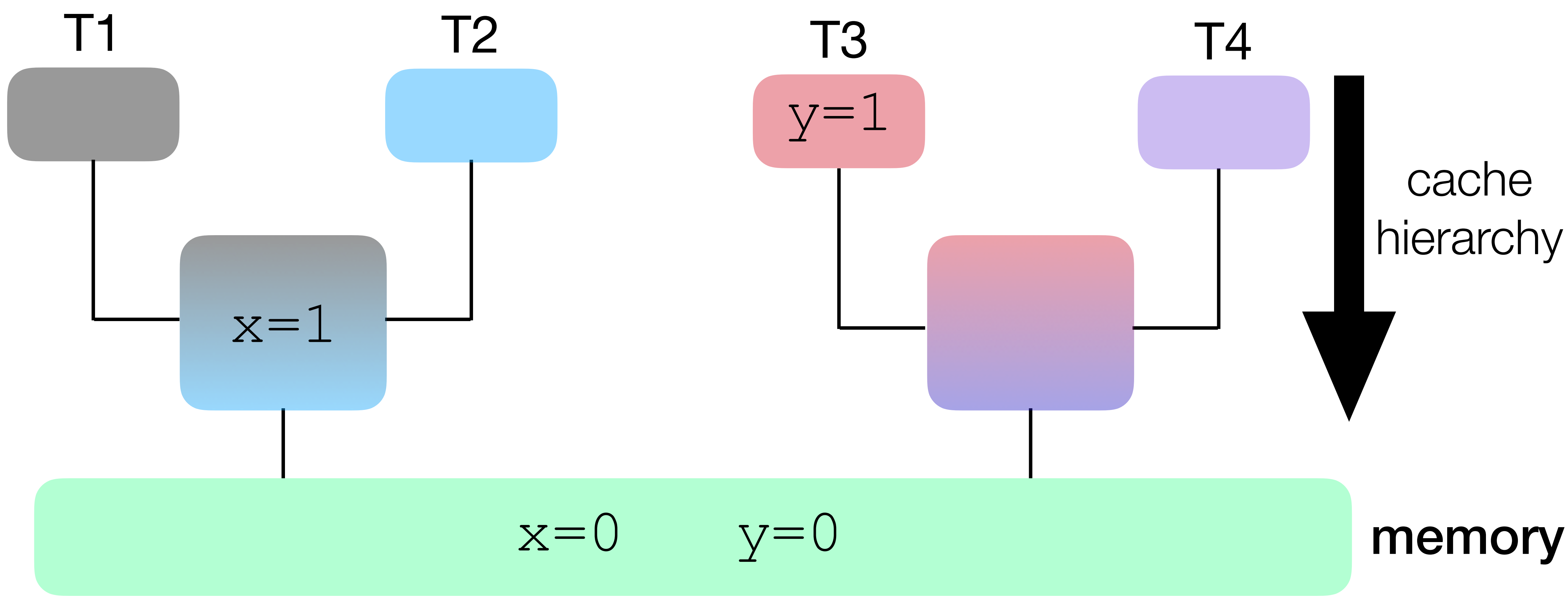
```
a x:=1 || b b:=x;    || d y:=1 || e e:=y;
           c c:=y    ||           f f:=x
```



non-multi-copy-atomic architecture

WMC: Independent Reads of Independent Writes

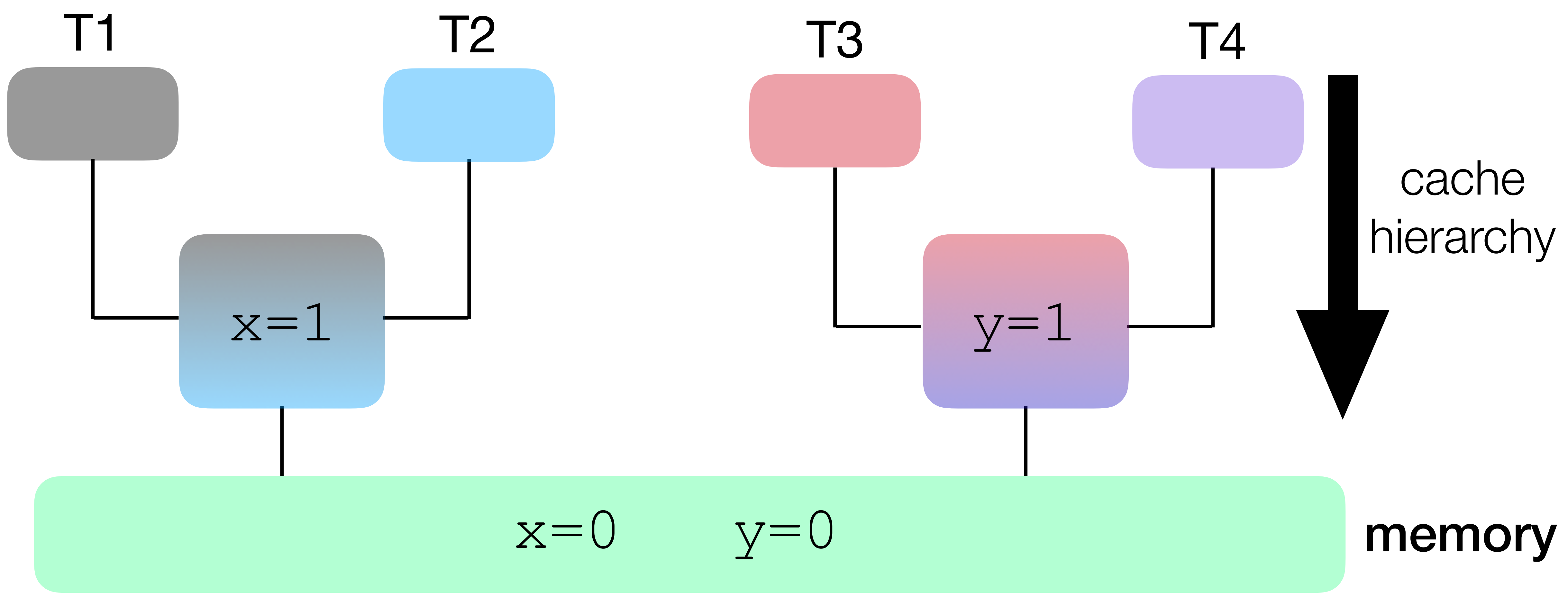
```
a x:=1 || b b:=x;      || d y:=1 || e e:=y;
            c c:=y     ||           f f:=x
```



non-multi-copy-atomic architecture

WMC: Independent Reads of Independent Writes

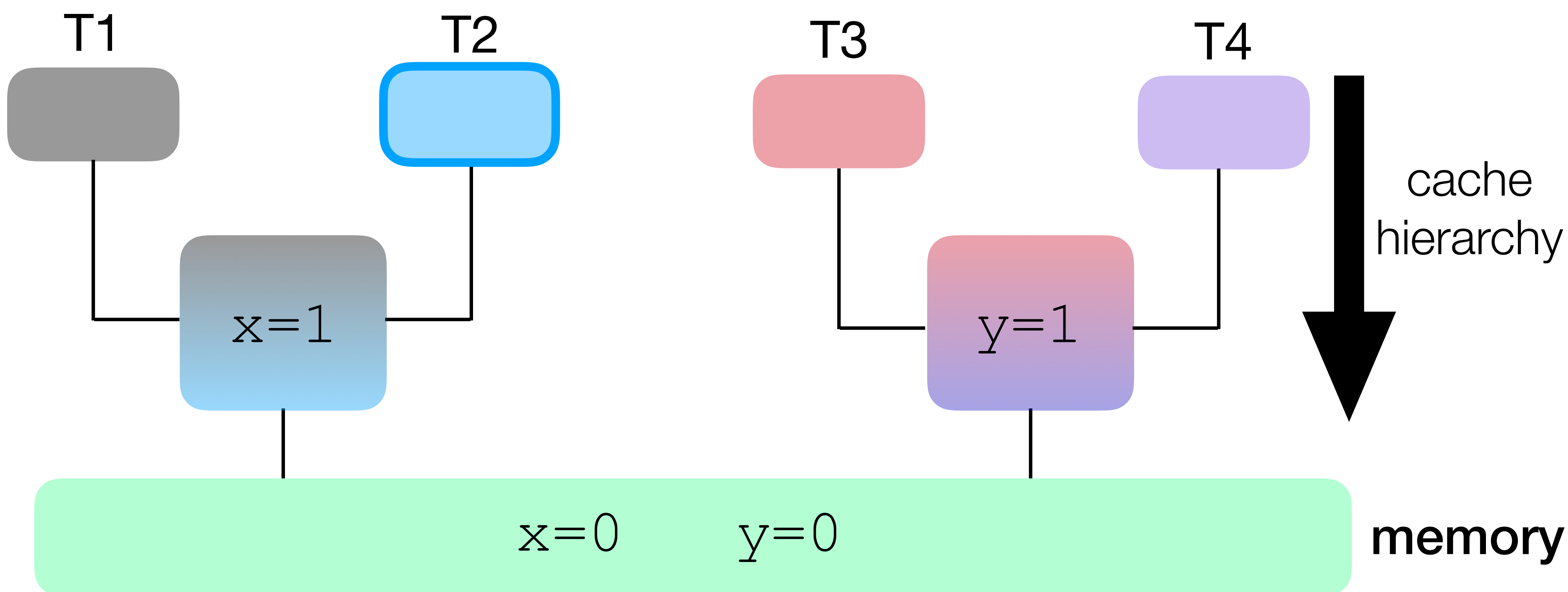
a `x:=1` ‖ b `b:=x;` c `c:=y` ‖ d `y:=1` ‖ e `e:=y;` f `f:=x`



non-multi-copy-atomic architecture

WMC: Independent Reads of Independent Writes

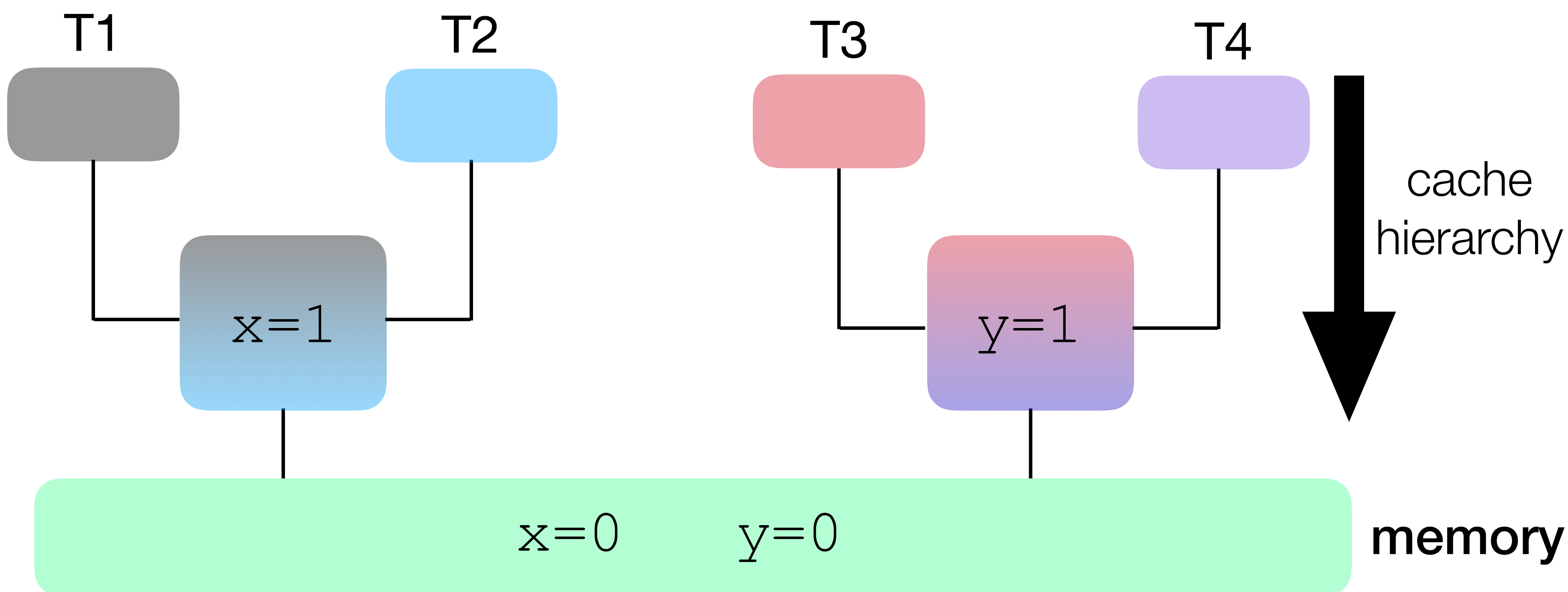
T1		T2		T3		T4
a `x:=1`	‖	b `b:=x;` c `c:=y`	‖	d `y:=1`	‖	e `e:=y;` f `f:=x`



non-multi-copy-atomic architecture

WMC: Independent Reads of Independent Writes

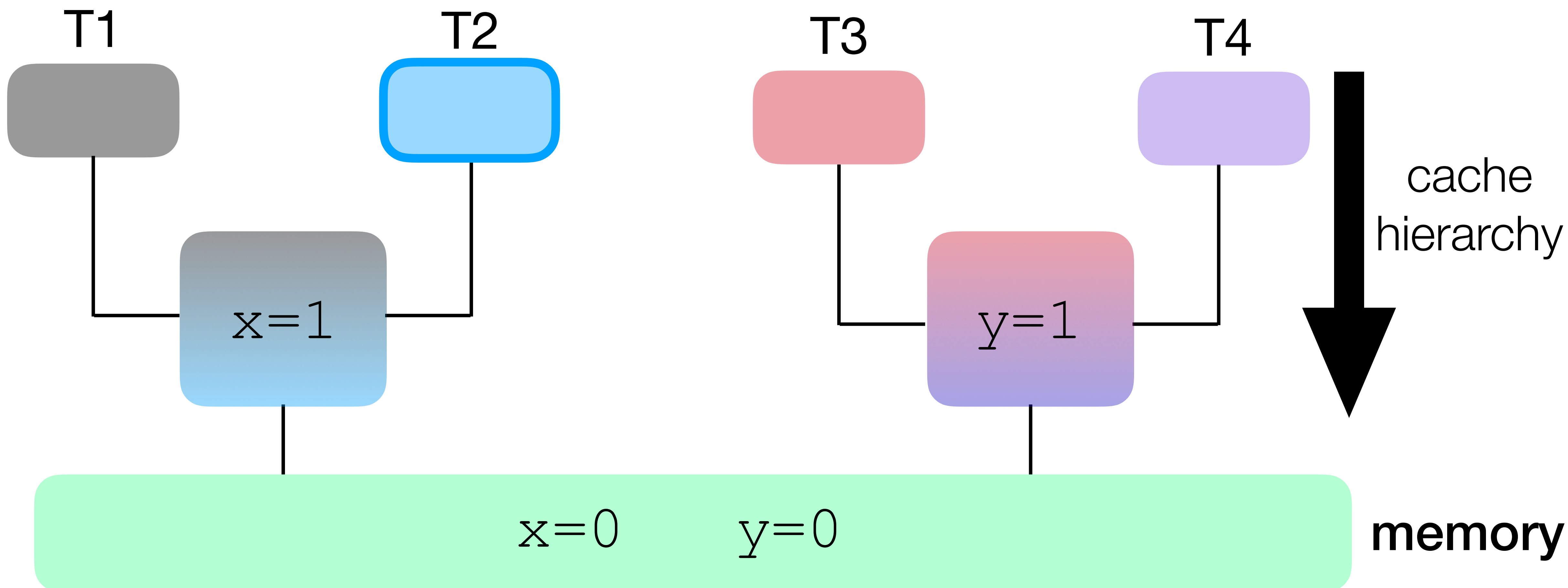
a x:=1 ‖ b b:=x; // 1 ‖ d y:=1 ‖ e e:=y;
c c:=y ‖ f f:=x



non-multi-copy-atomic architecture

WMC: Independent Reads of Independent Writes

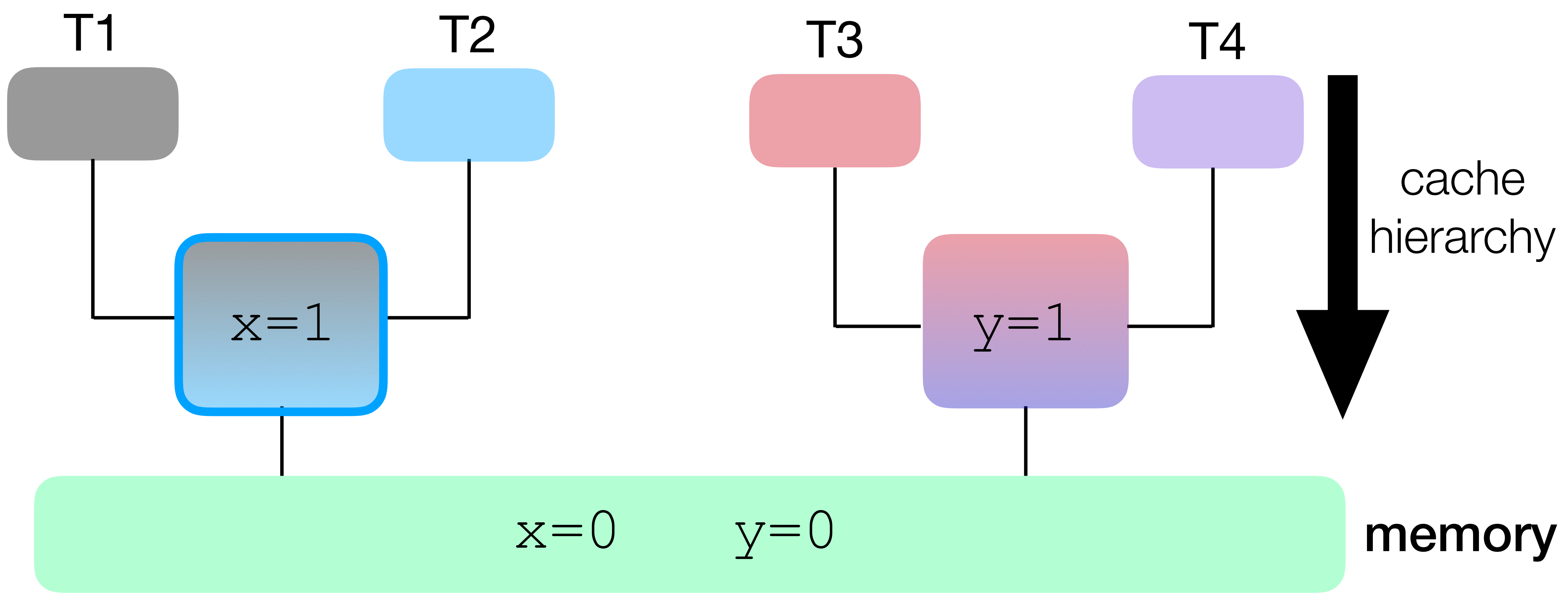
a `x:=1` ‖ b `b:=x; // 1` c `c:=y` ‖ d `y:=1` ‖ e `e:=y;` f `f:=x`



non-multi-copy-atomic architecture

WMC: Independent Reads of Independent Writes

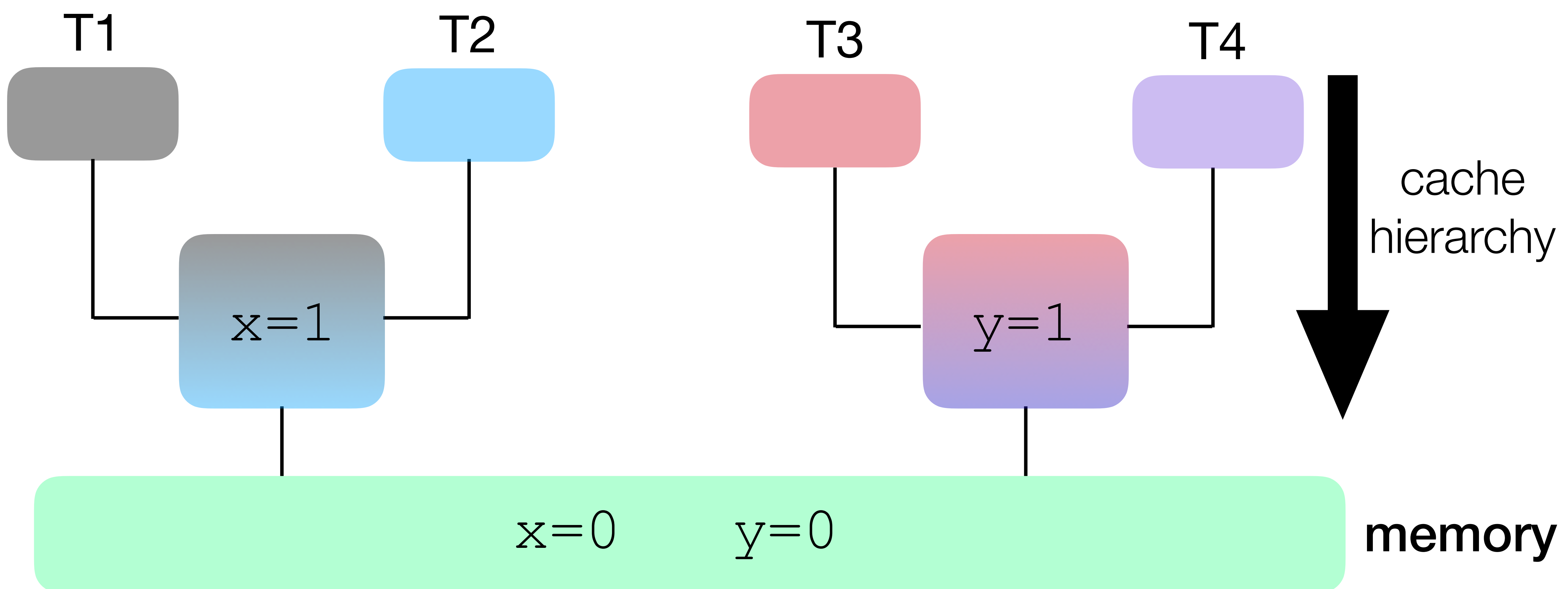
a `x:=1`	‖	b `b:=x;` // 1 c `c:=y`	‖	d `y:=1`	‖	e `e:=y;` f `f:=x`



non-multi-copy-atomic architecture

WMC: Independent Reads of Independent Writes

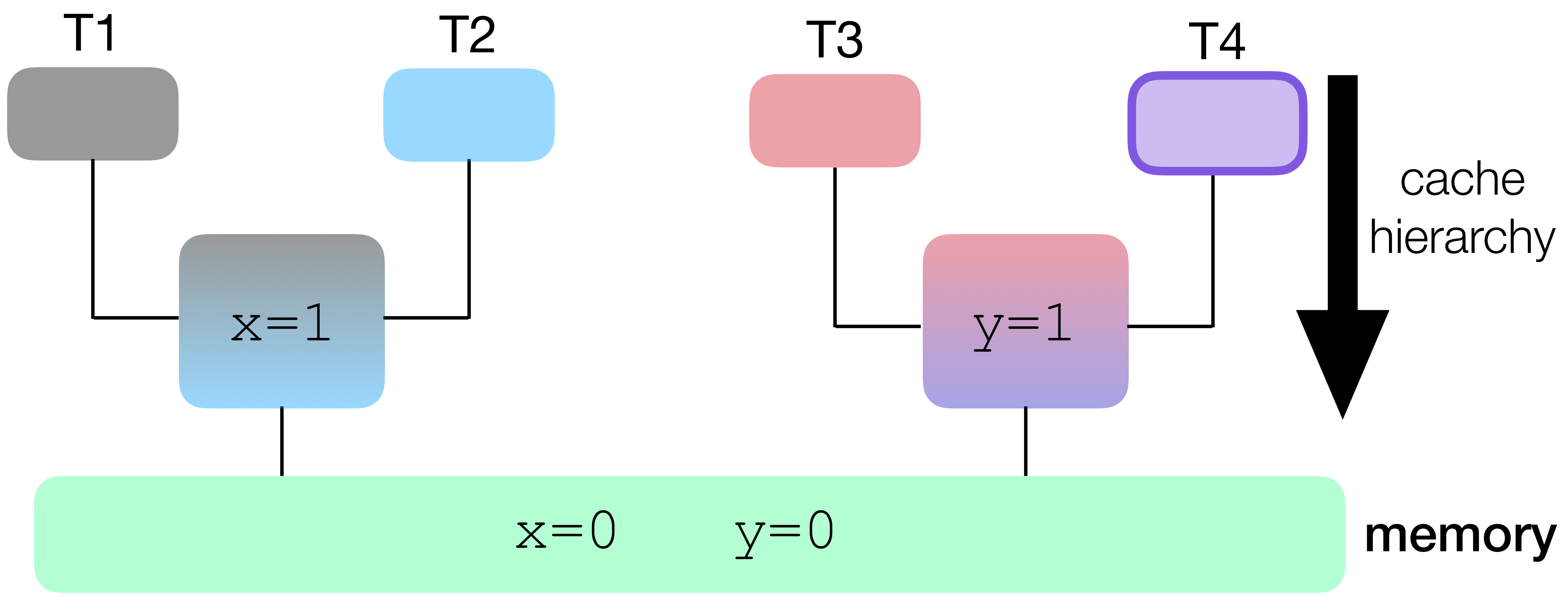
a `x:=1`	b `b:=x;` //1 c `c:=y` //0	d `y:=1`	e `e:=y;` f `f:=x`



non-multi-copy-atomic architecture

WMC: Independent Reads of Independent Writes

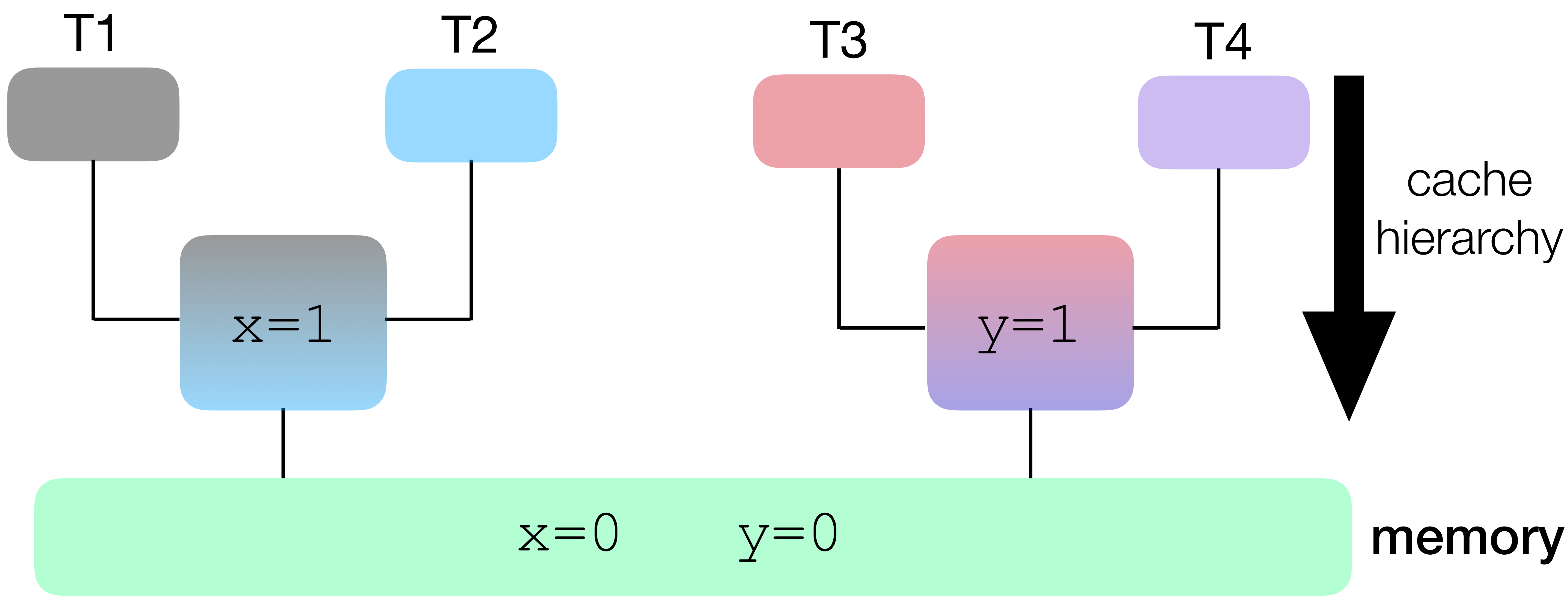
a `x:=1` || b `b:=x;` //1 ; c `c:=y` //0 || d `y:=1` || e `e:=y;` f `f:=x`



non-multi-copy-atomic architecture

WMC: Independent Reads of Independent Writes

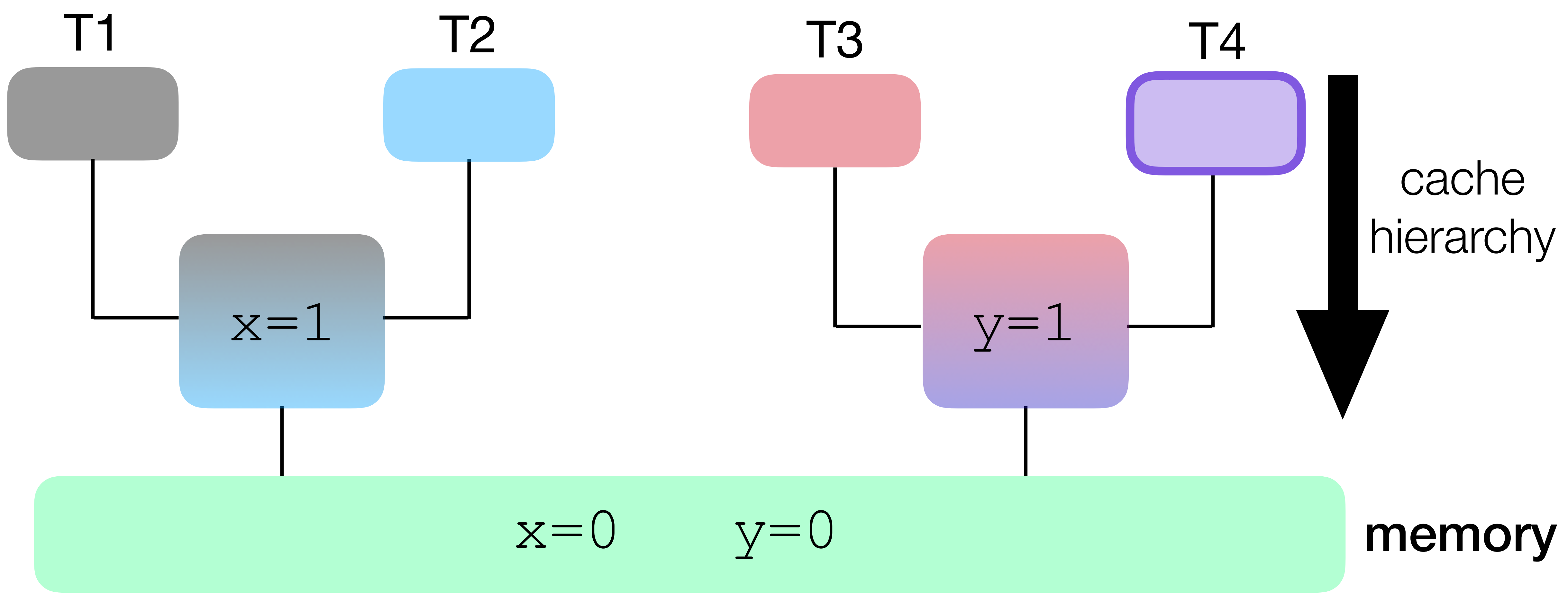
a `x:=1`	b `b:=x; //1` c `c:=y //0`	d `y:=1`	e `e:=y; //1` f `f:=x`



non-multi-copy-atomic architecture

WMC: Independent Reads of Independent Writes

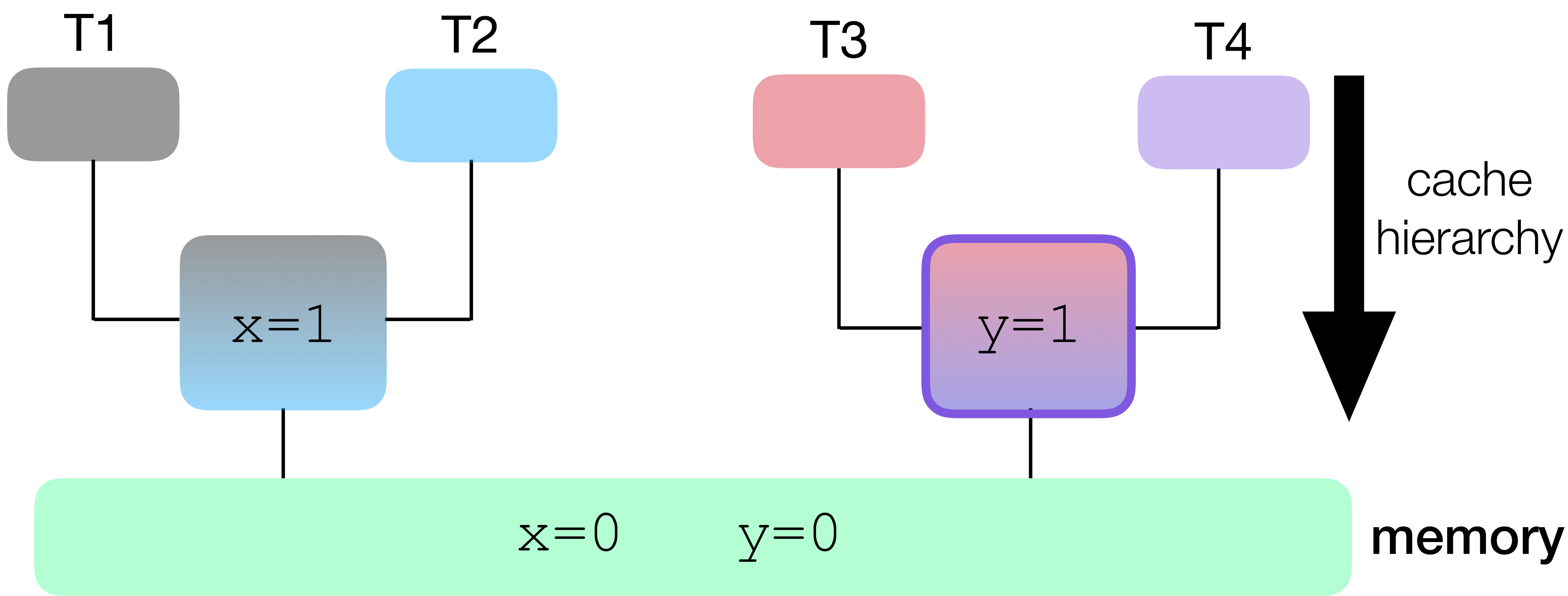
T1		T2		T3		T4
a `x:=1`	‖	b `b:=x; //1`	‖	d `y:=1`	‖	e `e:=y; //1`
		c `c:=y //0`				f `f:=x`



non-multi-copy-atomic architecture

WMC: Independent Reads of Independent Writes

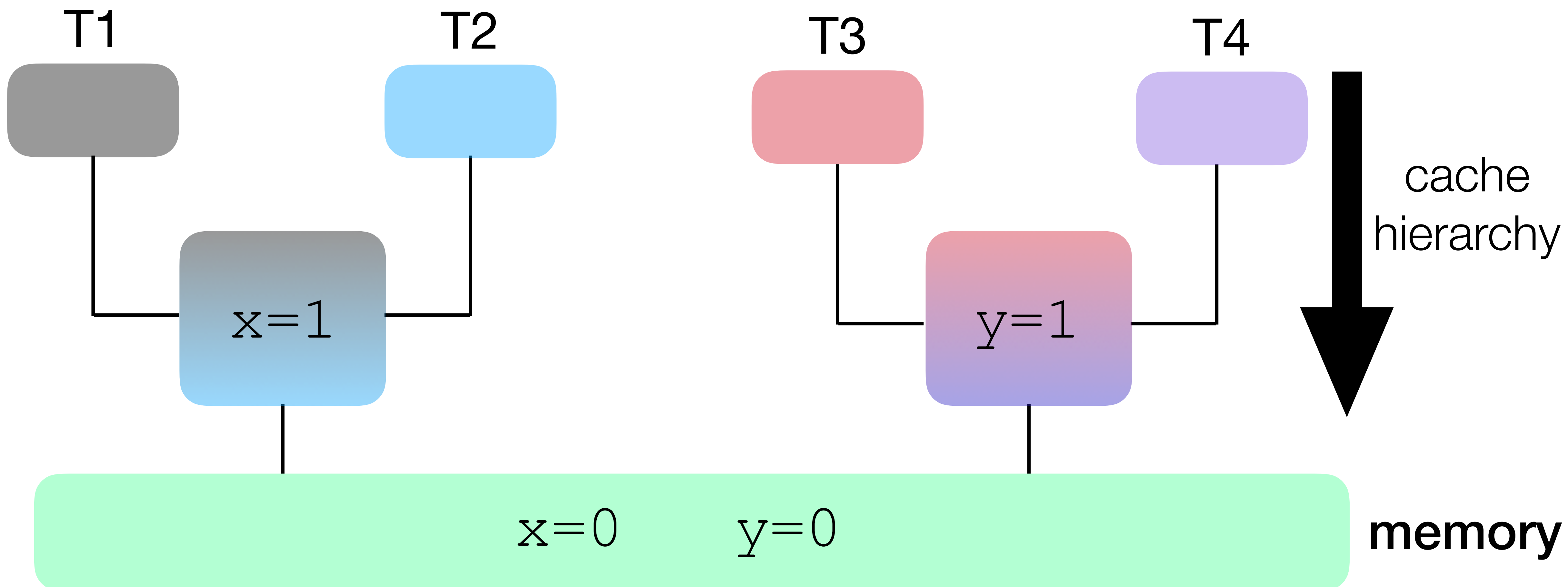
a `x:=1` ‖ b `b:=x;` //1 c `c:=y` //0 ‖ d `y:=1` ‖ e `e:=y;` //1 f `f:=x`



non-multi-copy-atomic architecture

WMC: Independent Reads of Independent Writes

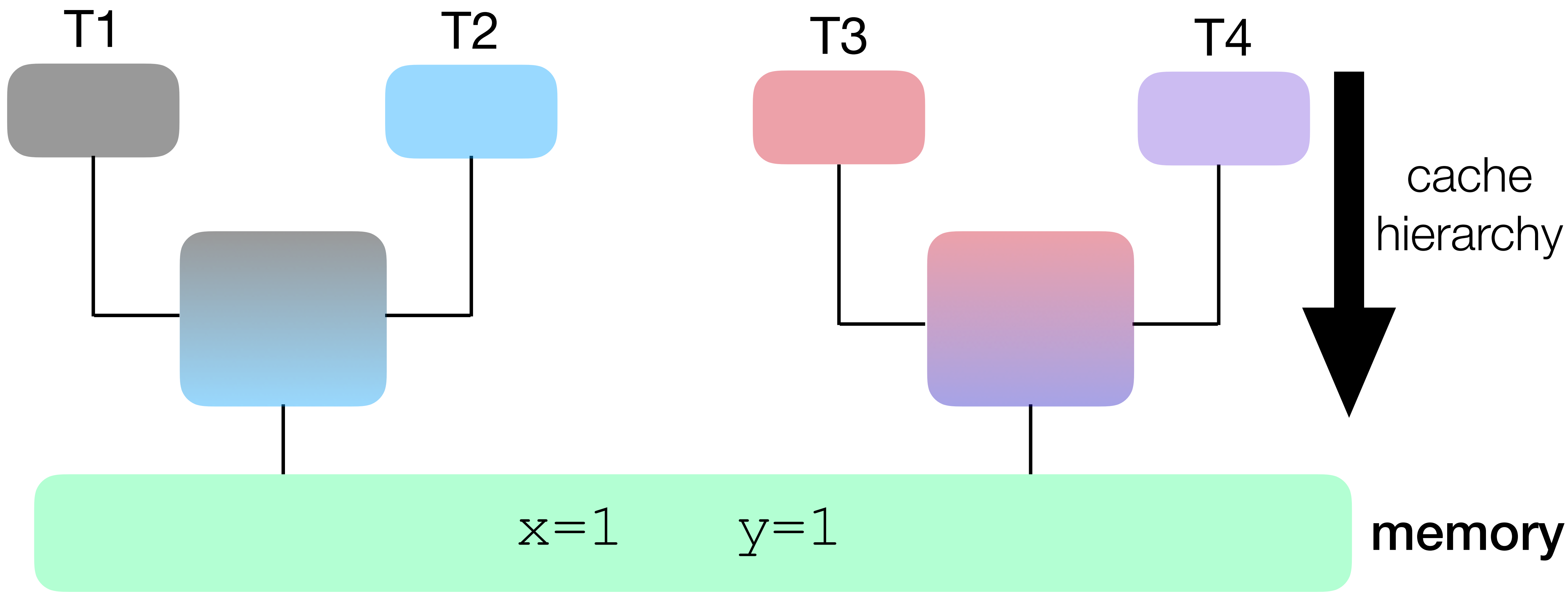
a `x:=1` ‖ b `b:=x;` //1 ; c `c:=y` //0 ‖ d `y:=1` ‖ e `e:=y;` //1 ; f `f:=x` //0



non-multi-copy-atomic architecture

WMC: Independent Reads of Independent Writes

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



non-multi-copy-atomic architecture

Weak Memory Concurrency (WMC)

No total execution order (*to*) ⇒

anomalies (***litmus tests***): behaviour absent under SC;
caused by:

- instruction ***reordering;***
  — e.g. store buffering (SB)
- ***different write propagation*** across cache hierarchy;
  — e.g. Independent Reads of Independent Writes (IRIW)

programming under WMC can be unintuitive and ***error-prone***

Weak Memory Concurrency (WMC)

No total execution order (*to*) ⟹

anomalies (***litmus tests***): behaviour absent under SC;
caused by:

- instruction
  — e.g. stor
- ***different w*** ...rchy;
  — e.g. Inde... s (IRIW)

progra... ***-prone***

Formal Methods

to the rescue!

WMC: State of the Art

- Formal Specification
  - ***Hardware*** (architecture) level WMC specification
    — e.g. x86-TSO, ARMv7, ARMv8, POWER, …
  - ***Software*** (language) level WMC specification
    — e.g. C/C++11, Java, …

WMC: State of the Art

- Formal Specification
  - ***Hardware*** (architecture) level WMC specification
    — e.g. x86-TSO, ARMv7, ARMv8, POWER, …
  - ***Software*** (language) level WMC specification
    — e.g. C/C++11, Java, …

- Formal Verification
  - Correctness of ***language to architecture compilation***, e.g. C11 to x86
  - Correctness of ***compiler transformations***
  - ***Verified compilers***, e.g. CompCertTSO
  - Case studies
    — Linux RCU, ARC, …

WMC: State of the Art

- Formal Specification
  - ***Hardware*** (architecture) level WMC specification
    — e.g. x86-TSO, ARMv7, ARMv8, POWER, …
  - ***Software*** (language) level WMC specification
    — e.g. C/C++11, Java, …

- Formal Verification
  - Correctness of ***language to architecture compilation***, e.g. C11 to x86
  - Correctness of ***compiler transformations***
  - ***Verified compilers***, e.g. CompCertTSO
  - Case studies
    — Linux RCU, ARC, …

low-level **!**

WMC: State of the Art

- Formal Specification
  - ***Hard***
    - e.g
  - ***Softw***
    - e.g
- Formal
  - Corre ... . C11 to x86
  - Corre
  - ***Verifi***
  - Case
    - Linux RCU, ARC, …

What about

high-level

concurrent library

specification & verification

under

WMC?

low-level **!**

Part I.

Concurrent Library ***Specification*** under WMC

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, ...

wanted

general

MM-agnostic

declarative

specification & verification

framework

Declarative Framework Desiderata

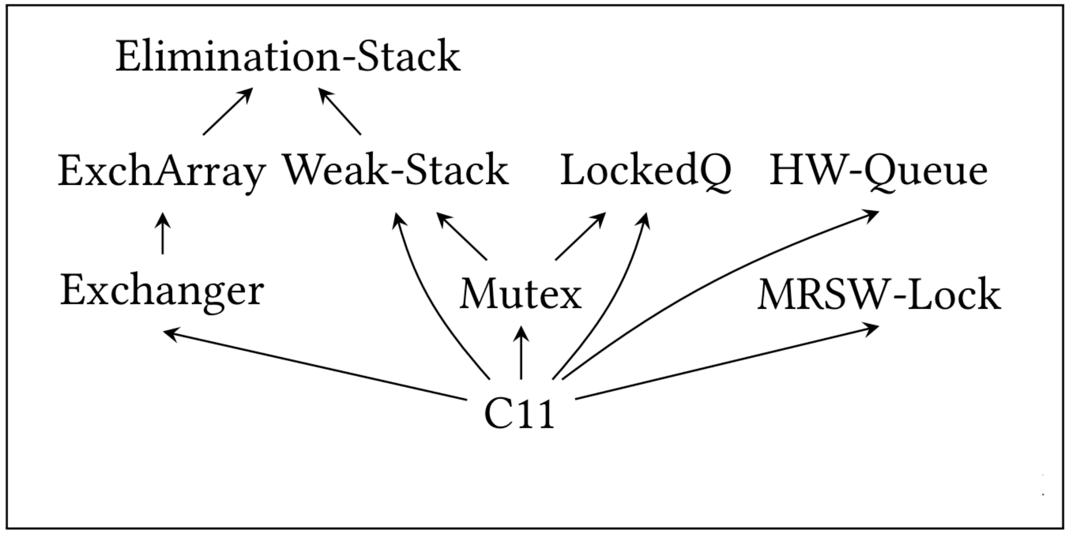
- ***Agnostic*** to memory model
  - support *both SC* and *WMC* specs
- ***General***
  - port *existing SC (linearisability) specs*
  - port *existing WMC specs* (e.g. C11, TSO)
  - built from the *ground up*: assume *no pre-existing* libraries or specs
- ***Compositional***
  - verify *client programs*

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

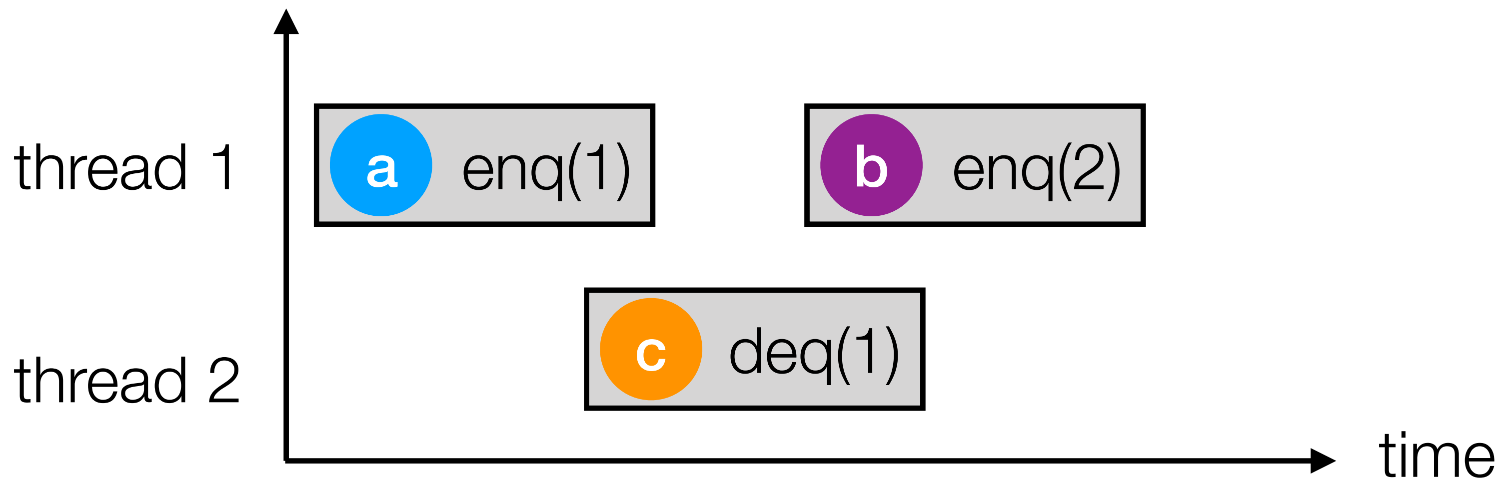
Declarative Framework Desiderata

- ***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
```



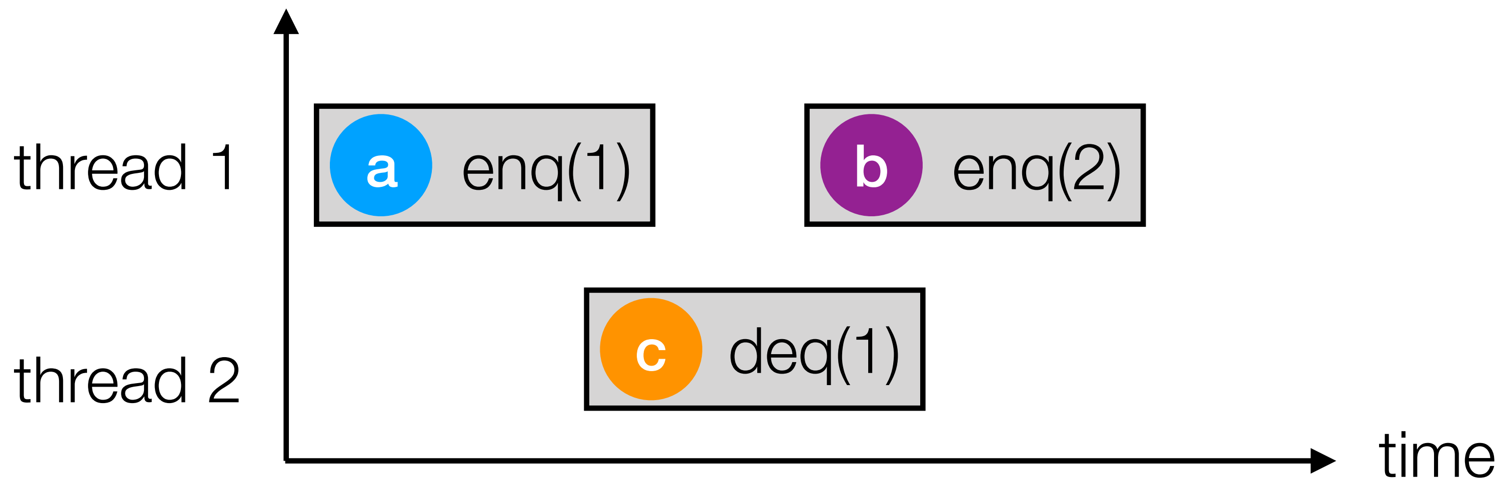
Linearisability



- 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) $\in$ *hb*   (a, c) $\notin$ *hb*

Linearisability

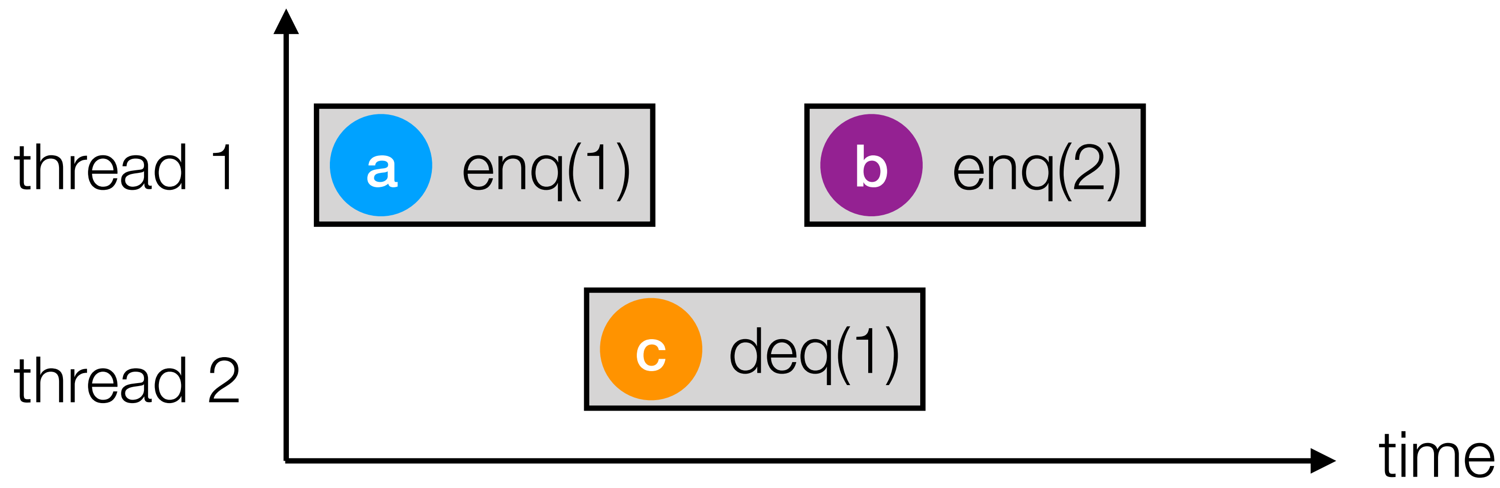


- Define a (partial) happens-before relation *hb* on events
  - $(e_1, e_2) \in hb \iff e_1.\text{end} <_{\text{time}} e_2.\text{begin}$

    -- 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

Linearisability



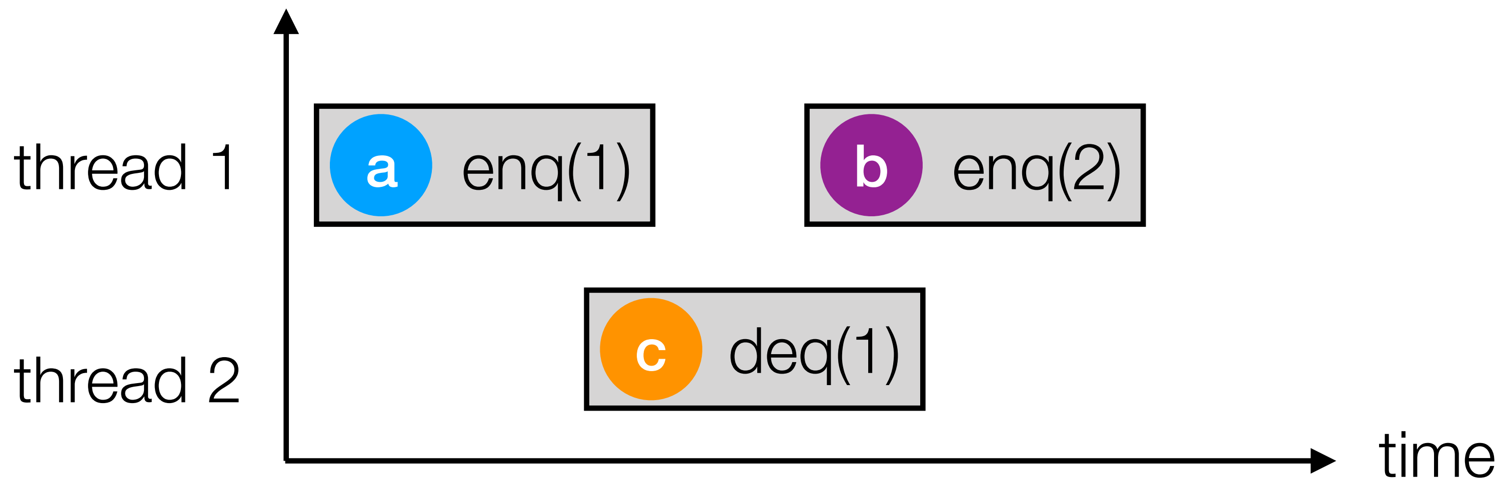
- Define a (partial) happens-before relation *hb* on events
  - $(e_1, e_2) \in hb \iff e_1.\text{end} <_{\textbf{time}} e_2.\text{begin}$

    -- e.g. (a, b) $\in$ *hb* (a, c) $\notin$ *hb*

- ***Linearisable*** $\iff \exists$ *to*. *to* ***totally*** orders events
  - *hb* $\subseteq$ *to*
  - *to* is a ***legal*** sequence (library-specific)

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

Linearisability

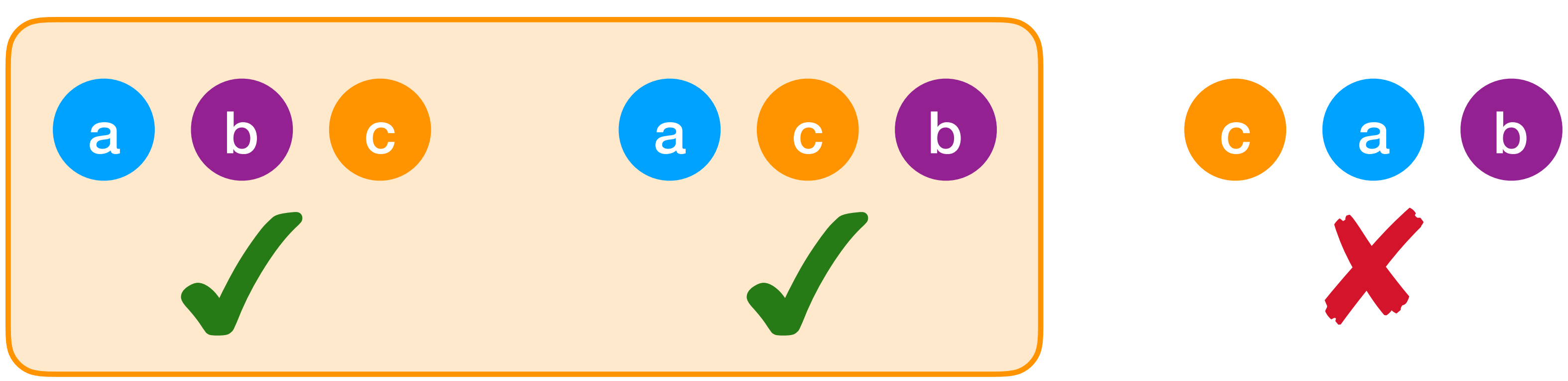


- Define a (partial) happens-before relation *hb* on events
  - $(e_1, e_2) \in hb \iff e_1.\text{end} <_{\text{time}} e_2.\text{begin}$

    -- 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

linearisable



Why Not Linearisability?

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

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

(SB)

✗ not linearisable

Why Not Linearisability?

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

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

(SB)

✗ not linearisable

$d \xrightarrow{to} a$ $\qquad$ $b \xrightarrow{to} c$

Why Not Linearisability?

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

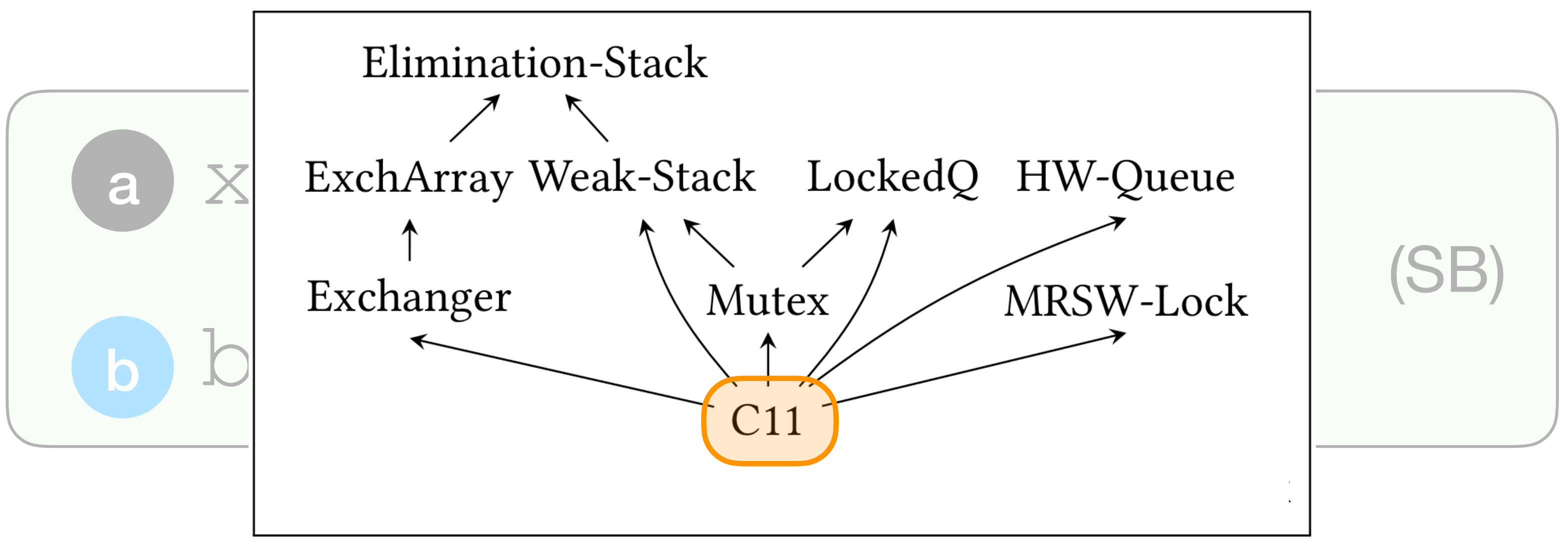
```
a x:=1;        ||  c y:=1;
b b:=y  // 0   ||  d d:=x  // 0
```

(SB)

✘ not linearisable

Why Not Linearisability?

- ✗ assumes $<_{\text{time}}$ order -- not present under WMC
- ✗ requires ***total*** order 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*** **<$_{\text{time}}$** 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 (⊥)
```

$$G_{queue} = < \underbrace{E}_{\text{events}}, po , so , hb >$$

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

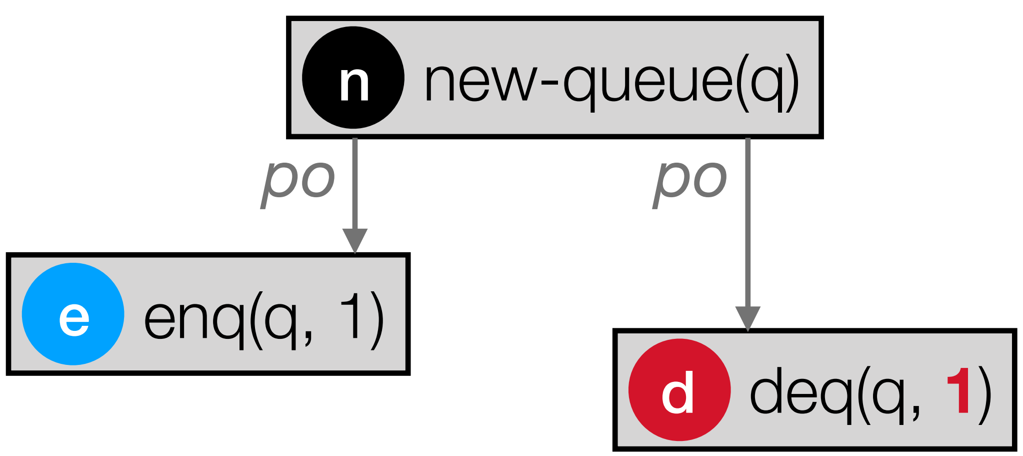
Example: ***Queue*** Library

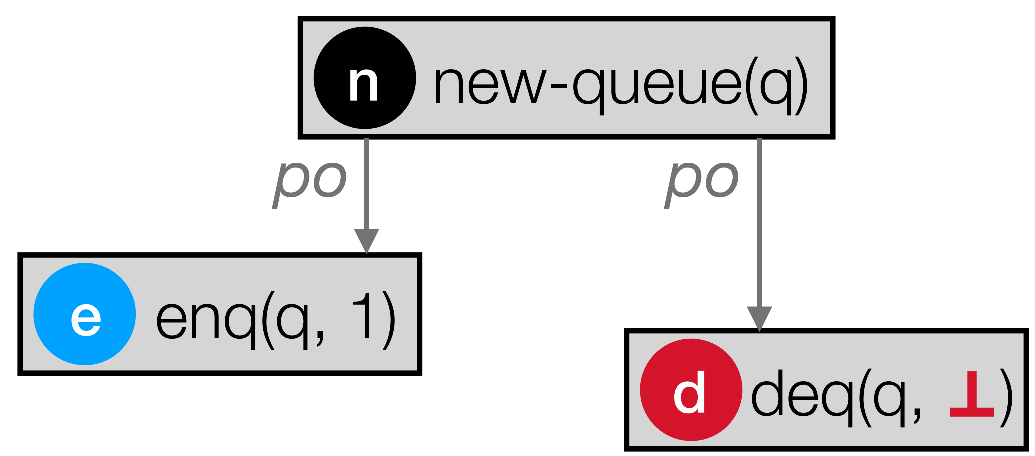
```
q:=new-queue();
s:=new-stack();

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

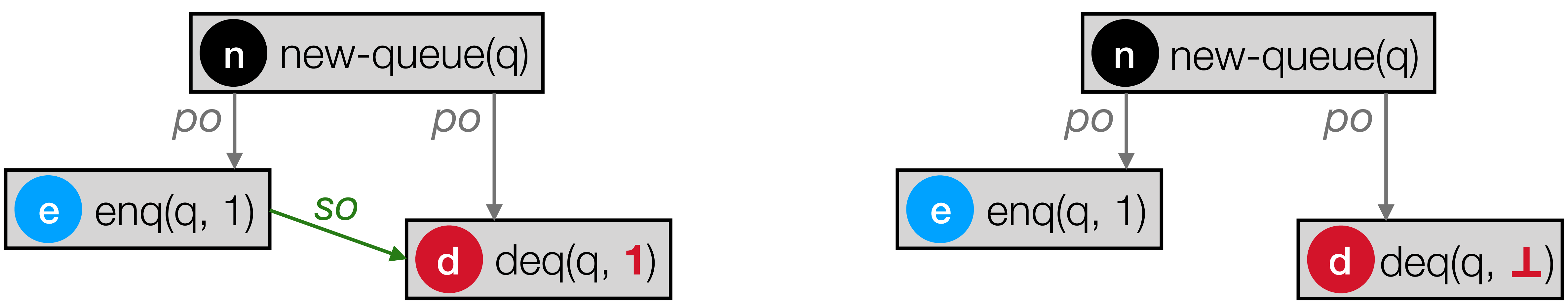
```
q:=new-queue();
s:=new-stack();

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

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

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 (⊥)
```

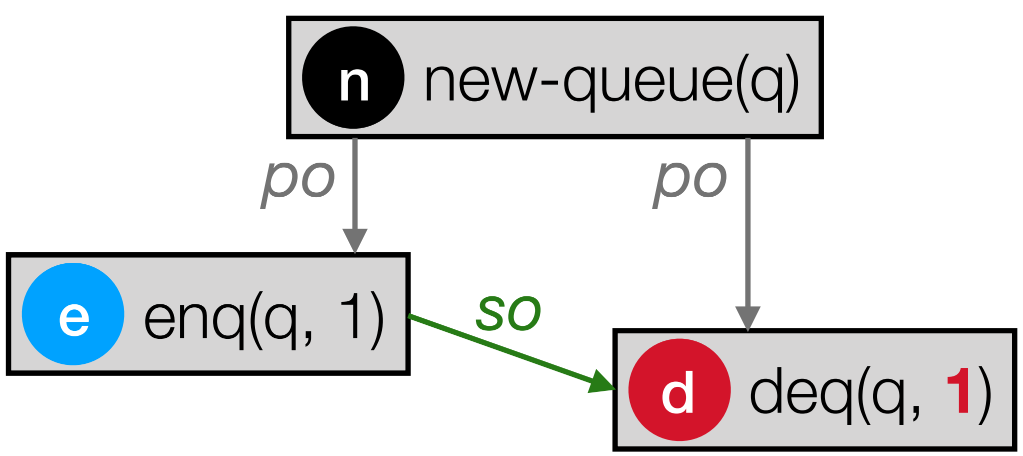
$$G_{queue} = \langle E, po, so, hb \rangle$$

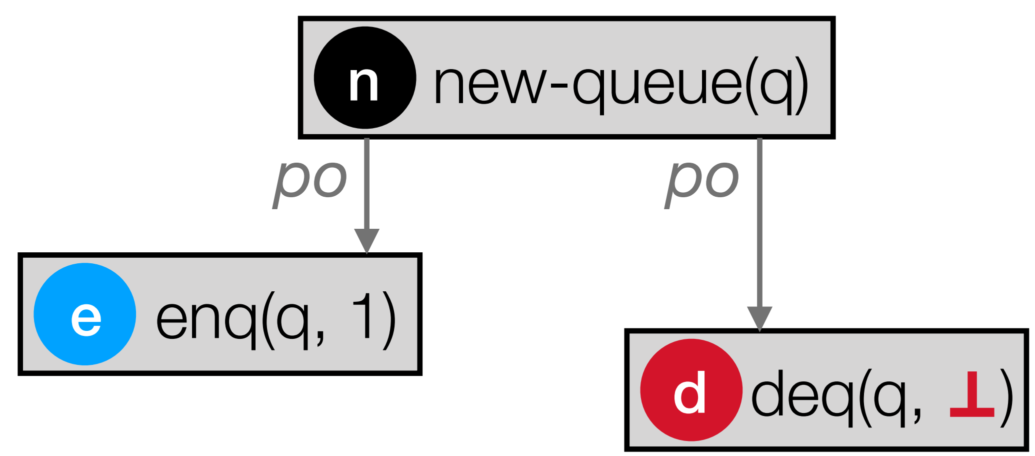
events

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 (⊥)
```

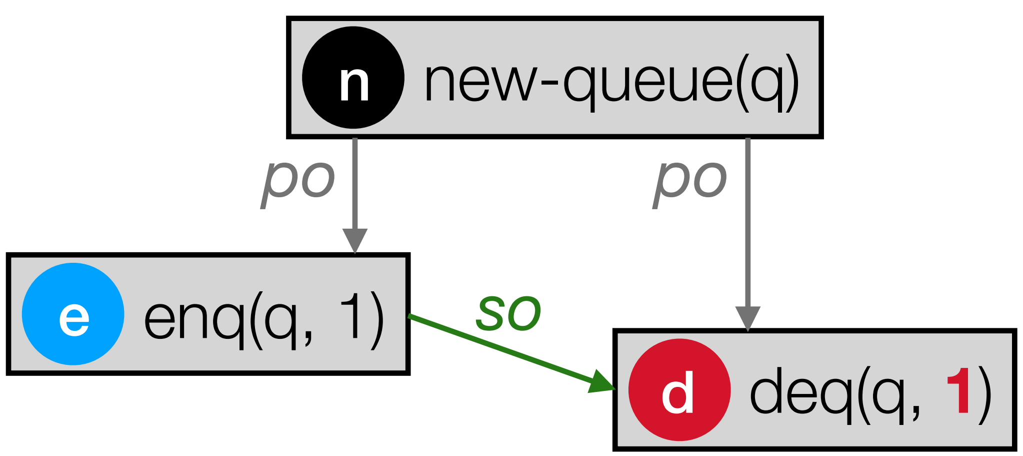
$$G_{queue} = < E, po, so, hb >$$

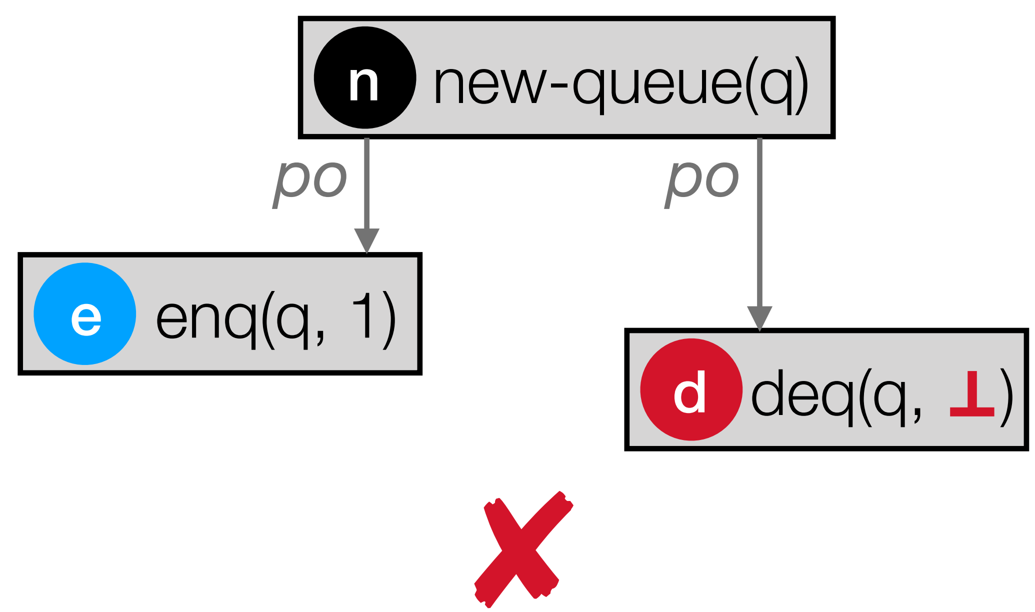
events

program-order

synchronisation-order

happens-before order (coming soon!)





Library Executions

Example: ***Queue*** Library

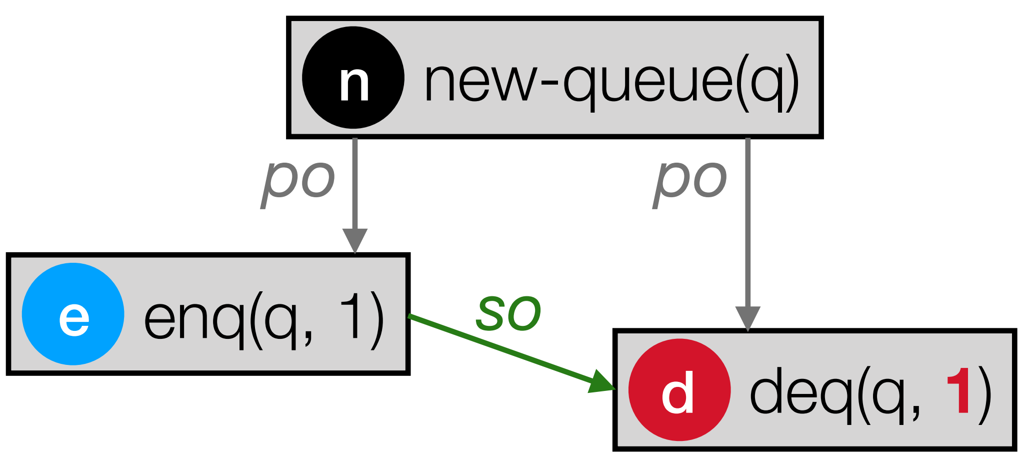
`q:=new-queue();`
`s:=new-stack();`

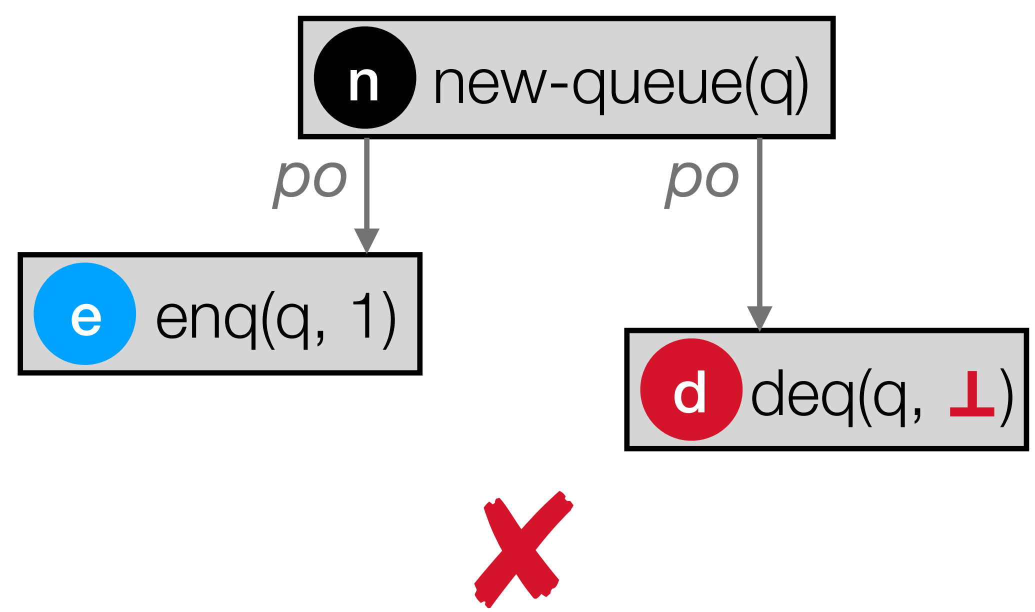
`enq(q,1);`
`push(s,2)`

||

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

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





Program Executions

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

semantics of P

execution of P

events

program-order

synchronisation-order
(library-specific)

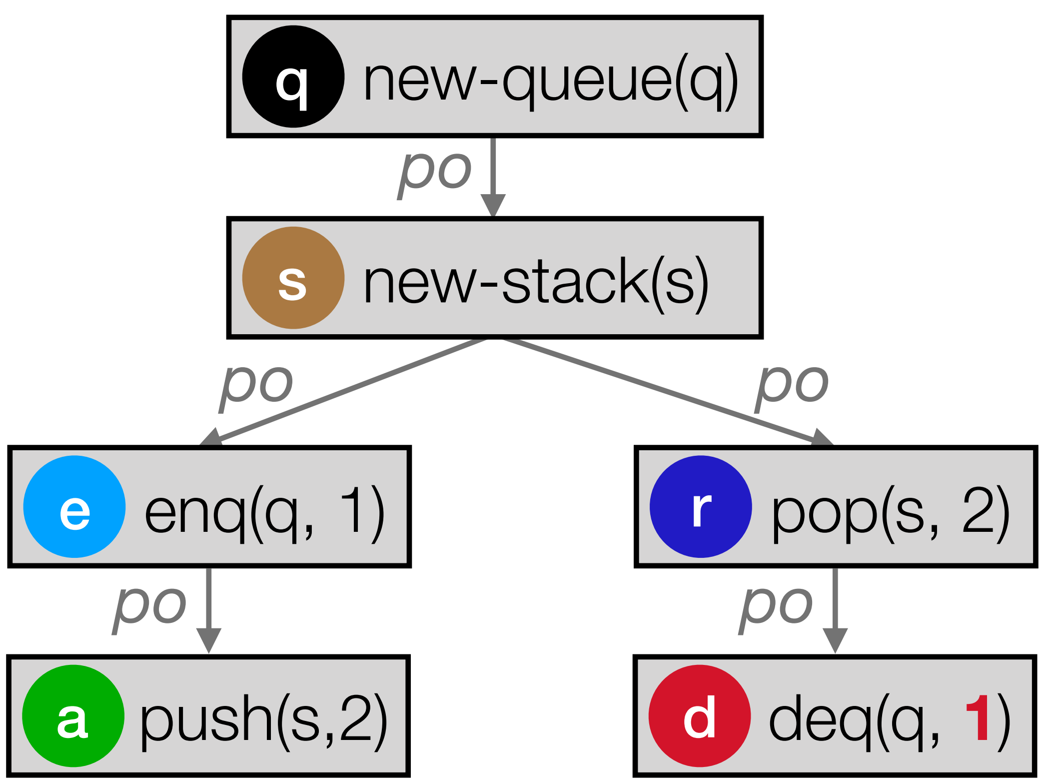
Program Executions

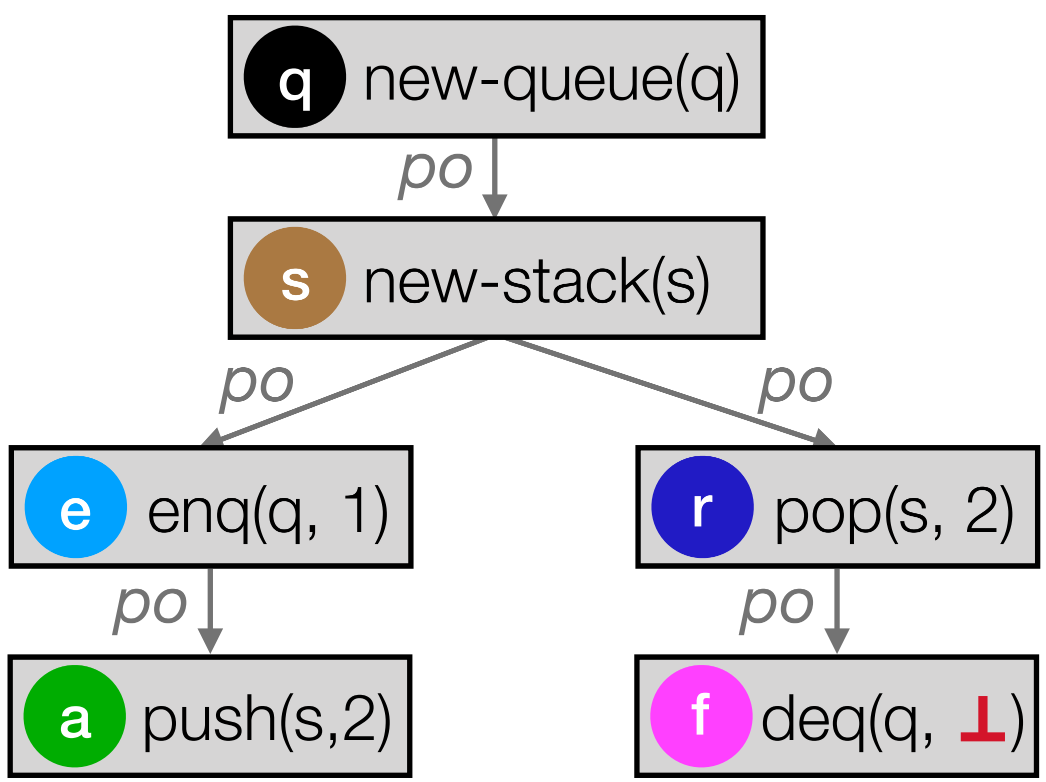
$$[\![P]\!] = \{ G_P = < E, po, so > \mid \cdots \}$$

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
```





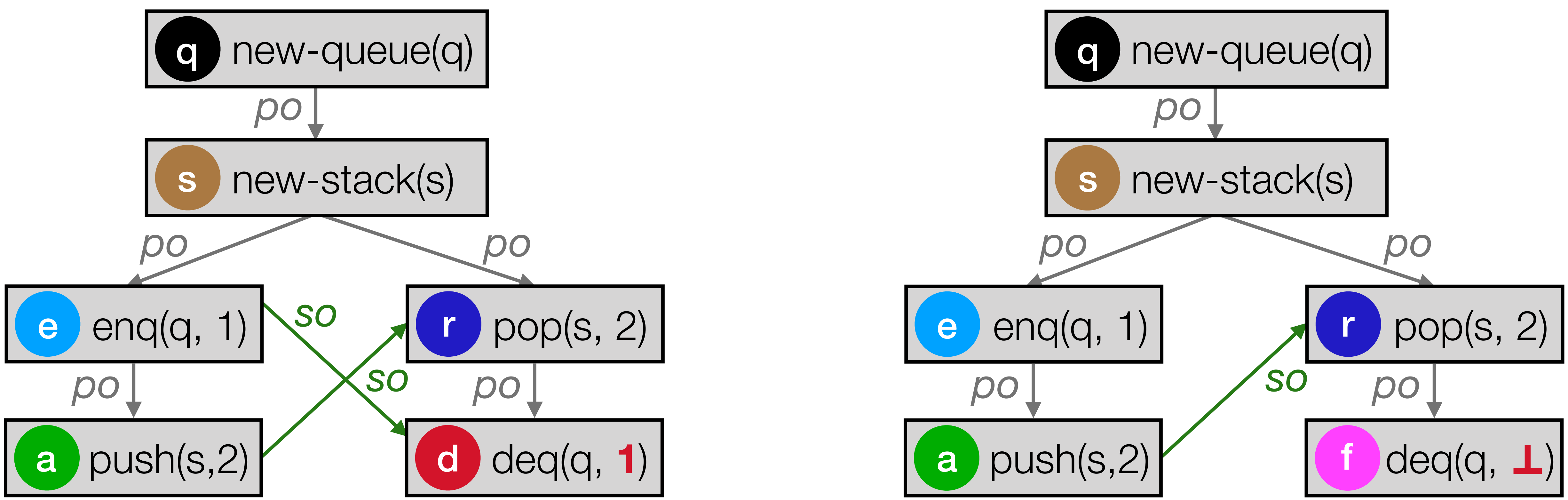
Program Executions

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

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

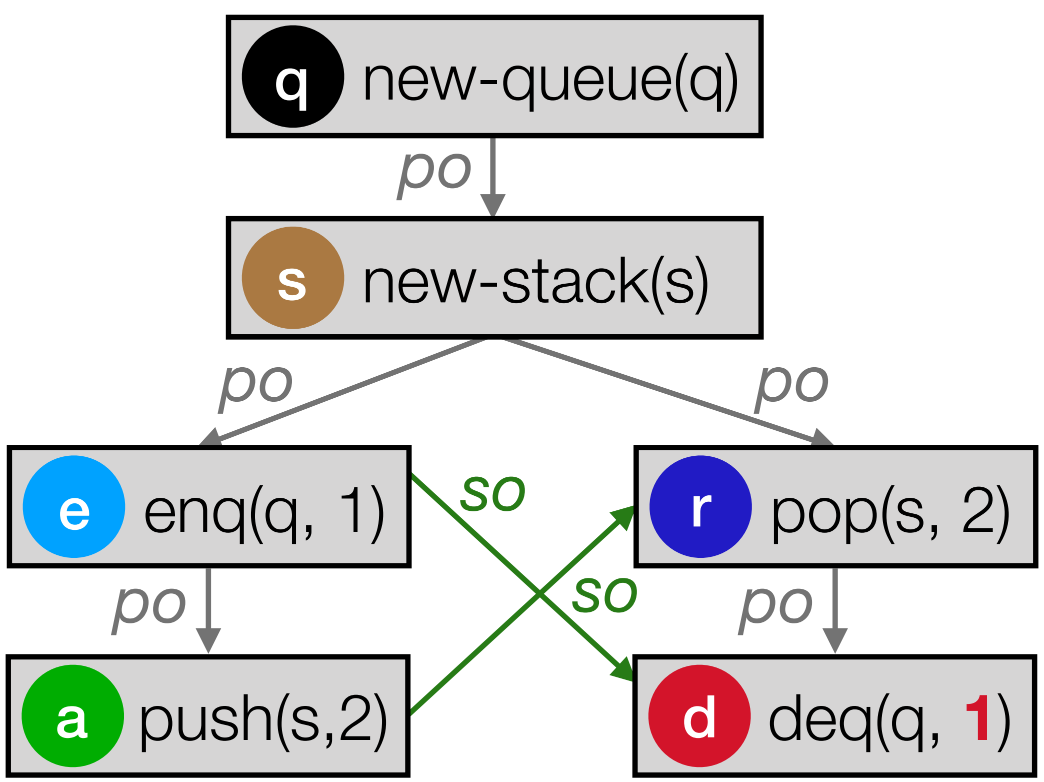
$$[\![P]\!] = \{ G_P = \langle\, E,\ po,\ so \,\rangle \mid \cdots \}$$

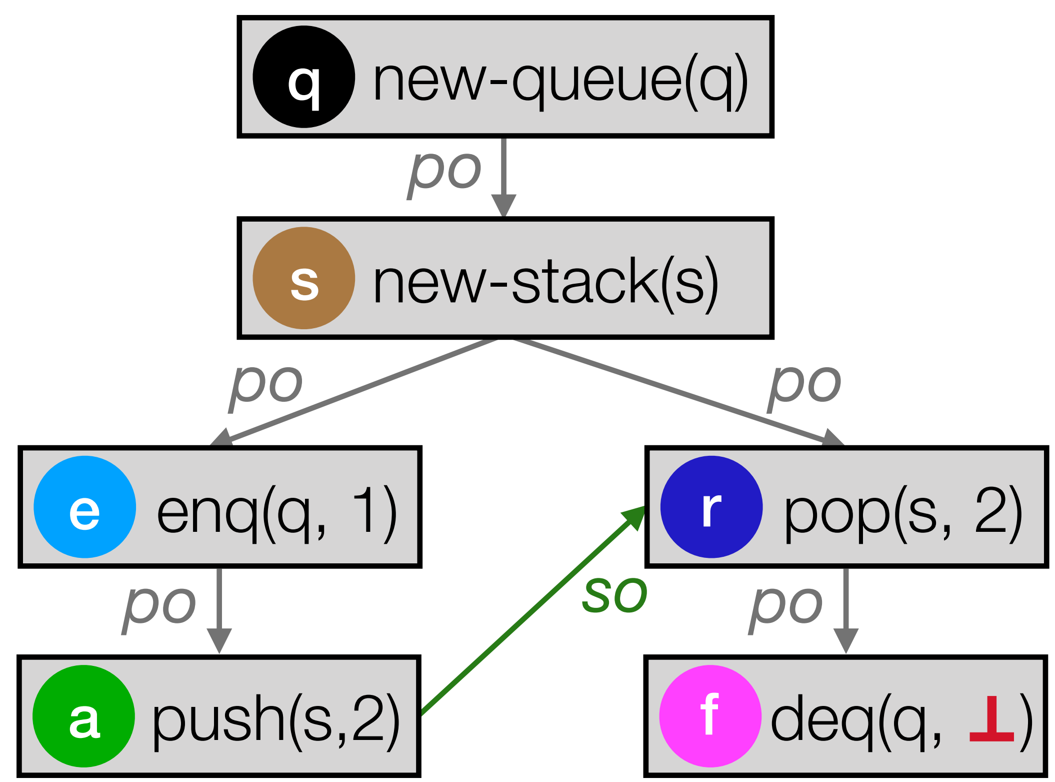
semantics of *P*

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

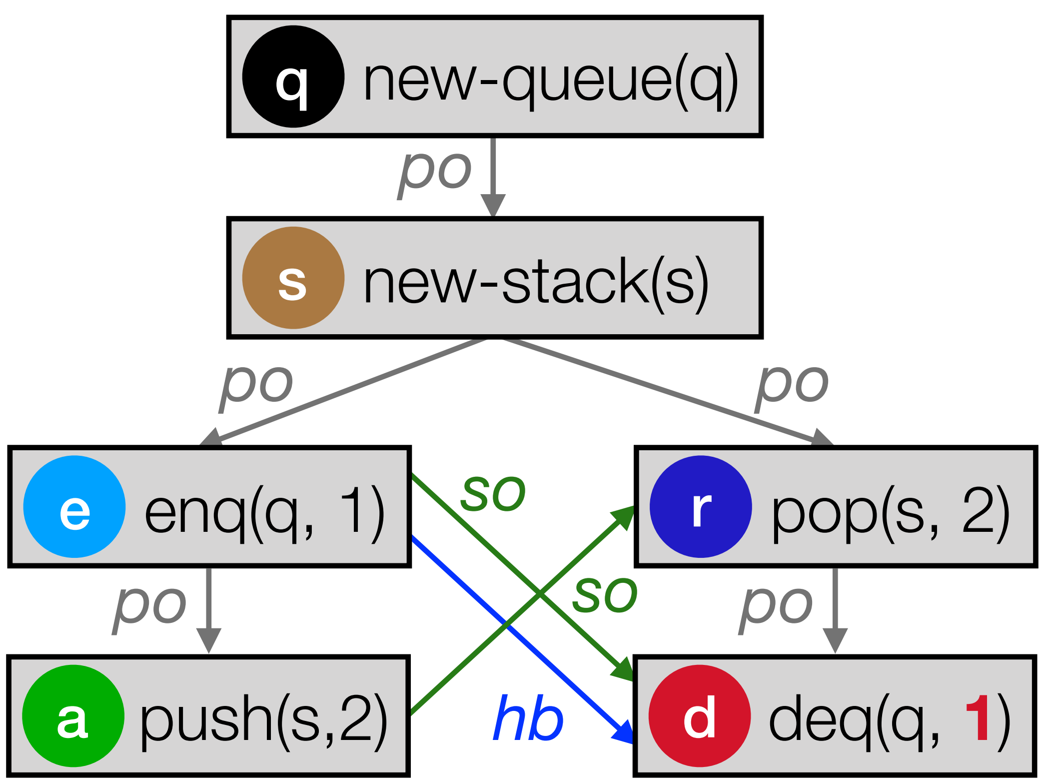
$$[\![P]\!] = \left\{ G_P = < E\ ,\ po\ ,\ so > \ \middle|\ \cdots \right\}$$

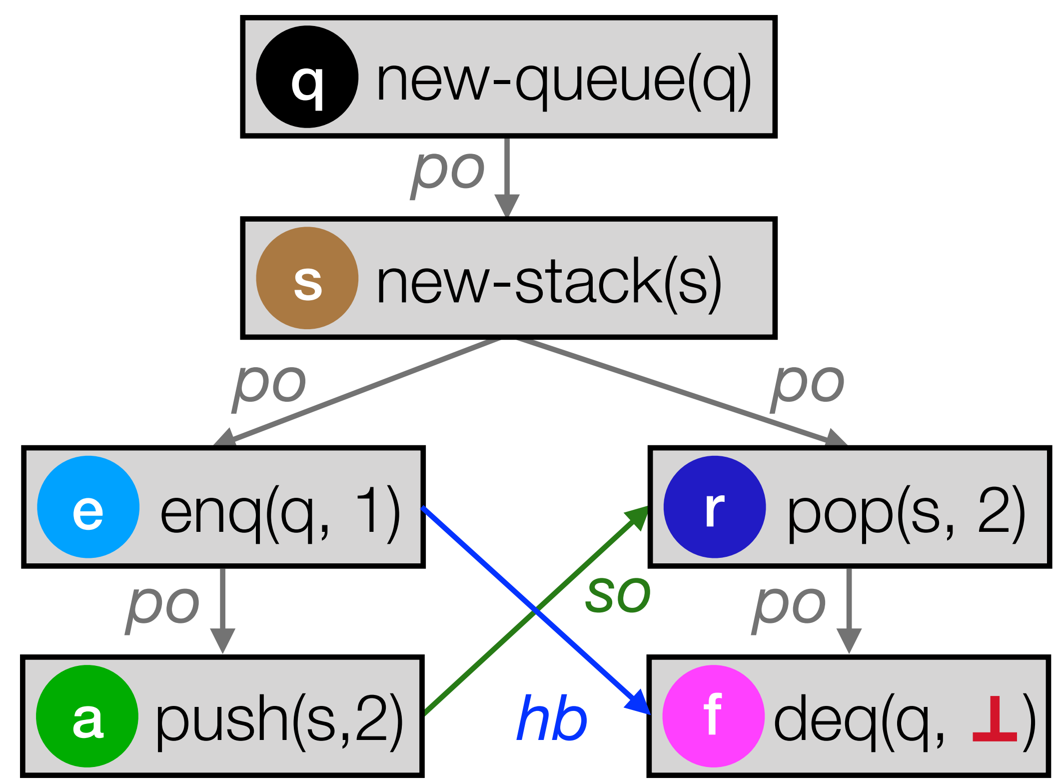
semantics of P

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

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

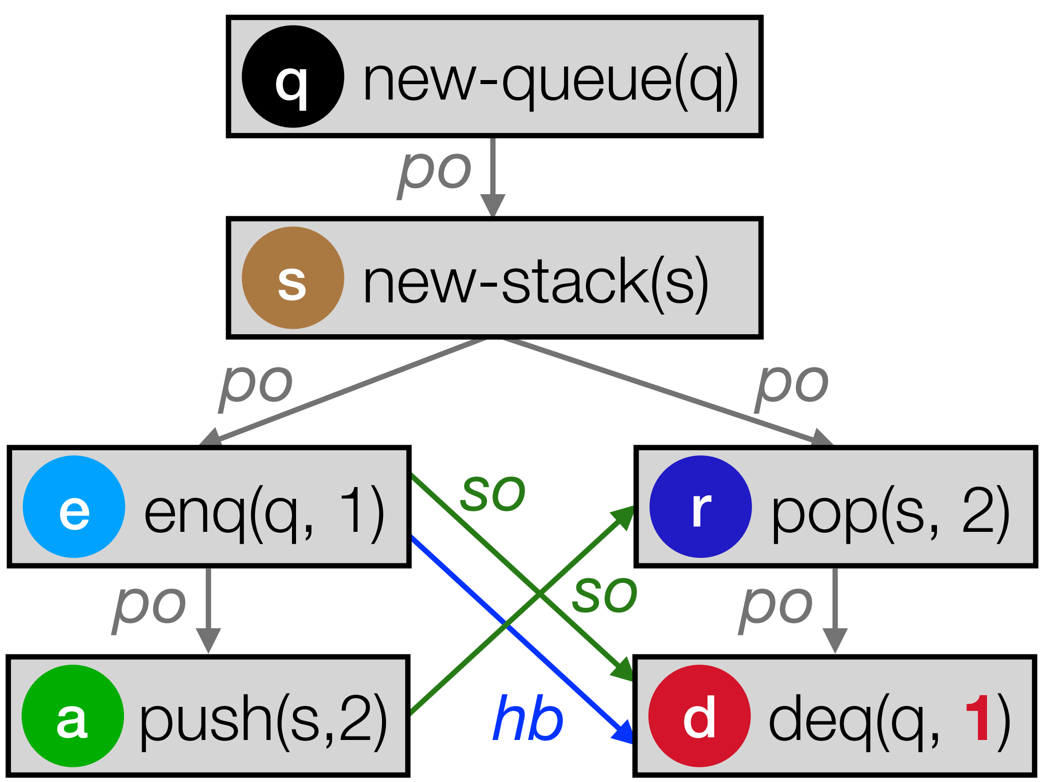
semantics of *P*

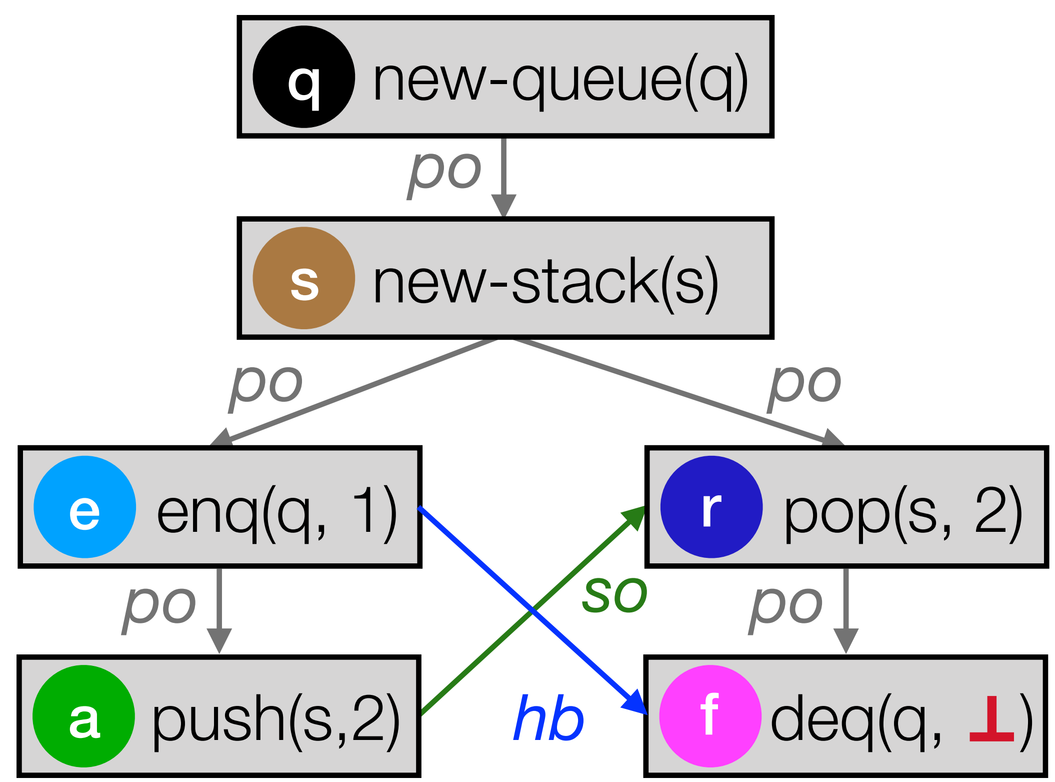
execution of *P*

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

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

How? *hb* defined on program executions

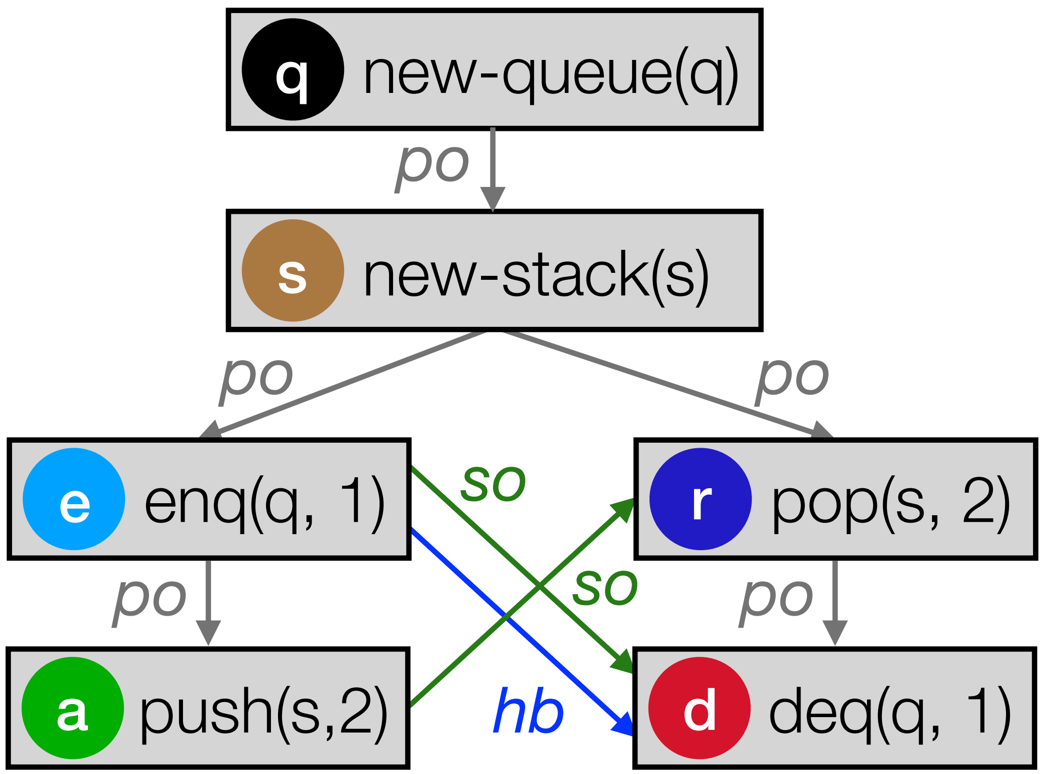




From *Program* to *Library* Executions

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

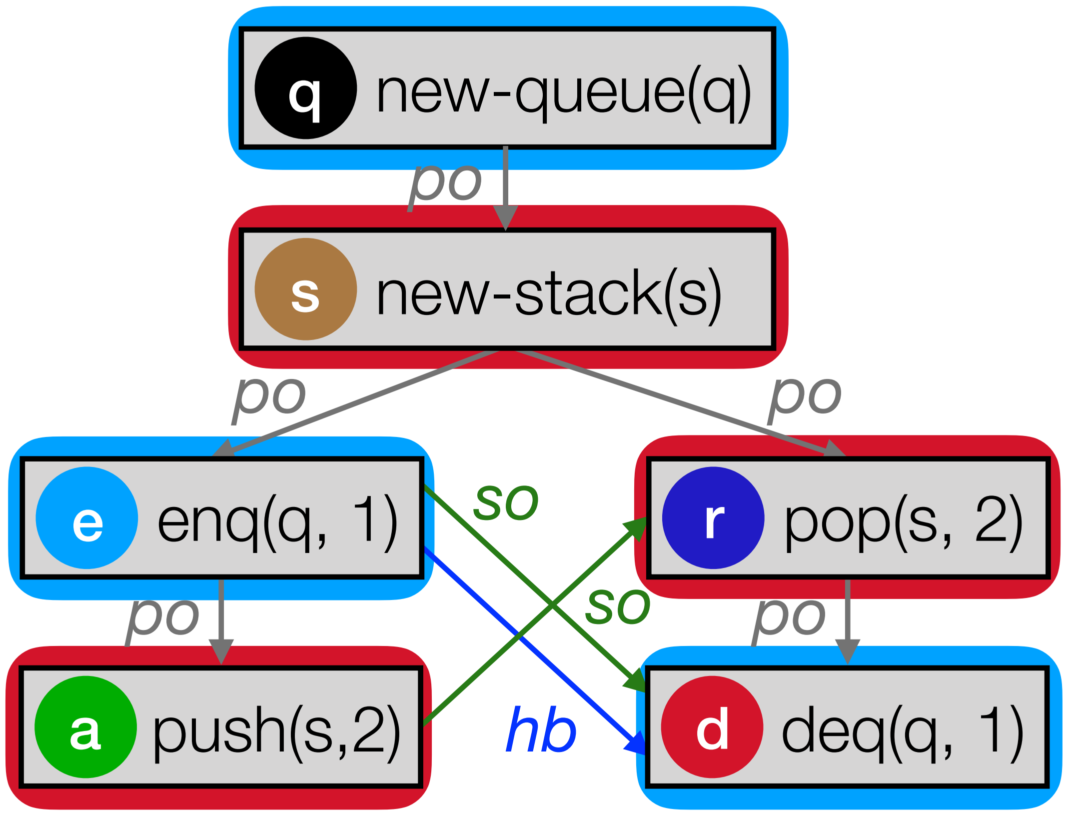
$hb = (\, po \cup so \,)^+$



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

$$G_P = \langle E, po, so \rangle \quad E = E_{queue} \cup E_{stack}$$

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

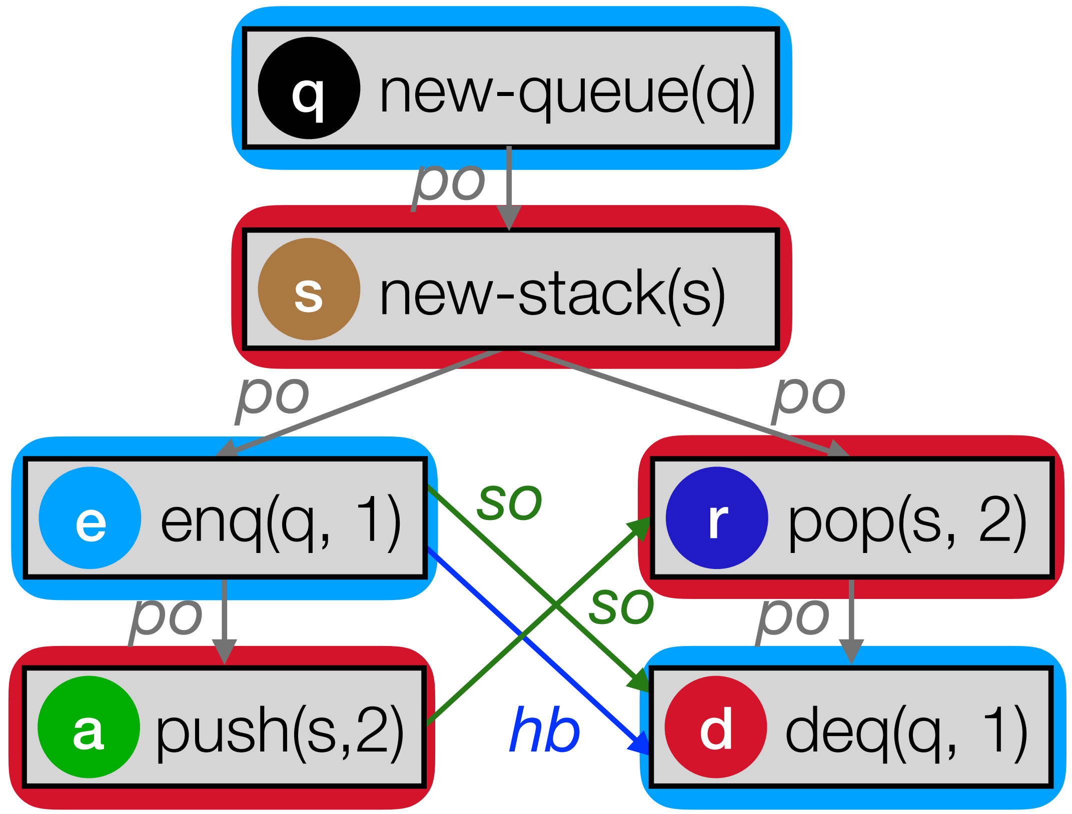


From ***Program*** to ***Library*** Executions

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

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

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



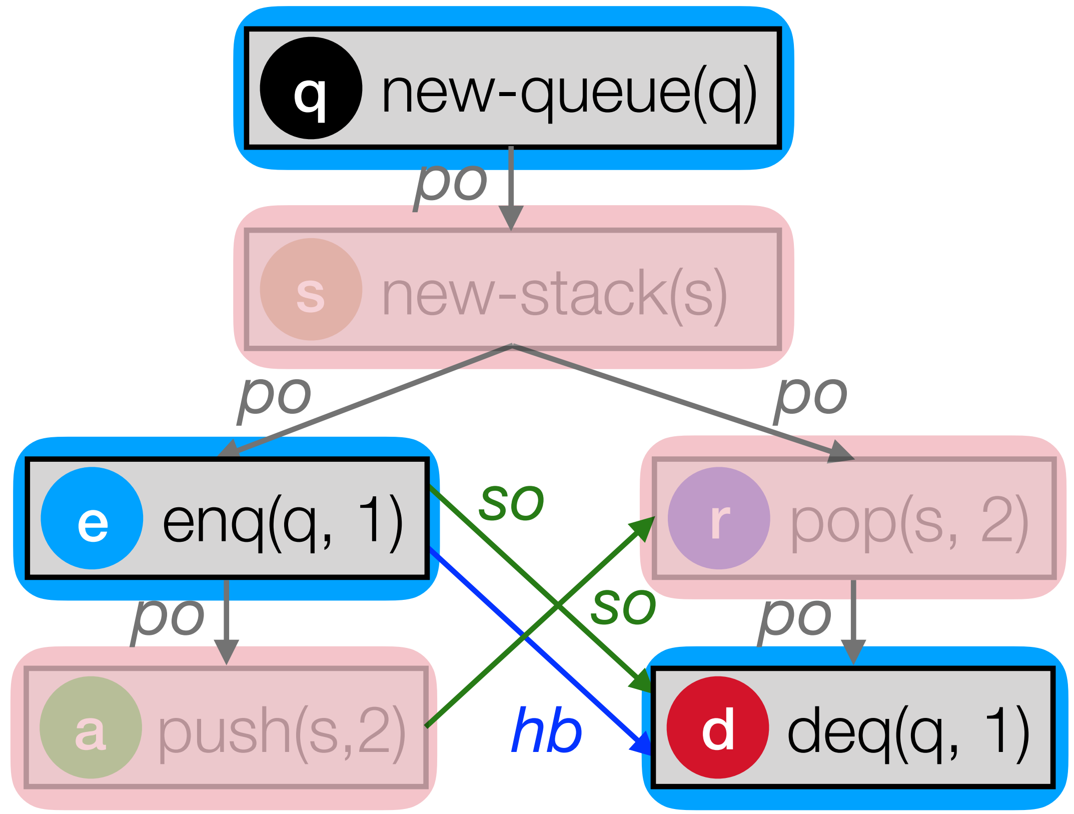
From ***Program*** to ***Library*** Executions

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

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

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

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



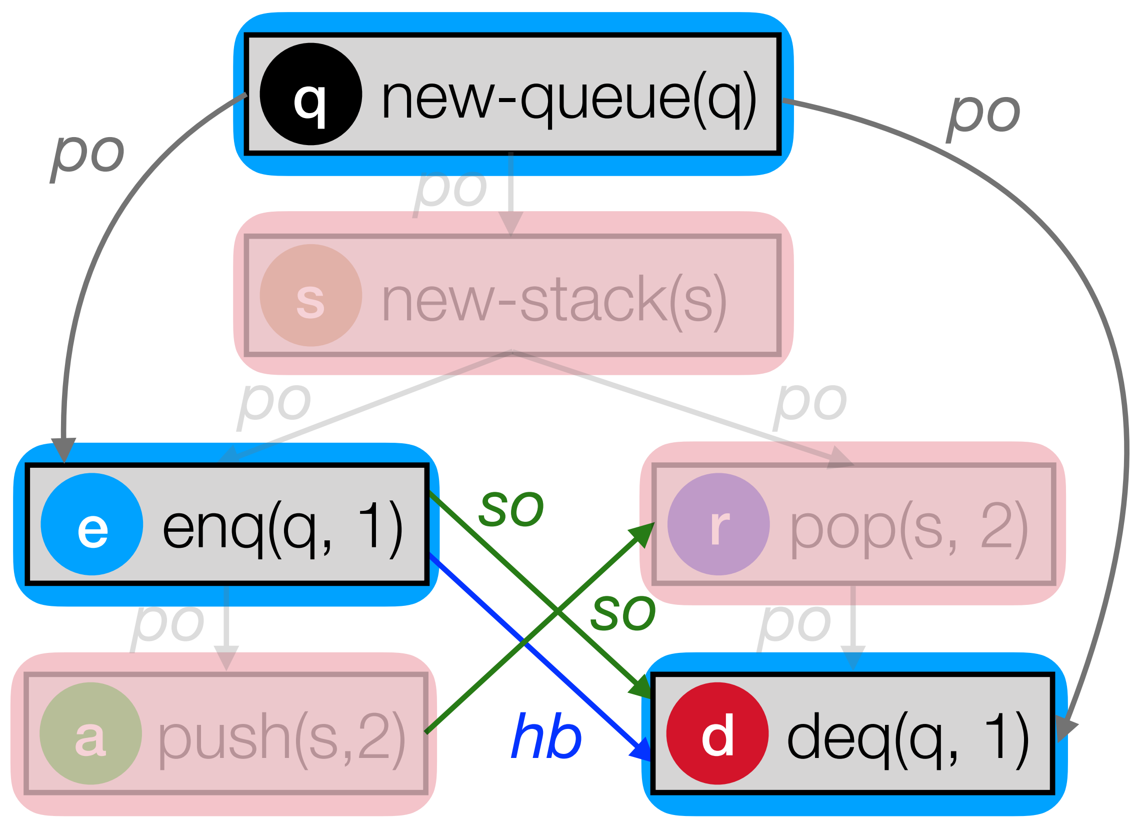
$$G_{queue} = \langle\ E_{queue}\ ,\ \qquad\qquad\qquad\qquad\qquad\qquad \rangle$$

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

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

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

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



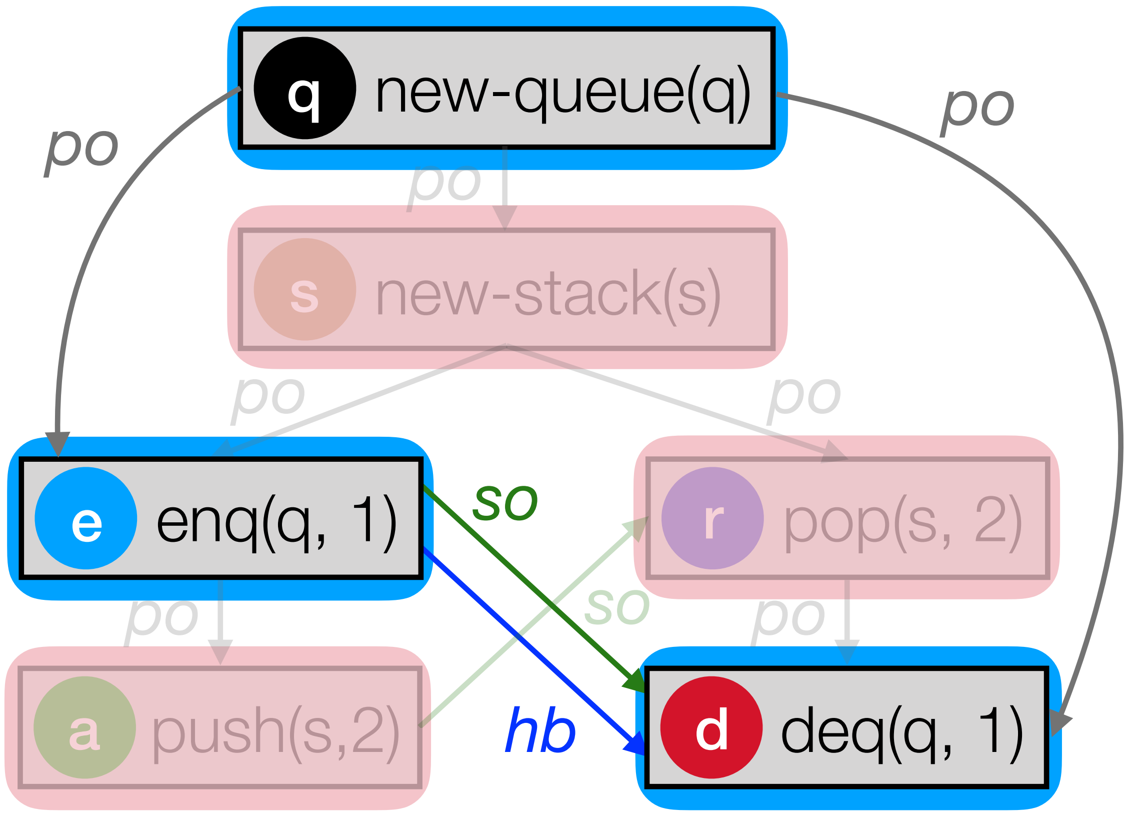
$G_{queue} = \langle\ E_{queue}\ ,\ po_{queue}\ ,\ \qquad \rangle$

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

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

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

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



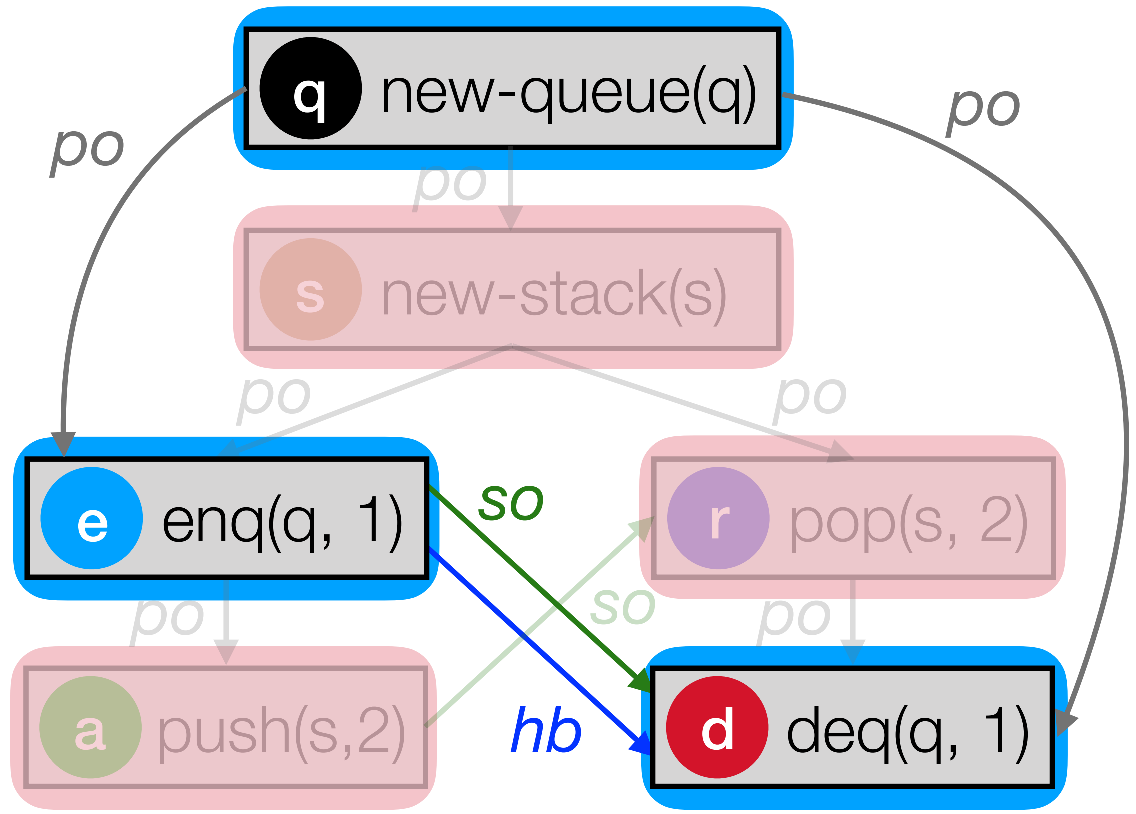
$G_{queue} = \langle\ E_{queue}\ ,\ po_{queue}\ ,\ so_{queue}\ , \qquad \rangle$

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

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

$G_P = G_{queue} \oplus G_{stack}$, $E = E_{queue} \cup E_{stack}$

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

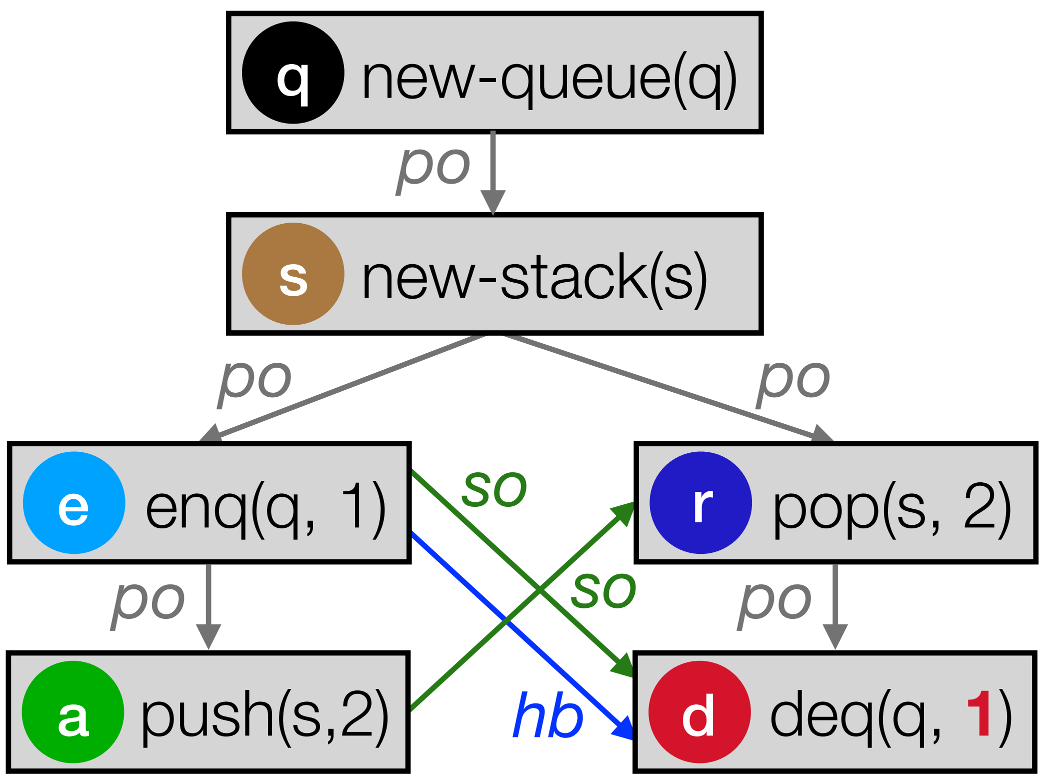


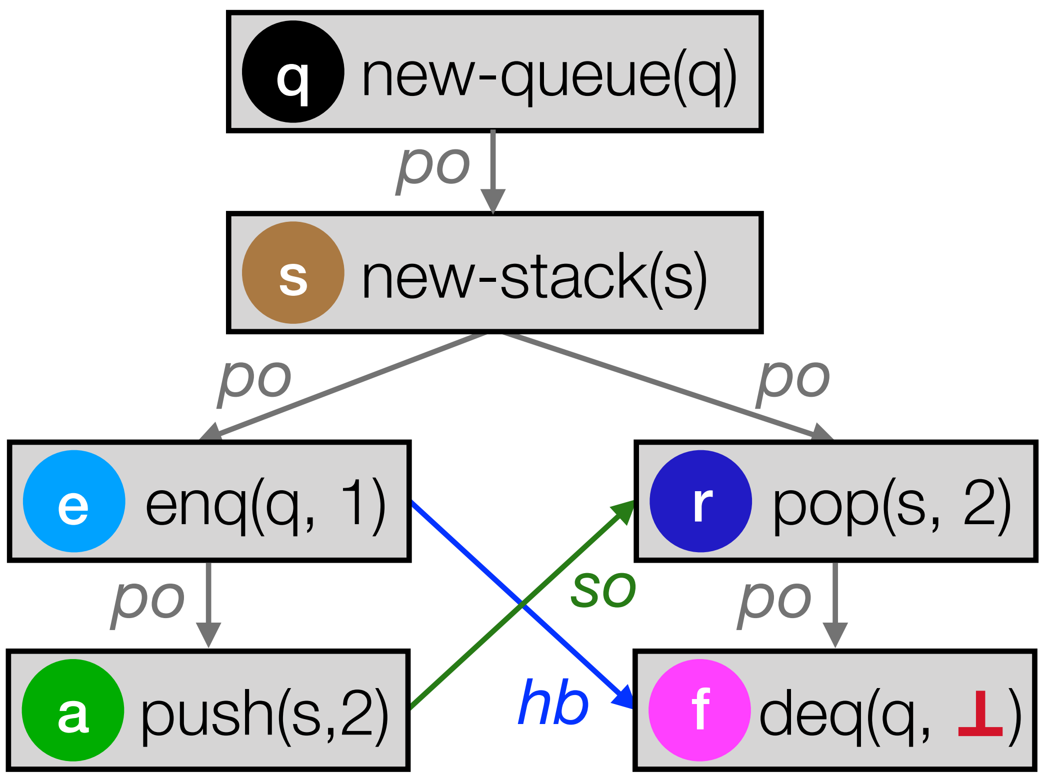
$$G_{queue} = \langle\ E_{queue}\ ,\ po_{queue}\ ,\ so_{queue}\ ,\ hb?\ \rangle$$

$hb? = hb_{queue}$

Example Revisited

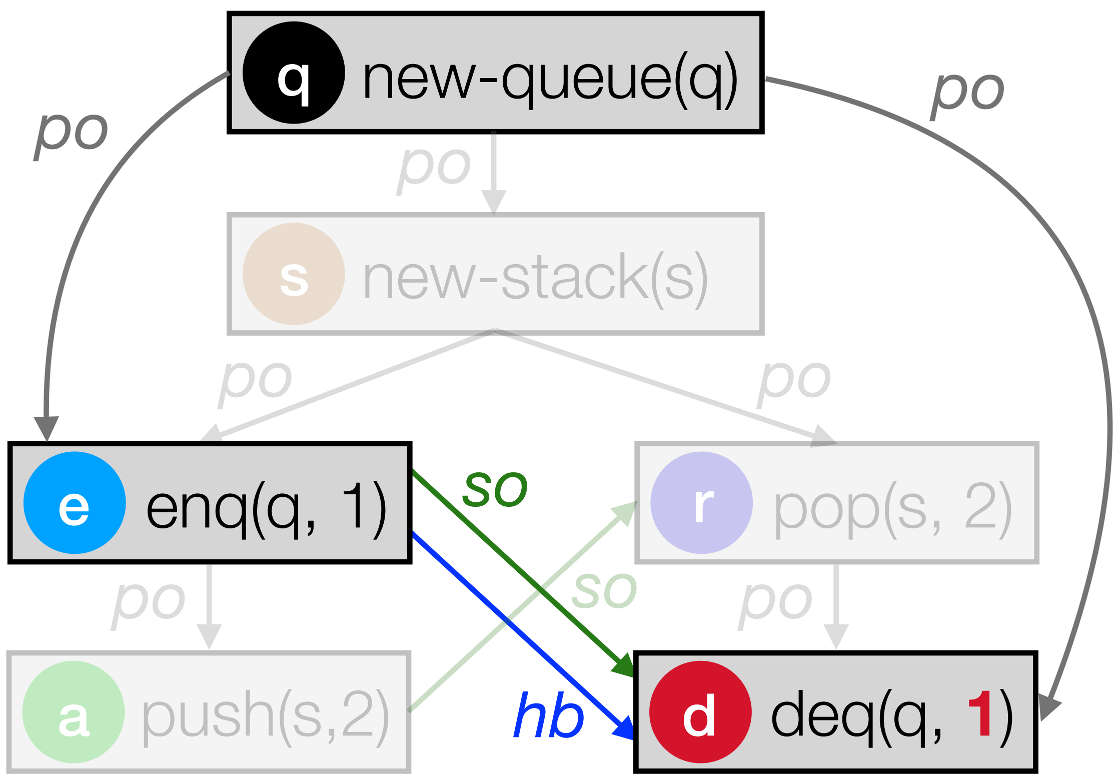
```
q:=new-queue();
s:=new-stack();

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

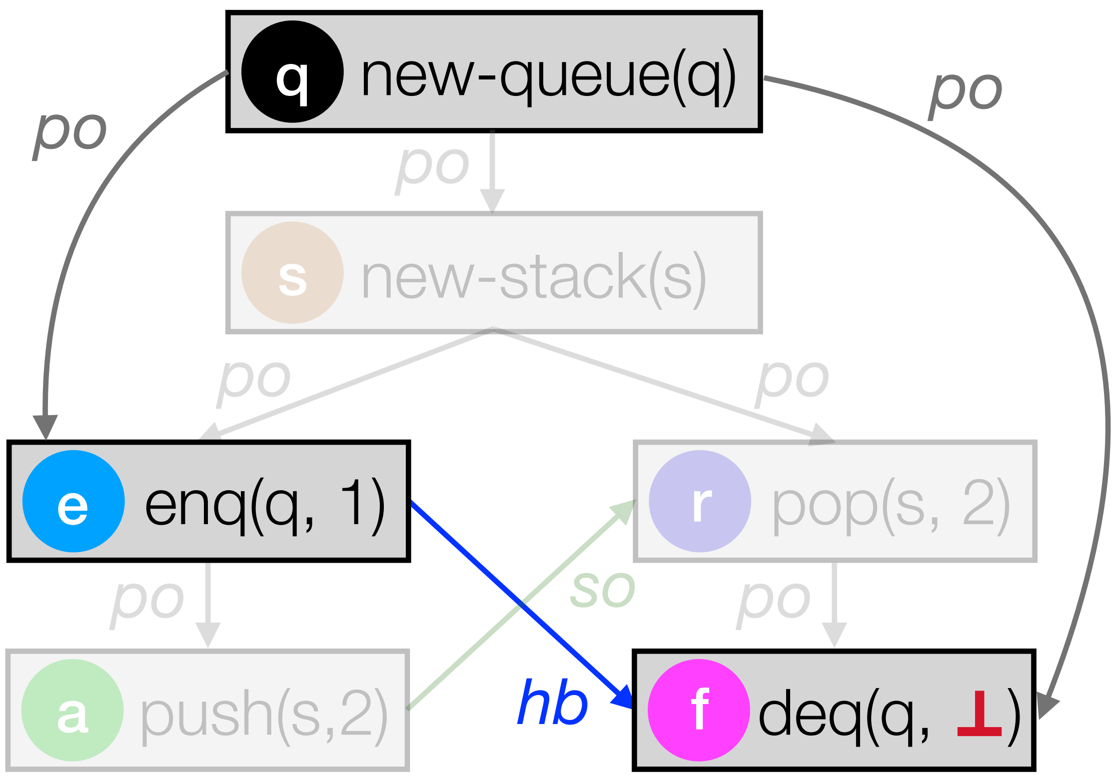




Example Revisited

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

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



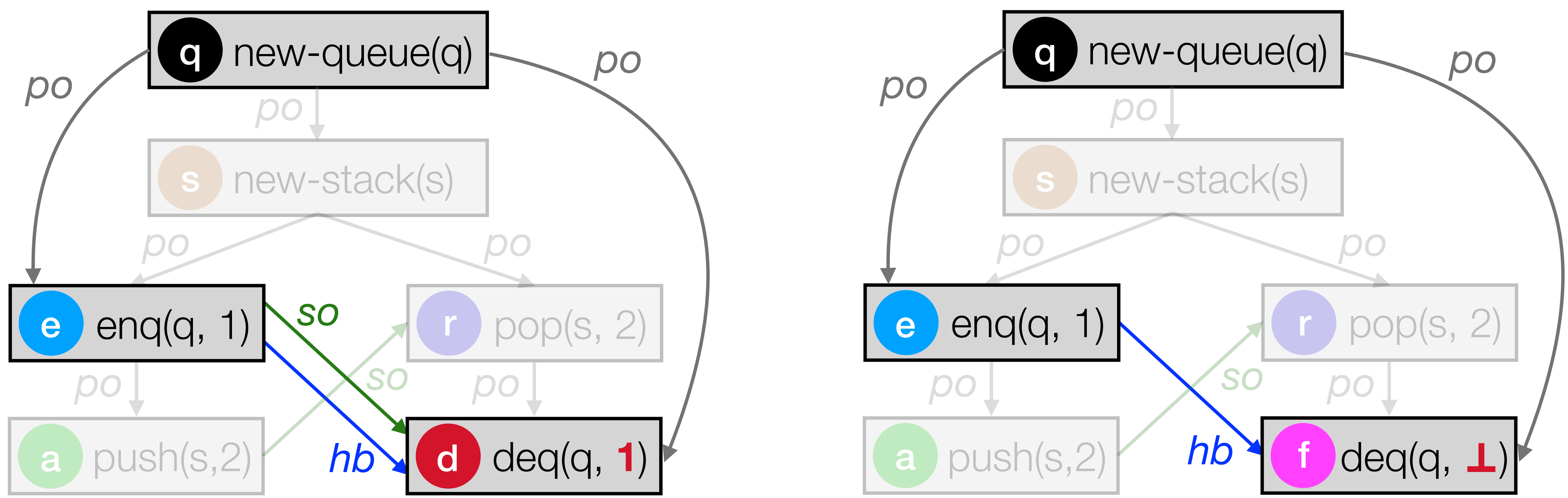


Example Revisited

```
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

$G = \langle\ E, po, so, hb\ \rangle$ 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 ∈ 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 = < *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

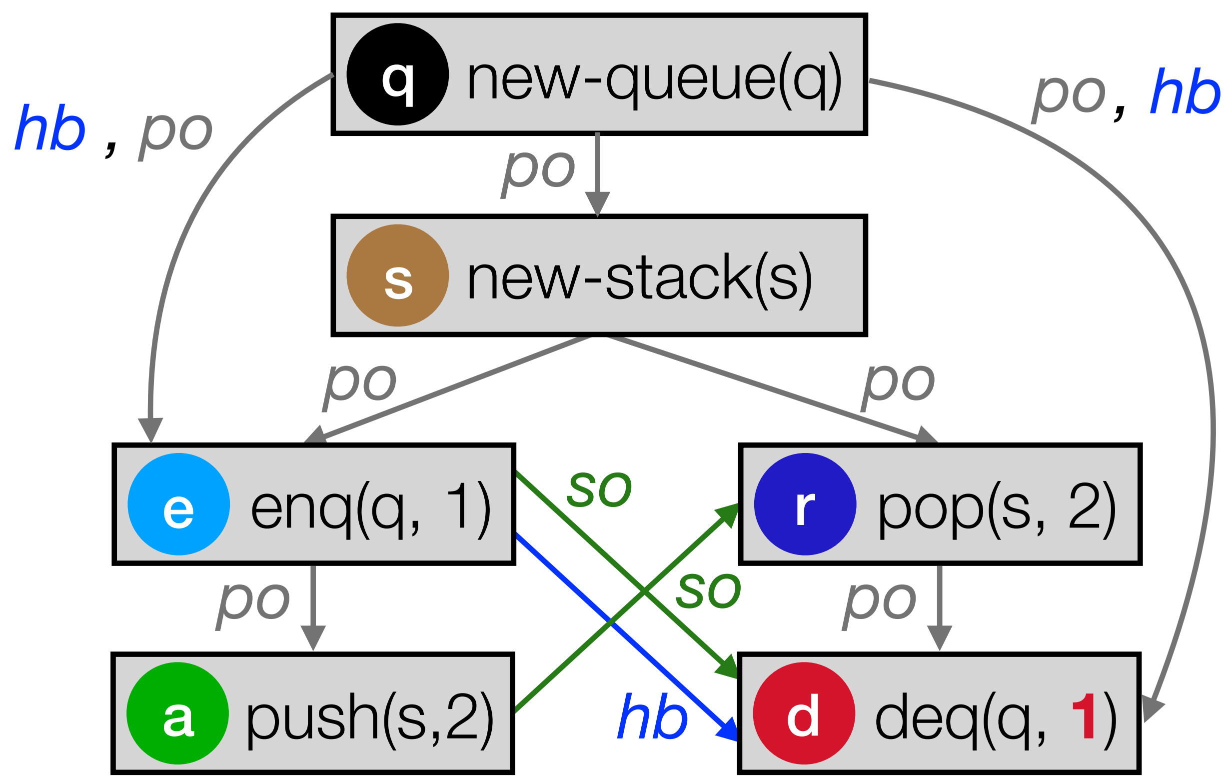
   e enq(q, v) $\xrightarrow{\cancel{so}}$ f deq(q, ⊥) *for all* v

3. ∃ *to*. *to* ***totally*** orders *E*,

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

Example Revisited

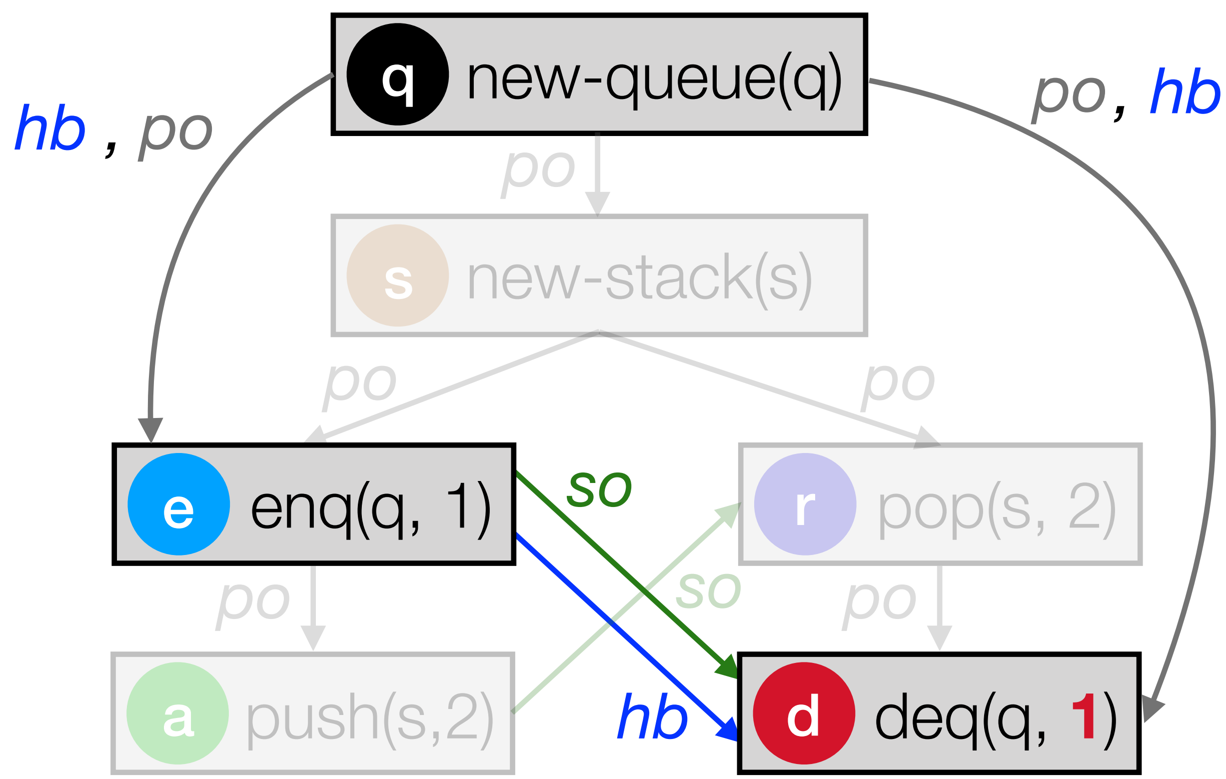
```
      q:=new-queue();
      s:=new-stack();
enq(q,1);  ||  a:=pop(s);
push(s,2)  ||  if(a==2)
                  b:=deq(q)   // should return 1
```



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

Example Revisited

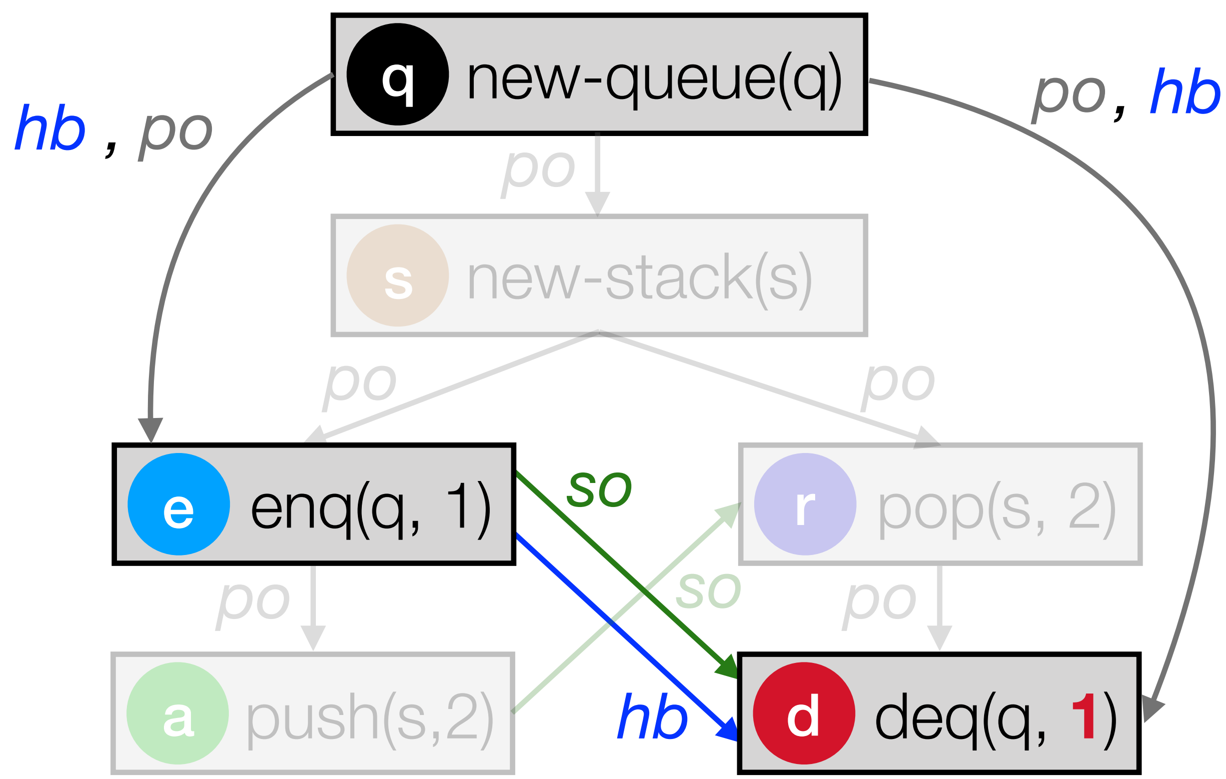
```
    q:=new-queue();
    s:=new-stack();

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

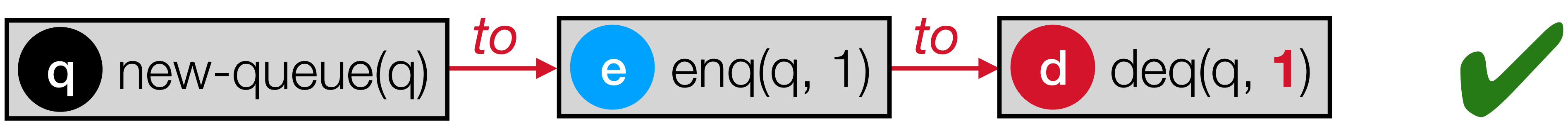


$hb = (po \cup so)^{+}$

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
```



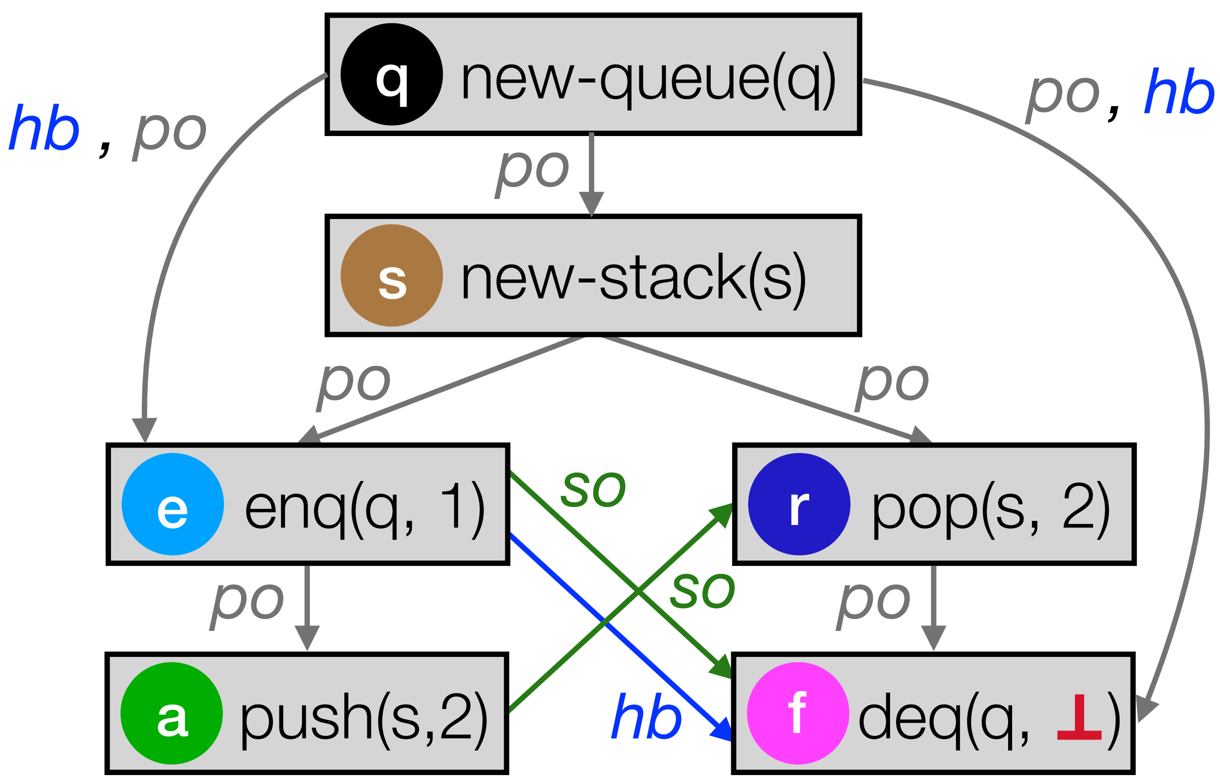


$hb = (po \cup so)^{+}$

$hb \subseteq to$

Example Revisited

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

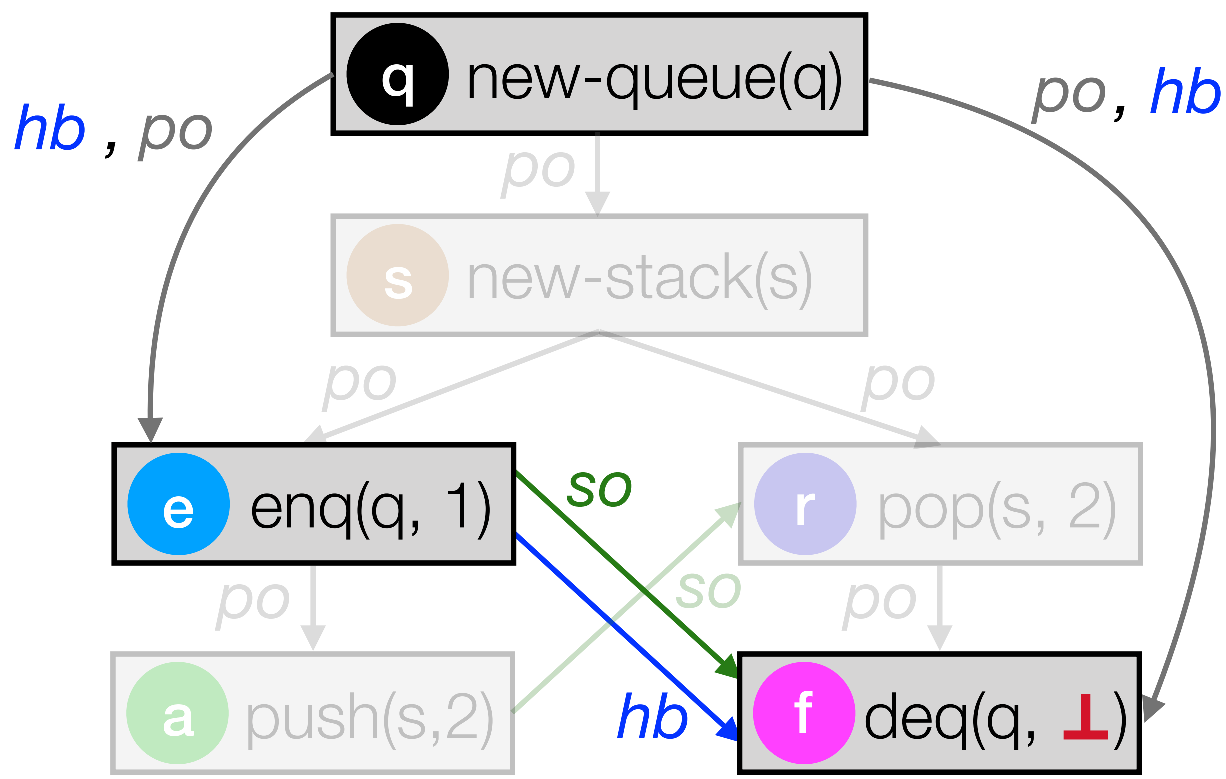


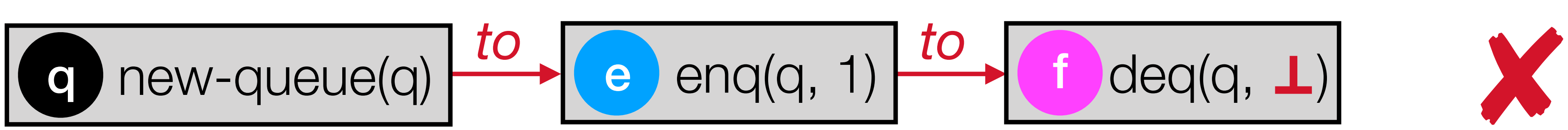
$hb = (po \cup so)^{+}$

$hb \subseteq to$

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
```





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

$hb \subseteq to$

Program Executions

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

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

Program Executions

$$[\![P]\!] = \left\{ G_P = \langle E, po, so \rangle \mid \boxed{\ldots} \right\}$$

?

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

Program Executions

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

?

queue & *stack* calls

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

Program Executions

$$[\![P]\!] = \left\{ G_P = \langle E, po, so \rangle \;\middle|\; \begin{array}{ll} G_{queue} & \textit{sats. queue axioms} \\ G_{stack} & \textit{sats. stack axioms} \end{array} \right\}$$

queue & *stack* calls

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

Program Executions

$$[\![P]\!] = \left\{ G_P = \langle E, po, so \rangle \;\middle|\; \begin{array}{l} G_{queue} \text{ sats. queue axioms} \\ G_{stack} \text{ sats. stack axioms} \end{array} \right\}$$

queue & *stack* calls

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

P calls $L_1 \ldots L_n$ and $G \in [\![P]\!]$

$\Longleftrightarrow$

G_{L_1} sats. L_1 axioms

...

G_{L_n} sats. L_n axioms

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

✗ difficult to find *to* witness

Strong Queue Axioms

C11 Herlihy-Wing Queue Implementation

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

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

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

Strong Queue Axioms

C11 Herlihy-Wing Queue Implementation

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

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

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

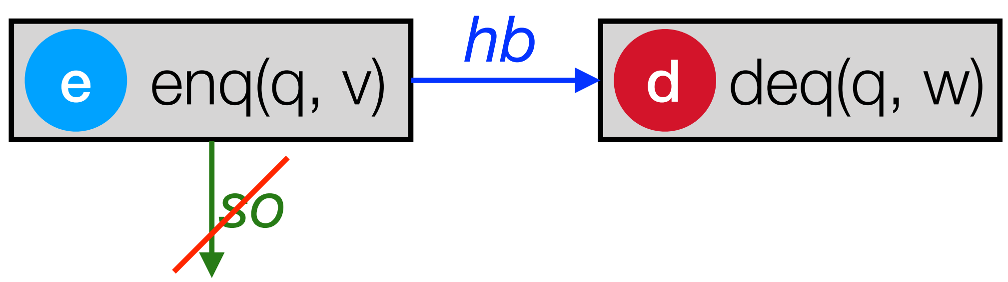
✔ satisfies ***strong*** queue axioms

Weak Queue Axioms

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

1. *E* contains queue events
2. *so* is ***1-to-1***; *so* relates ***matching*** `enq`/`deq` events
3. `deq` with *hb*-***earlier unmatched*** `enq` cannot return ⊥

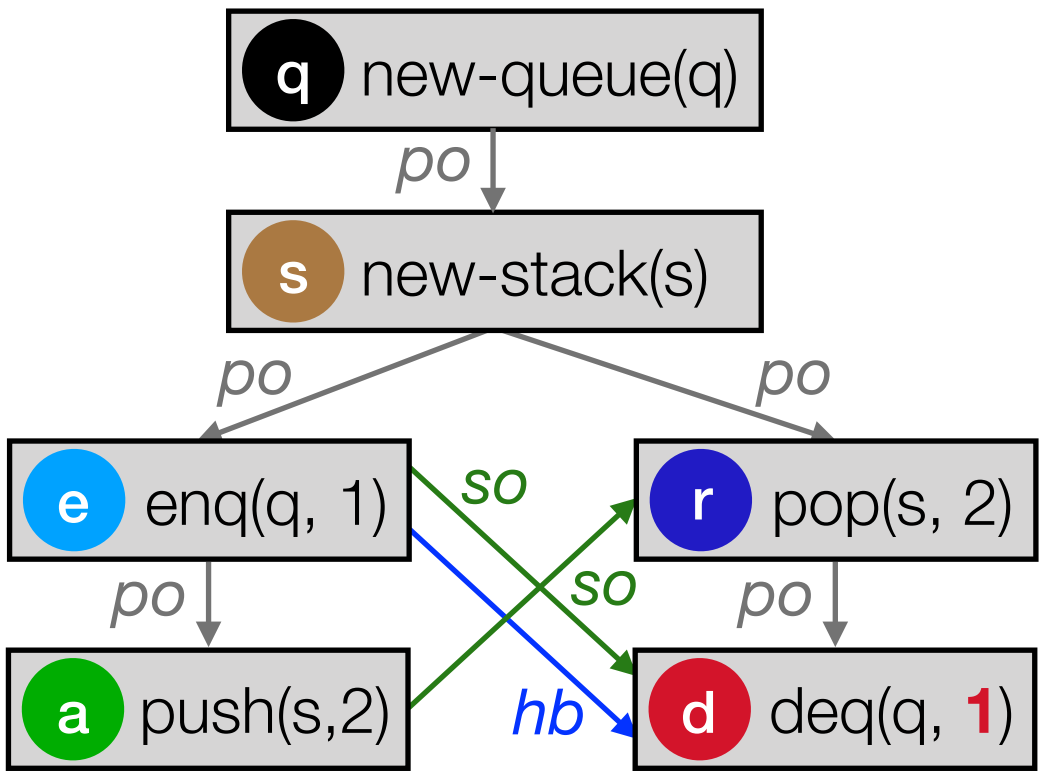
if

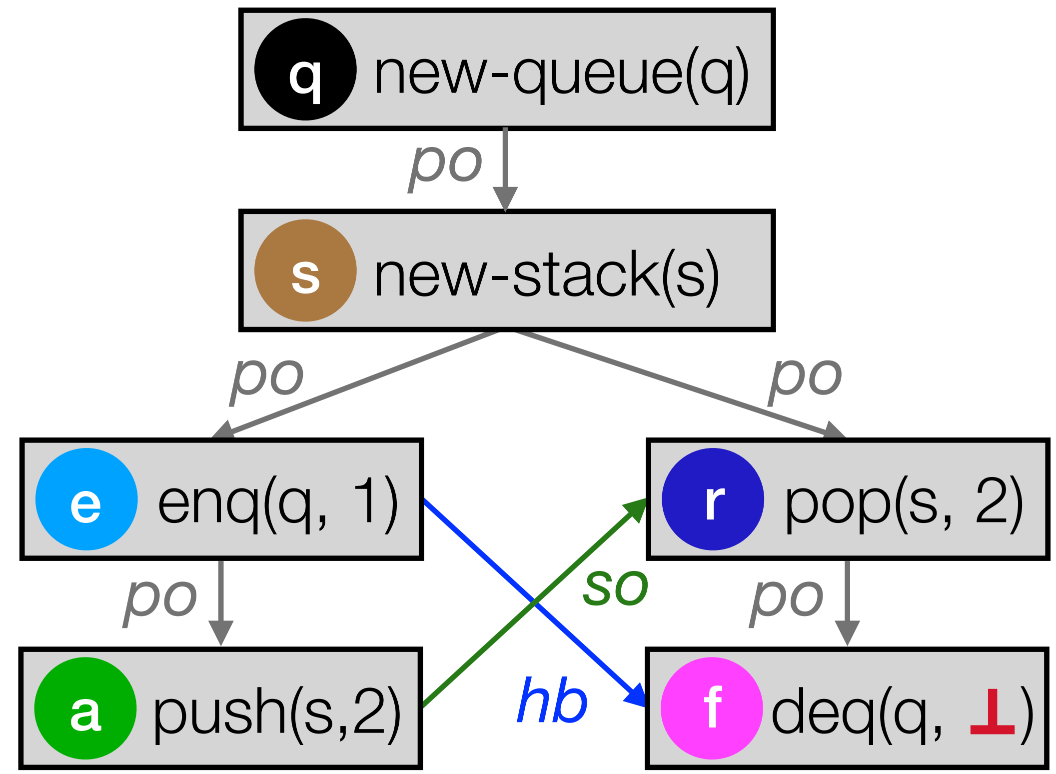


then w ≠ ⊥

Example Revisited

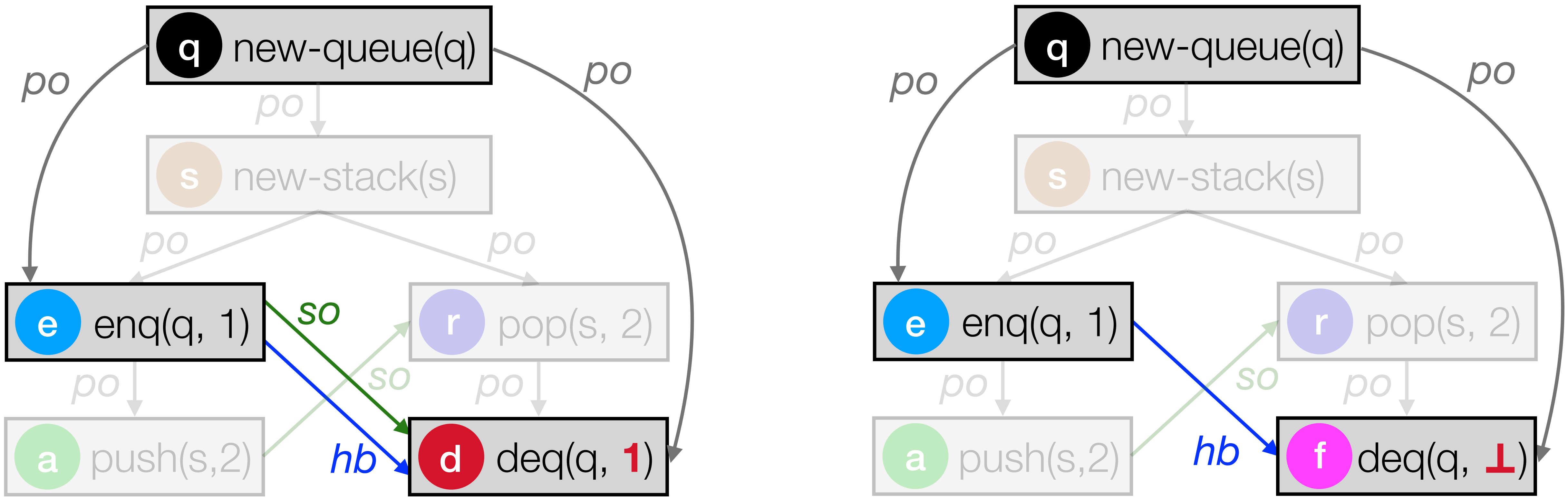
```
          q:=new-queue();
          s:=new-stack();
                          ||  a:=pop(s);
enq(q,1);                 ||  if(a==2)
push(s,2)                 ||     b:=deq(q)  // should return 1
```





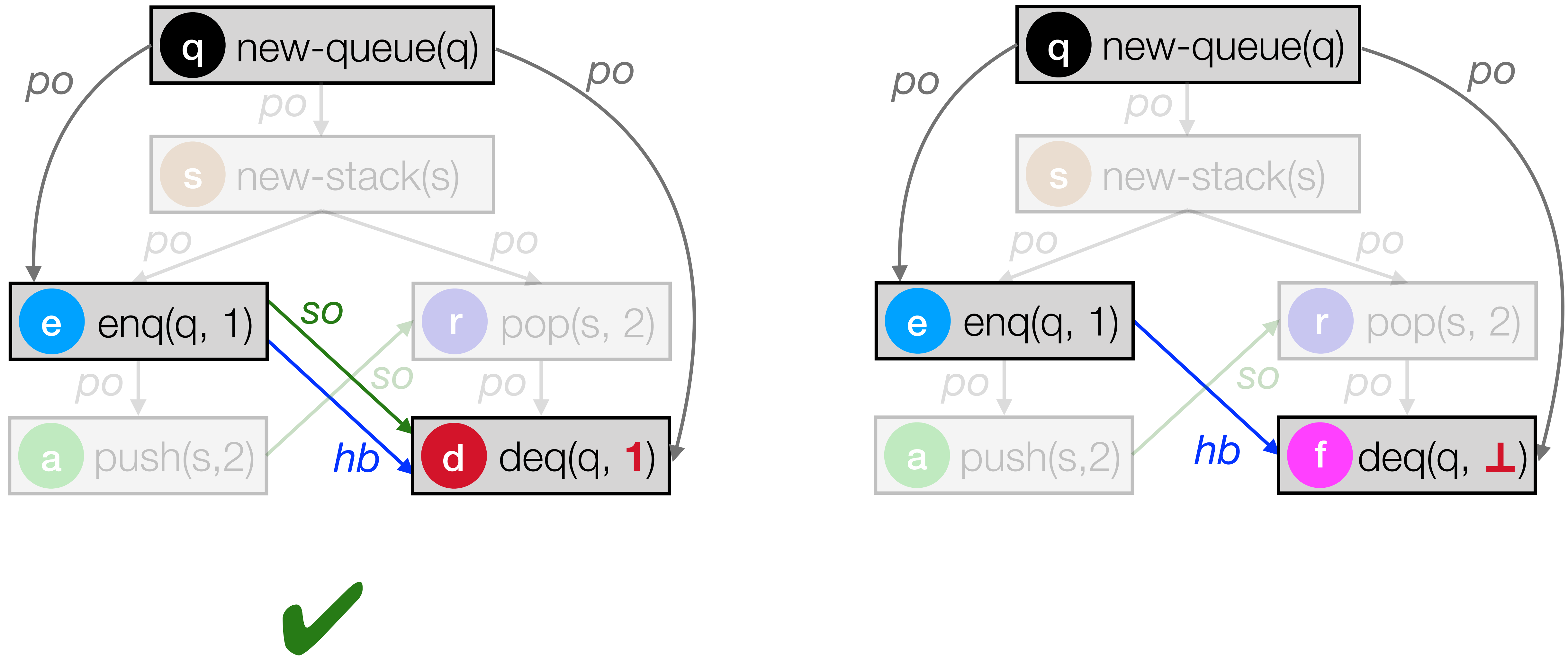
Example Revisited

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



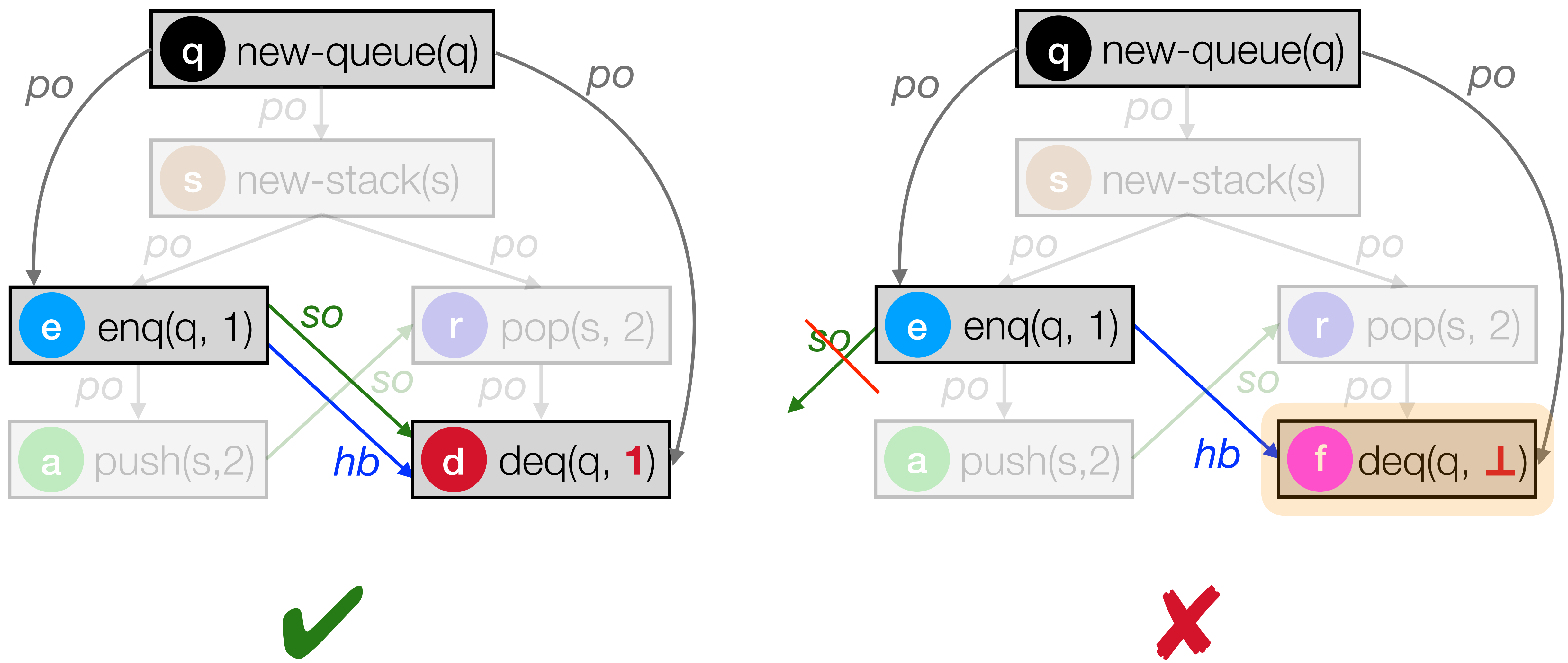
Example Revisited

```
     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
```

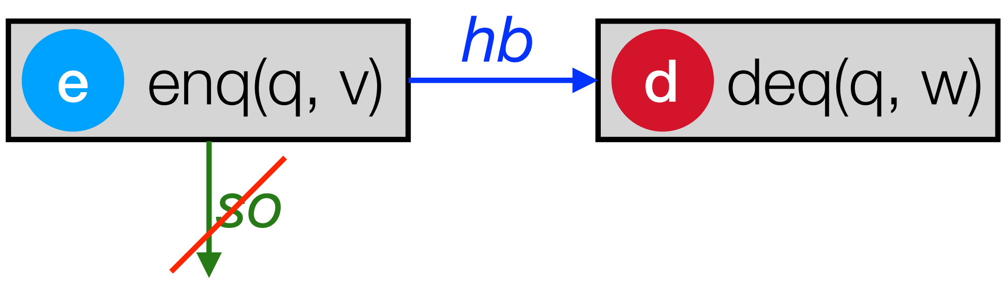


Weak Queue Axioms

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

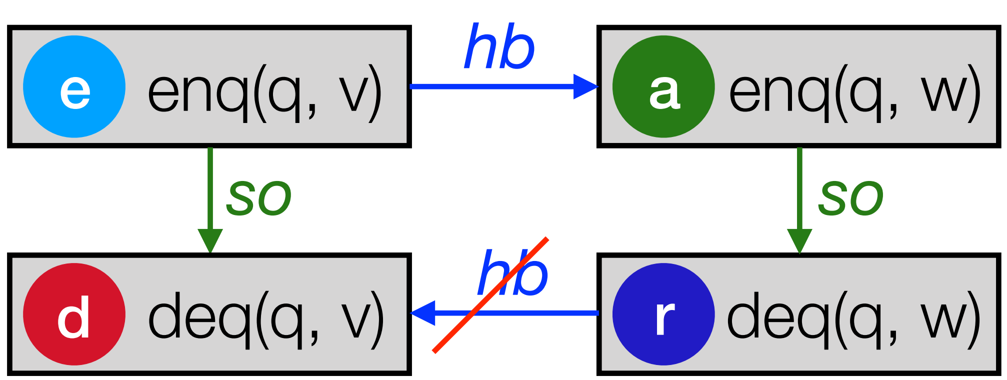
1. *E* contains queue events
2. *so* is ***1-to-1***; *so* relates ***matching*** `enq`/`deq` events
3. `deq` with *hb*-***earlier unmatched*** `enq` cannot return ⊥

if



then w ≠ ⊥

4. ***weak*** FIFO guarantee



Weak vs. ***Strong*** Queue Axioms

- ***Not*** equivalent

  $\Longrightarrow$ C11 ***Herlihy-Wing*** queue

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

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

✔ satisfies ***weak*** axioms

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

Weak vs. ***Strong*** Queue Axioms

- ***Not*** equivalent

  $\Longrightarrow$ C11 ***Herlihy-Wing*** queue

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

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

✔ satisfies ***weak*** axioms

✔ satisfies ***strong*** axioms

Weak vs. ***Strong*** Queue Axioms

- ***Not*** equivalent

  $\Longrightarrow$ C11 ***Herlihy-Wing*** queue

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

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

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

- Why weak axioms?

  $\Longrightarrow$ ***strong enough*** for certain uses: ***single-producer-single-consumer***

Alternative Strong 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

strong ***weak***

3. $\exists$ *to*. *to* ***totally*** orders E,
   $hb \subseteq to$ *to* is FIFO

3. `deq` with *hb*-***earlier unmatched*** `enq` cannot return $\perp$

4. ***weak*** FIFO guarantee
   (***weak acyclicity*** axiom)

Alternative Strong 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

strong / ***weak***

strong:

3. $\exists$ to. to ***totally*** orders E, $hb \subseteq to$ to is FIFO

✘ too strong ☛ weak axioms
✘ difficult to find to witness

weak:

3. `deq` with hb-***earlier unmatched*** `enq` cannot return $\perp$

4. ***weak*** FIFO guarantee (***weak acyclicity*** axiom)

Alternative Strong 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

strong ***weak***

3. $\exists$ to. to ***totally*** orders E, $hb \subseteq to$ to is FIFO

✘ too strong ☚ weak axioms
✘ difficult to find to witness

3. ***strong*** FIFO guarantee (***strong acyclicity*** axiom)

⇒ see our paper!

3. `deq` with hb-***earlier unmatched*** `enq` cannot return $\perp$
4. ***weak*** FIFO guarantee (***weak acyclicity*** axiom)

Recap

A declarative 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

Recap

A declarative framework:

✓ ***Agnostic*** to memory model

- ➡ support *both SC* and *WMC* specs

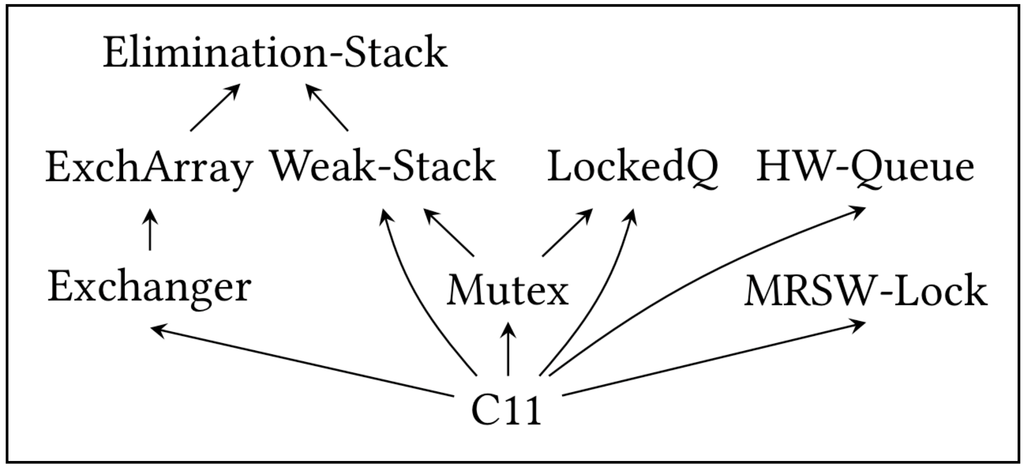
✓ ***General***

- ➡ 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***

- ? verify *client programs*
- ? verify *library implementations*

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



Part II.

Concurrent Library ***Verification*** under WMC

Program ***Outcomes***

$$[\![P]\!] = \{\, G_P \,|\, \ldots \}$$

$$outcomes(P) = \left\{ val(G_P) \;\middle|\; G_P \in [\![P]\!] \right\}$$

values of ***local*** variables

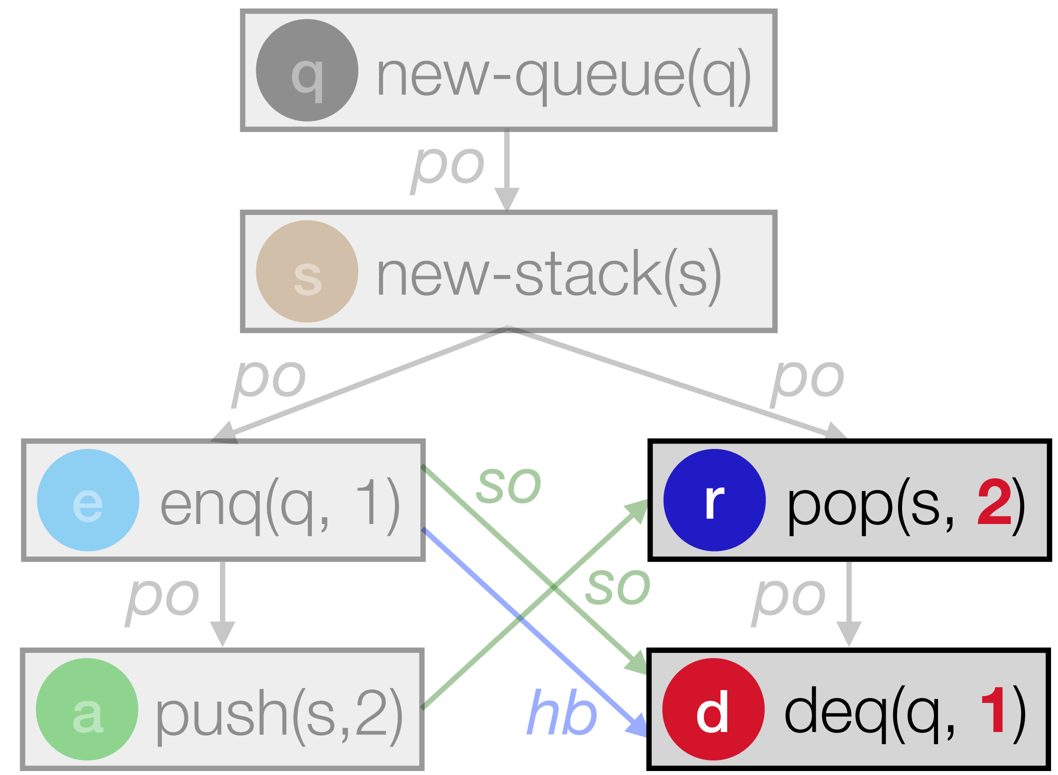
Program *Outcomes*

$[\![P]\!] = \{\, G_P \mid \ldots \}$

$outcomes(P) = \{\, val(G_P) \mid G_P \in [\![P]\!] \,\}$

values of ***local*** variables

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



a = 2 b = 1

Client Verification

$[\![P]\!] = \{\, G_P \mid \ldots \}$

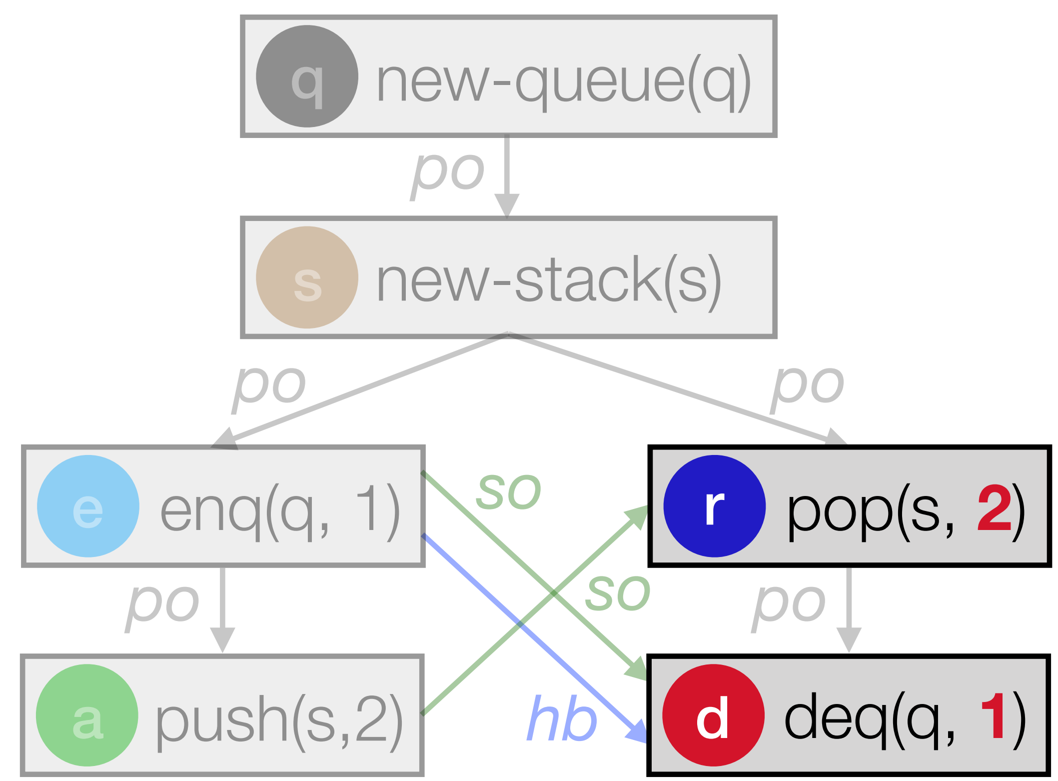
$outcomes(P) = \{\, val(G_P) \mid G_P \in [\![P]\!] \,\}$

values of ***local*** variables

assert (*true*)

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

assert $(\neg\,(a = 2 \ \wedge\ b = \perp))$



a = 2 b = 1

Implementation Verification?

$$
\begin{array}{l}
\mathit{new\text{-}queue}(\,) \triangleq \\
\quad \mathtt{let}\ q = \mathit{alloc}(+\infty)\ \mathtt{in}\ q
\end{array}
$$

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

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

Library *Implementation*

I_{queue} : *Method*$_{queue}$ $\rightarrow$ *Prog*

C11 Herlihy-Wing Implementation (HWQ)

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

I_{queue}(`new-queue()`)

$enq(q, v) \triangleq$

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

I_{queue}(`enq(q,v)`)

$deq(q) \triangleq$

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

I_{queue}(`deq(q)`)

Translation

$$\lfloor \,\cdot\, \rfloor_{I_{queue}} : \mathit{Prog} \to \mathit{Prog}$$

I_{queue}(new-queue())

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

I_{queue}(enq(q,1))

~~enq(q,1)~~;
push(s,2)

||

a:=pop(s);
if(a==2)
  b:=~~deq(q)~~

I_{queue}

I_{queue}(deq(q))

Implementation ***Soundness***

I_{queue} ***sound*** $\iff$ for all P :

$$\mathit{outcomes}(\lfloor P \rfloor_{I_{queue}}) \subseteq \mathit{outcomes}(P)$$

Implementation ***Soundness***

$\mathcal{I}_{queue}$ ***sound*** $\iff$ for all P :

$$outcomes(\llcorner P \lrcorner_{\mathcal{I}_{queue}}) \subseteq outcomes(P)$$

$$\Downarrow$$

$$\{ val(G_i) \mid G_i \in [\![\llcorner P \lrcorner_{\mathcal{I}_{queue}}]\!] \} \subseteq \{ val(G_s) \mid G_s \in [\![P]\!] \}$$

$$outcomes(P) = \{ val(G_P) \mid G_P \in [\![P]\!] \}$$

Implementation ***Soundness***

I_{queue} ***sound*** $\iff$ for all P :

$$outcomes(\lfloor P \rfloor_{I_{queue}}) \subseteq outcomes(P)$$

$$\Downarrow$$

$$\{ val(G_i) \mid G_i \in [\![\lfloor P \rfloor_{I_{queue}}]\!] \} \subseteq \{ val(G_s) \mid G_s \in [\![P]\!] \}$$

$$\Downarrow$$

$$\forall G_i \in [\![\lfloor P \rfloor_{I_{queue}}]\!] \,.\ \exists G_s \in [\![P]\!] \,.\ val(G_i) = val(G_s)$$

$$outcomes(P) = \{ val(G_P) \mid G_P \in [\![P]\!] \}$$

Example: HWQ Soundness

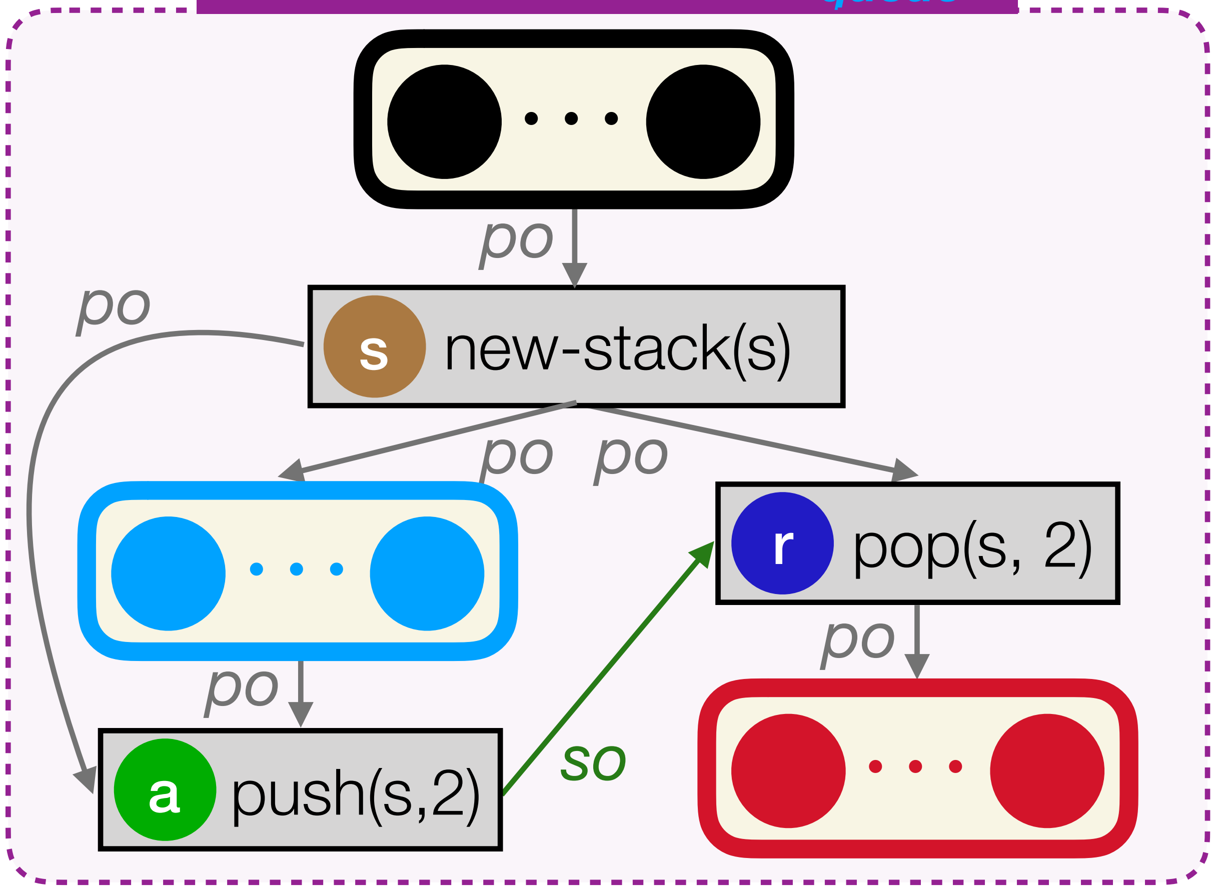
$\lfloor P \rfloor_{I_{queue}}$

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

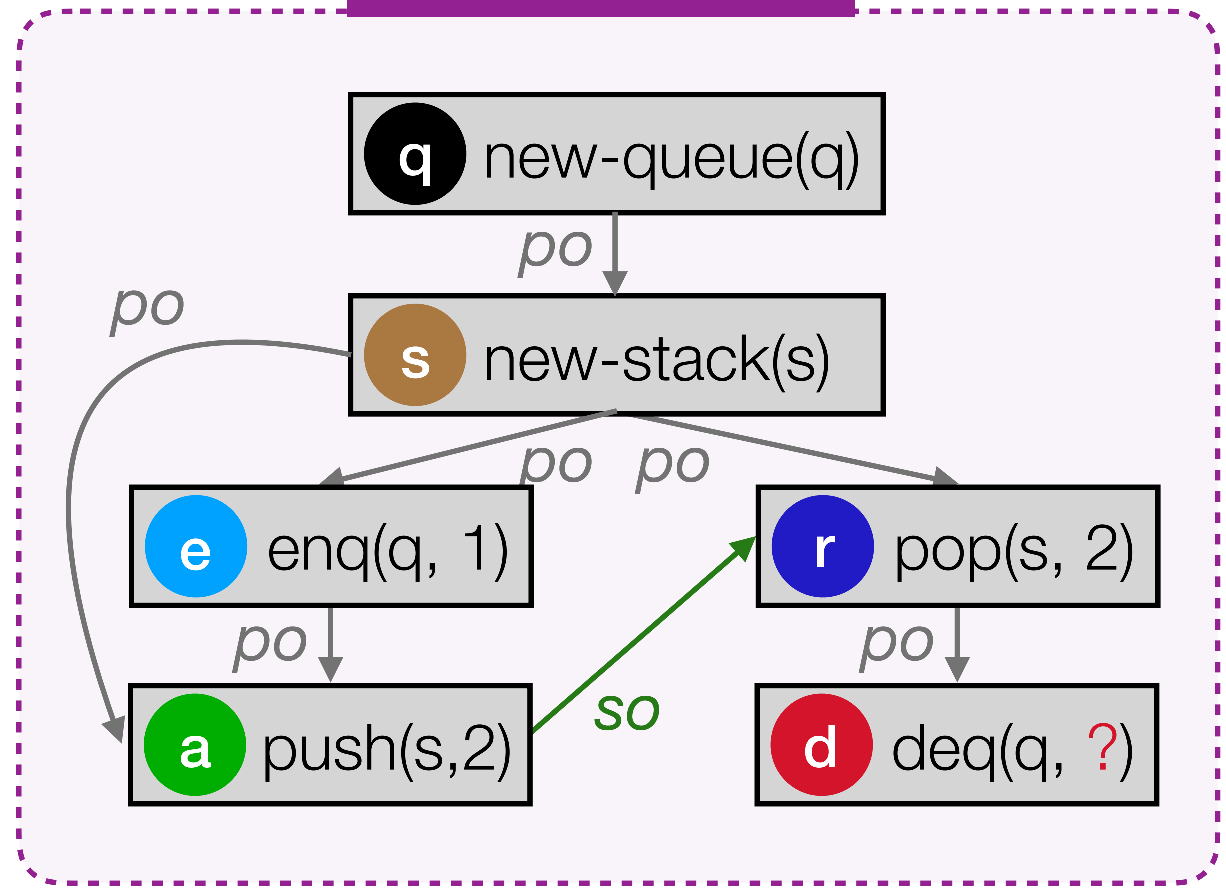
P

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

Given $G_i \in [\![\lfloor P \rfloor_{I_{queue}}]\!]$



Find $G_s \in [\![P]\!]$



such that: $val(G_i) = val(G_s)$

Example: HWQ Soundness

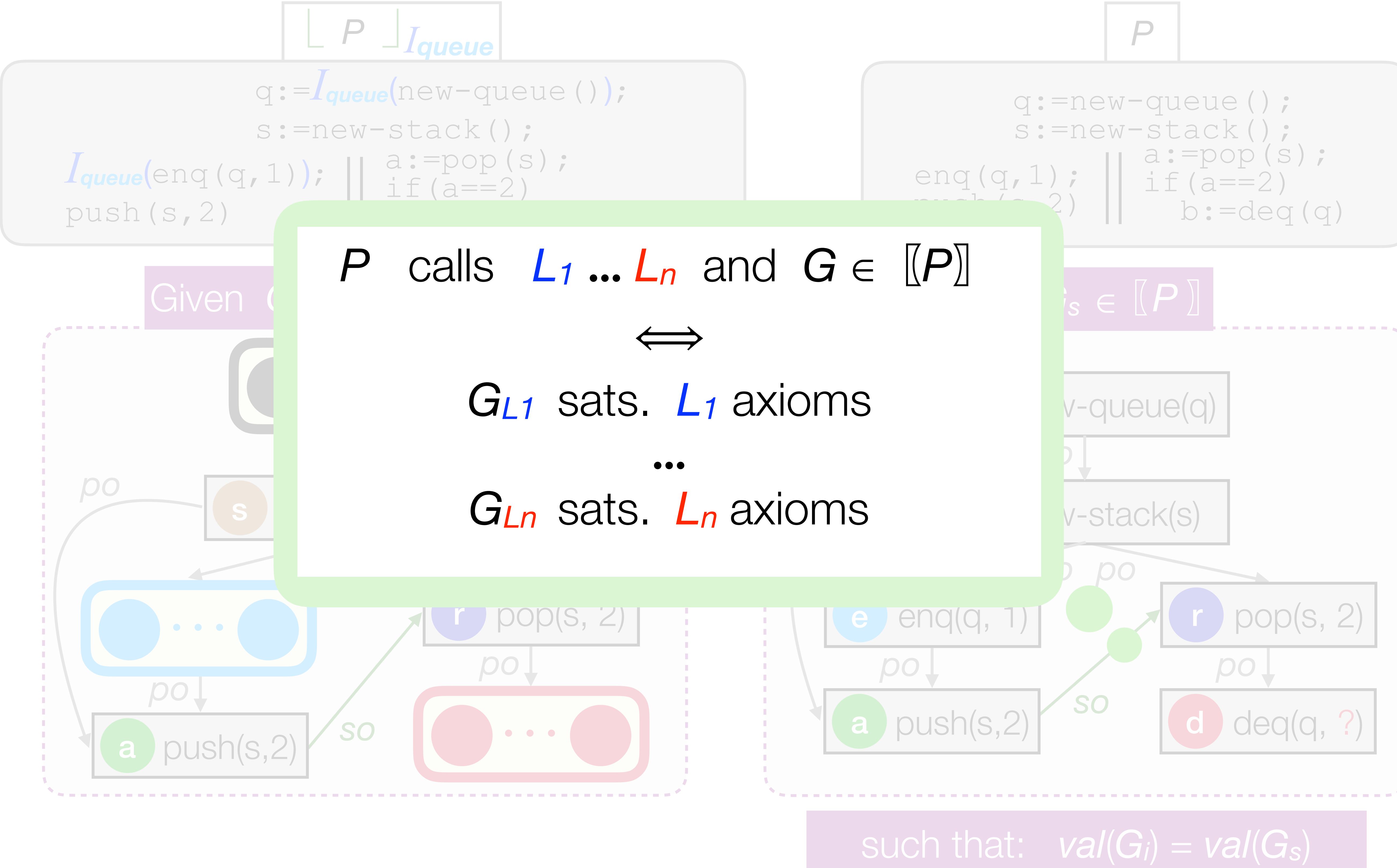
P calls $L_1 \dots L_n$ and $G \in [\![P]\!]$

$$\iff$$

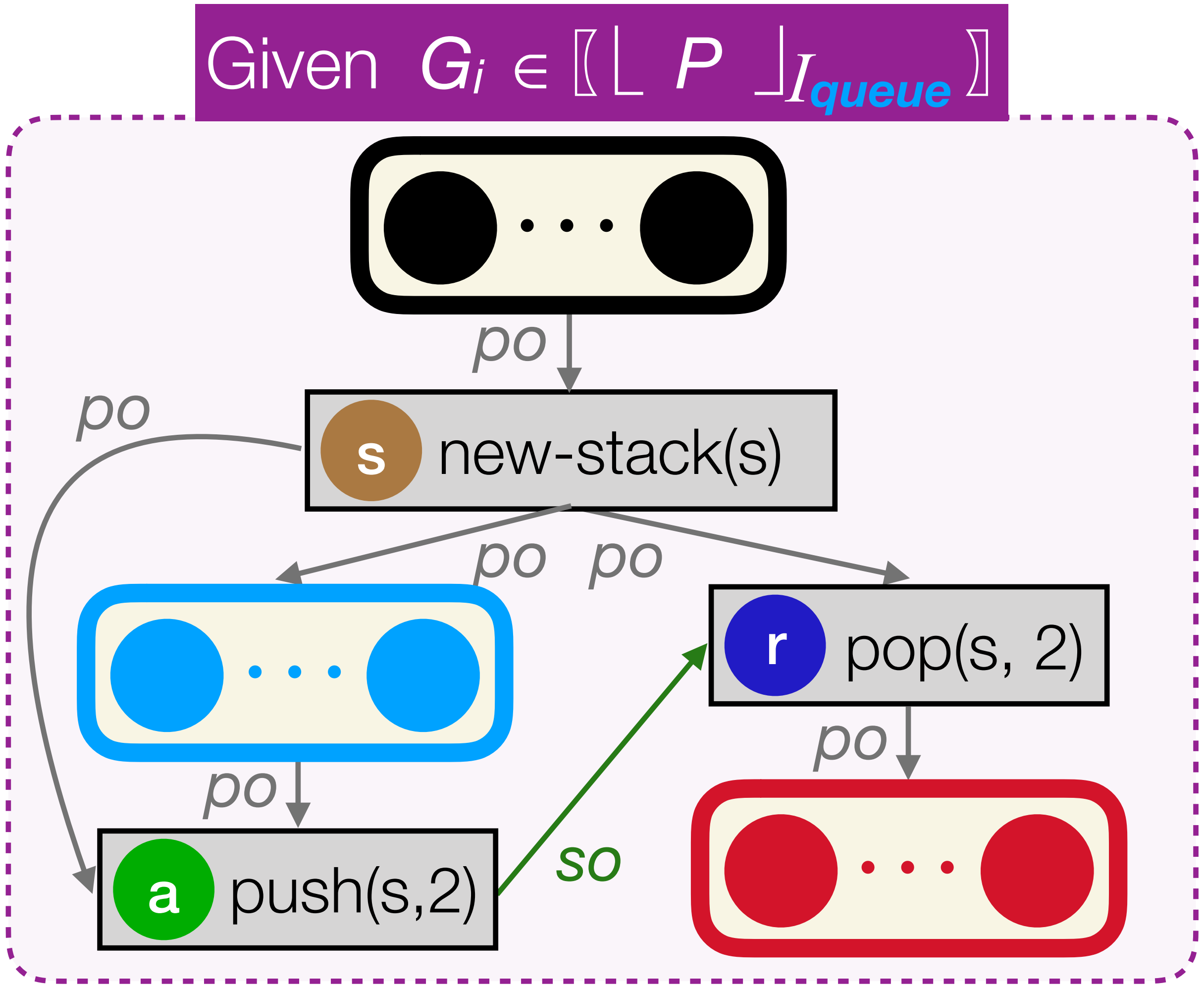
G_{L_1} sats. L_1 axioms

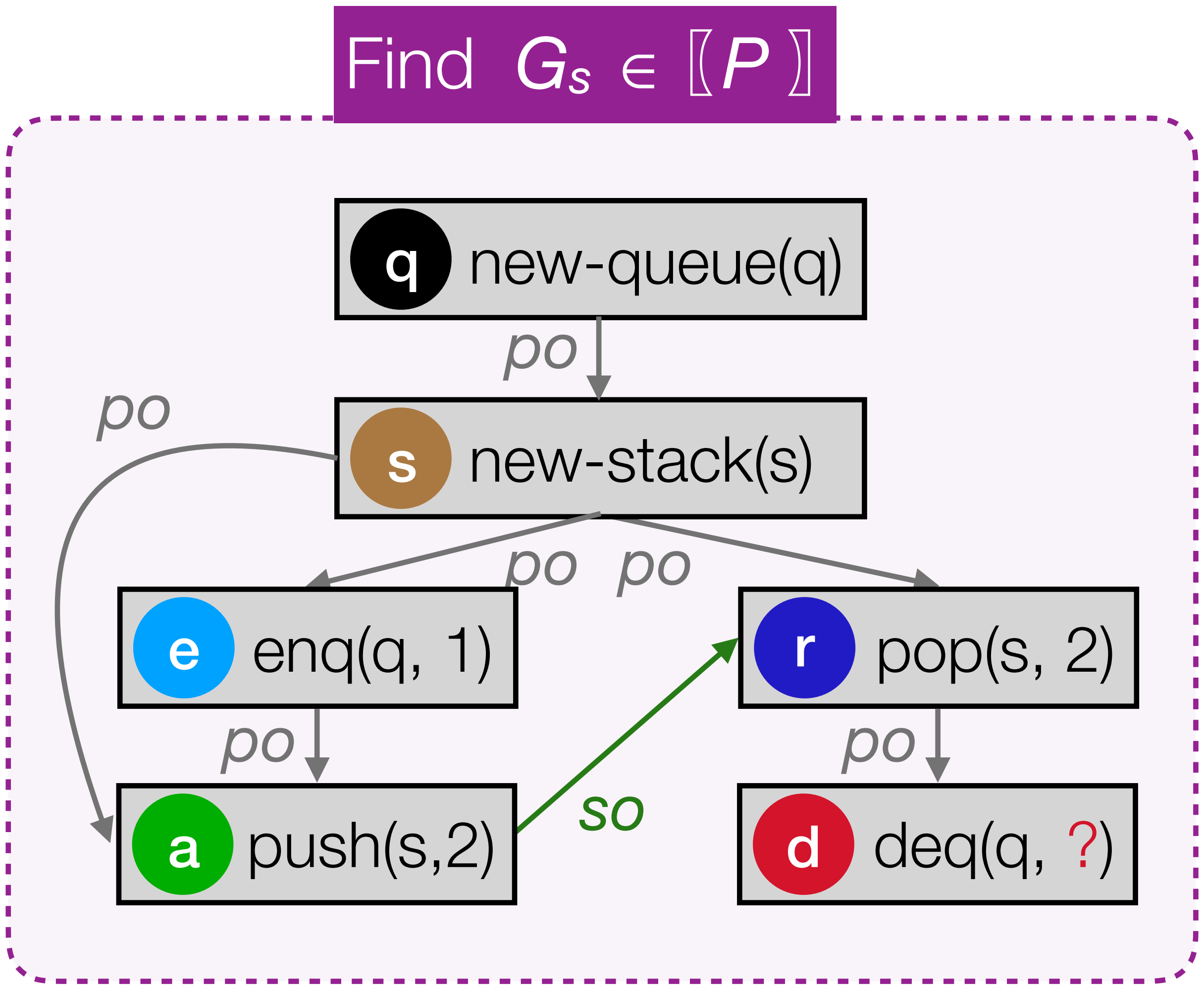
...

G_{L_n} sats. L_n axioms



Example: HWQ Soundness





We know:

$(G_i)_{C11}$ sats. *C11* axioms

$(G_i)_{stack}$ sats. *stack* axioms

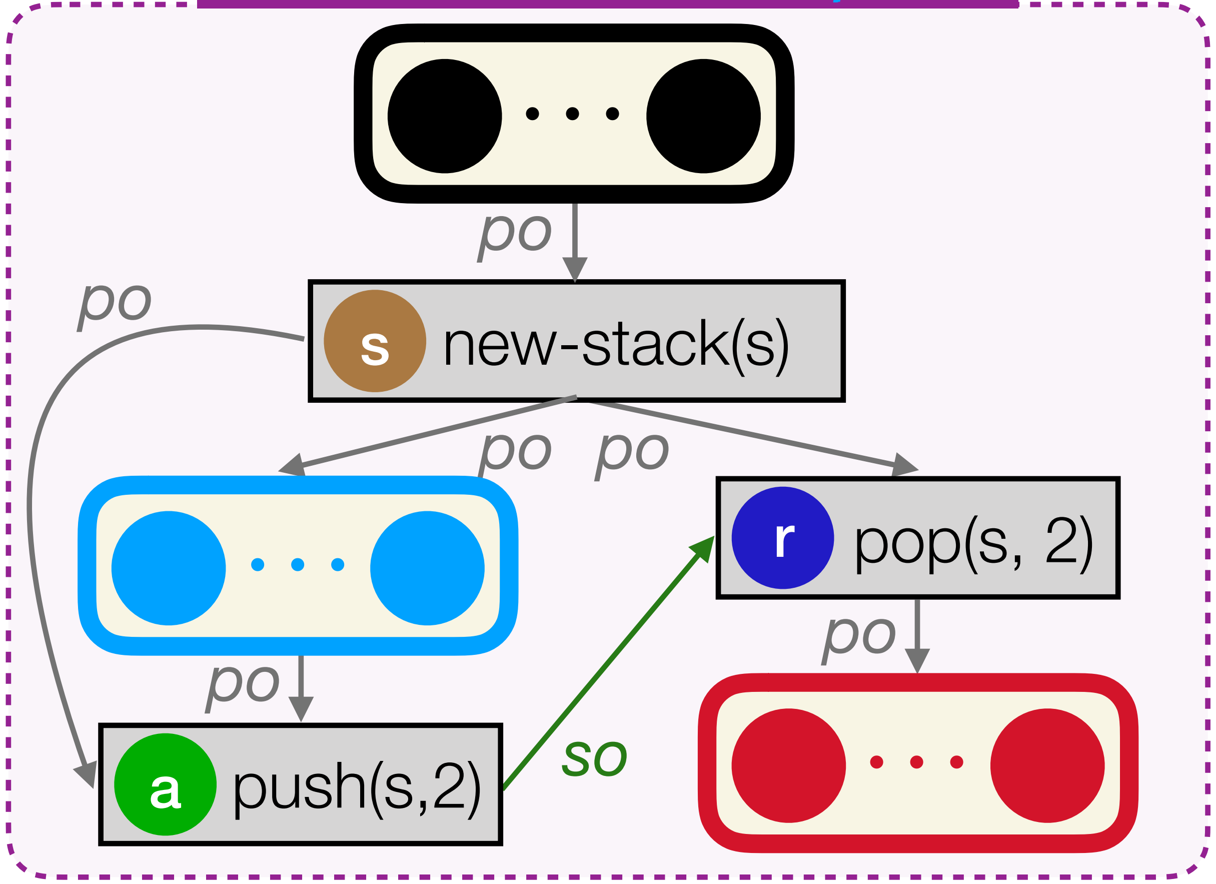
Show:

$(G_s)_{queue}$ sats. *queue* axioms

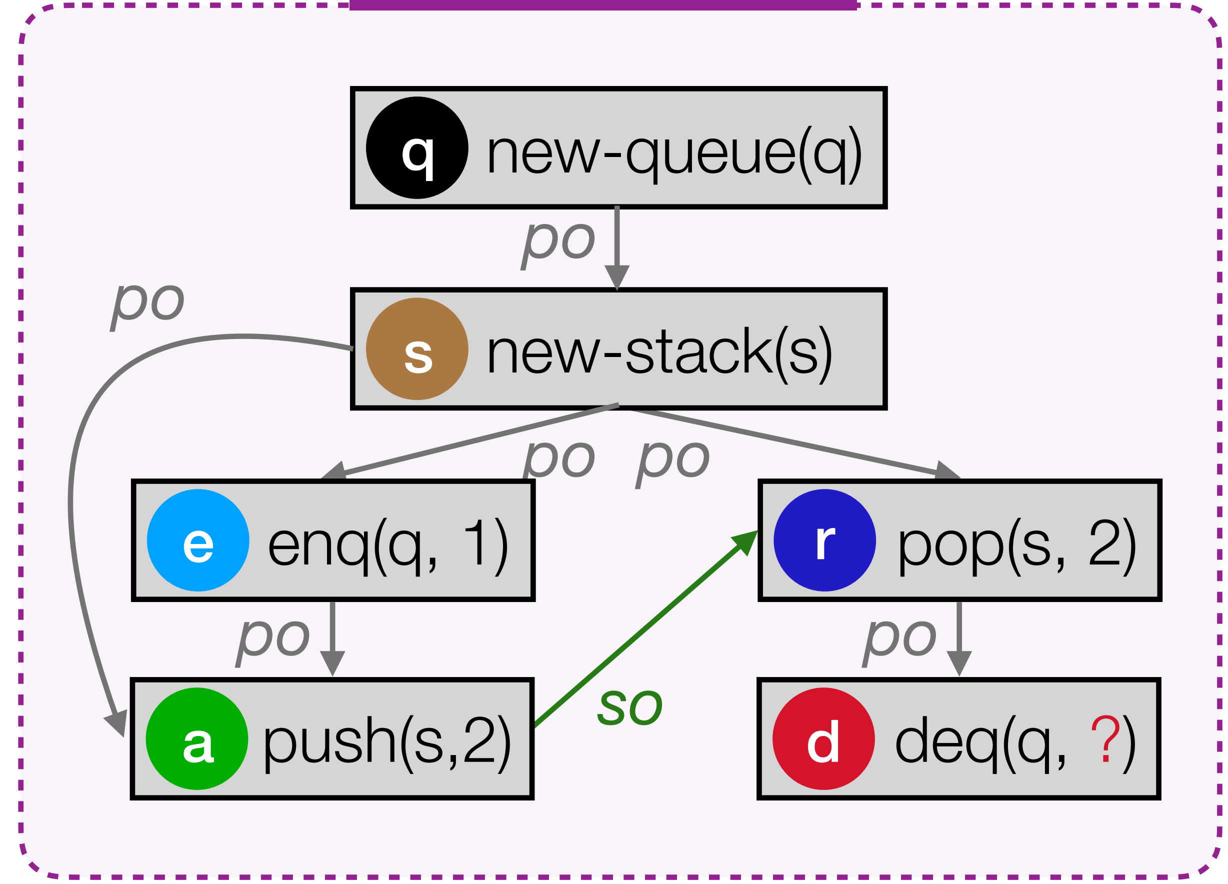
$(G_s)_{stack}$ sats. *stack* axioms

Example: HWQ Soundness

Given $G_i \in [\![\lfloor P \rfloor_{I_{queue}}]\!]$



Find $G_s \in [\![P]\!]$



We know:

$(G_i)_{C11}$ sats. *C11* axioms

$(G_i)_{stack}$ sats. *stack* axioms

Show:

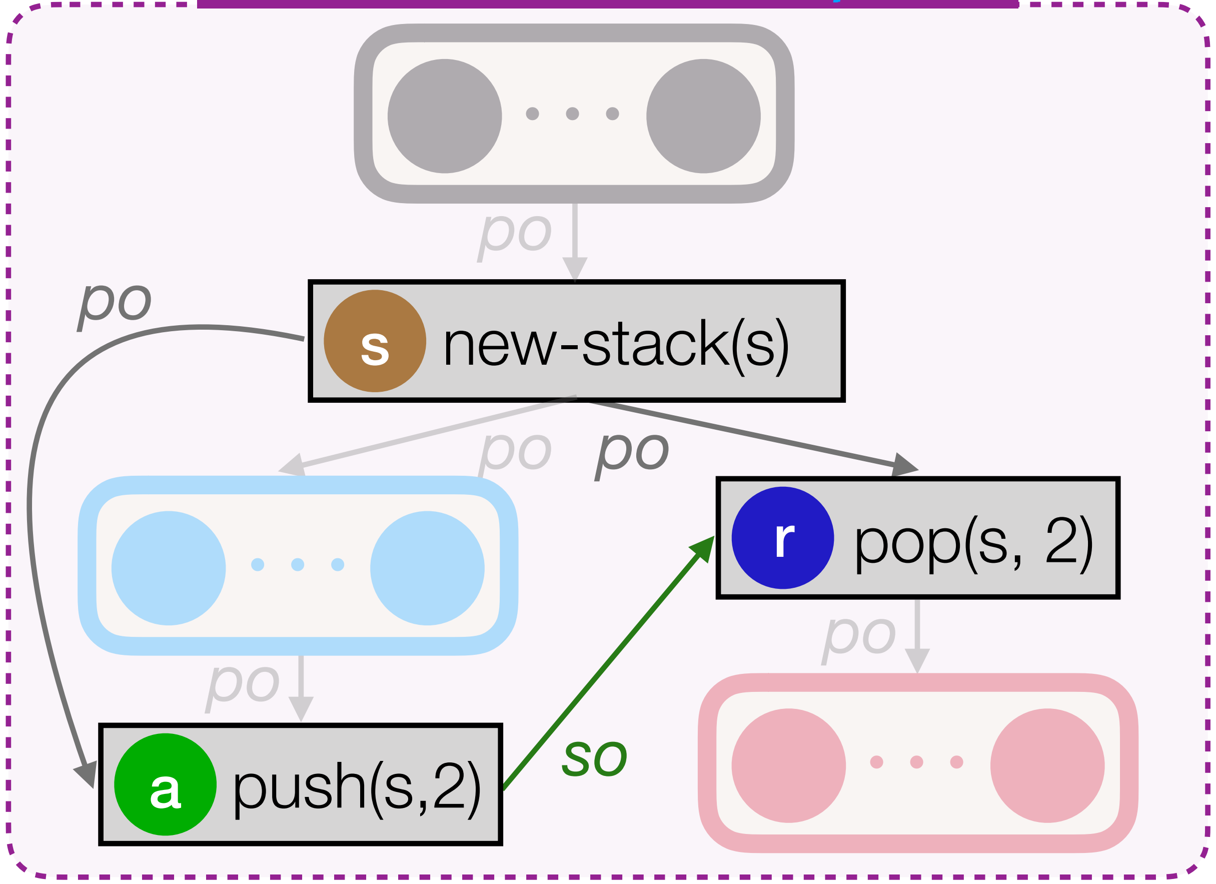
main proof obligation

$(G_s)_{queue}$ sats. *queue* axioms

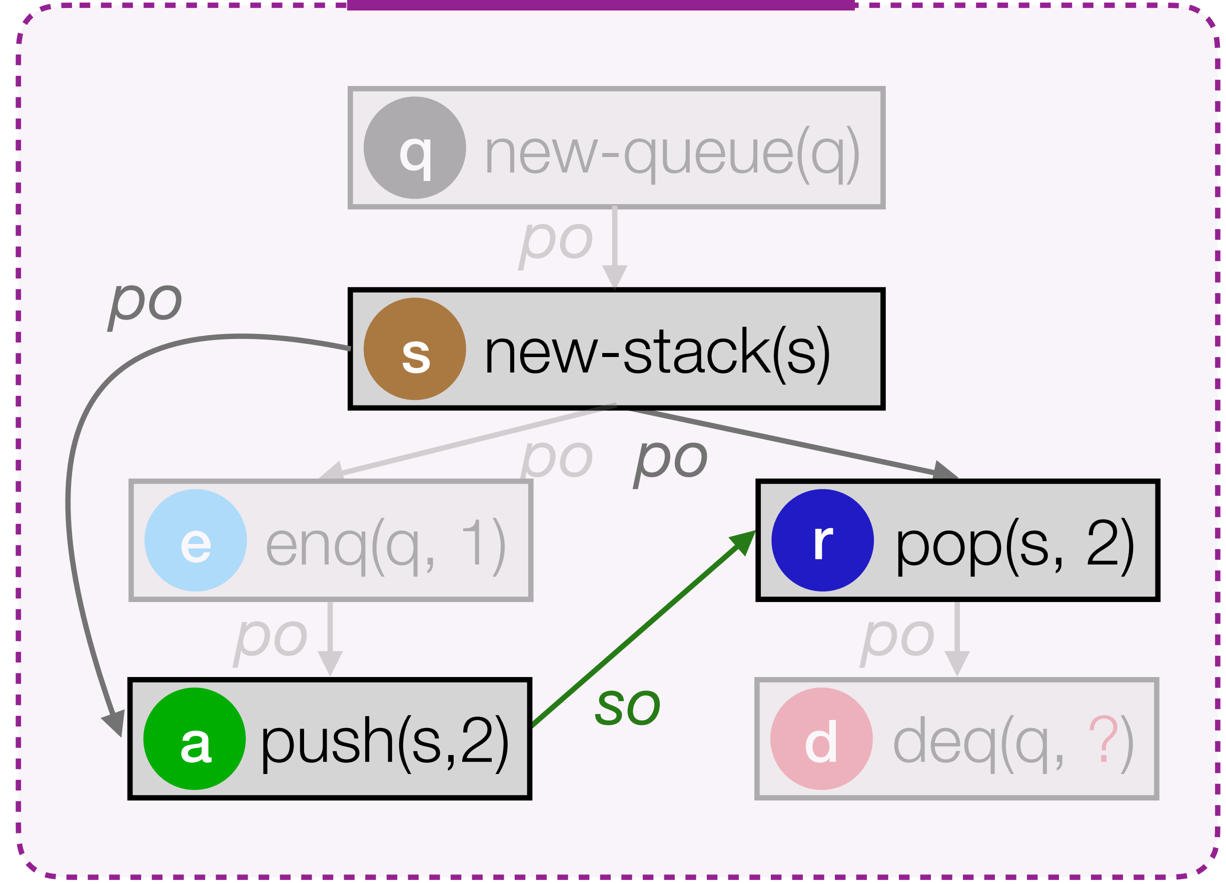
$(G_s)_{stack}$ sats. *stack* axioms

Example: HWQ Soundness

Given $G_i \in \llbracket \lfloor P \rfloor_{I_{queue}} \rrbracket$



Find $G_s \in \llbracket P \rrbracket$



We know:

$(G_i)_{C11}$ sats. *C11* axioms

$(G_i)_{stack}$ sats. *stack* axioms

Show:

main proof obligation

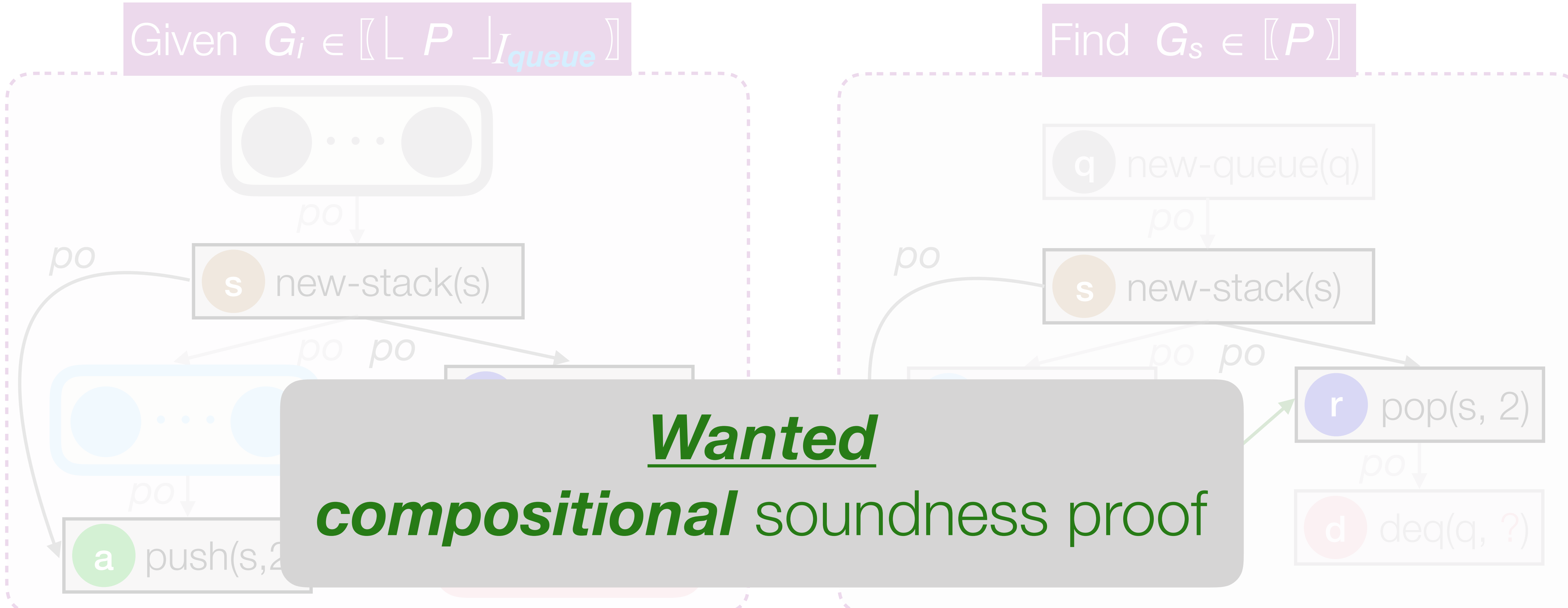
$(G_s)_{queue}$ sats. *queue* axioms

$(G_s)_{stack}$ sats. *stack* axioms

almost redundant :

$(G_i)_{stack}$ =?= $(G_s)_{stack}$

Example: HWQ Soundness



<u>Wanted</u>

compositional soundness proof

We know:

$(G_i)_{C11}$ sats. *C11* axioms

$(G_i)_{stack}$ sats. *stack* axioms

Show:

main proof obligation

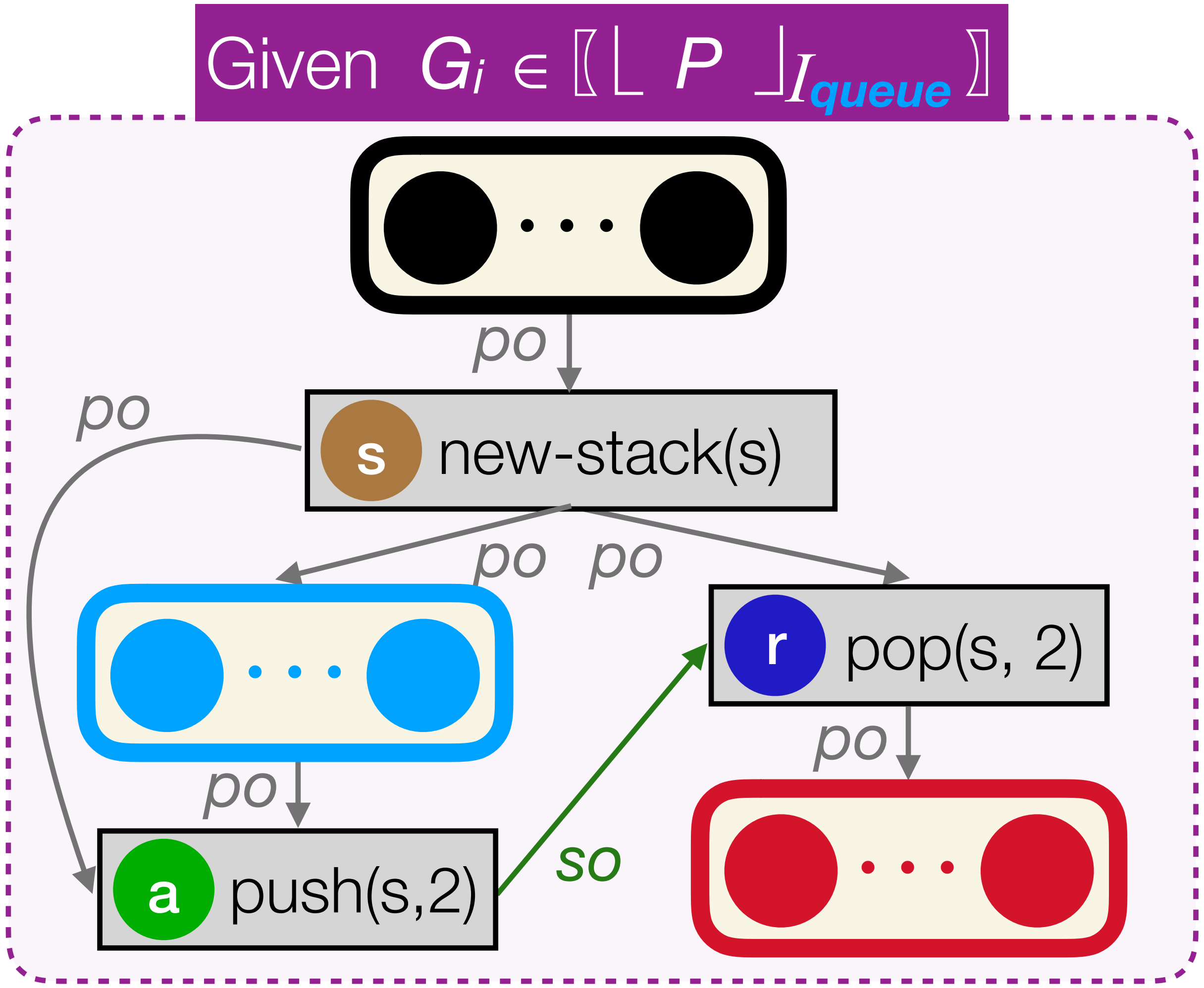
$(G_s)_{queue}$ sats. *queue* axioms

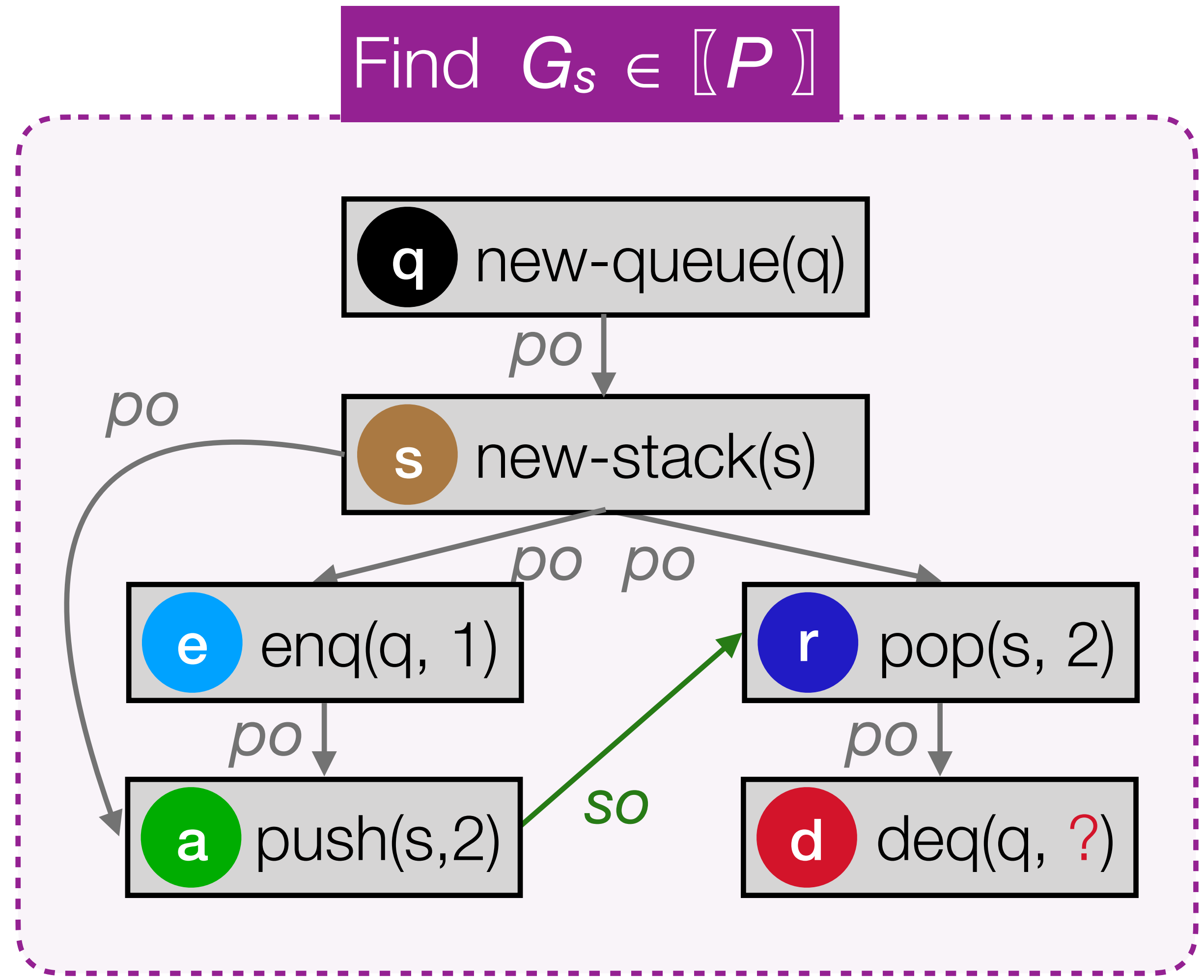
$(G_s)_{stack}$ sats. *stack* axioms

almost redundant :

$(G_i)_{stack}$ =?= $(G_s)_{stack}$

Example: HWQ Soundness

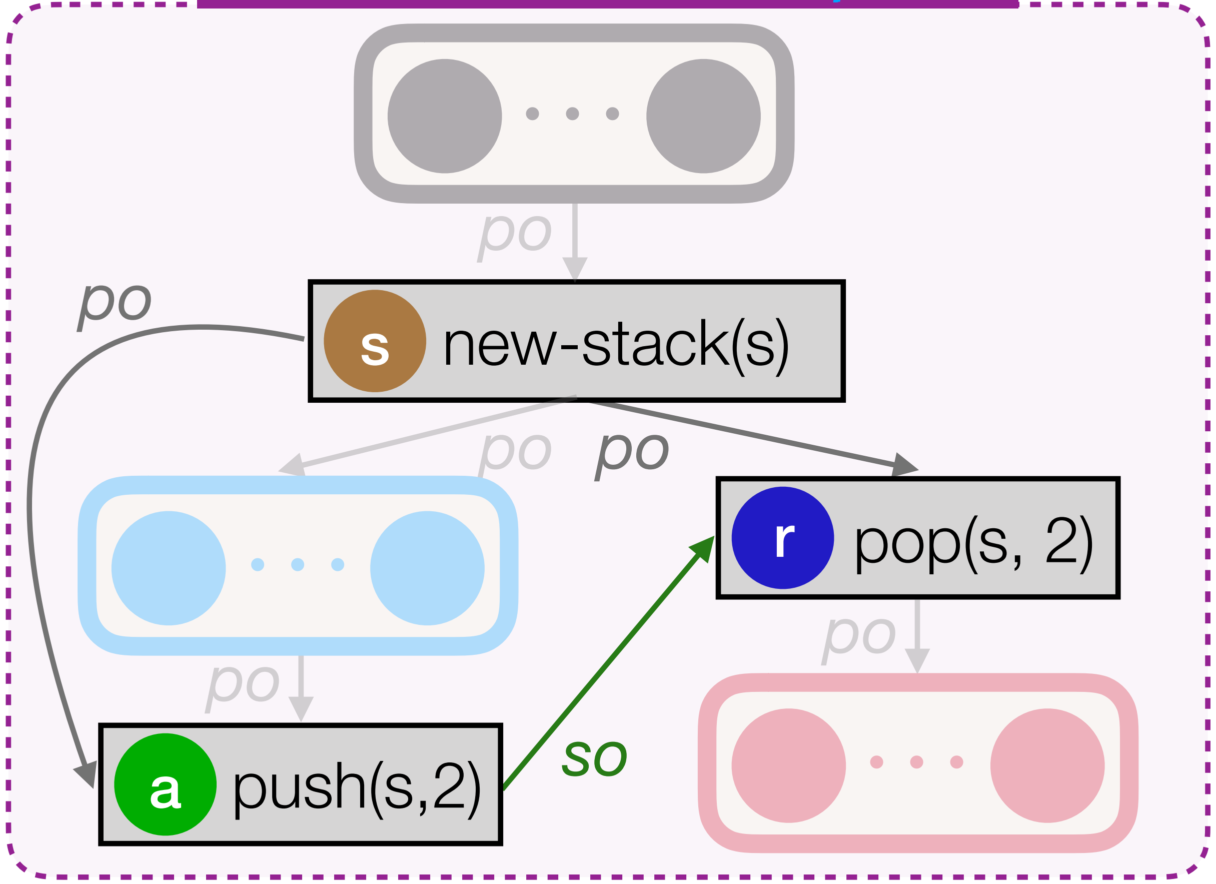




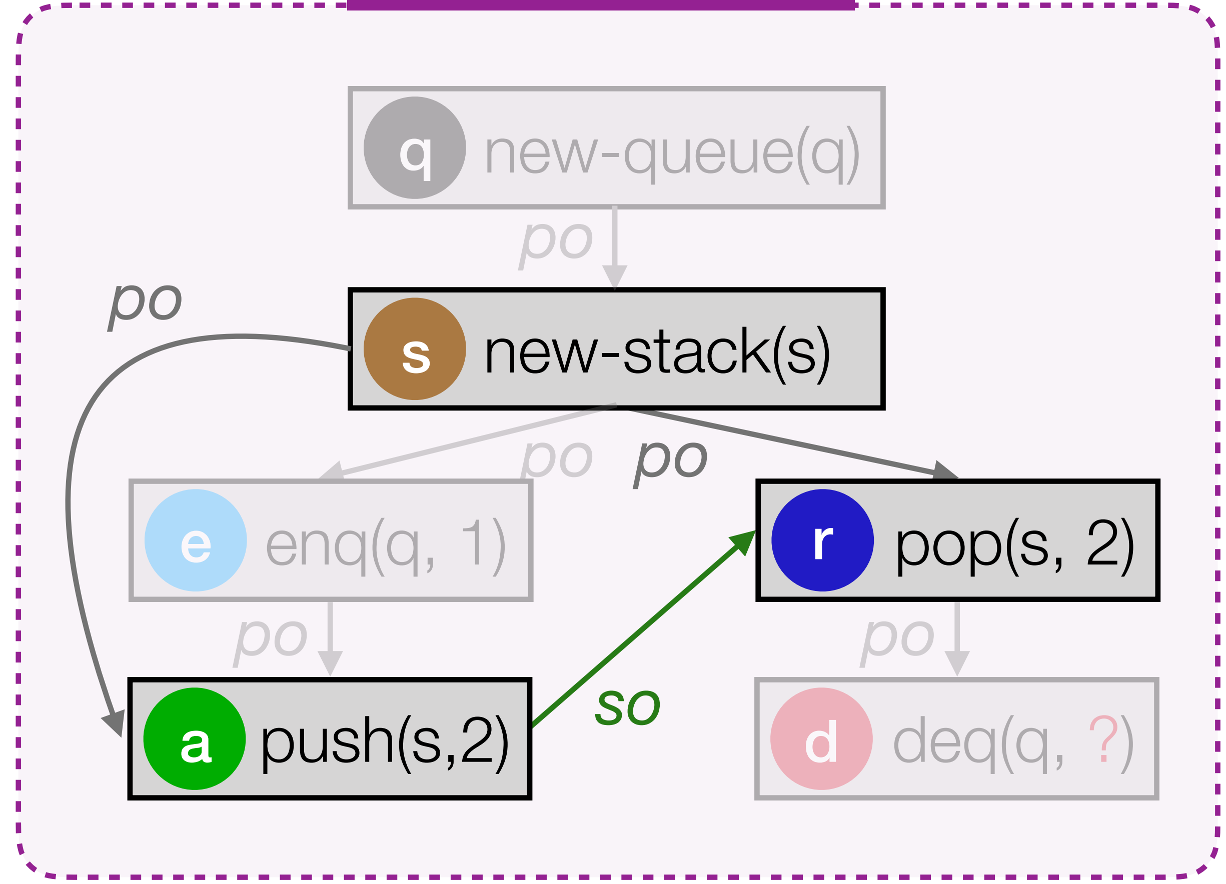
$(G_i)_{stack} = \langle E_{stack}, po_{stack}, so_{stack}, (hb_i)_{stack} \rangle$ $(G_s)_{stack} = \langle E_{stack}, po_{stack}, so_{stack}, (hb_s)_{stack} \rangle$

Example: HWQ Soundness

Given $G_i \in [\![\lfloor P \rfloor_{I_{queue}}]\!]$

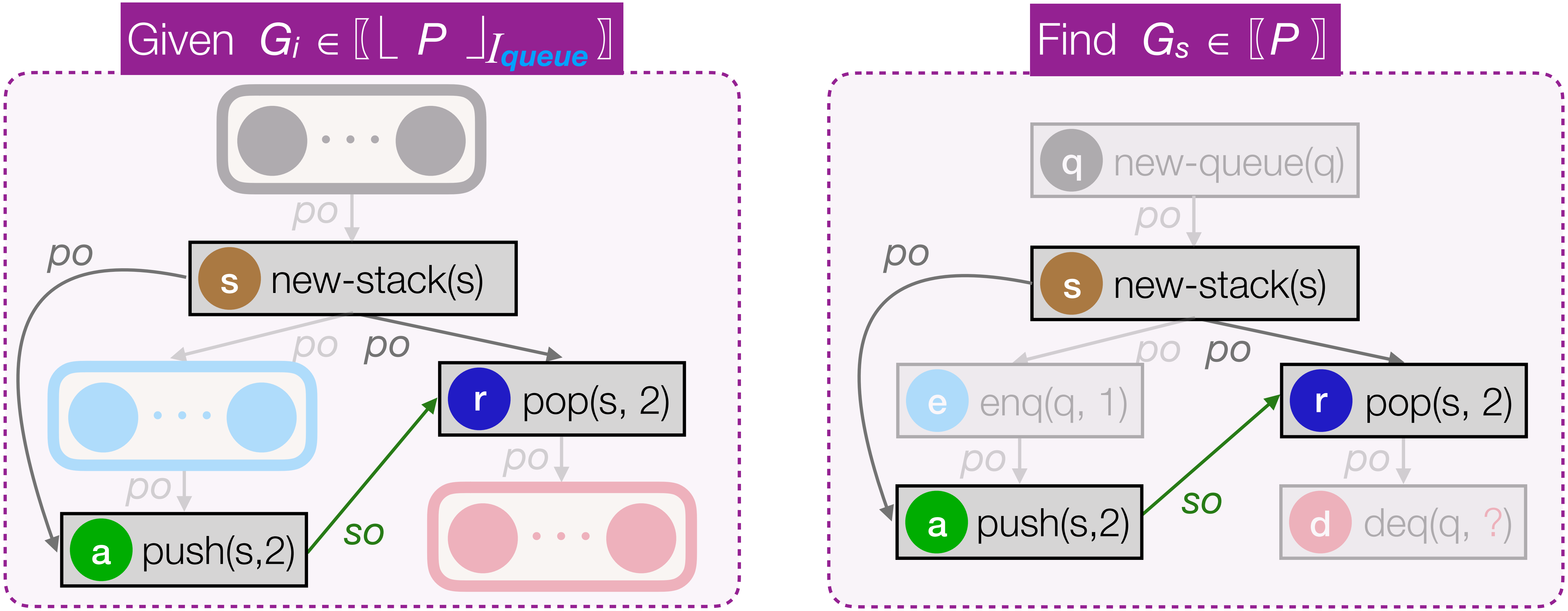


Find $G_s \in [\![P]\!]$



$(G_i)_{stack} = \langle E_{stack}, po_{stack}, so_{stack}, (hb_i)_{stack} \rangle$ $\quad$ $(G_s)_{stack} = \langle E_{stack}, po_{stack}, so_{stack}, (hb_s)_{stack} \rangle$

Example: HWQ Soundness



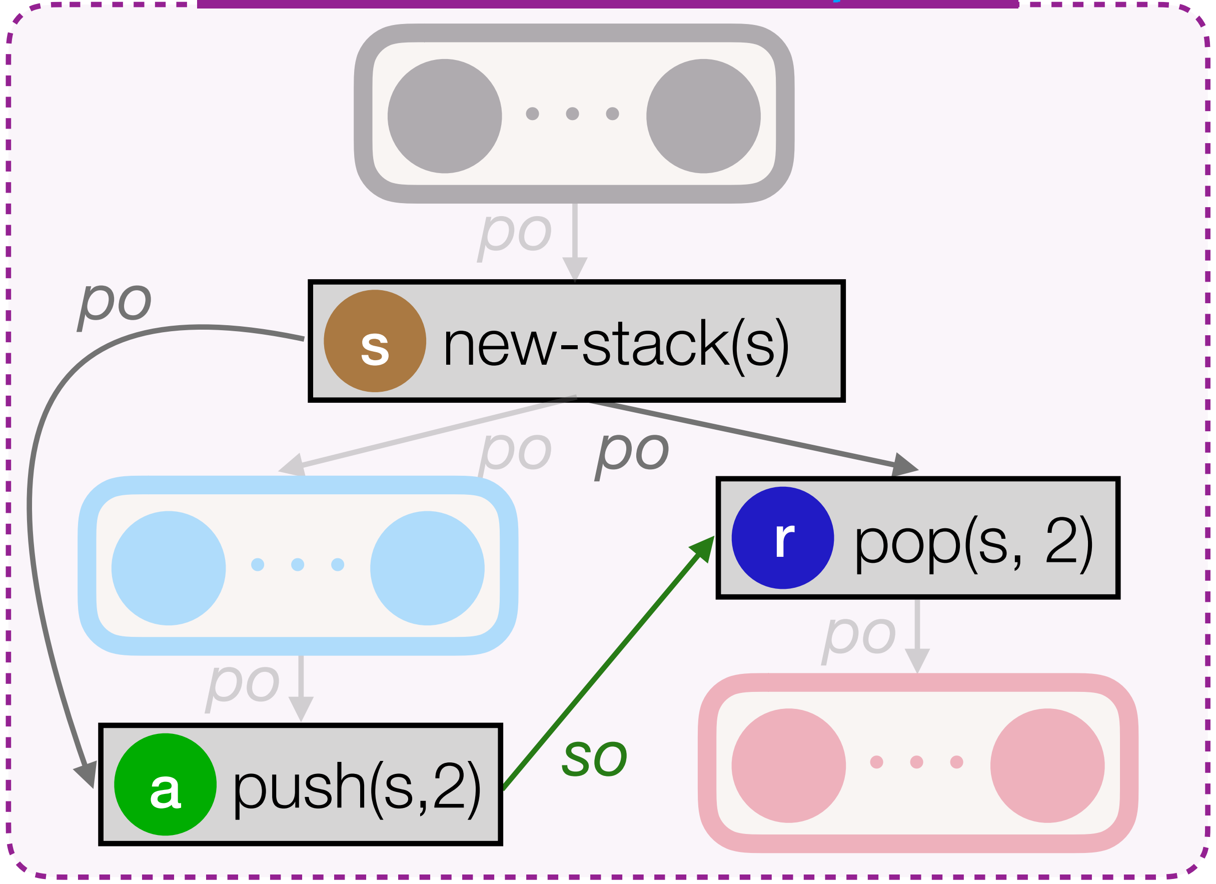
$(G_i)_{stack} = \langle E_{stack}, po_{stack}, so_{stack}, (hb_i)_{stack} \rangle$ $\quad (G_s)_{stack} = \langle E_{stack}, po_{stack}, so_{stack}, (hb_s)_{stack} \rangle$

show: $hb_i = hb_s$

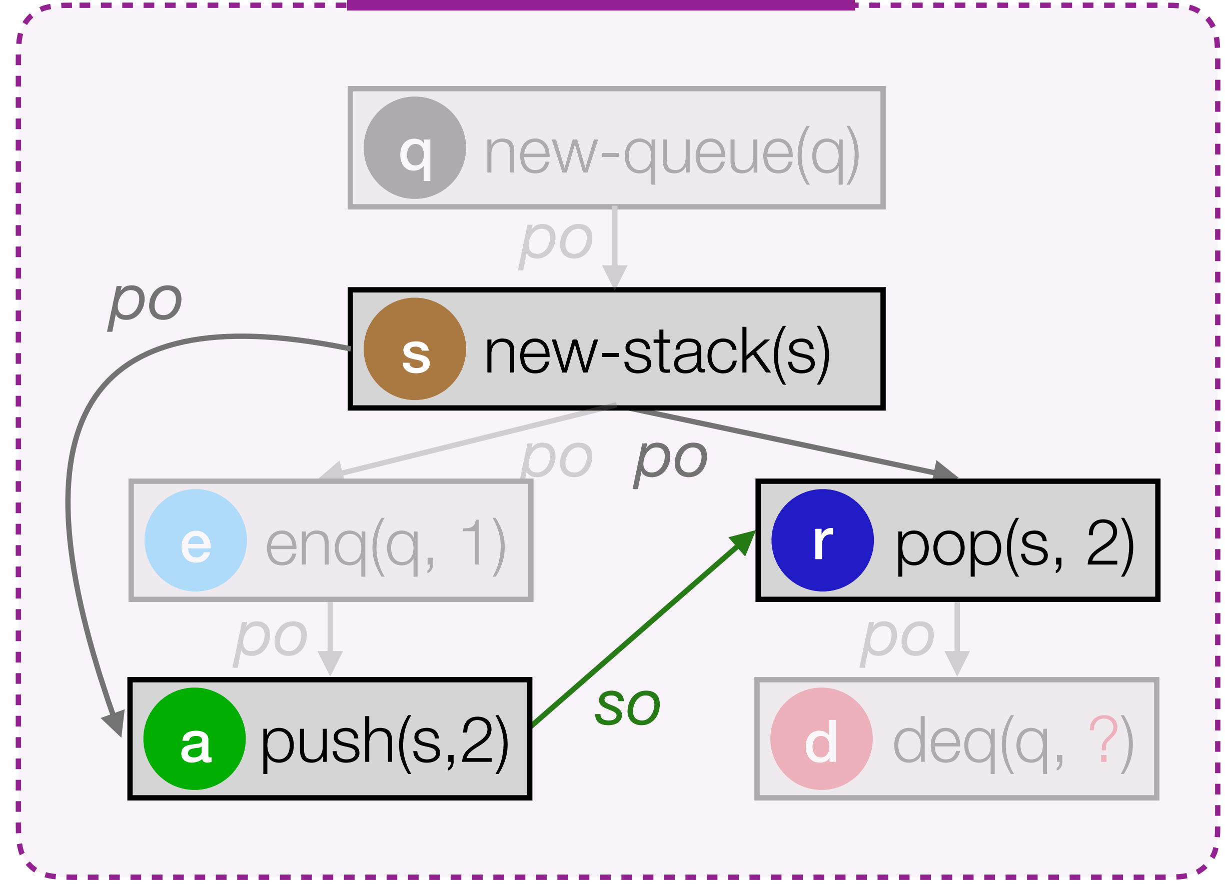
$hb_s = (hb)_{stack} = ((po \cup so)^+)_{stack}$

Example: HWQ Soundness

Given $G_i \in [\![\lfloor P \rfloor_{I_{queue}}]\!]$



Find $G_s \in [\![P]\!]$



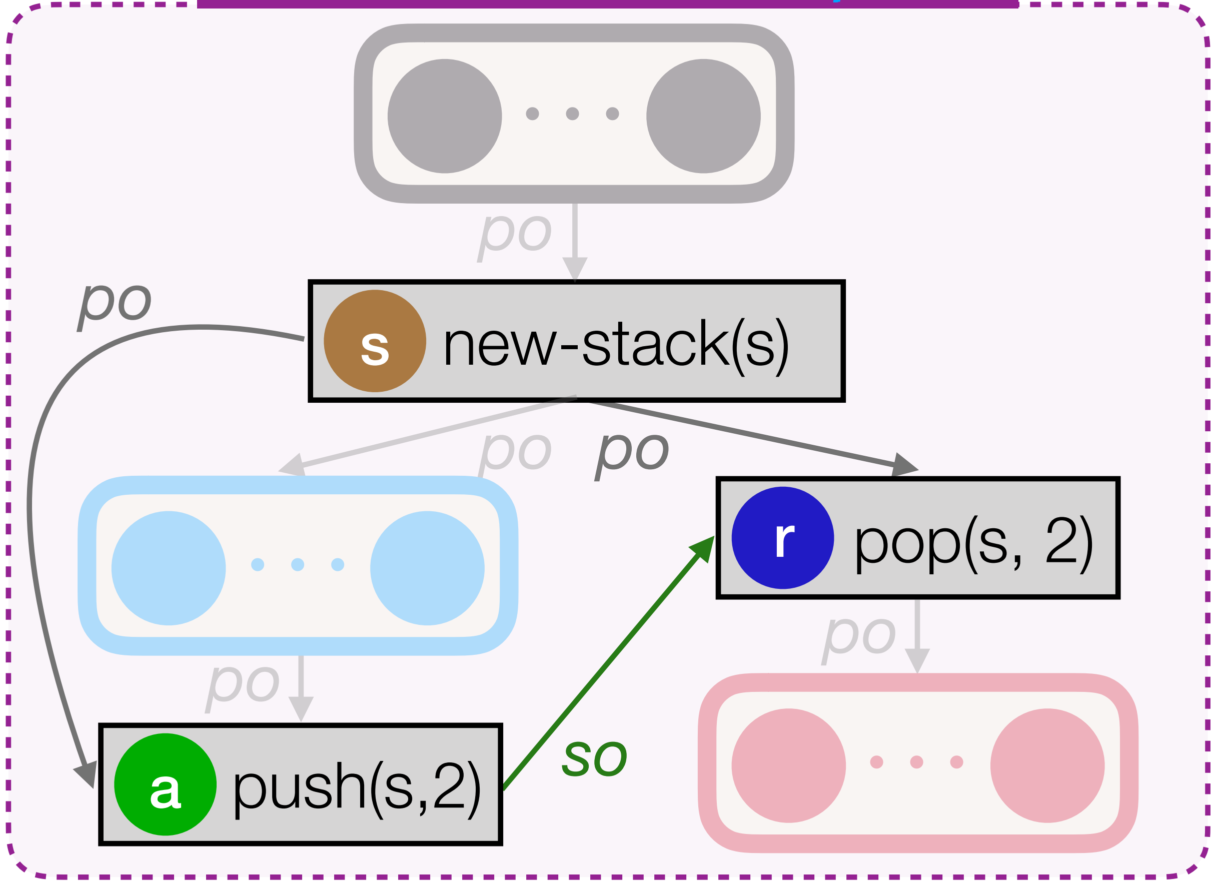
$(G_i)_{stack} = \langle E_{stack}, po_{stack}, so_{stack}, (hb_i)_{stack} \rangle$ $\quad$ $(G_s)_{stack} = \langle E_{stack}, po_{stack}, so_{stack}, (hb_s)_{stack} \rangle$

show: $hb_i = hb_s$

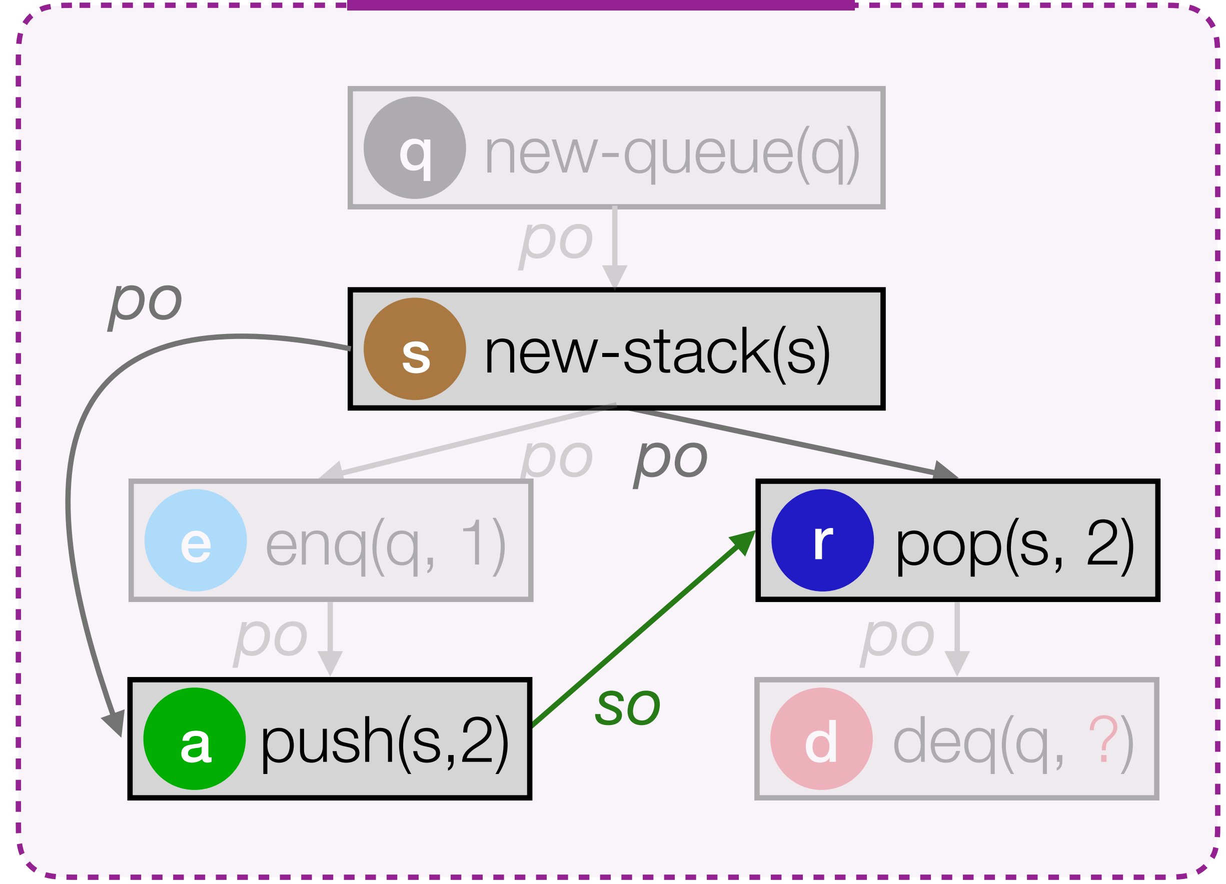
$hb_s = (hb)_{stack} = ((po \cup so_{queue} \cup so_{stack})^+)_{stack}$

Example: HWQ Soundness

Given $G_i \in [\![\lfloor P \rfloor_{I_{queue}}]\!]$



Find $G_s \in [\![P]\!]$



$(G_i)_{stack} = \langle E_{stack}, po_{stack}, so_{stack}, (hb_i)_{stack} \rangle$ $\quad$ $(G_s)_{stack} = \langle E_{stack}, po_{stack}, so_{stack}, (hb_s)_{stack} \rangle$

show: $hb_i = hb_s$

sufficient: $G_s . so_{queue} \subseteq G_i . hb_i$

$hb_s = (hb)_{stack} = ((po \cup so_{queue} \cup so_{stack})^+)_{stack}$

Local Soundness

I_{queue} sound $\iff$ for all P :

$$\forall G_i \in [\![\lfloor P \rfloor_{I_{queue}}]\!] .\ \exists G_s \in [\![P]\!] .\ val(G_i) = val(G_s)$$

Local Soundness

$\mathcal{I}_{queue}$ sound $\iff$ for all P :

$$\forall G_i \in [\![\lfloor P \rfloor_{\mathcal{I}_{queue}}]\!] .\ \exists G_s \in [\![P]\!] .\ val(G_i) = val(G_s)$$

$\Downarrow$ (when P calls queue, L_1, ..., L_n)

$\mathcal{I}_{queue}$ sound $\iff$ for all P :

$$\forall G_i \in [\![\lfloor P \rfloor_{\mathcal{I}_{queue}}]\!] .\ \exists G_s .\ val(G_i) = val(G_s)$$

$(G_s)_{queue}$ sats. queue axioms

$(G_s)_{L_1}$ sats. L_1 axioms

...

$(G_s)_{L_n}$ sats. L_n axioms

Local Soundness

(when P calls *queue*, L_1, ..., L_n)

$\mathcal{I}_{queue}$ sound $\Longleftrightarrow$ for all P :

$$\forall G_i \in [\![\lfloor P \rfloor_{\mathcal{I}_{queue}}]\!] .\ \exists G_s .\ \begin{array}{l} val(G_i) = val(G_s) \\ (G_s)_{queue} \text{ sats. } queue \text{ axioms} \\ (G_s)_{L_1} \text{ sats. } L_1 \text{ axioms} \\ \quad\quad\quad \dots \\ (G_s)_{L_n} \text{ sats. } L_n \text{ axioms} \end{array}$$

$\mathcal{I}_{queue}$ ***locally sound*** $\Longleftrightarrow$ for all P :

$$\forall G_i \in [\![\lfloor P \rfloor_{\mathcal{I}_{queue}}]\!] .\ \exists G_s .\ \begin{array}{l} val(G_i) = val(G_s) \\ (G_s)_{queue} \text{ sats. } queue \text{ axioms} \\ G_s . so_{queue} \subseteq G_i . hb \end{array}$$

Local Soundness

(when P calls *queue*, L_1, ..., L_n)

$\mathcal{I}_{queue}$ sound $\iff$ for all P :

$$\forall G_i \in [\![\lfloor P \rfloor_{\mathcal{I}_{queue}}]\!] .\ \exists G_s .\ val(G_i) = val(G_s)$$

$(G_s)_{queue}$ *sats.* *queue axioms*

Given for free!

$(G_s)_{L_1}$ *sats.* L_1 *axioms*

...

$(G_s)_{L_n}$ sats. L_n axioms

$\mathcal{I}_{queue}$ ***locally sound*** $\iff$ for all P :

$$\forall G_i \in [\![\lfloor P \rfloor_{\mathcal{I}_{queue}}]\!] .\ \exists G_s .\ val(G_i) = val(G_s)$$

$(G_s)_{queue}$ *sats.* *queue axioms*

$G_s . so_{queue} \subseteq G_i . hb$

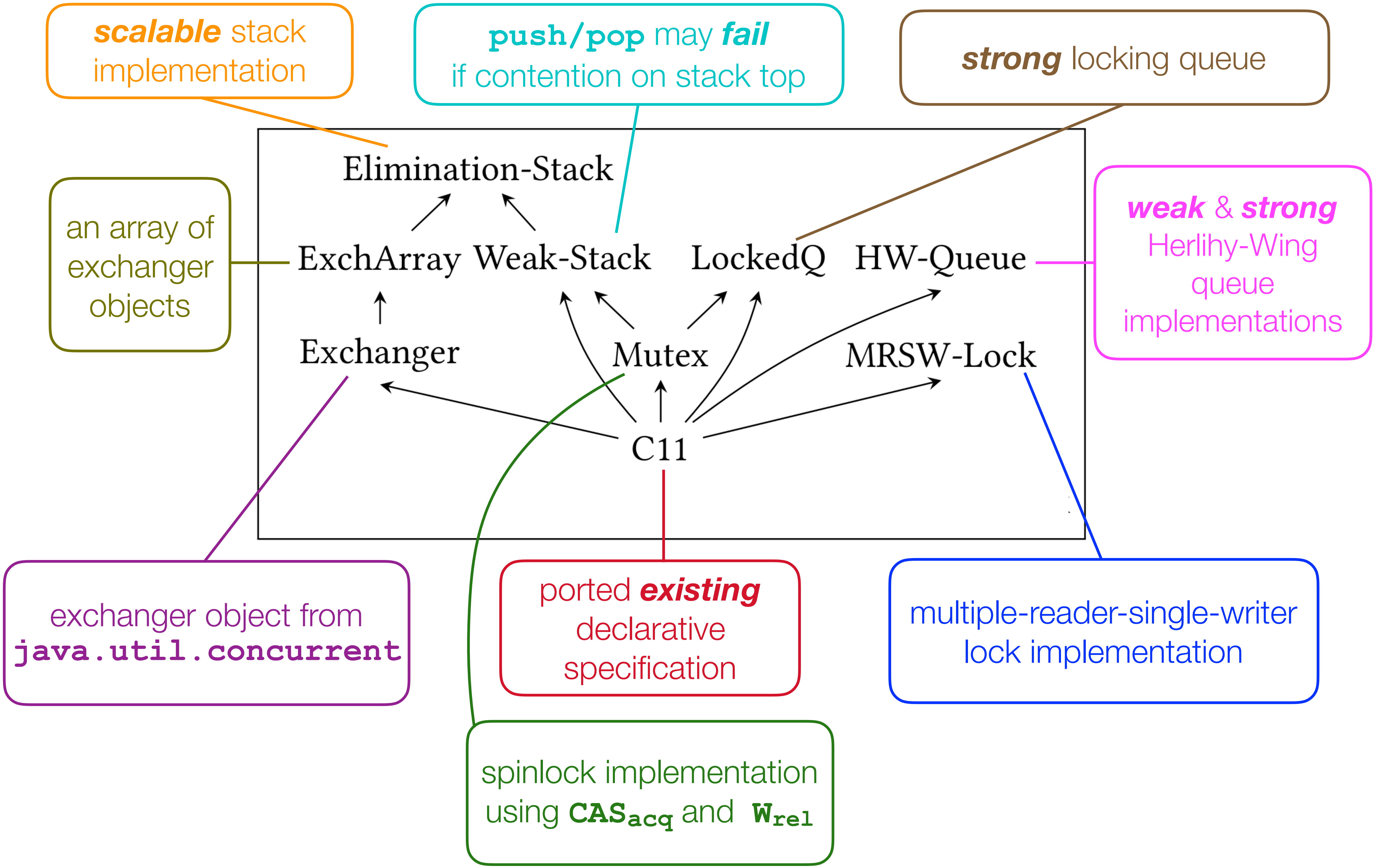
Compositionality

Theorem (Compositionality)

$$I \text{ locally-sound} \implies I \text{ sound}$$

Proof. Mechanised in Coq.

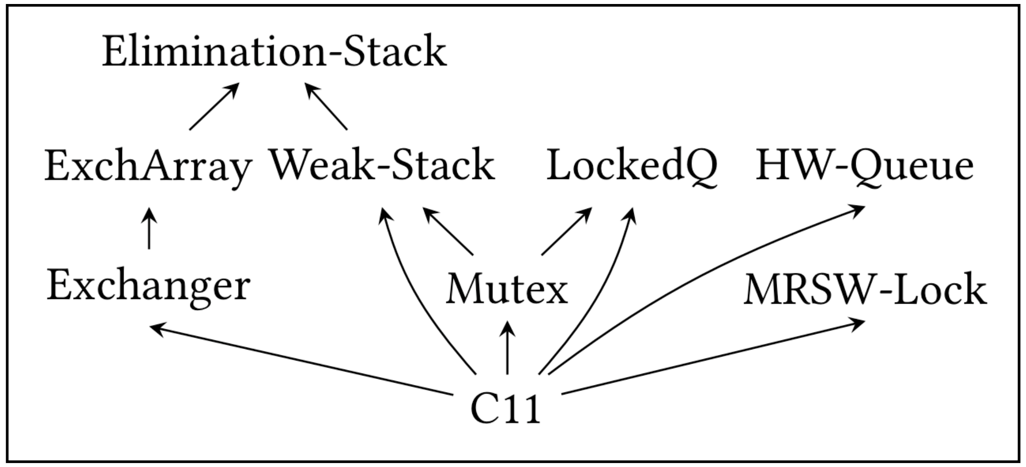
Case Studies



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*



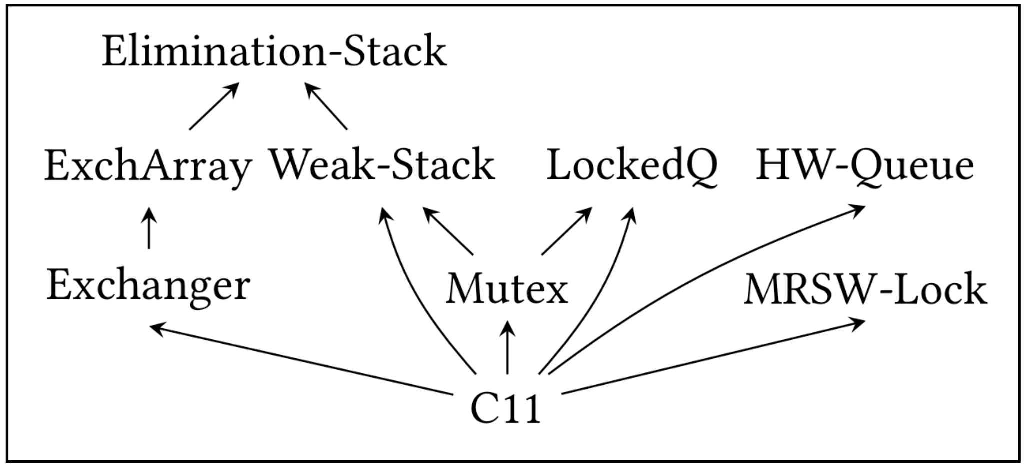
Future Work

- Support ***load buffering*** behaviour
  - allow (*po* ∪ *so*)$^+$ cycles
- MM-agnostic, general ***program logic***
  - port existing WMC logics, e.g. RSL, GPS, …
- MM-agnostic (stateless) ***model-checking***
  - a generalisation of existing approaches, e.g. RCMC, …

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