RPCs & Linearizability

RPCs

- Why?
  - The most common way that processes in implemented distributed systems interact
    - ↳ Convenient way to define
      - (a) The structure of messages
        - CONTENTS
        - ENCODING
      - (b) HIDE NETWORKING DETAILS

- Some of the papers we read describe protocols in terms of RPC calls & responses

NOTE: The model in the paper & what we talk about closely corresponds to what GRPC, THRIFT, etc. provide

```
on recv (:echo, m)
    from s :

    send s, m
```

⟷

```
msg echo(msg m){
    return m;
}
```

```
send a, (:echo, "hello")

recv "hello" from a
```

```
ret = a.echo("hello")
assert(ret == "hello")
```

SEEMS SIMPLE ENOUGH, BUT THERE ARE DIFFERENCES

① RE-ENTRANCY/ATOMICITY

alice add_and_ret(5)

```
on recv n from y:
    a = a + n
    b = 2*a - 1
    send(z, (:up_a, a))
    send(z, (:up_b, b))

    send(y, a+b)
```

```
int add_and_ret(int n){
    a = a + n;
    b = 2*a - 1;
    z.up_a(a);
    z.up_b(b);
    return a+b;
}
```

both add_and_ret(1)

Most of the protocols we encounter that are expressed in messages

– Assume 1 message processed at

– Assume 1 message ...
a time

↳ Assume all state updates & accesses happen atomically.

RPC: Making a call ⇒

Caller must wait (BLOCK) for response

Runtime (usually) can process other calls while blocked.

↳ Handling other RPC calls was mandated by Birrell & Nelson

```
X
int rA(int) {
    ...
    int z = Y.rB(...);
    ...
}

Y
int rB(...) {
    ...
    int h = X.rC(...);
    ...
}

int rC(...) {
    ...
```

Allowing concurrent RPC calls helps with

}

local reasoning

↳ Do not need to consider implementation of function being called.

② Effect Of Remote Failures

```
X
int rA(int) {
    ...
    int z = Y.rB(...);   ①
    ...
}
```

```
Y                        ②
int rB(...) {
    ...
    int h = X.rC(...);
    ...
}
```

Y crashes before returning to X
responding

Message model: X never receives from Y

- on recv... logic never executed

RPCs: Implementation dependent but

↳ Call will timeout &

fail

→ Return value indicates failure
(of call not process)

→ Exception thrown

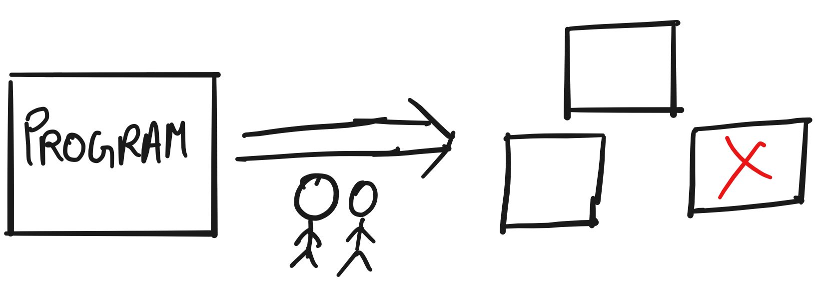
→ ...

Will show up in papers as
RPC request fails OR
No RESPONSE IS RECEIVED

LINEARIZABILITY

Motivation/Goal:

One reason to build Distributed Systems
is to tolerate faults



Dist. Program

Q: How does program behavior ... to dist. program behavior?

Differences

- Concurrency
- Failures

Linearizability: Consistency model

Given ADT: Queue
Stack
...

How specified: Sequential specification

Queue: FIFO container

enq(5)

deq() → 5

enq(6)

enq(7)

enq(8)

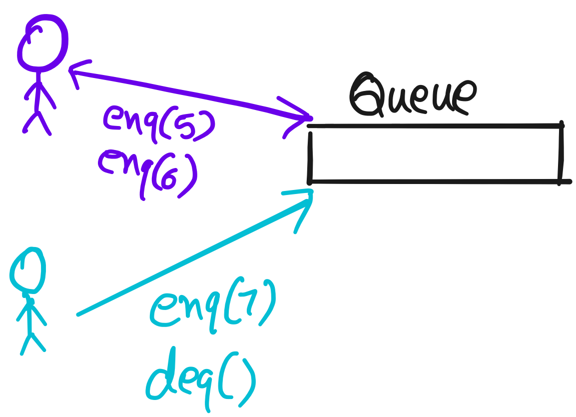
deq() → 6

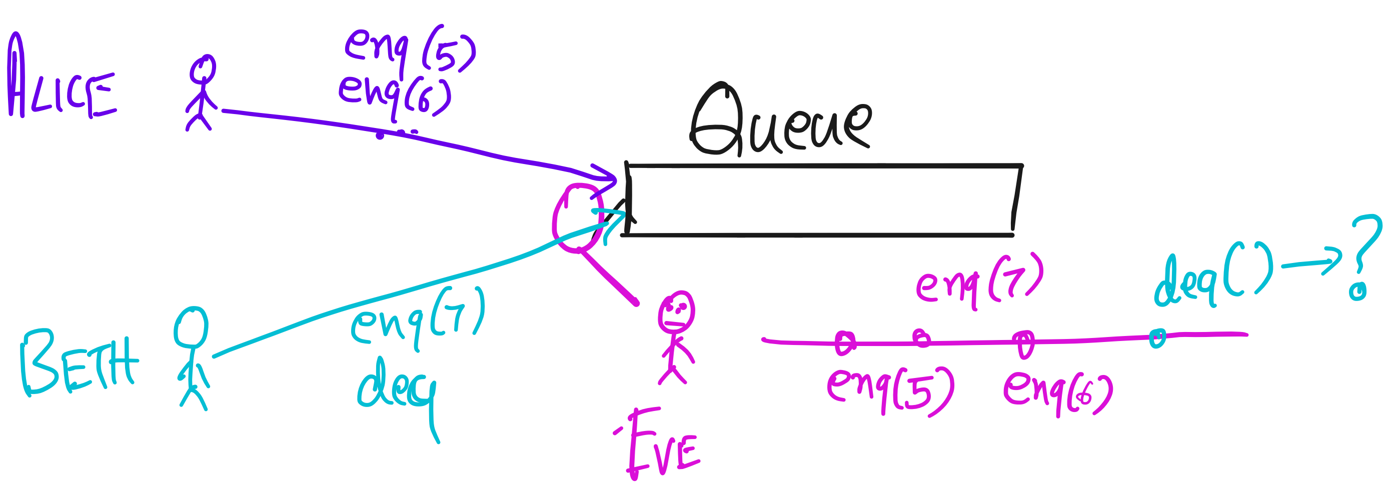
deq() → 7

Stack: LIFO

push(1)
push(2)
pop()
. . .

Q: How does a concurrent queue or stack behave?





Many possibilities

(a) Eve's observed order

(b) Alice - Beth - Alice - Beth . . . .

(c) LINEARIZABILITY

(d) SEQ CST

(e) CAUSAL CONSISTENCY

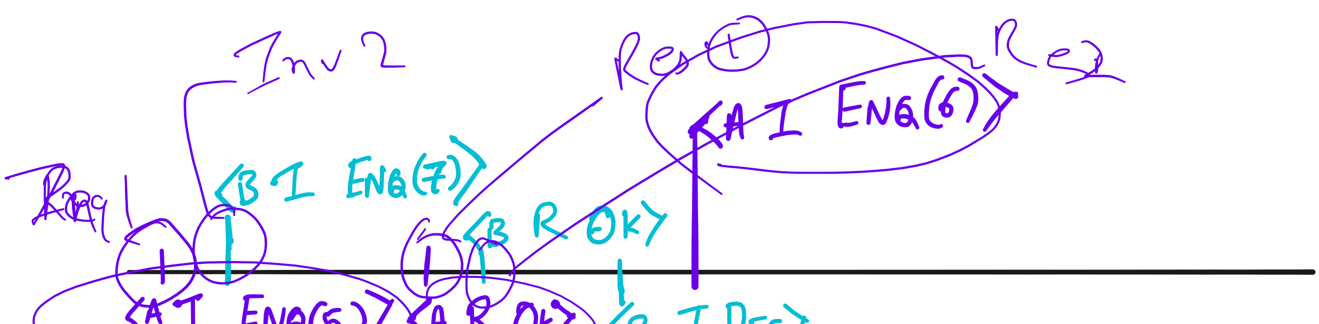
. . .

Defining Linearizability

(1) ALICE ENQ(5) ⟶ <ALICE Inv ENQ(5)>
<ALICE Res OK()>

(2) History :- Sequence of inv, res interactions

Q. From whose perspective?



$<_H$: History partial order

$$C_1 <_H C_2 \Rightarrow \text{Res } C_1 < \text{Inv } C_2$$

[A Enq(5)] 1

2 [B Enq(7)]

[A Enq(6)]

[B Deq()]

Some other useful definitions

H equivalent to H':

$$\Rightarrow H|p = H'|p \quad \forall \; p \in \Pi$$

Complete(H) → Maximal subsequence where every request has a response.

H sequential ⇒ First event is invocation

No concurrent requests ($<_H$ is total order)

Assume $H|p$ sequential $\forall\ p \in \Pi$

Defn

History H is linearizable if

H can be EXTENDED to H' s.t

1 $\exists$ Sequential history S, where

S is equivalent to complete(H')

; 2 $<_H \subseteq <_S$ & 3 S valid for ADT Seq. spec

Note, 2 is wrong: Sela, Herlihy, Petrank PODC '21 [Reading for next week]

Should be

$$<_{\text{complete}(H')} \subseteq <_S$$

PROPERTIES

① Locality: $O_1, O_2, \ldots O_n$ objects

Linearizable $H \iff$

$H|O_1, H|O_2, \ldots H|O_n$ linearizable

② Non-Blocking.

S ⟸ H linearizable, contains invocation $\langle A \text{ Inv } Deq() \rangle$

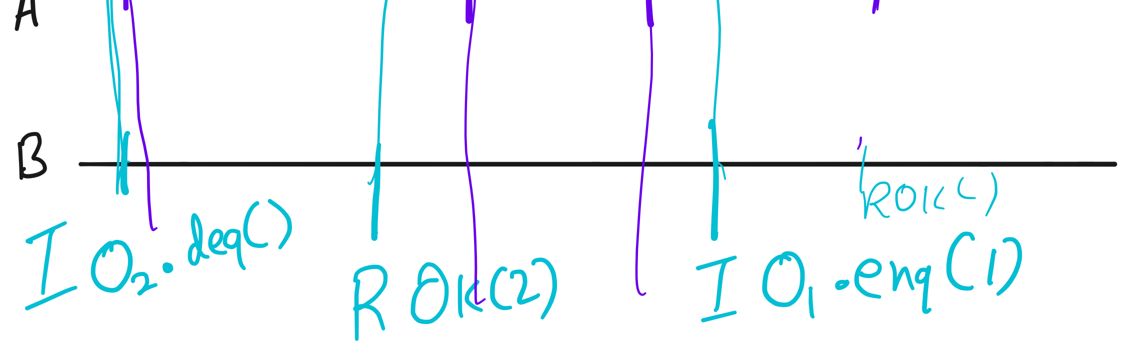
$\exists$ response R s.t.

$H \cdot \langle A \text{ Inv } Deq() \rangle \langle A \text{ Res } R \rangle$

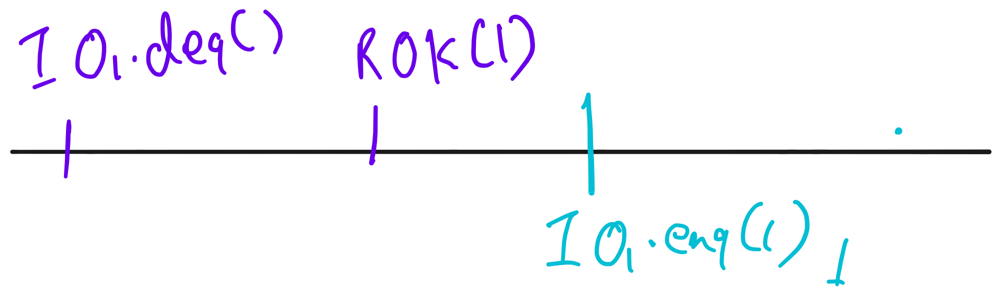
is linearizable.

Instructive to see examples where this is not true

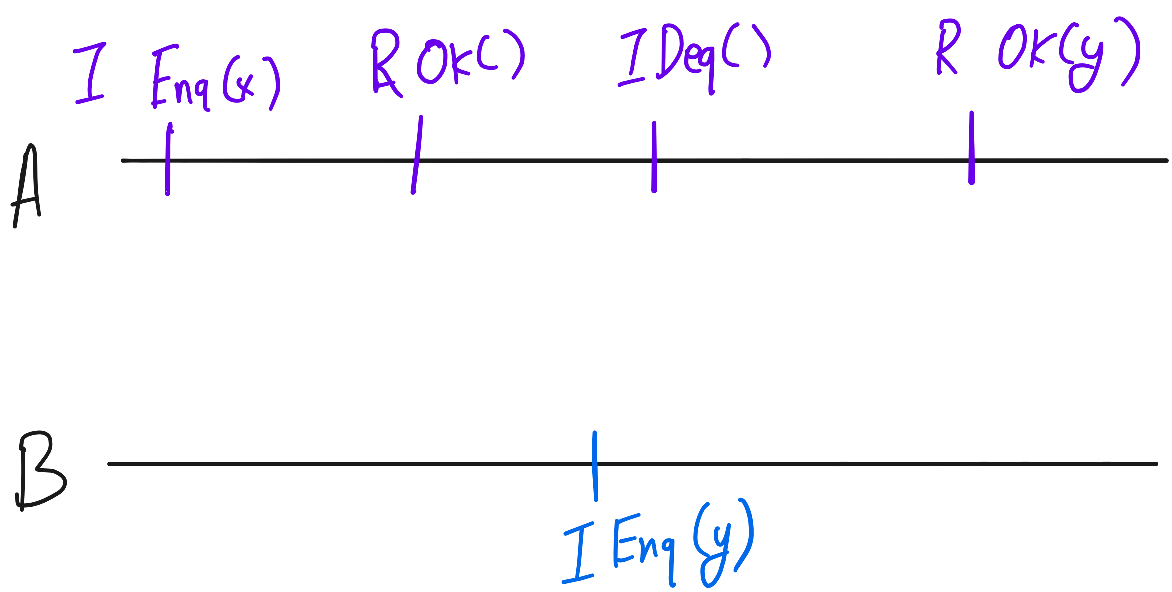
I O_1.deq() R OK(1) I O_2.enq(2) R OK

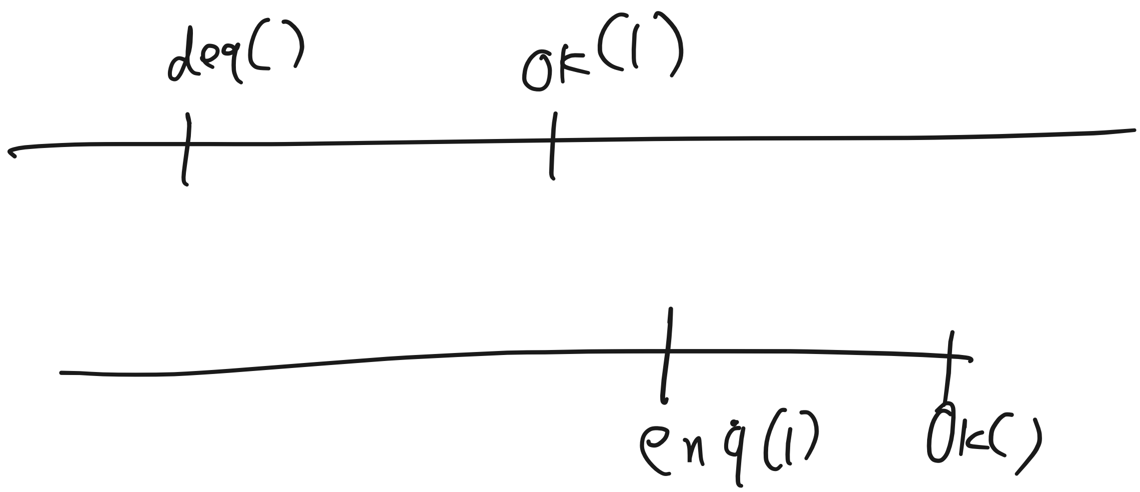

A
B
I O₂.deq()
R Ok(2)
I O₁.enq(1)
R Ok()

$H|O_1$

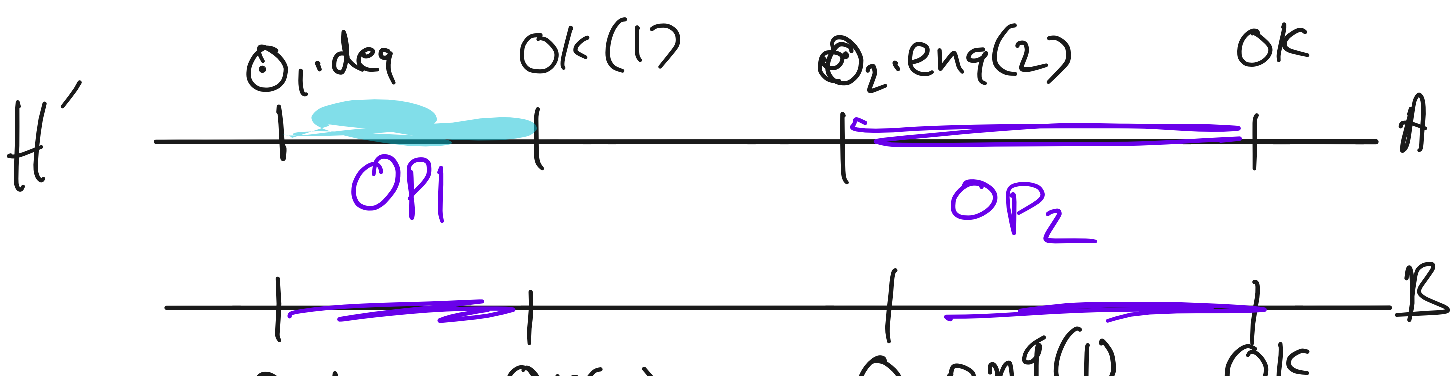

I O₁.deq()
R Ok(1)
I O₁.enq(1)

$H|O_2$


I Enq(x)
R Ok()
I Deq()
R Ok(y)
A
B
I Enq(y)



deq()
ok(1)
enq(1)
Ok()



H'
O1.deq
Ok(1)
O2.enq(2)
OK
A
OP1
OP2
B

O_2.deq   OK(2) $O_?$.enq(1)

OP3   OP4

Complete(H') = H'

$S|A = H'|A$

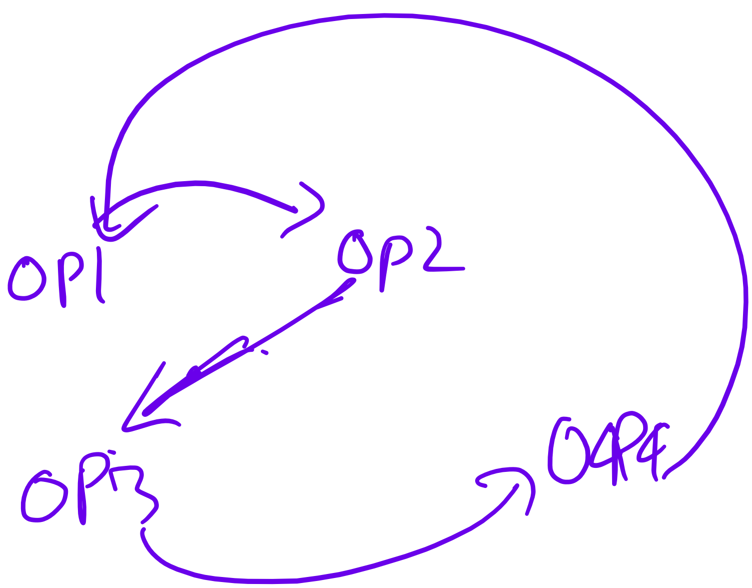
$S|B = H'|B$

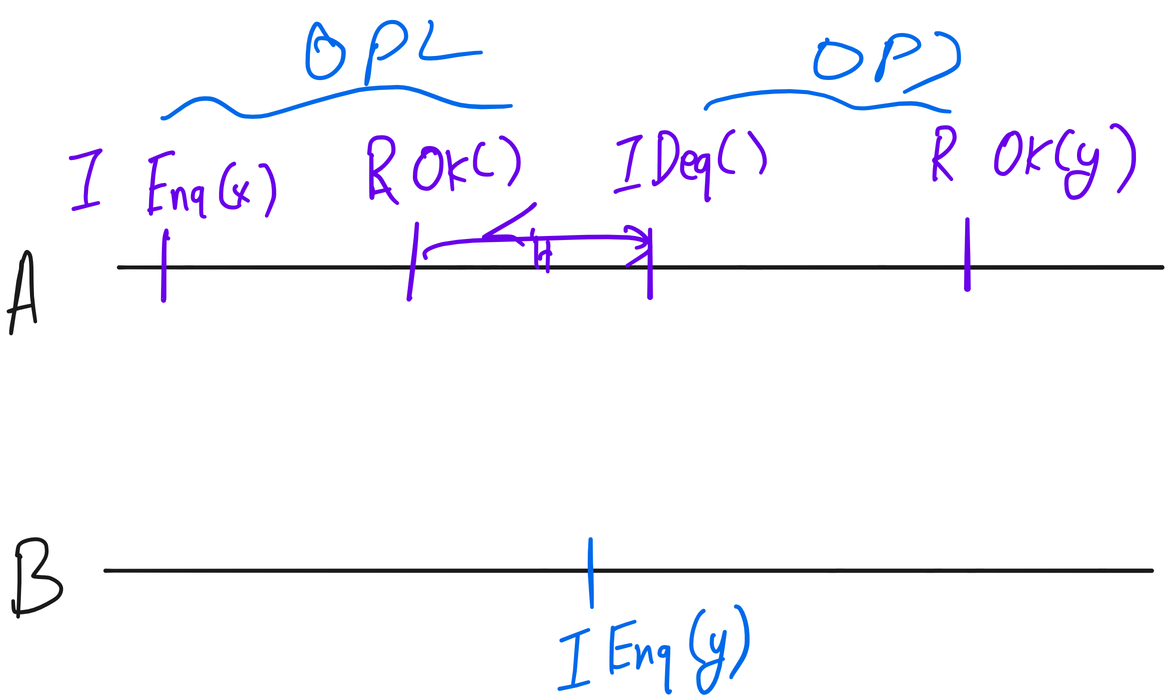
OP1 < OP2

OP3 < OP4

OP2 < OP3

OP4 < OP1



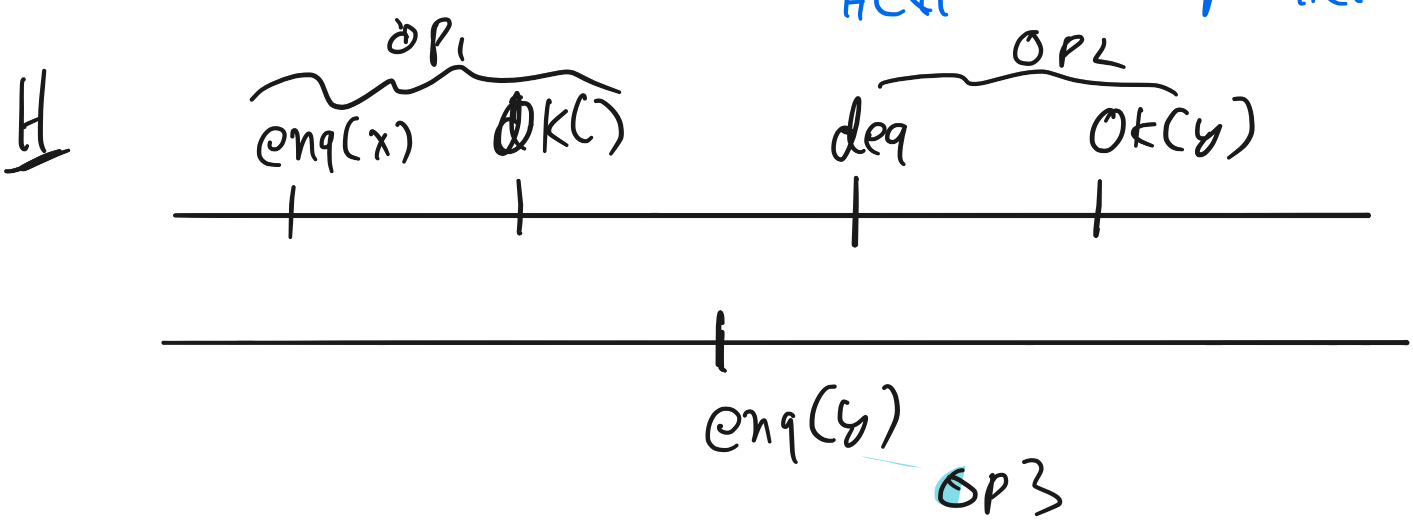


Op1

Op1   Op2   Op3

$$H_{ext} = H \cdot \langle R \ ok() \rangle$$

$<_{Hext}$ $Op1 <_{Hext} Op2$



$$<_H : \{(Op_1, Op_2)\}$$

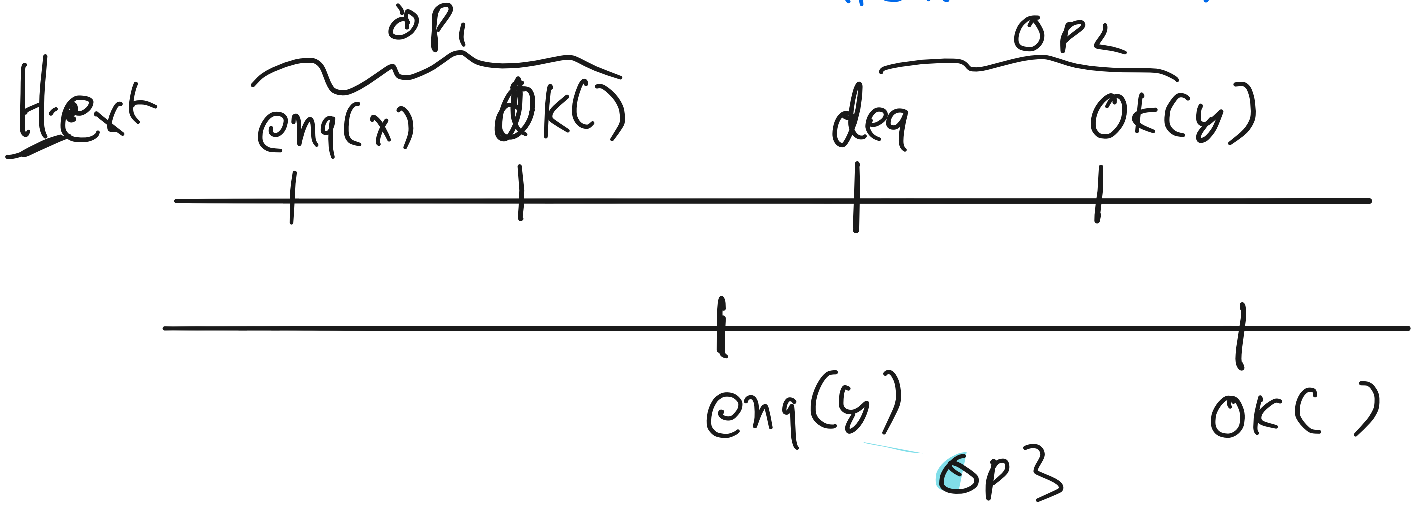
H buggy-line $H \to H'$

$$H \cdot \{ok()\}$$

Complete(H') = H

$$S = \{Op3, Op_1, Op_2)$$

Hext



$$<_{Hext} = \{ (op1, op_2), (op1, op3) \}$$

S