Reducing Linearizability to Classic Verification Problems

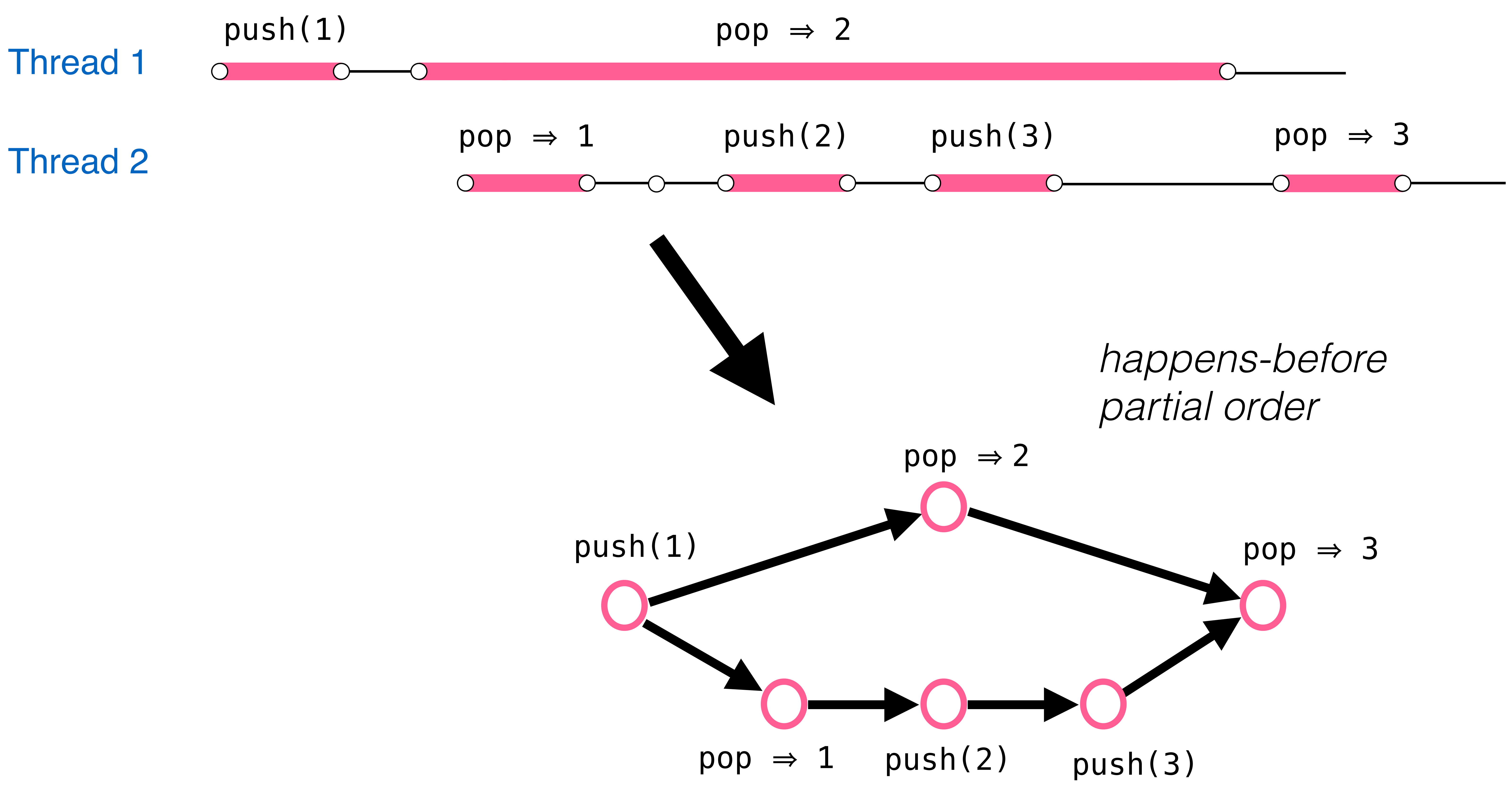
Constantin Enea

Ecole Polytechnique

Checking Lin. using “bad patterns”

- Reduce linearizability checking to reachability (EXPSPACE-complete):
  - Define (sequential) data-structure S using inductive rules
  - S is data independent and closed under projection
  - Characterize sequential executions of S using bad patterns
  - Characterize concurrent executions, linearizable w.r.t. S using bad patterns (one per rule)
  - Define a regular automaton A_i for each bad pattern
  - Reduce“L is linearizable w.r.t. S” to: for all i, $L \cap A_i = \emptyset$

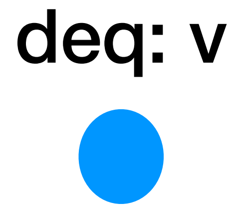
Histories = Posets of events



Concurrent Queues [ICALP'15]

Linearizability ≡ Exclusion of **bad patterns** (assuming each value is enqueued at most once - sound under data independence)

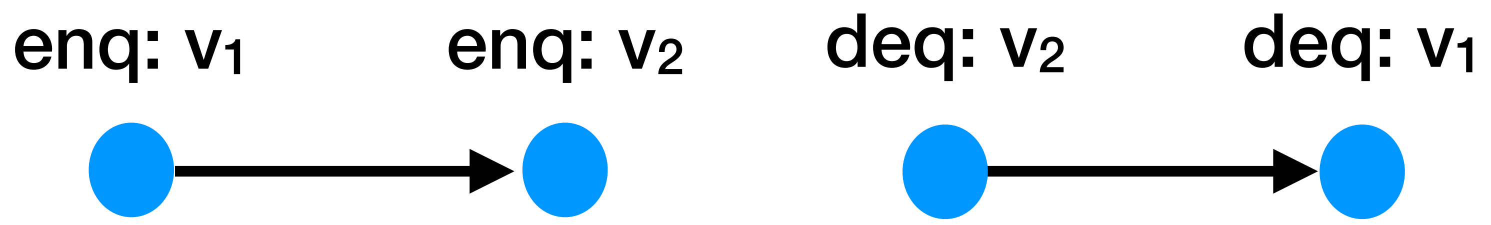
"Value v dequeued without being enqueued"



"Value v dequeued before being enqueued"

"Value v dequeued twice"

"Values dequeued in the wrong order"

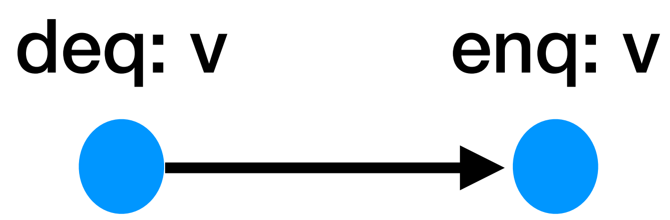


Concurrent Queues [ICALP’15]

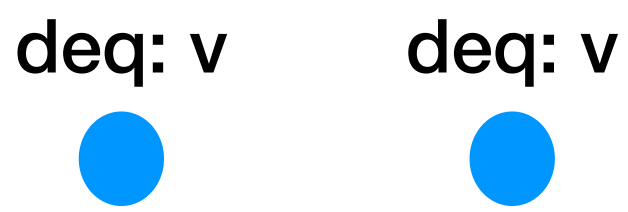
Linearizability ≡ Exclusion of **bad patterns** (assuming each value is enqueued at most once - sound under data independence)

“Value v dequeued without being enqueued”

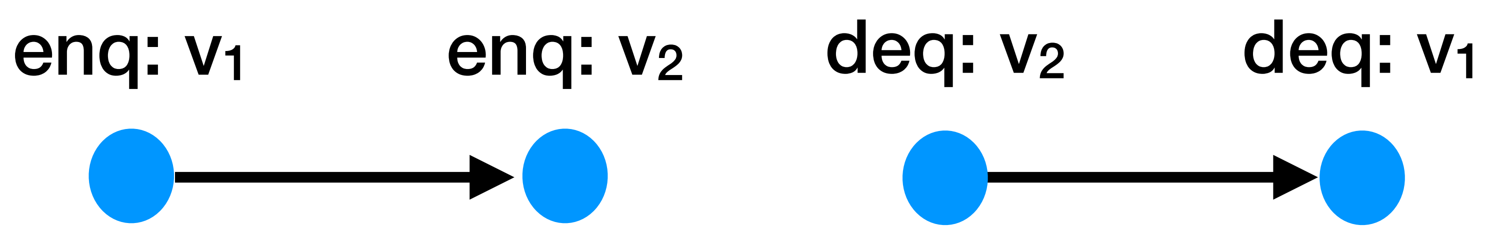
“Value v dequeued before being enqueued”



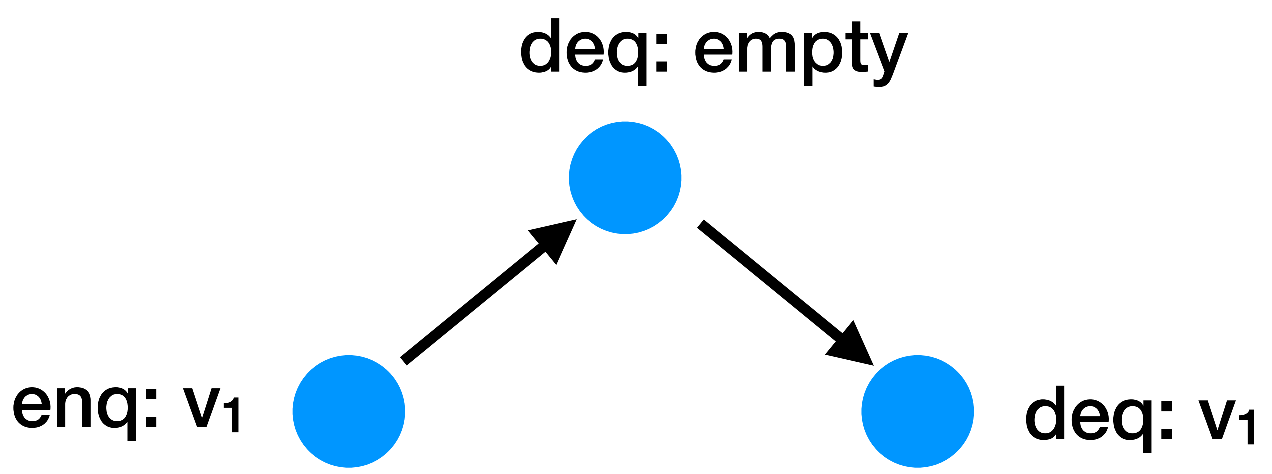
“Value v dequeued twice”



“Values dequeued in the wrong order”



“Dequeue wrongfully returns empty”

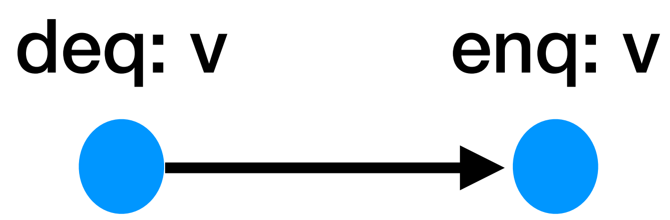


Concurrent Queues [ICALP'15]

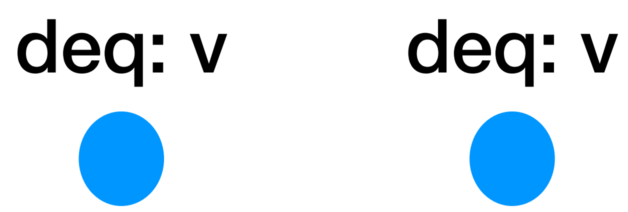
Linearizability ≡ Exclusion of **bad patterns** (assuming each value is enqueued at most once - sound under data independence)

"Value v dequeued without being enqueued"

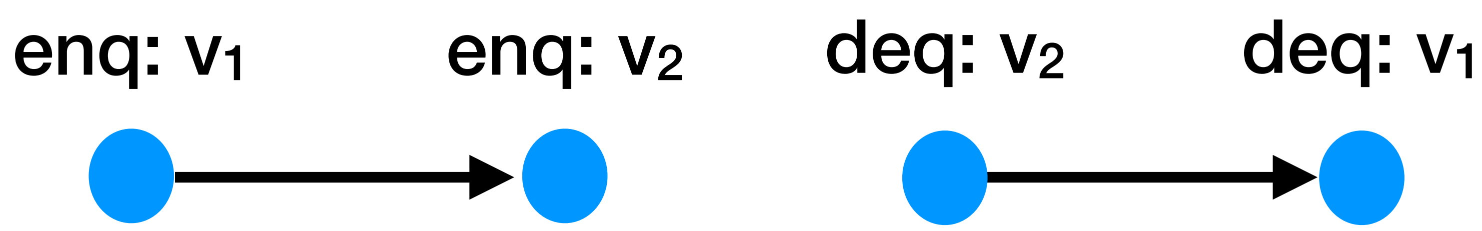
"Value v dequeued before being enqueued"



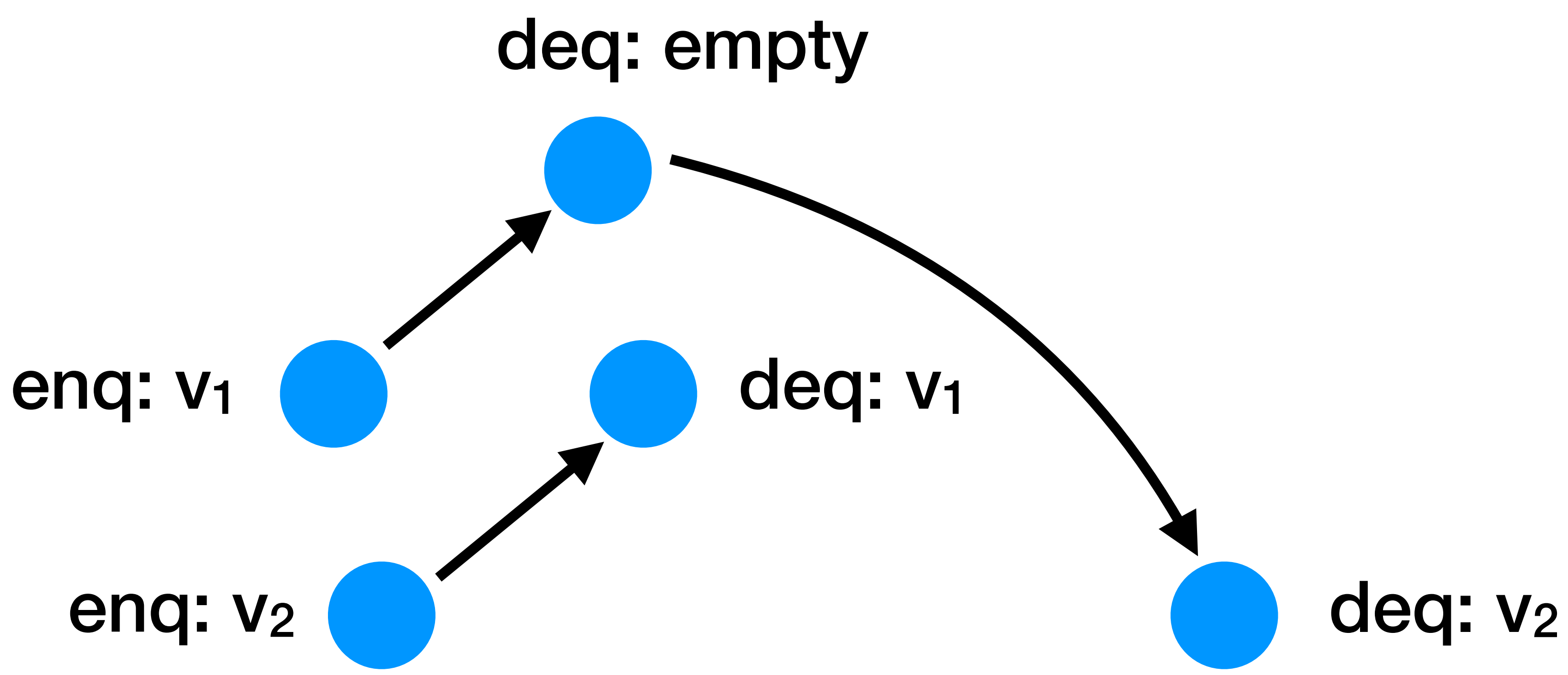
"Value v dequeued twice"



"Values dequeued in the wrong order"



"Dequeue wrongfully returns empty"



Concurrent Queues [ICALP'15]

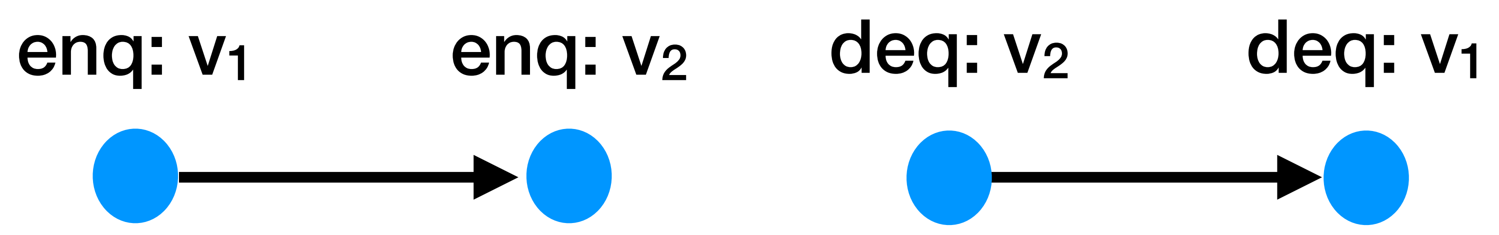
Linearizability ≡ Exclusion of **bad patterns** (assuming each value is enqueued at most once - sound under data independence)

"Value v dequeued without being enqueued"

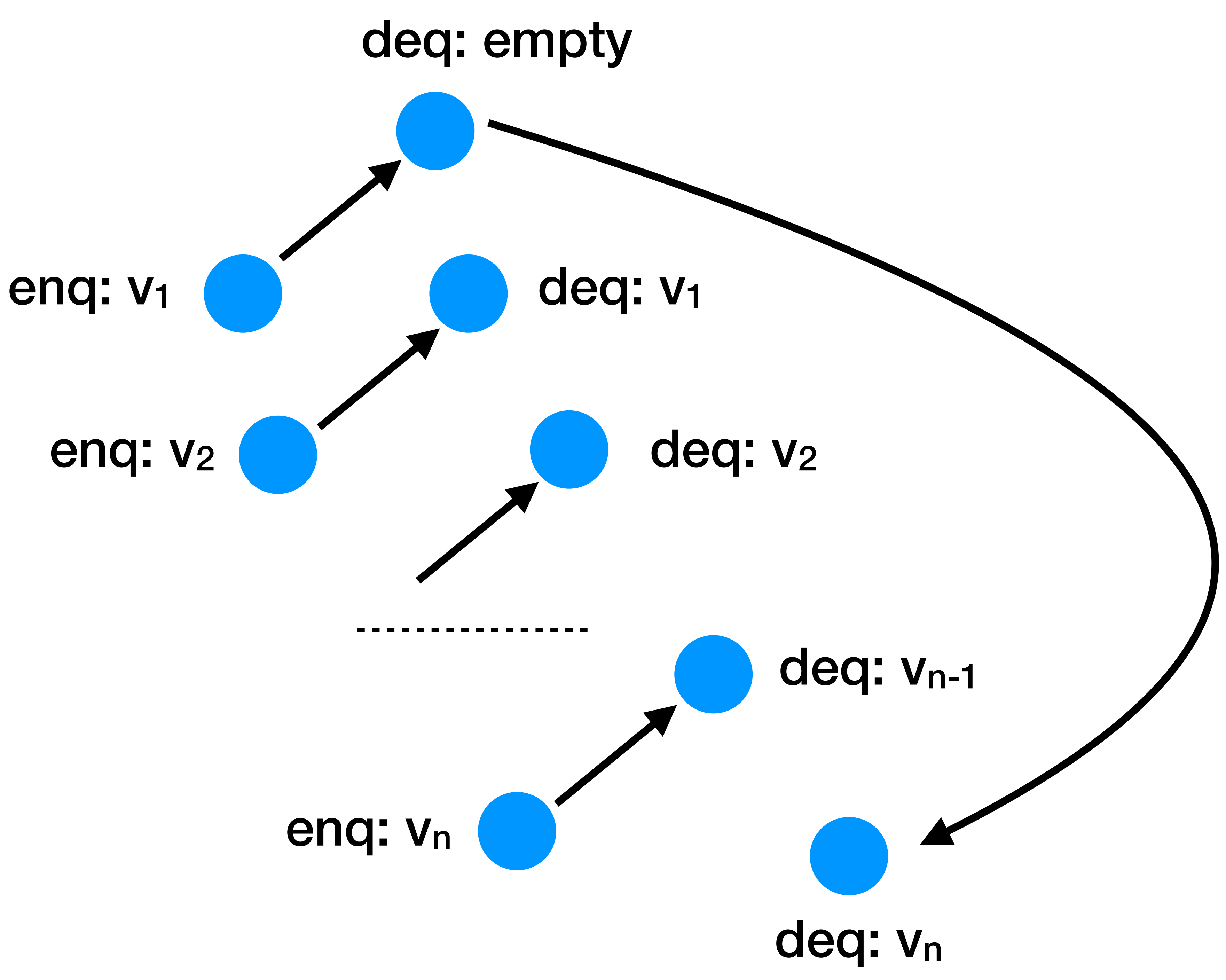
"Value v dequeued before being enqueued"

"Value v dequeued twice"

"Values dequeued in the wrong order"



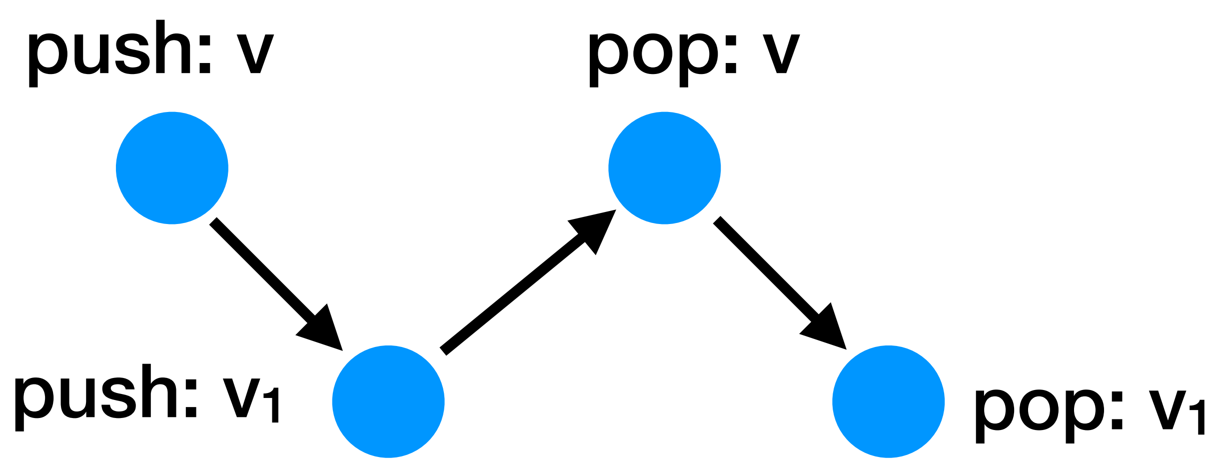
"Dequeue wrongfully returns empty"



Concurrent Stacks [ICALP'15]

Linearizability ≡ Exclusion of **bad patterns** (assuming each value is enqueued at most once, which is sound under data independence)

"Pop doesn't return the top of the stack"



"Value v popped without being pushed"

"Value v popped before being pushed"

"Value v popped twice"

"Pop wrongfully returns empty"

Concurrent Stacks [ICALP'15]

Linearizability ≡ Exclusion of **bad patterns** (assuming each value is enqueued at most once, which is sound under data independence)

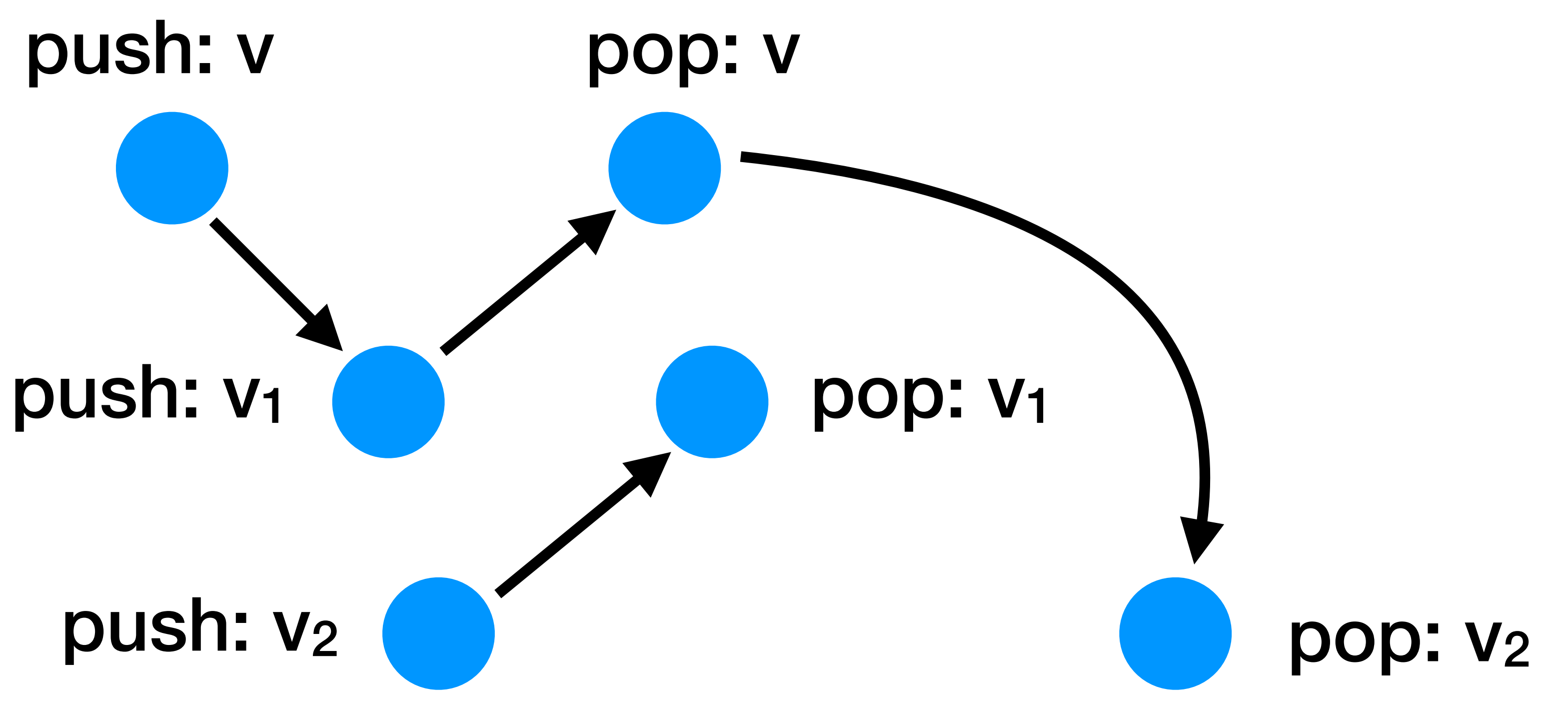
"Value v popped without being pushed"

"Value v popped before being pushed"

"Value v popped twice"

"Pop wrongfully returns empty"

"Pop doesn't return the top of the stack"



Concurrent Stacks [ICALP'15]

Linearizability ≡ Exclusion of **bad patterns** (assuming each value is enqueued at most once, which is sound under data independence)

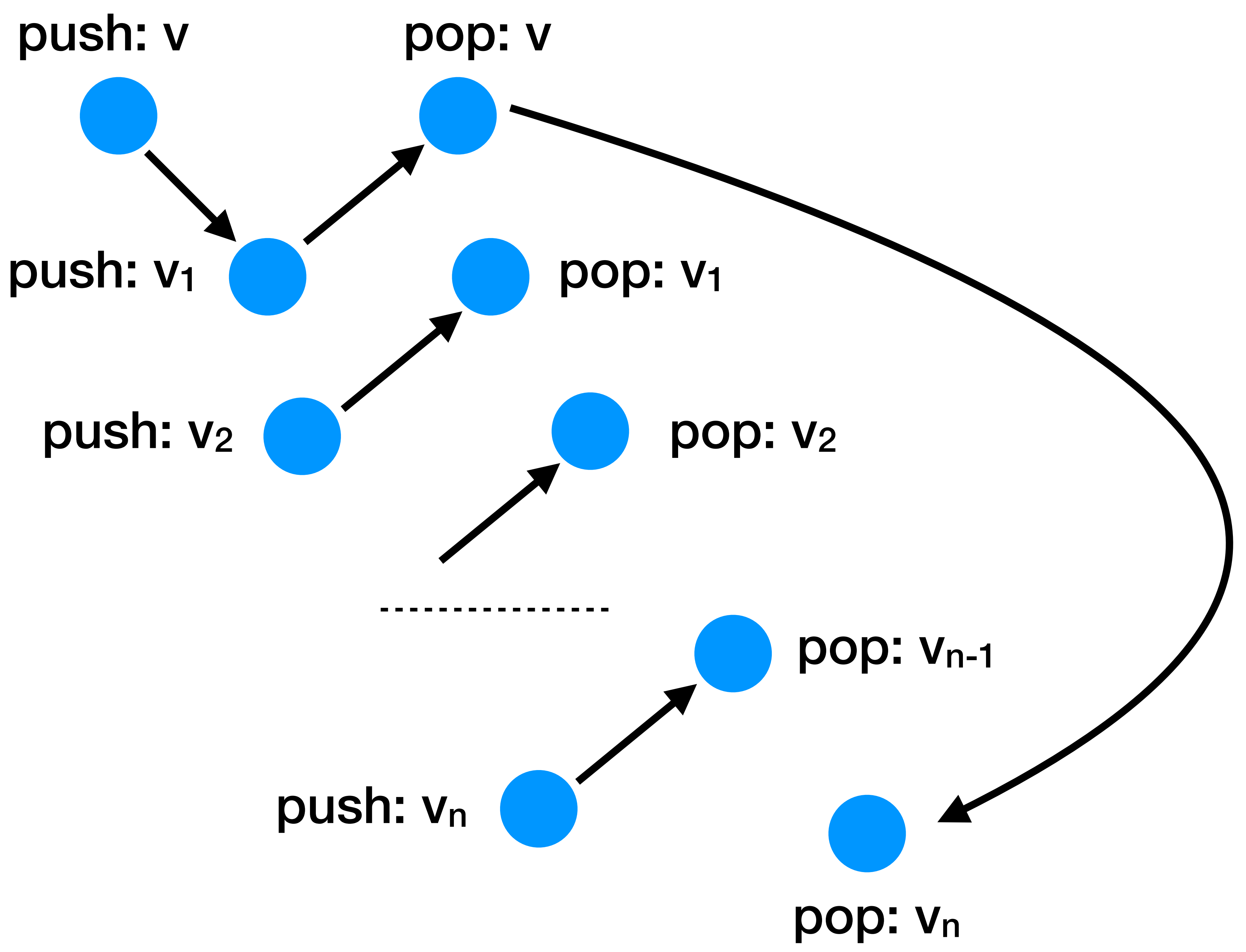
"Value v popped without being pushed"

"Value v popped before being pushed"

"Value v popped twice"

"Pop wrongfully returns empty"

"Pop doesn't return the top of the stack"



Data Independence

- Input methods = methods taking an argument
- A sequential execution u is called *differentiated* if for all input methods m and every x, u contains at most one invocation m(x)
- $S_{\neq}$ is the set of differentiated executions in S

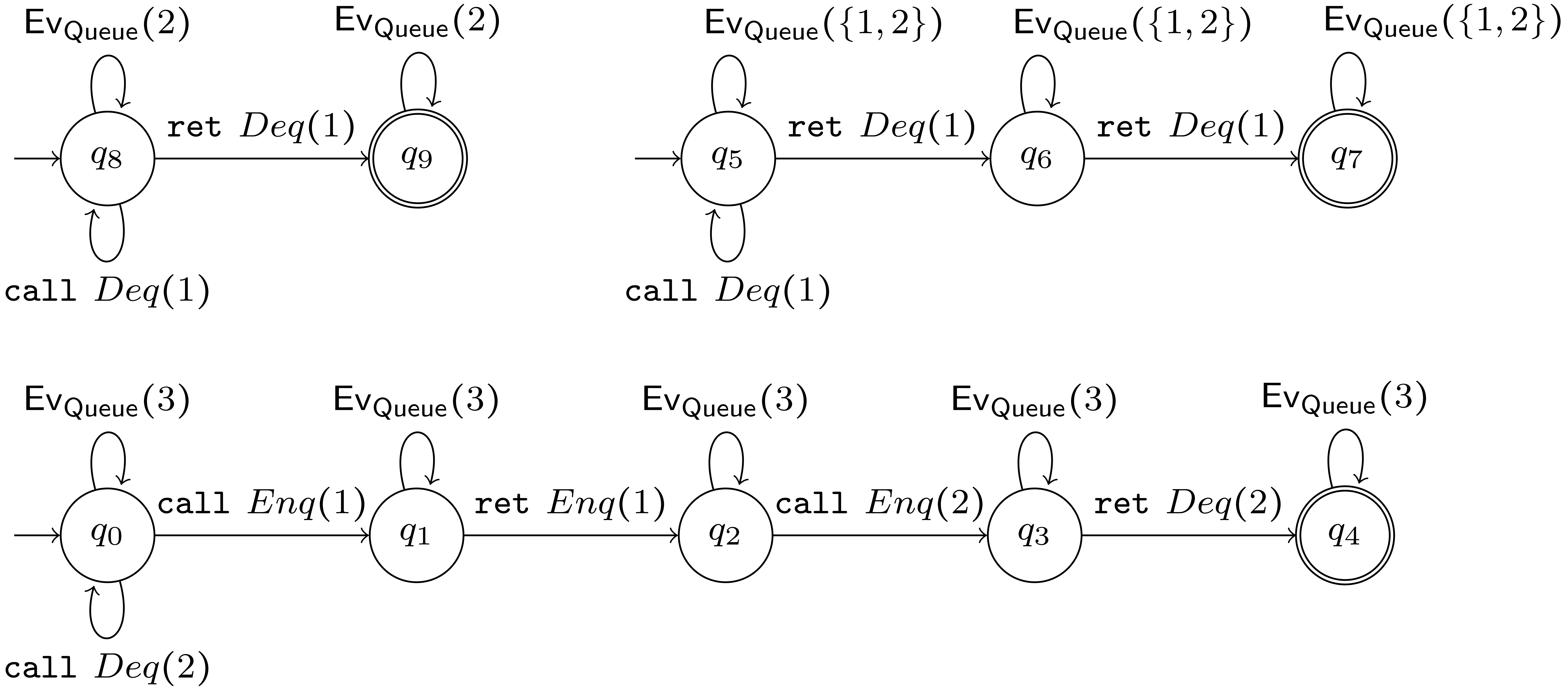
A *renaming* r is a function from $\mathbb{D}$ to $\mathbb{D}$. Given a sequential execution (resp., execution or history) u, we denote by $r(u)$ the sequential execution (resp., execution or history) obtained from u by replacing every data value x by $r(x)$.

Definition 6. *The set of sequential executions (resp., executions or histories) S is* data independent *if:*

- *for all $u \in S$, there exists $u' \in S_{\neq}$, and a renaming r such that $u = r(u')$,*
- *for all $u \in S$ and for all renaming r, $r(u) \in S$.*

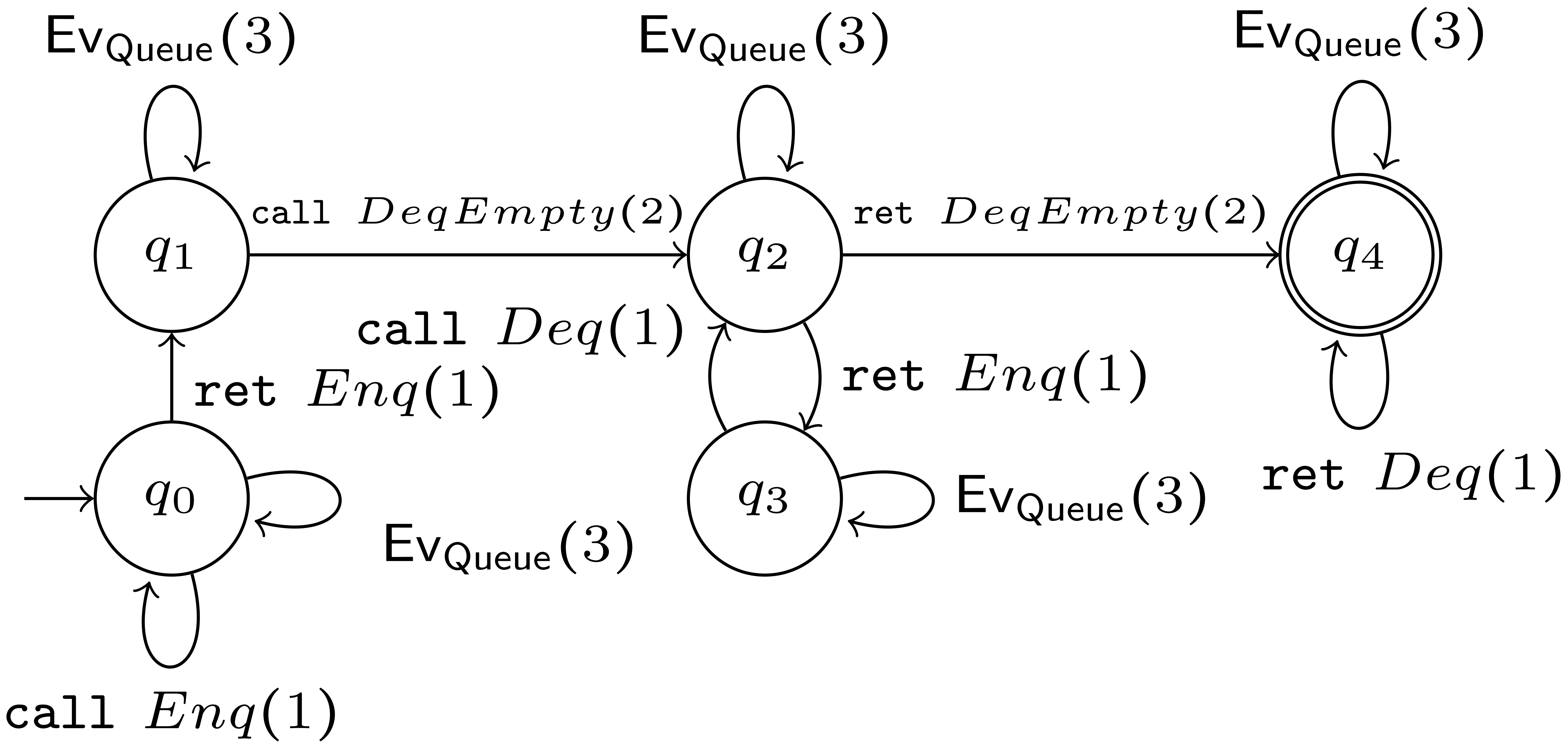
Theorem: A data-independent implementation I is linearizable w.r.t. a data-independent specification S iff $I_{\neq}$ is linearizable w.r.t. $S_{\neq}$

Characterization of concurrent executions



Characterization of concurrent executions

REMP



we assume that all actions call Enq(1) occur at the beginning

Exercices

We consider a sequential specification defined by the language $S = (\texttt{a()})^*(\texttt{b()})^*$ where all the invocations of `a()` occur before invocations of `b()`.

1. Describe a reduction of checking linearizability w.r.t. the specification S to a reachability problem. More precisely, describe a labeled transition system (monitor) that accepts exactly all the histories of a given implementation (sequences of call and return actions) that are *not* linearizable w.r.t. S. The synchronized product between a transition system representing an implementation and this monitor (where the synchronization actions are call and returns) reaches an accepting state of the monitor iff the implementation is not linearizable.

Exercices

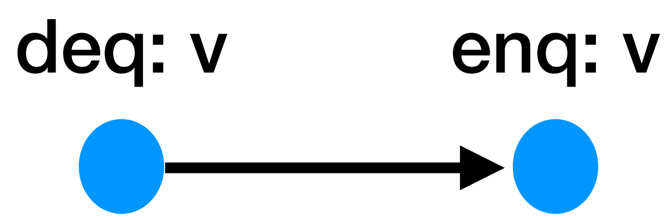
- What is the complexity of checking linearizability of a *differentiated* history *of a* concurrent queue?

Exercices

- What is the complexity of checking linearizability of a *differentiated* history *of a* concurrent queue?

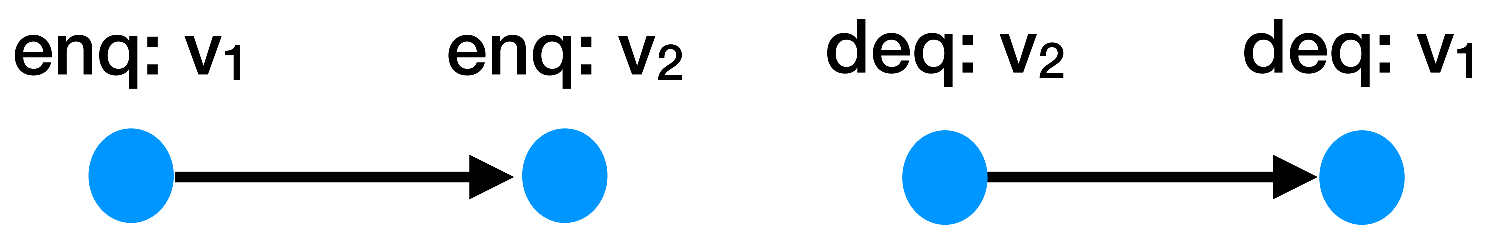
“Value v dequeued without being enqueued”

“Value v dequeued before being enqueued”

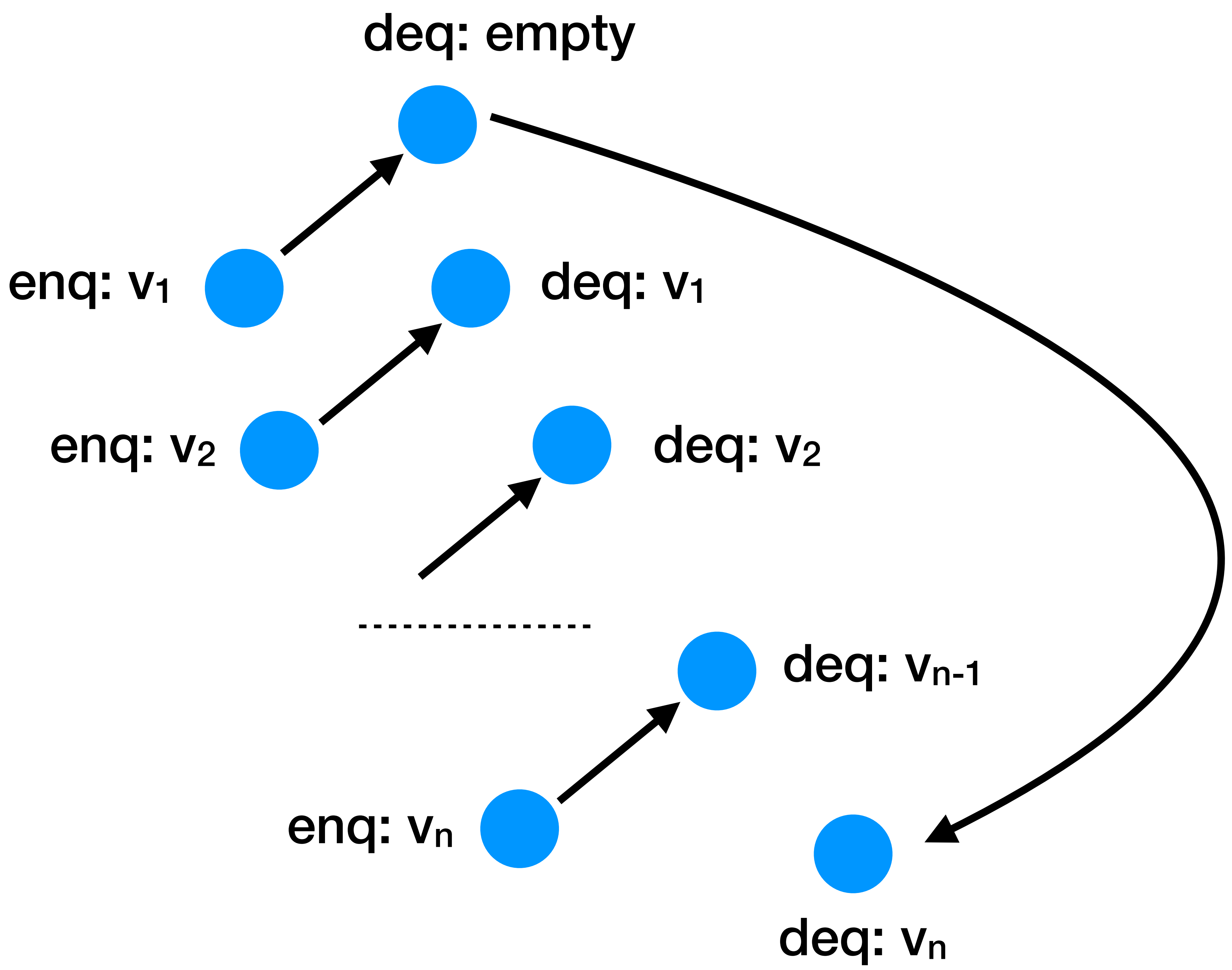


“Value v dequeued twice”

“Values dequeued in the wrong order”



“Dequeue wrongfully returns empty”



Observational Refinement

<=>

Linearizability/ Refinement

Observational Refinement

Reference implementation

```
class AtomicStack {
  cell∗ top;
  Lock l;

  void push (int v) {
    l.lock();
    top->next = malloc(sizeof ∗x);
    top = top->next;
    top->data = v;
    l.unlock();
  }

  int pop () {
    ...
  }
}
```

minimize contention

Efficient implementation

```
class TreiberStack {
  cell∗ top;

  void push (int v) {
    cell∗ t;
    cell∗ x = malloc(sizeof ∗x);
    x->data = v;
    do {
      t = top;
      x->next = top;
    } while (!CAS(&top,t,x));
  }

  int pop () {
    ...
  }
}
```

For every Client,

Client x Impl included in Client x Spec

Formalizing Libraries/Programs

We fix an arbitrary set $\mathbb{O}$ of operation identifiers, and for given sets $\mathbb{M}$ and $\mathbb{V}$ of methods and values, we fix the sets

$$C = \{m(v)_o : m \in \mathbb{M}, v \in \mathbb{V}, o \in \mathbb{O}\}, \text{ and}$$
$$R = \{\mathsf{ret}(v)_o : v \in \mathbb{V}, o \in \mathbb{O}\}$$

of *call actions* and *return actions*; each call action $m(v)_o$ combines a method $m \in \mathbb{M}$ and value $v \in \mathbb{V}$ with an *operation identifier* $o \in \mathbb{O}$. Operation identifiers are used to pair call and return actions.

A sequence in (C ∪ R)* is **well-formed** if every return is preceeded by a matching call, each identifier is used at most once

A sequence in (C ∪ R)* is **sequential** if there exists a return between every successive two calls

Formalizing Libraries/Programs

Definition 3.1. *A* library L *is an LTS over alphabet* $C \cup R$ *such that each execution* $e \in E(L)$ *is well formed, and*

- *Call actions* $c \in C$ *cannot be disabled:*
  $e \cdot e' \in E(L)$ *implies* $e \cdot c \cdot e' \in E(L)$ *if* $e \cdot c \cdot e'$ *is well formed.*
- *Call actions* $c \in C$ *cannot disable other actions:*
  $e \cdot a \cdot c \cdot e' \in E(L)$ *implies* $e \cdot c \cdot a \cdot e' \in E(L)$.
- *Return actions* $r \in R$ *cannot enable other actions:*
  $e \cdot r \cdot a \cdot e' \in E(L)$ *implies* $e \cdot a \cdot r \cdot e' \in E(L)$.

Definition 3.2. *A* program P *over actions* Σ *is an LTS over alphabet* $(\Sigma \uplus C \uplus R)$ *where each execution* $e \in E(P)$ *is well formed, and*

- *Call actions* $c \in C$ *cannot enable other actions:*
  $e \cdot c \cdot a \cdot e' \in E(P)$ *implies* $c \mapsto a$ *or* $e \cdot a \cdot c \cdot e' \in E(P)$.
- *Return actions* $r \in R$ *cannot disable other actions:*
  $e \cdot a \cdot r \cdot e' \in E(P)$ *implies* $a \mapsto r$ *or* $e \cdot r \cdot a \cdot e \in E(P)$.
- *Return actions* $r \in R$ *cannot be disabled:*
  $e \cdot e' \in E(P)$ *implies* $e \cdot r \cdot e' \in E(L)$ *if* $e \cdot r \cdot e'$ *is well formed.*

Observational Refinement

Definition 3.3. *The library* L_1 refines L_2, *written* $L_1 \leq L_2$, *iff*

$$E(P \times L_1)|\Sigma \ \subseteq \ E(P \times L_2)|\Sigma$$

for all programs P *over actions* Σ.

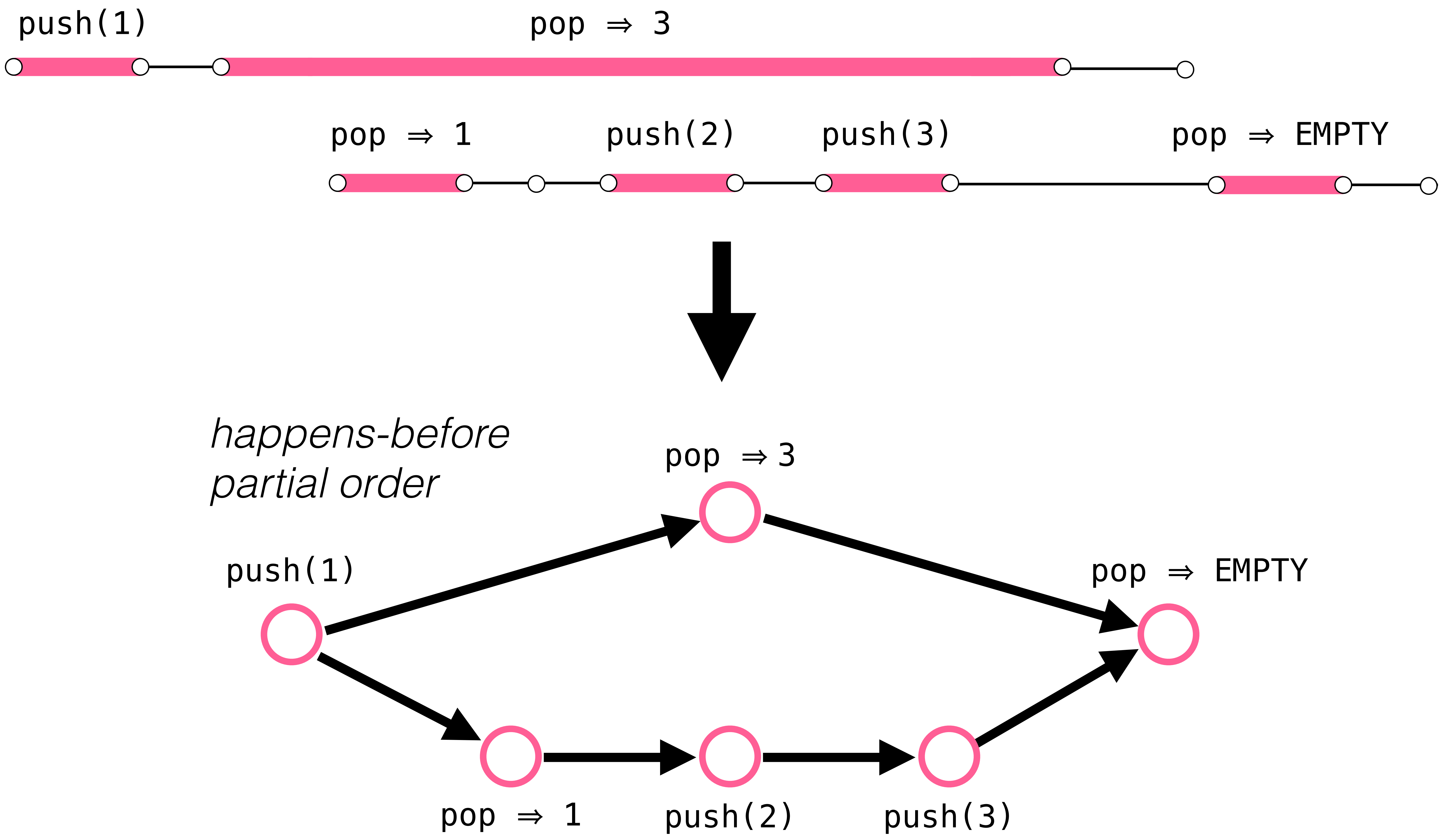
Histories

For given sets $\mathbb{M}$ and $\mathbb{V}$ of methods and values, we fix a set $\mathbb{L} = \mathbb{M} \times \mathbb{V} \times (\mathbb{V} \cup \{\bot\})$ of *operation labels*, and denote the label $\langle m, u, v \rangle$ by $m(u) \Rightarrow v$. A *history* $h = \langle O, <, f \rangle$ is a partial order $<$ on a set $O \subseteq \mathbb{O}$ of operation identifiers labeled by $f : O \to \mathbb{L}$ for which $f(o) = m(u) \Rightarrow \bot$ implies o is maximal in $<$. The *history* $H(e)$ of a well-formed execution $e \in \Sigma^*$ labels each operation with a method-call summary, and orders non-overlapping operations:

- $O = \{\mathsf{op}(e_i) : 0 \leq i < |e| \text{ and } e_i \in C\}$,
- $\mathsf{op}(e_i) < \mathsf{op}(e_j)$ iff $i < j$, $e_i \in R$, and $e_j \in C$.
- $f(o) = \begin{cases} m(u) \Rightarrow v & \text{if } m(u)_o \in e \text{ and } \mathsf{ret}(v)_o \in e \\ m(u) \Rightarrow \bot & \text{if } m(u)_o \in e \text{ and } \mathsf{ret}(\_)_o \notin e \end{cases}$

The histories admitted by a library L are H(L) = { H(e): e ∈ E(L) }

Histories


push(1)
pop ⇒ 3
pop ⇒ 1
push(2)
push(3)
pop ⇒ EMPTY
happens-before
partial order
pop ⇒ 3
push(1)
pop ⇒ EMPTY
pop ⇒ 1
push(2)
push(3)

Histories

Definition 4.2. *Let* $h_1 = \langle O_1, <_1, f_1 \rangle$ *and* $h_2 = \langle O_2, <_2, f_2 \rangle$*. We say* h_1 *is* weaker than h_2*, written* $h_1 \preceq h_2$*, when there exists an injection* $g : O_2 \to O_1$ *such that*

- $o \in \mathrm{range}(g)$ *when* $f_1(o) = m(u) \Rightarrow v$ *and* $v \neq \bot$*,*
- $g(o_1) <_1 g(o_2)$ *implies* $o_1 <_2 o_2$ *for each* $o_1, o_2 \in O_2$*,*
- $f_1(g(o)) \ll f_2(o)$ *for each* $o \in O_2$*.*

where $(m_1(u_1) \Rightarrow v_1) \ll (m_2(u_2) \Rightarrow v_2)$ *iff* $m_1 = m_2$*,* $u_1 = u_2$*, and* $v_1 \in \{v_2, \bot\}$*. We say* h_1 *and* h_2 *are* equivalent *when* $h_1 \preceq h_2$ *and* $h_2 \preceq h_1$*.*

Examples ?

Equivalent histories need not be distinguished

Histories

$$\textit{If } h_1 \in H(L) \textit{ and } h_2 \preceq h_1 \textit{ then } h_2 \in H(L).$$

$$E(L) = \{e \in (C \cup R)^* : H(e) \in H(L)\}.$$

History Inclusion

THEOREM

L_1 refines L_2 $\Leftrightarrow$ $H(L_1) \subseteq H(L_2)$ $\Leftrightarrow$ $E(L_1) \subseteq E(L_2)$

- (=>) Given h in Hist(L_1), construct a program P_h that imposes all the happen-before constraints of h.
- (<=) Clients cannot distinguish executions with the same history. History inclusion implies Execution Inclusion

History Inclusion (=>)

We construct $P_h = \langle Q, \Sigma, q_0, \delta \rangle$ over alphabet $\Sigma = C \cup R \cup \{a\}$ whose states $Q : O \to \mathbb{B}^2$ track operations called/completed status. The initial state is $q_0 = \{o \mapsto \langle \bot, \bot \rangle : o \in O\}$. Transitions are given by,

for each $q \in Q, o \in O, m \in \mathbb{M}, v \in \mathbb{V}$

if $f(o) = m(v) \Rightarrow \_$ and $q(o')$ for all $o' < o$ then

$$q[o \mapsto \bot, \bot] \xrightarrow{m(v)_o} q[o \mapsto \top, \bot]$$

preserving happens-before

if $f(o) = m(\_) \Rightarrow v$ then

$$q[o \mapsto \top, \bot] \xrightarrow{\mathsf{ret}(v)_o} \cdot \xrightarrow{a} q[o \mapsto \top, \top]$$

counting ops completed in h

if $f(o) = m(\_) \Rightarrow \bot$ then

$$q[o \mapsto \top, \bot] \xrightarrow{\mathsf{ret}(v)_o} q[o \mapsto \top, \top]$$

ops that are pending in h (an execution may have more completed ops and less pending - no call for pending)

$$(??)\ \forall e \in E(P_h).\ |(e|\Sigma)| = n \implies h \preceq H(e)$$

= a^n

nb of completed ops in h

History Inclusion (=>)

$$(??)\ \forall e \in E(P_h).\ |(e|\Sigma)| = n \implies h \preceq H(e)$$

nb of completed ops in h

For every execution $e_1 \in E(P_h \times L_1)$ with $e_1 | \Sigma = n$,

there must exist an execution $e_2 \in E(P_h \times L_2)$ such that $e_2 | \Sigma = e_1 | \Sigma$

(by observational refinement)

Therefore, $h \preccurlyeq H(e_2)$.

Since $e_2 | (C \cup R) \in E(L_2)$, we have that $H(e_2) \in H(L_2)$

By closure under weakening, $h \in H(L_2)$

History Inclusion (<=)

THEOREM

L_1 refines L_2 $\Leftrightarrow$ $H(L_1) \subseteq H(L_2) \Leftrightarrow E(L_1) \subseteq E(L_2)$

Let $e \in E\,(P \times L_1)$

$e \,|\, (C \cup R) \in E(L_1)$ implies $H(e) \in H(L_1)$ implies $H(e) \in H(L_2)$

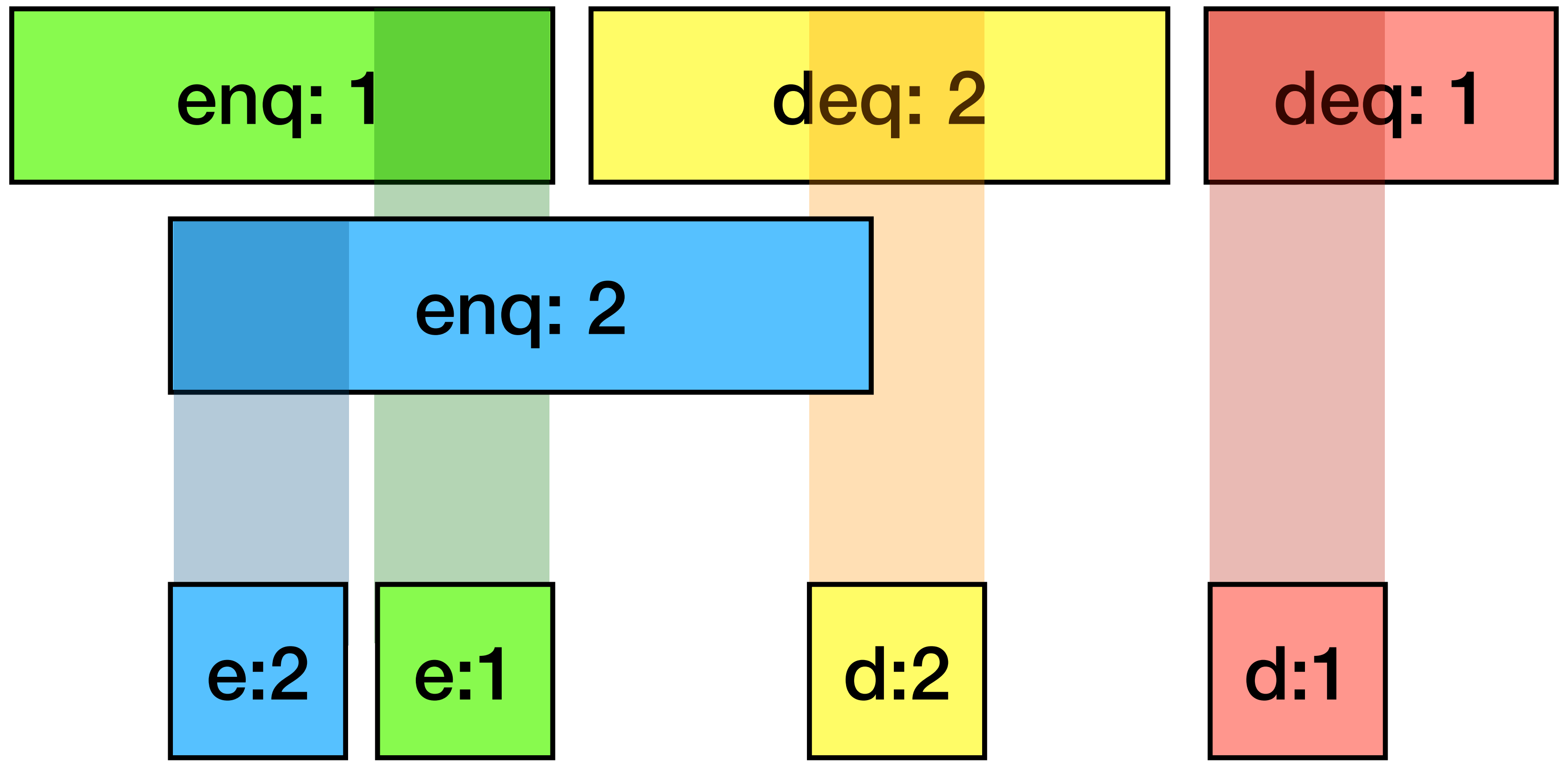
Therefore, $e \,|\, (C \cup R) \in E(L_2)$ which by definition of the product $P \times L_2$,

implies $e \in E\,(P \times L_2)$

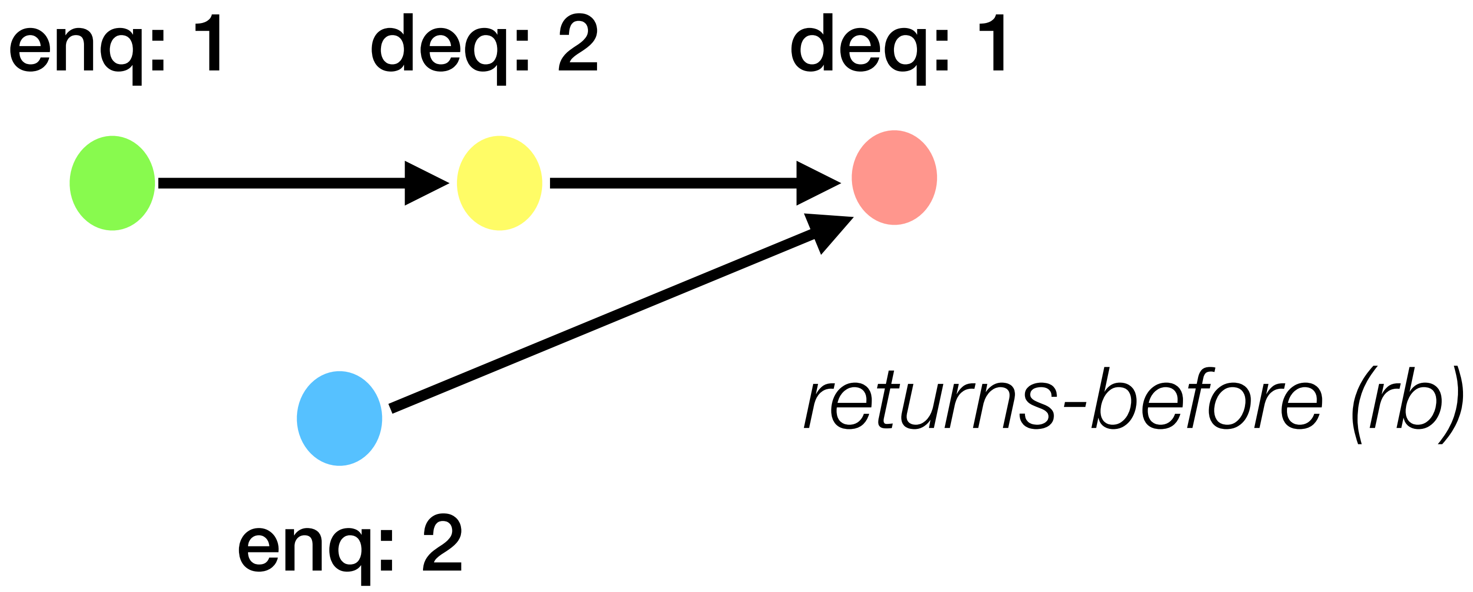
Linearizability [Herlihy&Wing 1990]

Effects of each invocation appear to occur instantaneously

Execution history



Linearization admitted by Queue ADT



$\exists$ lin. rb $\subseteq$ lin $\wedge$ lin $\in$ Queue ADT

About Linearizability

History inclusion $H(L_1) \subseteq H(L_2)$ equiv. to linearizability when L_2 is **atomic**

Definition 3.1. *A* library L *is an LTS over alphabet* $C \cup R$ *such that each execution* $e \in E(L)$ *is well formed, and*

- *Call actions* $c \in C$ *cannot be disabled:*
  $e \cdot e' \in E(L)$ *implies* $e \cdot c \cdot e' \in E(L)$ *if* $e \cdot c \cdot e'$ *is well formed.*
- *Call actions* $c \in C$ *cannot disable other actions:*
  $e \cdot a \cdot c \cdot e' \in E(L)$ *implies* $e \cdot c \cdot a \cdot e' \in E(L)$.
- *Return actions* $r \in R$ *cannot enable other actions:*
  $e \cdot r \cdot a \cdot e' \in E(L)$ *implies* $e \cdot a \cdot r \cdot e' \in E(L)$.

We write $e_1 \rightsquigarrow e_2$ when e_2 can be derived from e_1 by applying zero or more of the above rules. The *closure* of a set E of executions under $\rightsquigarrow$ is denoted $\overline{E}$.

A library L is called *atomic* if it is defined by the closure of some set E of sequential executions, i.e., $E(L) = \overline{E}$.

About Linearizability

History inclusion $H(L_1) \subseteq H(L_2)$ equiv. to linearizability when L_2 is **atomic**

Linearizability is defined by an execution order: $e_1 \sqsubseteq e_2$ iff there exists a well-formed execution e_1' obtained from e_1 by appending return actions, and deleting call actions, such that:

e_2 is a permutation of e_1' that preserves the order between return and call actions, i.e., a given return action occurs before a given call action in e_1' iff the same holds in e_2.

An execution e_1 is *linearizable* w.r.t. a library L_2 iff there exists a sequential execution $e_2 \in E(L_2)$, with only completed operations, such that $e_1 \sqsubseteq e_2$. A library L_1 is *linearizable* w.r.t. L_2, written $L_1 \sqsubseteq L_2$, iff each execution $e_1 \in E(L_1)$ is linearizable w.r.t. L_2.

About Linearizability

History inclusion $H(L_1) \subseteq H(L_2)$ equiv. to linearizability when L_2 is **atomic**

Linearizability compares execs of L_1 with pending ops. with execs of L_2 with only completed ops => problematic when L_2 contains non-terminating methods

Example 5.1. *Let L be the library whose kernel contains the single execution $e = m(u)_1\ m'(u)_2\ \mathsf{ret}(v)_1$, in which the call to m' is pending. Although L refines itself, since refinement is reflexive, L is not linearizable w.r.t. itself, since e could only be linearizable w.r.t. L if $E(L)$ were to contain one of the following executions:*

$$m(u)_1\ \mathsf{ret}(v)_1 \quad m(u)_1\ m'(u)_2\ \mathsf{ret}(v)_1\ \mathsf{ret}(\_)_2$$

$$m(u)_1\ \mathsf{ret}(v)_1\ m'(u)_2\ \mathsf{ret}(\_)_2 \quad m'(u)_2\ \mathsf{ret}(\_)_2 m(u)_1\ \mathsf{ret}(v)_1.$$

Yet $E(L) = \overline{\{e\}}$ clearly contains none of them.

About Linearizability

History inclusion $H(L_1) \subseteq H(L_2)$ equiv. to linearizability when L_2 is **atomic**

Lemma 5.1. *$e_1 \sqsubseteq e_2$ iff $H(e_1) \preceq H(e_2)$.*

Theorem 2. *$L_1 \sqsubseteq L_2$ iff $H(L_1) \subseteq H(L_2)$, if L_2 is atomic.*

Proof. (=>) Let $h \in H(L_1)$. Then, every execution e_1 with $H(e_1) = h$ is linearizable w.r.t. some execution $e_2 \in L_2$

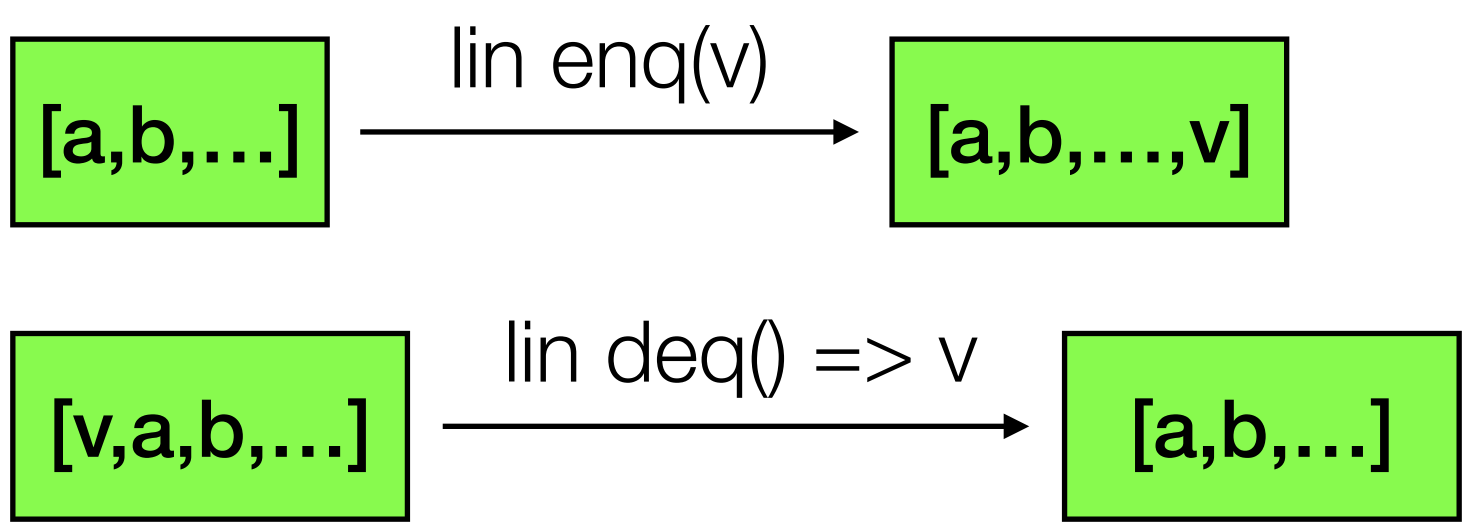
By the lemma above, $H(e_1) \preccurlyeq H(e_2)$. By closure under weakening, if $H(e_2) \in H(L_2)$ then any weakening, h in particular, belongs to $H(L_2)$.

(<=) Let $e_1 \in E(L_1)$. By hypothesis, $H(e_1) \in H(L_2)$, which implies $e_1 \in E(L_2)$. Since L_2 is atomic, there exists a sequential $e_2 \in E(L_2)$ with only completed ops such that $H(e_1) \in H(L_2)$ such that e_1 is lin. w.r.t. e_2.

Linearizability Proofs based on Forward Simulations

Linearizability vs Refinement

- Modelling concurrent objects with Labeled Transition Systems (LTSs)
- Linearizability is a property of sequences of **call/return actions**
- Given an ADT A, define a **reference implementation Spec(A)** which admits all histories linearizable w.r.t. A
  - standard reference implementations (atomic method bodies): call, return, and **linearization point** actions

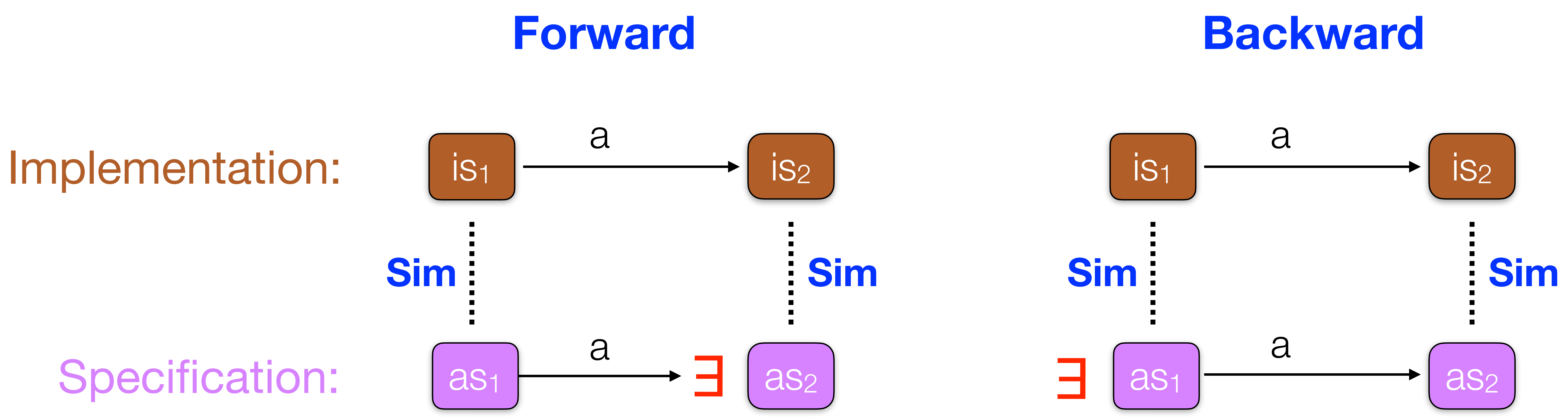


- Linearizability = inclusion of traces with call/return actions (these are the only common actions) between **Impl** and **Spec(A)**
  - the actions included in traces are called **observable**

Proving Refinement

Inductive reasoning for proving **refinement**: forward/backward simulations

Simulations: relations between states of the impl. and spec., relating initial states and



Proving Refinement

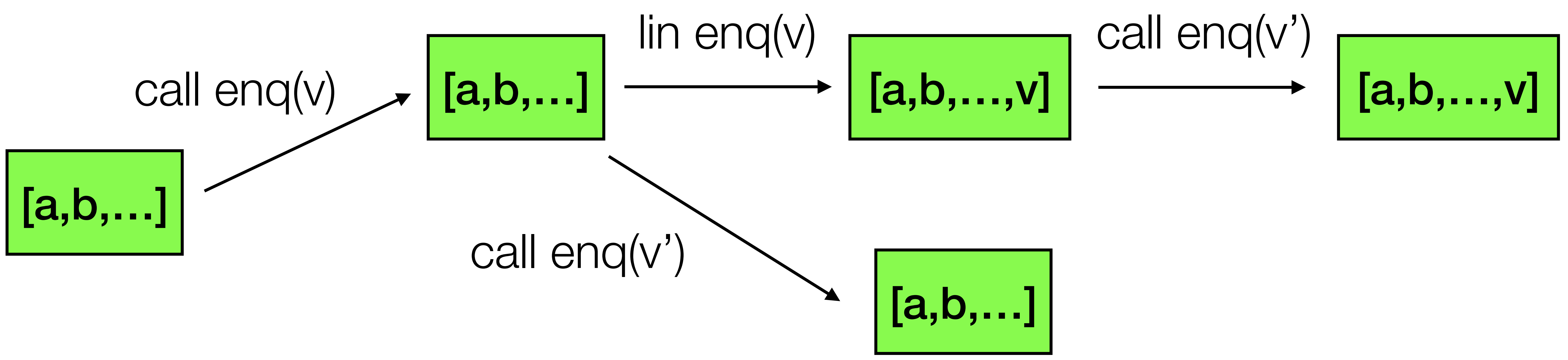
- Given two LTSs A and B such that A refines B [Abadi et al.'91, Lynch et al.'95]

	Frw Sim (FS)	Bckw Sim (BS)
exists if	*B* deterministic	*A* forest
exists if we add	Prophecy vars to *A*	History vars to *A*

- Forward simulations are easier to derive and establish (standard invariant checking)

Proving Linearizability

- **Impl** is **linearizable** w.r.t. A iff **Impl** **refines** **Spec(A)**
    - refinement = inclusion of traces with call/return actions (observable actions)
- **Spec(A)** is **not deterministic** when projected on observable actions => backward simulations are unavoidable in general



- Classes of implementations for which forward simulations are sufficient - associate linearization points with statements of the implementation
    - the linearization point actions become **observable**
    - **Spec(A)** is deterministic assuming that A is **deterministic**

Fixed Linearization Points

- **Fixed** linearization points: the linearization point is fixed to a particular statement in the code

```
class Node {
  Node tl;
  int val;
}
```

```
class NodePtr {
  Node val;
} TOP
```

Treiber Stack

```
void push(int e){
  Node y, n;
  y = new();
  y->val = e;
  while(true) {
    y->tl = n;
    if (cas(TOP->val, n, y))
      break;
  }
}
```

```
int pop(){
  Node y,z;
  while(true) {
    y = TOP->val;
    if (y==0) return EMPTY;
    z = y->tl;
    if (cas(TOP->val, y, z))
      break;
  }
  return y->val;
}
```

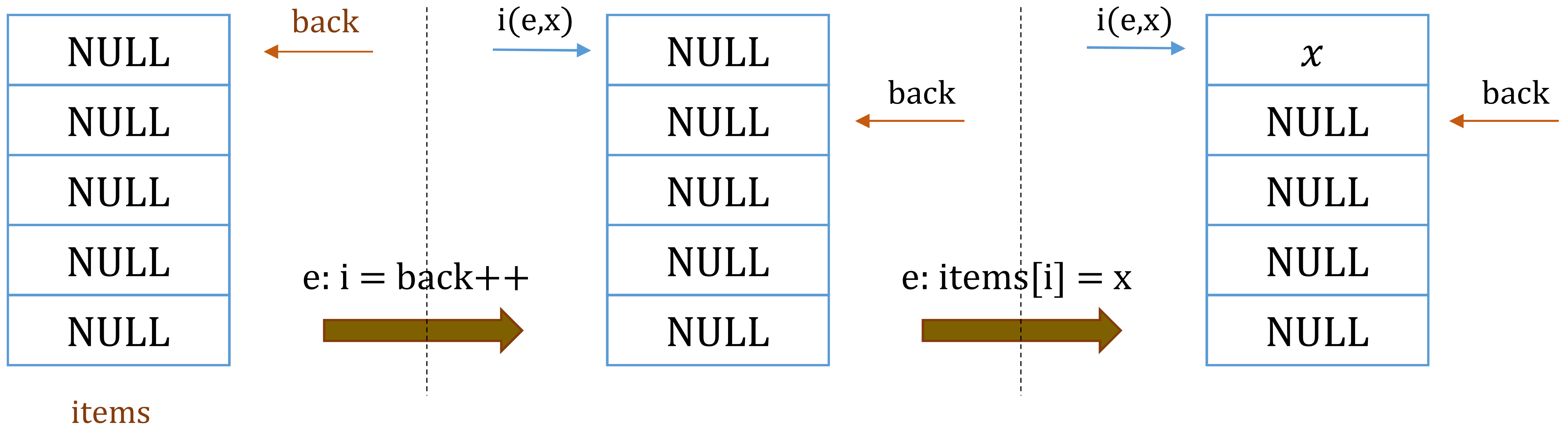
Herlihy & Wing Queue

```
void enq(int x) {
  i = back++; items[i] = x;
}
int deq() {
  while (1) {
    range = back - 1;
    for (int i = 0; i <= range; i++) {
      x = swap(items[i],null);
      if (x != null) return x;
} } }
```

Non-fixed Linearization Points

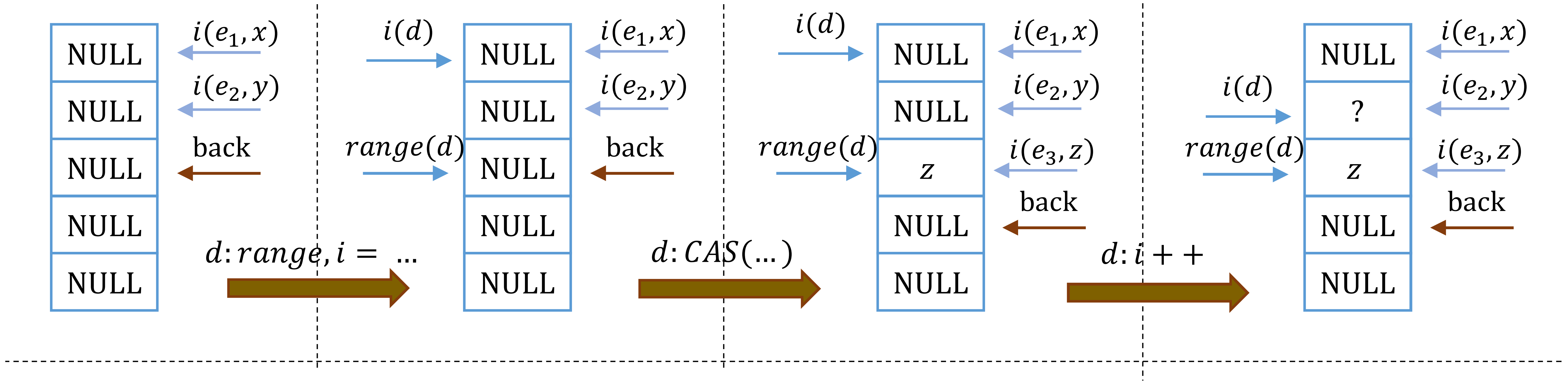
Herlihy & Wing Queue

Enqueue

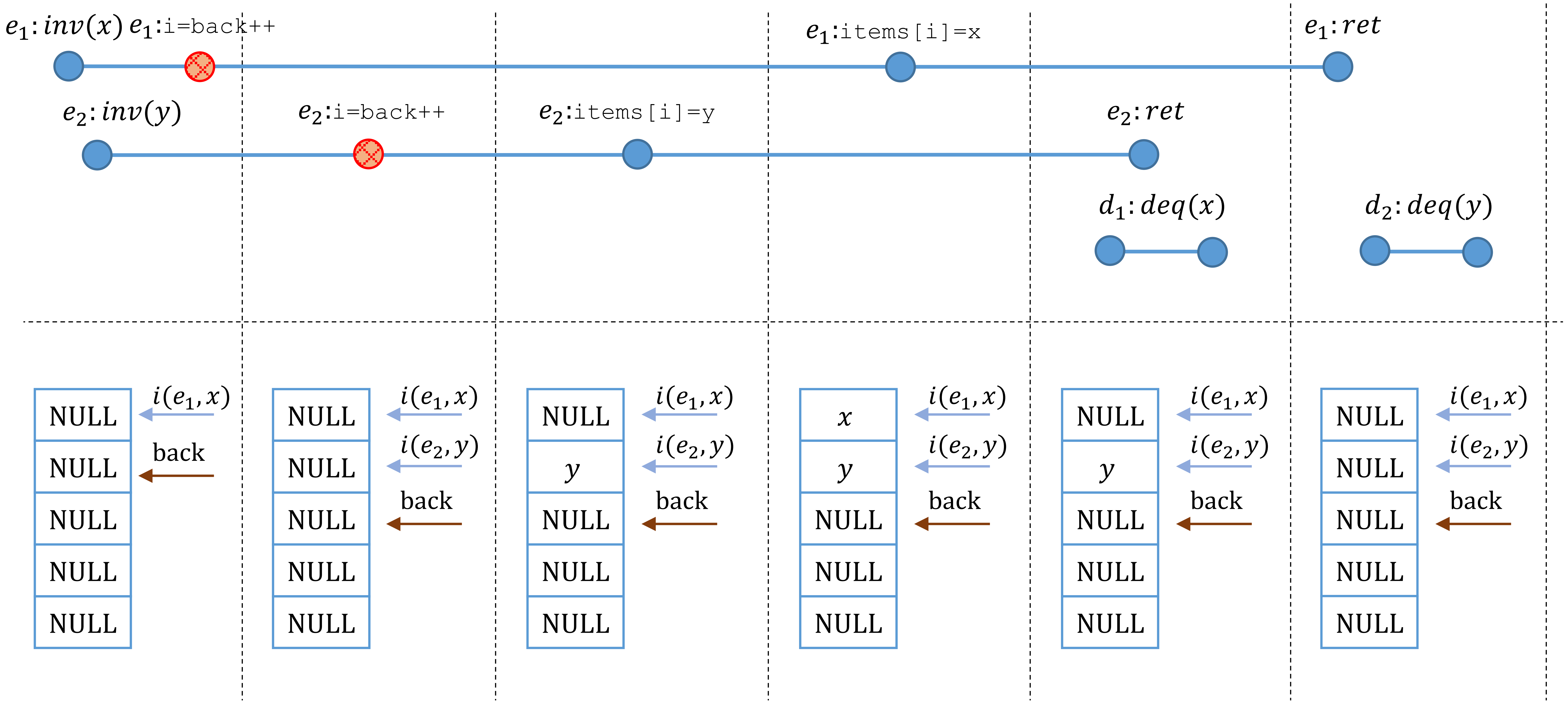


$i(e,x)$: index i of enqueue with id e that will insert item x

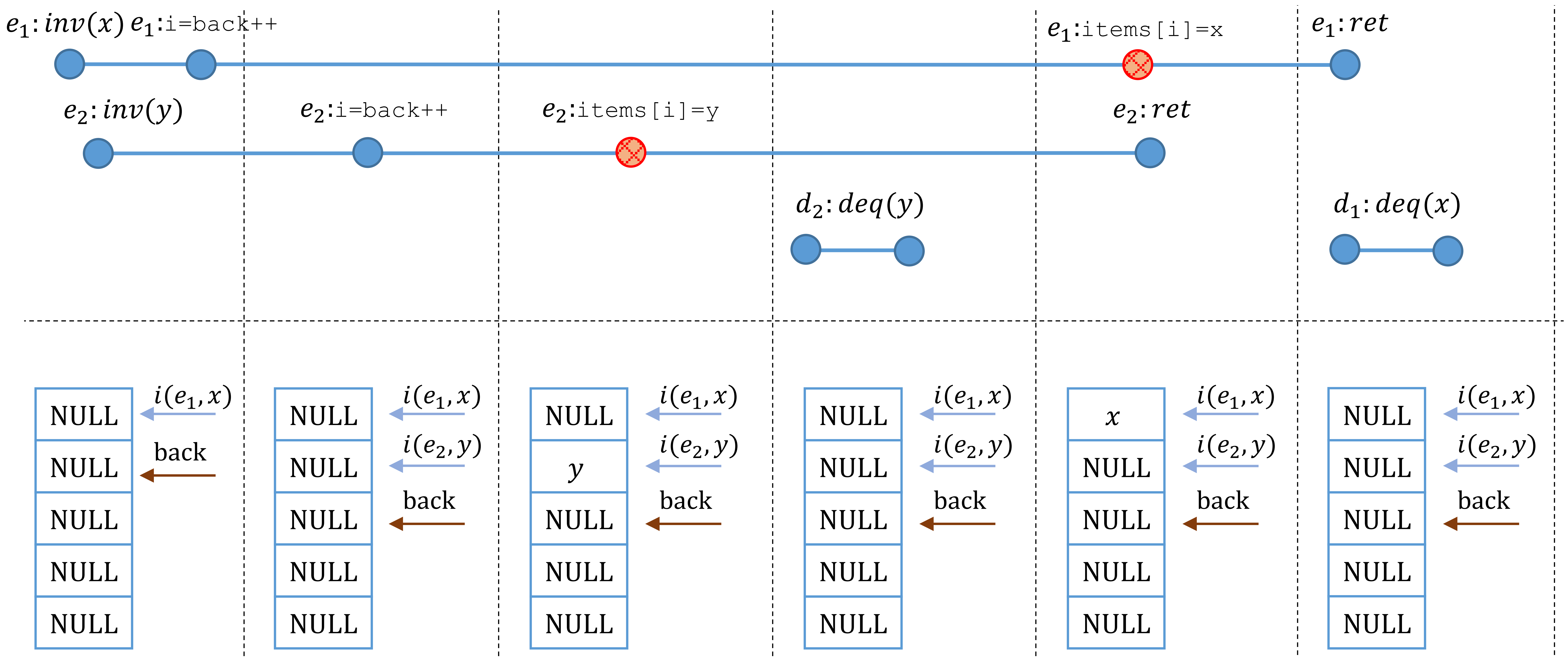
Dequeue



Non-fixed Linearization Points



Non-fixed Linearization Points



Snapshot

```
procedure update(i,data)
  mem[i] = data;

procedure scan()
  for i = 1 to n do r1[i] = mem[i];
  repeat
    r2 = r1;
    for i = 1 to n do r1[i] = mem[i];
  until r1 == r2
  return r1;
```

Fixed linearization points ?

Snapshot

```
1 procedure update(i,data)
2   mem[i] = data;

4 procedure scan()
5   for i = 1 to n do r1[i] = mem[i];
6   repeat
7     r2 = r1;
8     for i = 1 to n do r1[i] = mem[i];
9   until r1 == r2
10  return r1;
```

Specification

```
procedure update(i,data)
  mem[i] = data;

procedure scan()
  while ( nondet )
    r = atomic_snapshot();
    snaps = snaps · r;
  return r1 ∈ snaps;
```

(s,s') ∈ F iff they contain the same **mem** and for each invocation k, its local state s[k] (valuation of r1, r2, and prog. counter pc) is related to s'[k] (a valuation of snaps) as follows:

$$\bigwedge_{\mathtt{r}\in\{\mathtt{r1},\mathtt{r2}\}} \mathsf{valid}(\mathtt{r},\mathtt{pc}) \wedge \mathsf{fst}(\mathtt{r2},\mathtt{n}) \leq \mathsf{fst}(\mathtt{r1},0) \wedge \mathsf{last}(\mathtt{snaps}) = \mathtt{mem} \wedge (\mathtt{pc} = 10 \implies \mathtt{r1} \in \mathtt{snaps})$$

$$\mathsf{valid}(\mathtt{r},\mathtt{pc}) ::= \forall i,j.\ i < j \implies \mathsf{fst}(\mathtt{r},i) \leq \mathsf{fst}(\mathtt{r},j)$$

fst(v,i) = smallest index of a snapshot in snaps which contains v at index i

fst(r1,i) = ∞ if index i not set, or fst(r1[i],i) otherwise
fst(r2,i) = -∞ if index i not set, or fst(r2[i],i) otherwise

update in impl. => update in spec. +
every pending scan takes a snapshot

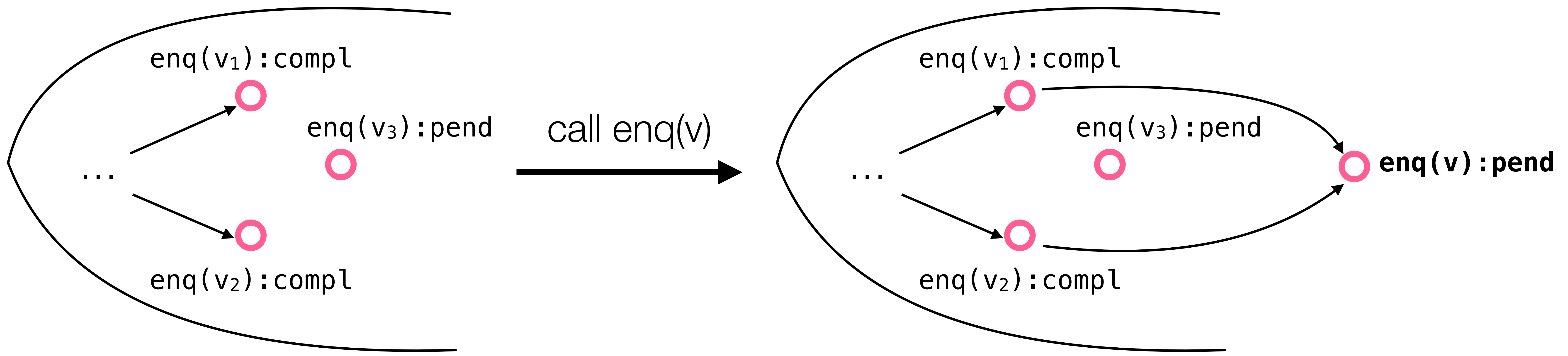
scan steps in impl. => "epsilon" steps

Non-fixed Linearization Points

Non-fixed linearization points => proofs based on **forward simulations** are impossible in general

Possible for certain ADTs, **queues and stacks** [BE**E**M-CAV'17]

- assuming **fixed** linearization points only for **dequeue/pop**
- reference implementations whose states are **partial orders** of enq/push



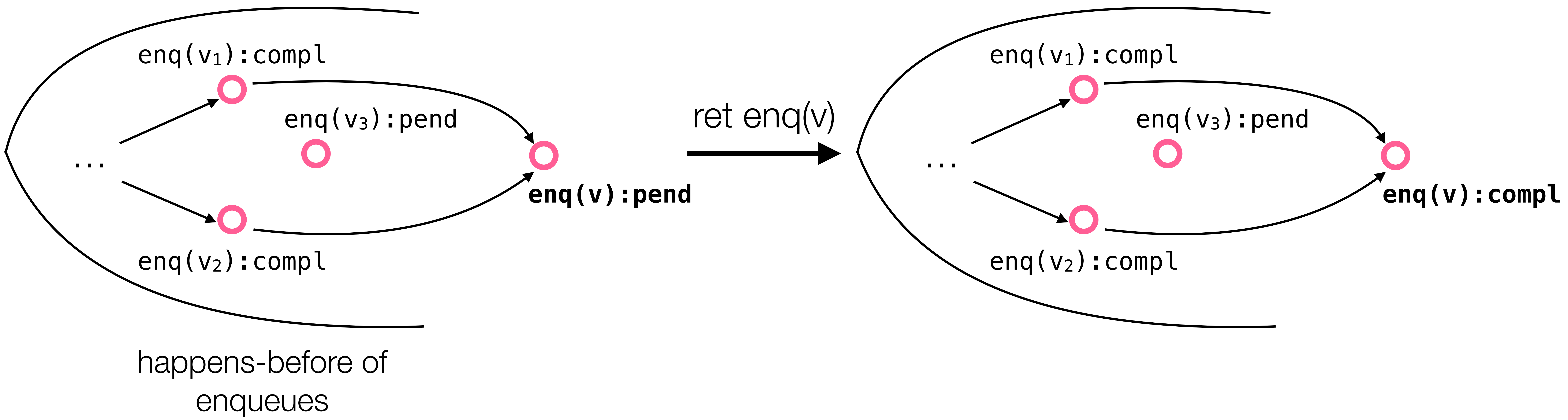
happens-before of enqueues

Non-fixed Linearization Points

Non-fixed linearization points => proofs based on **forward simulations** are impossible in general

Possible for certain ADTs, **queues and stacks** [BE**E**M-CAV'17]

- assuming **fixed** linearization points only for **dequeue/pop**
- reference implementations whose states are **partial orders** of enq/push

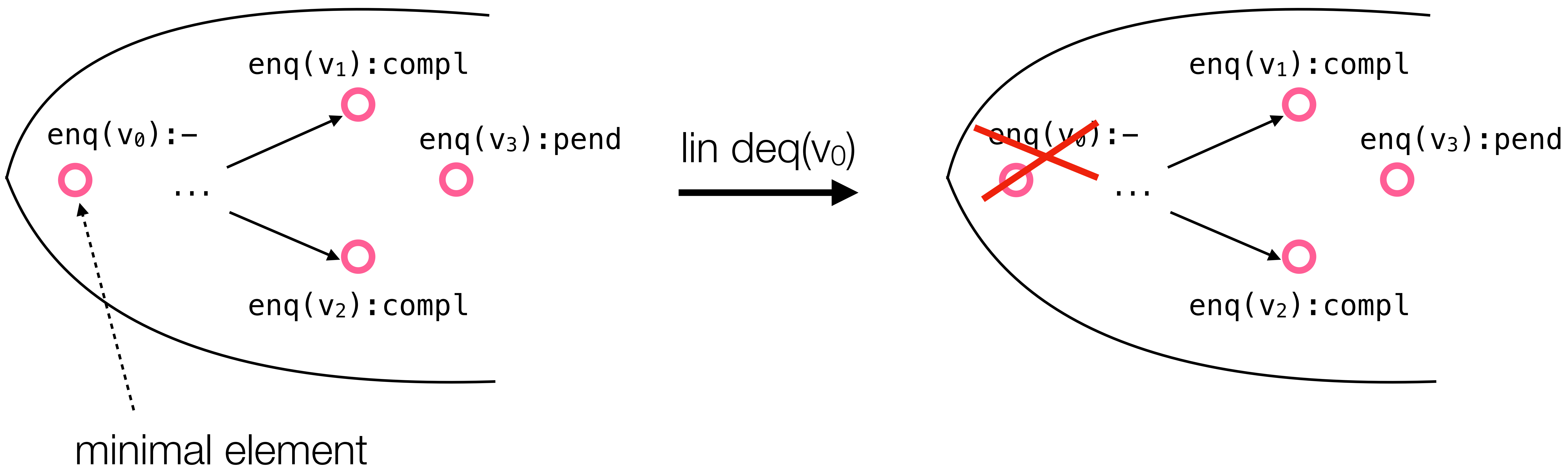


Non-fixed Linearization Points

Non-fixed linearization points => proofs based on **forward simulations** are impossible in general

Possible for certain ADTs, **queues and stacks** [BE**E**M-CAV'17]

- assuming **fixed** linearization points only for **dequeue/pop**
- reference implementations whose states are **partial orders** of enq/push



Forward Sim. for H&W Queue

FS f between HWQ and *AbsQ*. Given a HWQ state s and an *AbsQ* state t, $(s, t) \in f$ iff:

- Pending enqueues in s are pending and maximal in t.
- Order in t is consistent with the positions reserved in `items` of s.
- For two enqueues e_1, e_2 and dequeue d, if e_1 reserves a position before e_2, d is visiting an index in between and d can remove e_2 in s, then e_1 cannot be ordered before e_2 in t.