

interpreters everywhere!

Lindsey Kuper @ UC Santa Cruz

Northeastern Software Seminar • January 22, 2026

featuring the work of some amazing students:

Gan Shen, Shun Kashiwa,  
Jonathan Castello, Patrick Redmond

*Dan Friedman*



↙ 2008-2014

my PhD: **shared-memory** deterministic concurrency  
via **monotonic data structures**

↙ 2008-2014

my PhD: **shared-memory** deterministic concurrency  
via **monotonic data structures**

Hey, wait a second...

Distributed systems are **fascinating!**



CRDT verification OOPSLA '20

Causal broadcast protocol verification IFL'22

Library-level choreographic programming ICFP '23, PLDI '25

Causal separation diagrams OOPSLA '24

CRDT emulation ICFP '25

Hey, WAIT a second...

it was all about interpreters the whole time!





What is an interpreter?

What is an interpreter?

What is an interpreter?



What is an interpreter?

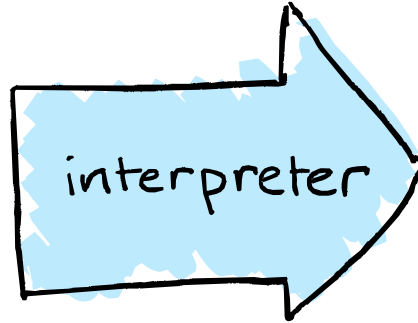


What is an interpreter?



What is an interpreter?

SYNTAX



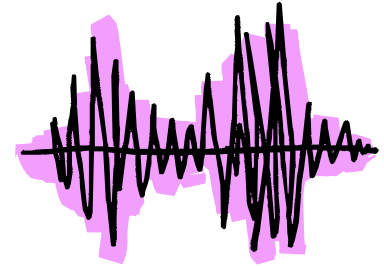
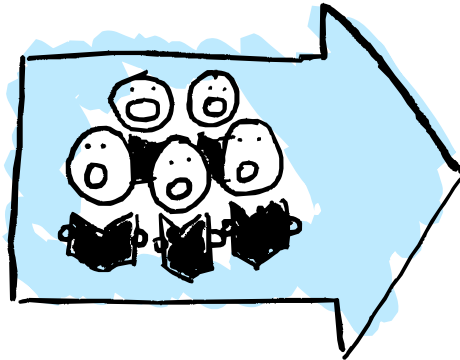
SEMANTICS

# What is an interpreter?

**SYNTAX**

interpreter

**SEMANTICS**

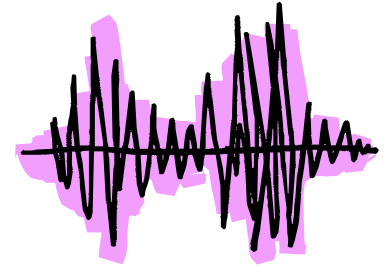
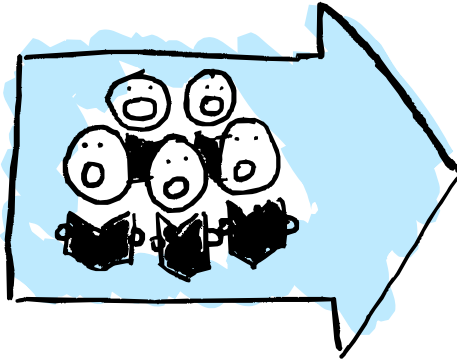


# What is an interpreter?

**SYNTAX**

interpreter

**SEMANTICS**



$\lambda$

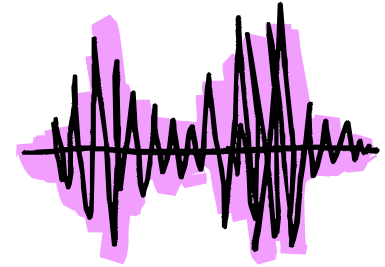
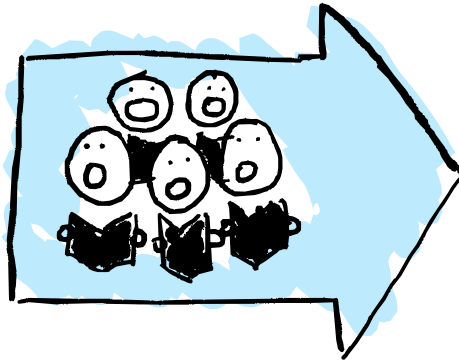
"Hello, world!"

# What is an interpreter?

**SYNTAX**

interpreter

**SEMANTICS**



$\lambda$

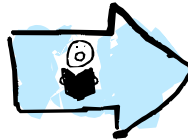
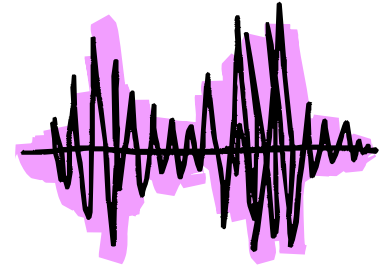
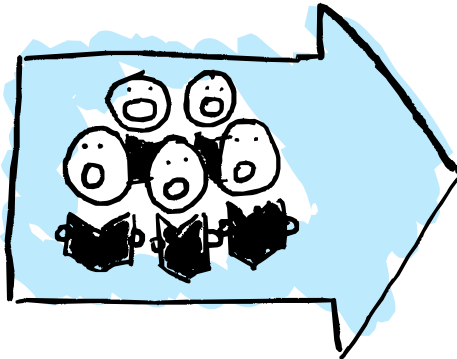


# What is an interpreter?

**SYNTAX**

interpreter

**SEMANTICS**



Hey, wait a second...

Distributed systems are fascinating!



CRDT verification OOPSLA '20

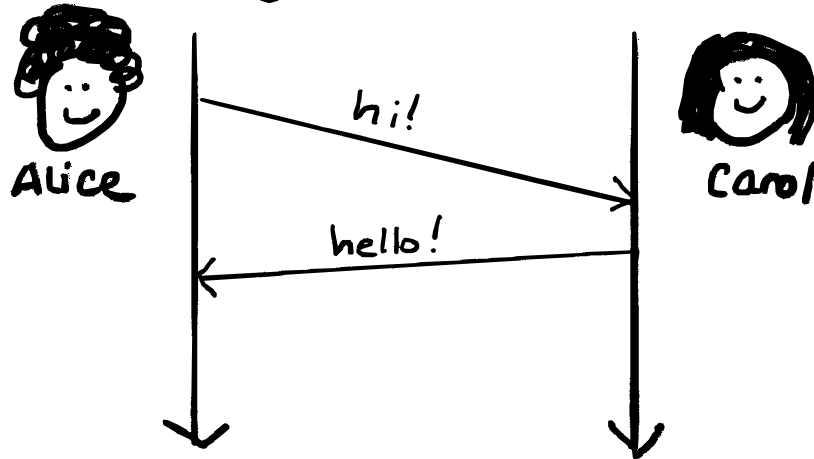
Causal broadcast protocol verification IFL '22

Library-level choreographic programming ICFP '23, PLDI '25

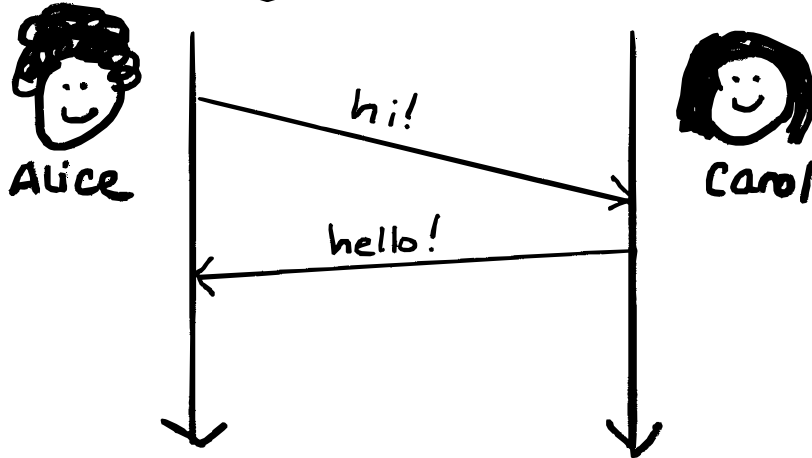
Causal separation diagrams OOPSLA '24

CRDT emulation ICFP '25

How to accomplish something **global**  
by taking only **local** actions?



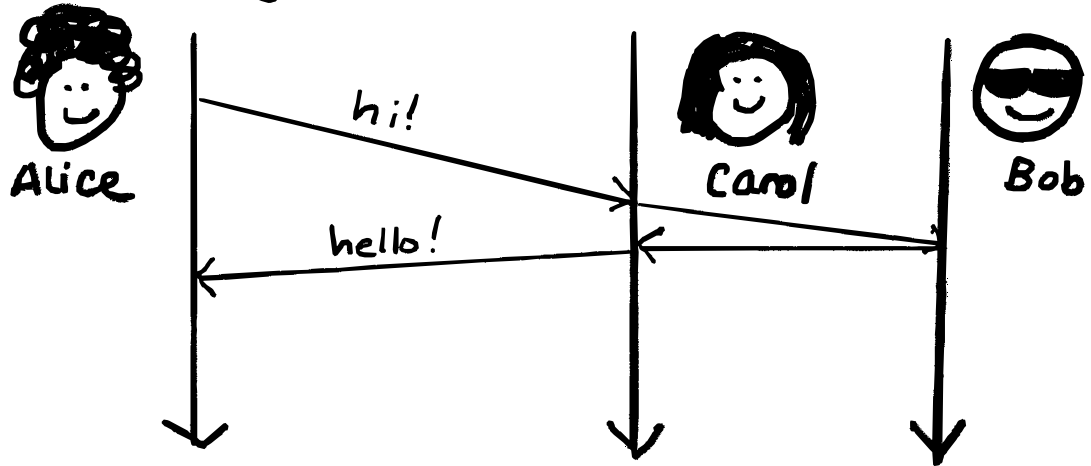
How to accomplish something **global**  
by taking only **local** actions?



```
send(request, Carol)
y = recv(carol)
```

```
request = recv(Alice)
response = handle(request)
send(response, Alice)
```

How to accomplish something **global**  
by taking only **local** actions?



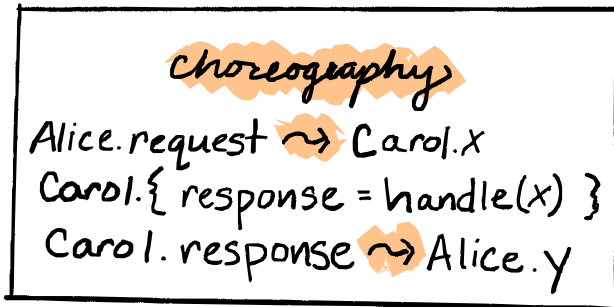
"A **distributed system** is a system in which the failure of a computer that you didn't even know existed can render your own computer unusable."

- Leslie Lamport

How to accomplish something **global**  
by taking only **local** actions?

**Choreographic programming's** answer:

Make the implicit **global** behavior  
explicit in the code!



ENDPOINT PROJECTION



Alice

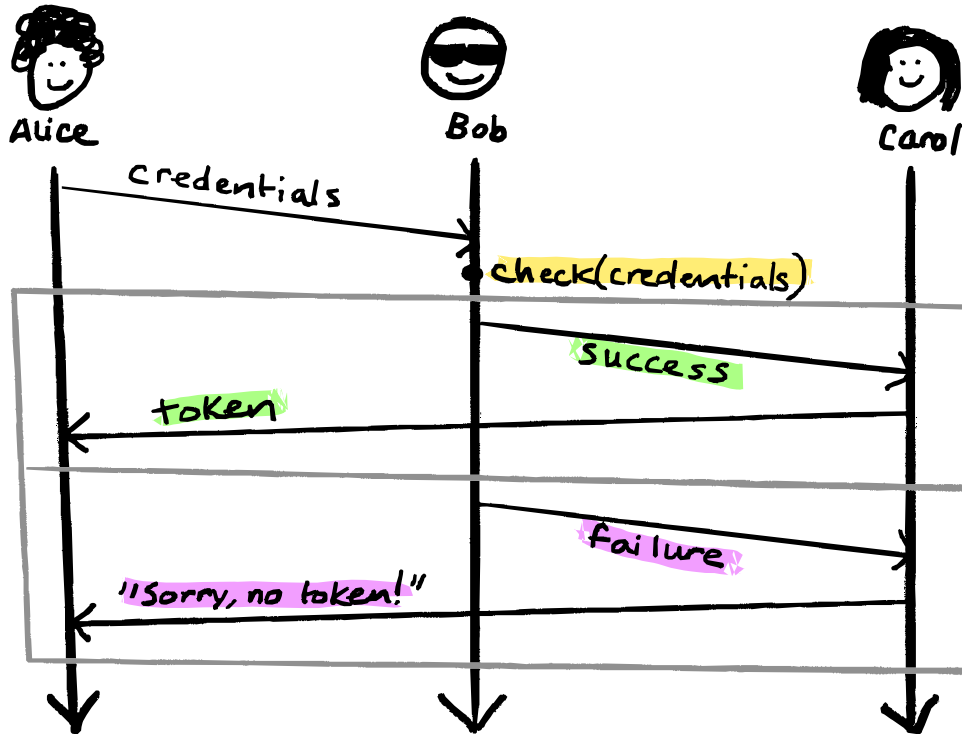
```
Send(request, Carol)  
y = recv(Carol)
```



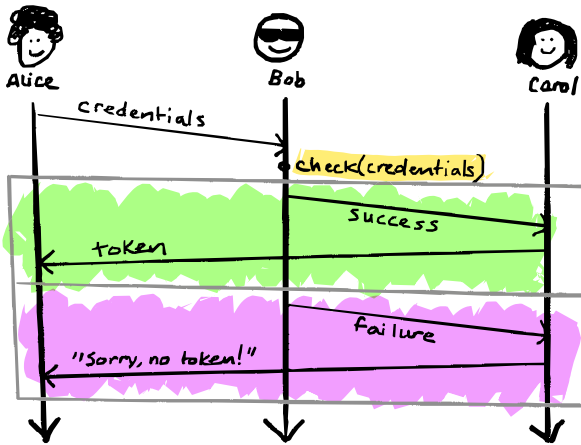
Carol

```
request = recv(Alice)  
response = handle(request)  
send(response, Alice)
```

# Choreographies with **Conditionals**



# Choreographies with conditionals



Alice. credentials  $\rightarrow$  Bob. authRequest  
if Bob. check(authRequest) then  
Bob. Success  $\rightarrow$  Carol. decision  
Carol. newToken()  $\rightarrow$  Alice. result  
else  
Bob. Failure  $\rightarrow$  Carol. decision  
Carol. noTokenMessage  $\rightarrow$  Alice. result

# Choreographies with conditionals

## Choreography

```
Alice.credentials → Bob.authRequest
if Bob.check(authRequest) then
  Bob.Success → Carol.decision
  Carol.newToken() → Alice.result
else
  Bob.Failure → Carol.decision
  Carol.noTokenMessage → Alice.result
```

ENDPOINT PROJECTION

```
send(credentials, Bob)
result = recv(Carol)
```



Alice

```
authRequest = recv(Alice)
if check(authRequest) then:
  send(Success, Carol)
else:
  send(Failure, Carol)
```



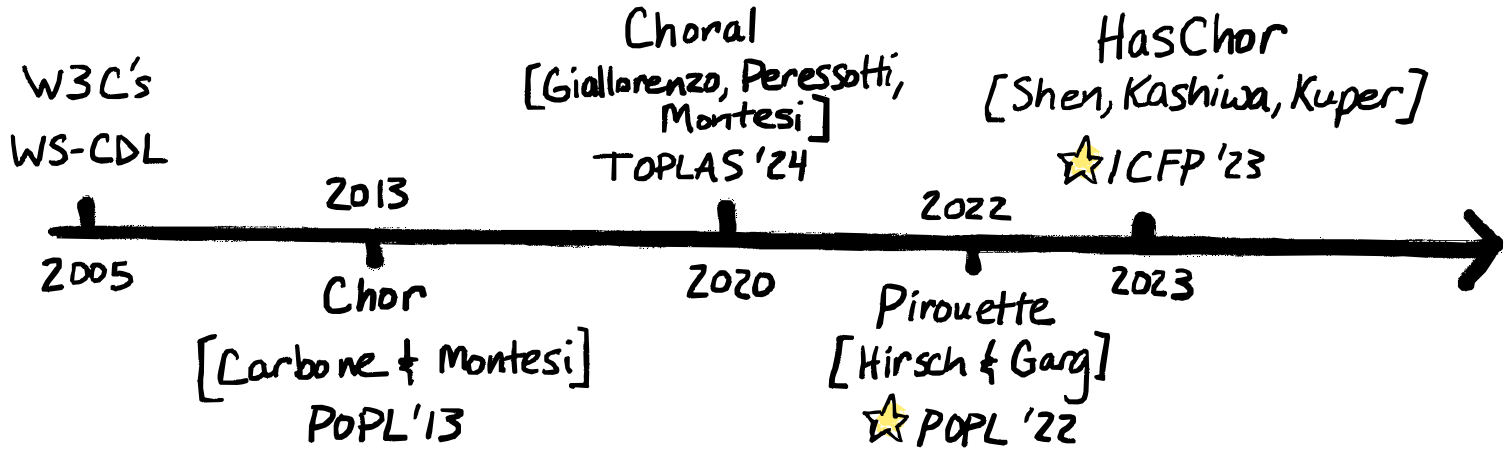
Bob

```
decision = recv(Bob)
switch (decision)
case Success:
  send(newToken(), Alice)
case Failure:
  send("Sorry, no token!", Alice)
```



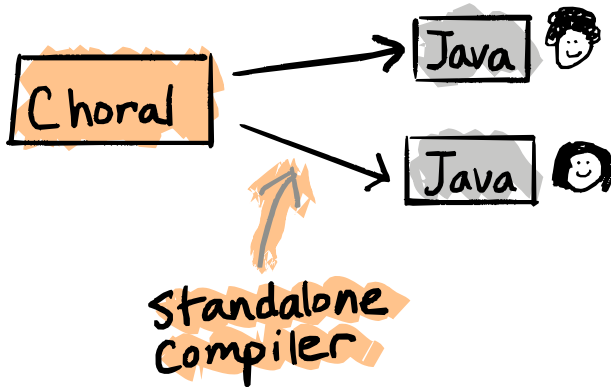
Carol

# 20 years of choreographies



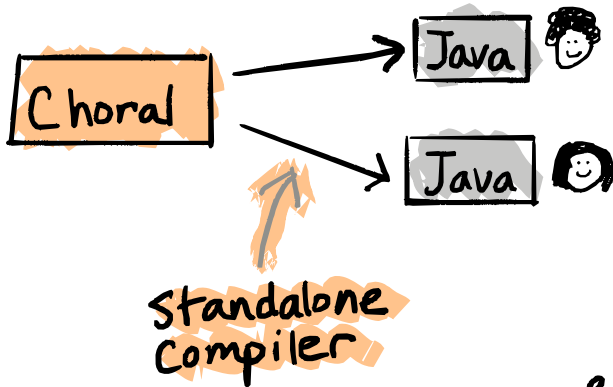
# HasChor's contribution: library-level choreographic programming

Traditional CP:

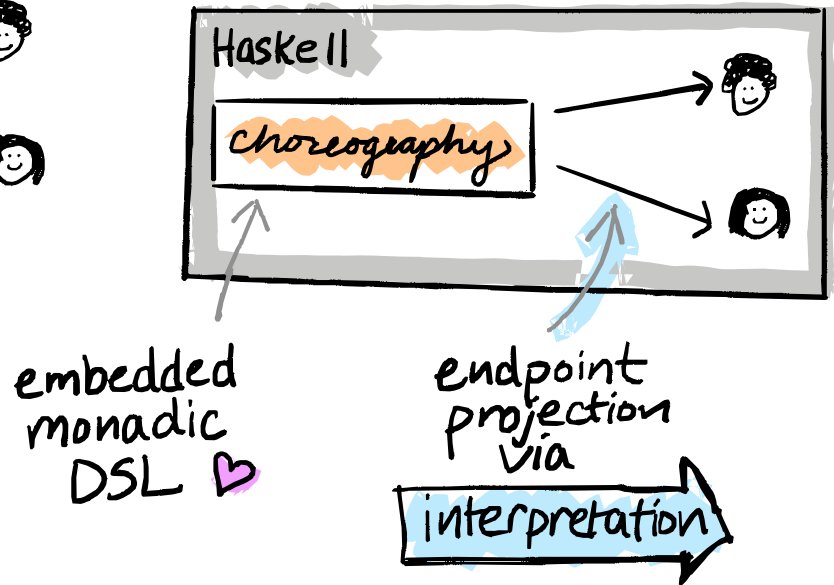


# HasChor's contribution: library-level choreographic programming

Traditional CP:



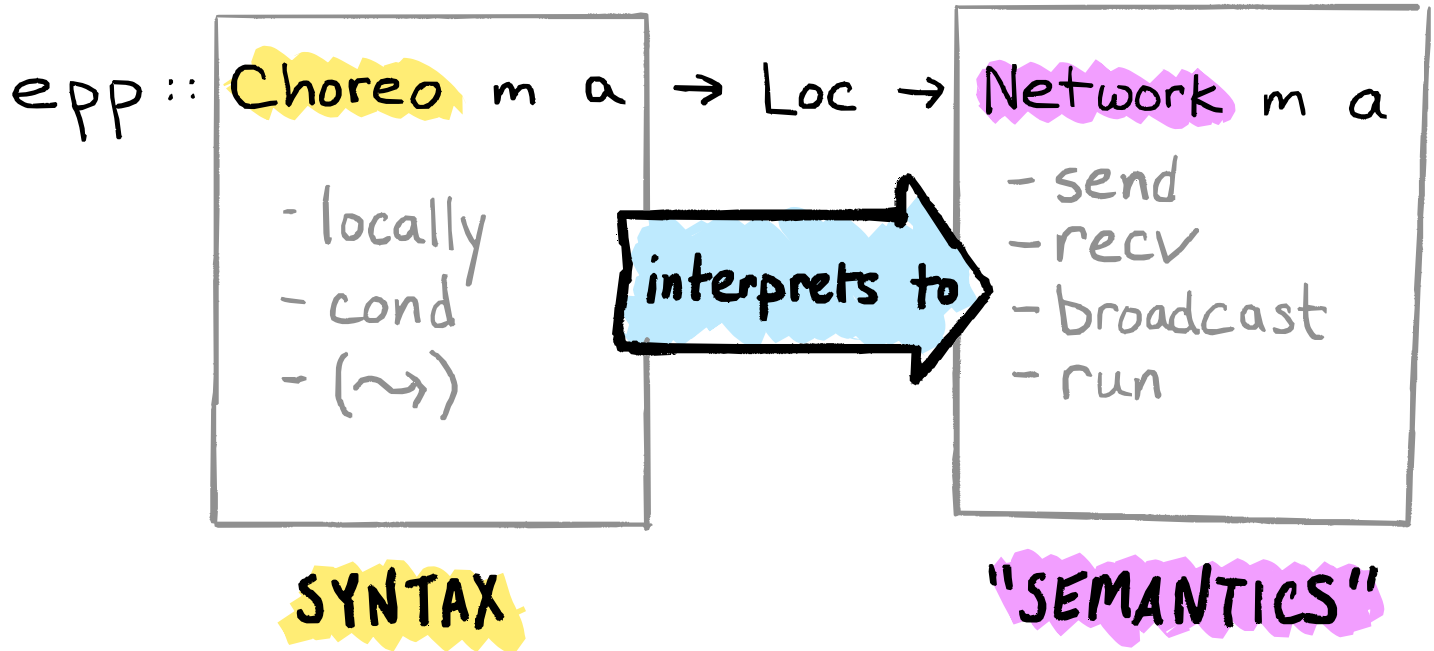
HasChor approach:



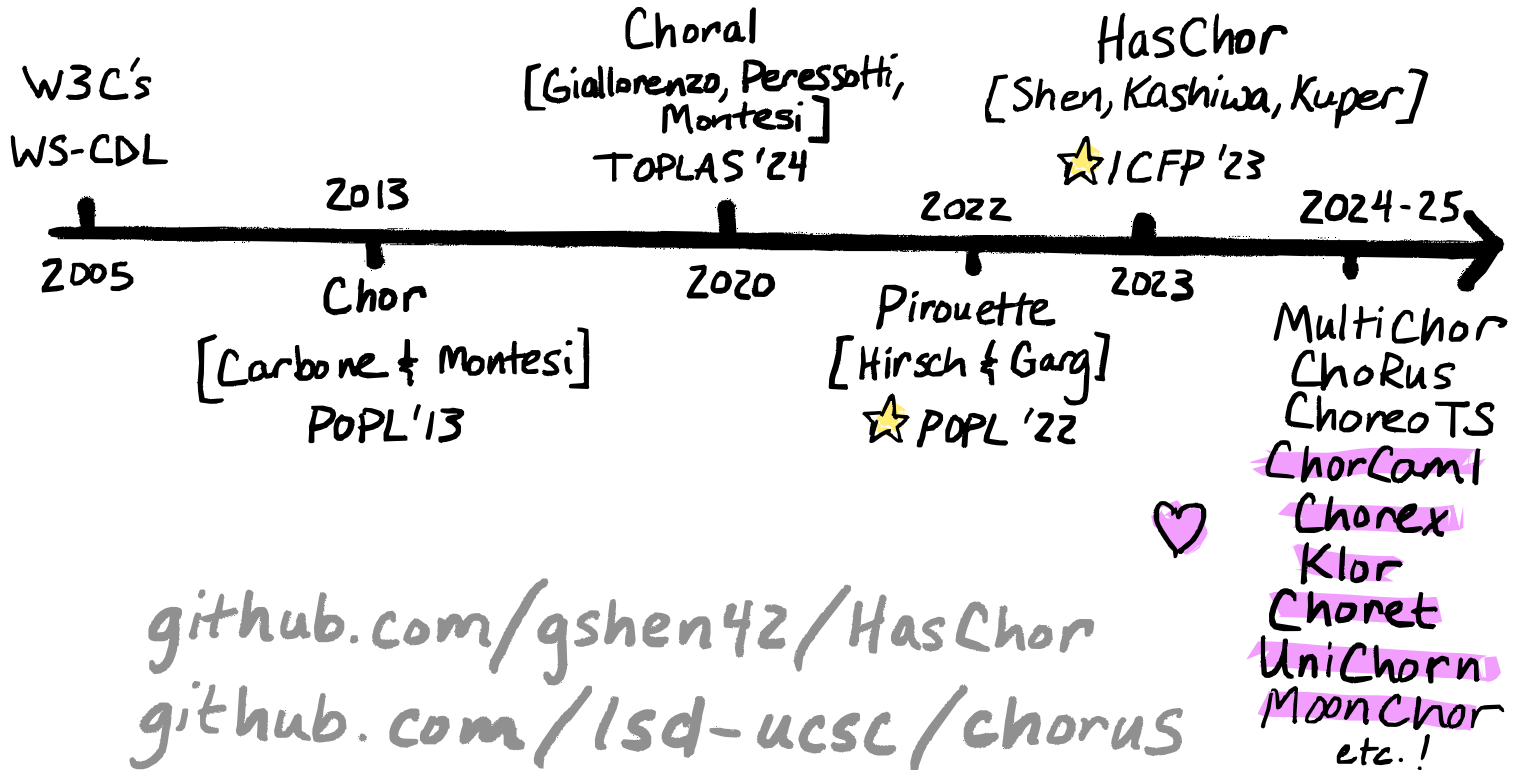
# Endpoint Projection is Just an Interpreter

$epp :: \text{Choreo } m \ a \rightarrow \text{Loc} \rightarrow \text{Network } m \ a$

# Endpoint Projection is Just an Interpreter



# Tons of new CP libraries!



Hey, wait a second...

Distributed systems are fascinating!



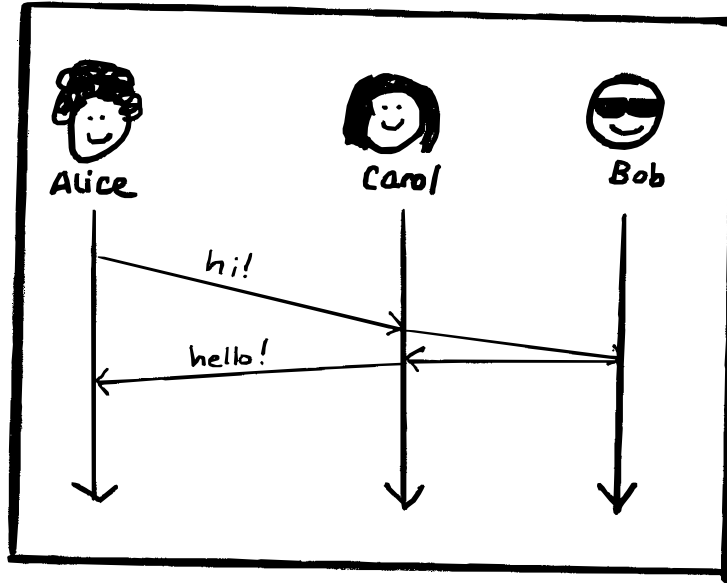
CRDT verification OOPSLA '20

Causal broadcast protocol verification IFL '22

Library-level choreographic programming ICFP '23, PLDI '25

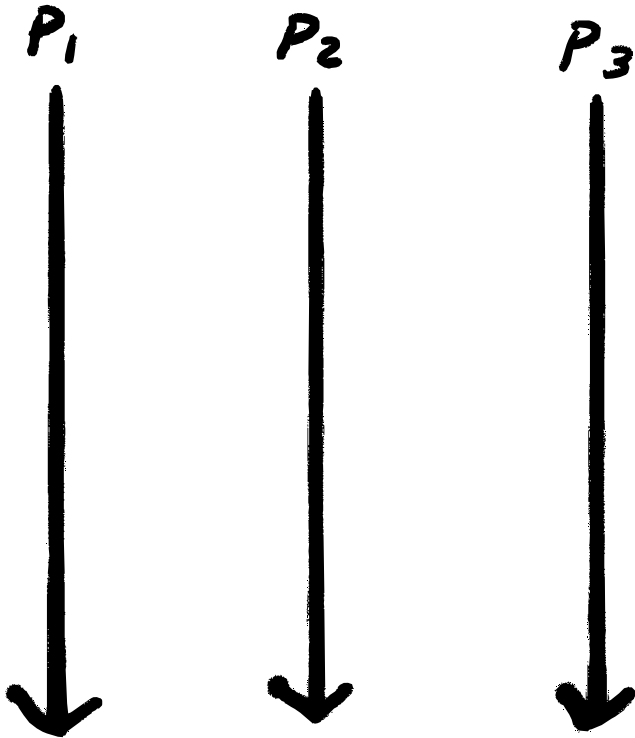
Causal separation diagrams OOPSLA '24

CRDT emulation ICFP '25



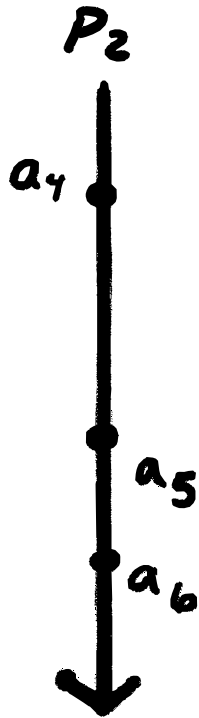
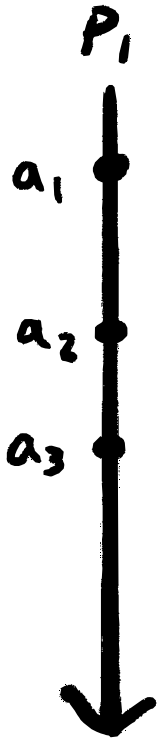
↗  
Lamport diagrams are everywhere  
(aka: sequence diagrams,  
time diagrams,  
...)

How to formalize executions?



$$P = \{P_1, P_2, P_3\}$$

# How to formalize executions?



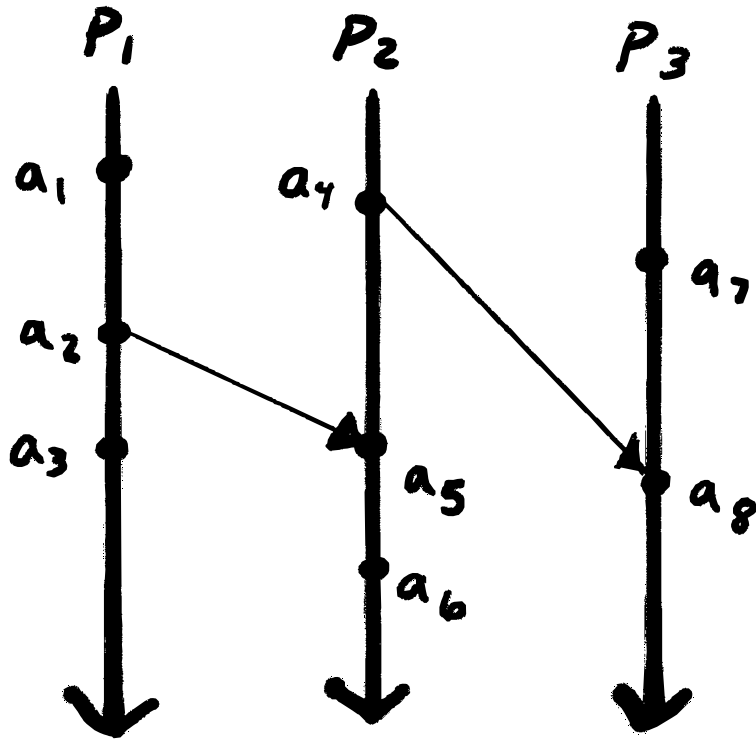
$$P = \{P_1, P_2, P_3\}$$

$$P_1 = [a_1, a_2, a_3]$$

$$P_2 = [a_4, a_5, a_6]$$

$$P_3 = [a_7, a_8]$$

# How to formalize executions?



$$P = \{P_1, P_2, P_3\}$$

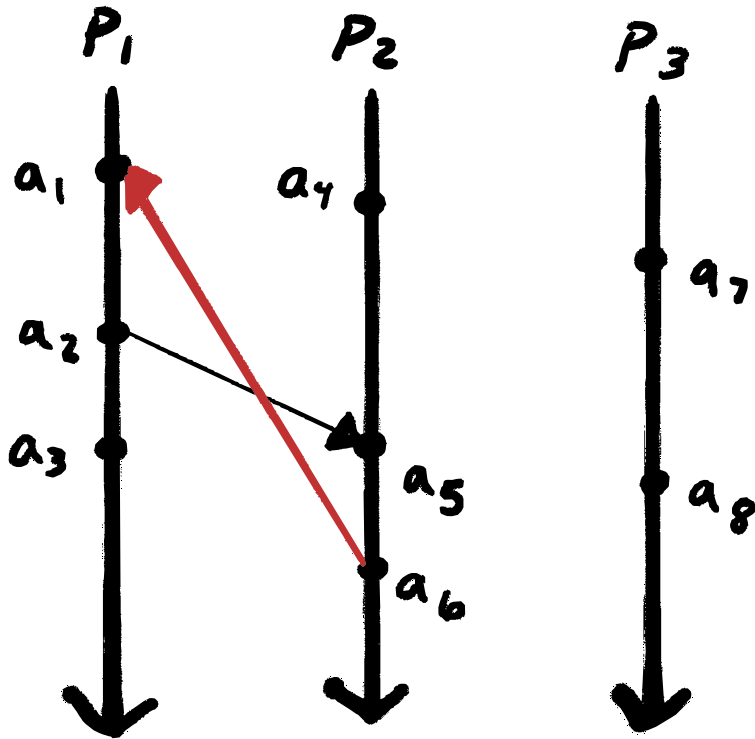
$$P_1 = [a_1, a_2, a_3]$$

$$P_2 = [a_4, a_5, a_6]$$

$$P_3 = [a_7, a_8]$$

$$m = \{(a_2, a_5), (a_4, a_8)\}$$

# How to formalize executions?



$$P = \{P_1, P_2, P_3\}$$

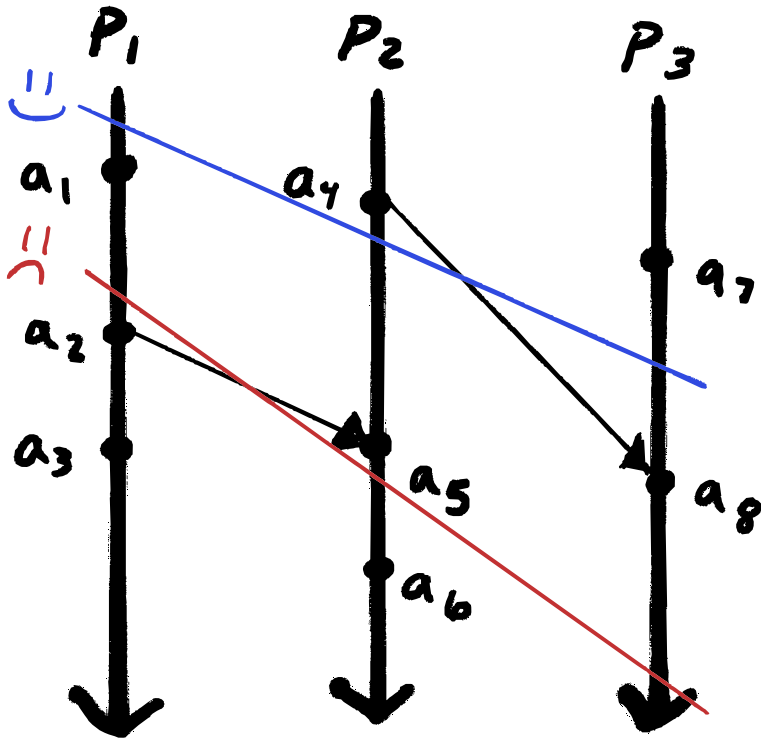
$$P_1 = [a_1, a_2, a_3]$$

$$P_2 = [a_4, a_5, a_6]$$

$$P_3 = [a_7, a_8]$$

$$m = \{(a_2, a_5), \\ (a_6, a_1)\}$$

# How to formalize executions?



$$P = \{P_1, P_2, P_3\}$$

$$P_1 = [a_1, a_2, a_3]$$

$$P_2 = [a_4, a_5, a_6]$$

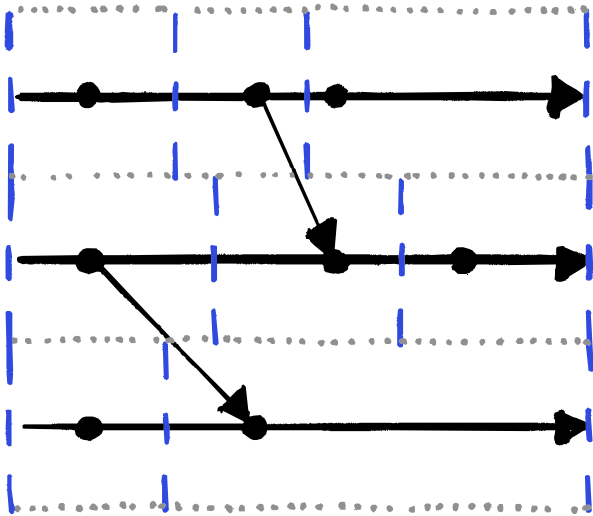
$$P_3 = [a_7, a_8]$$

$$M = \{(a_2, a_5), (a_4, a_8)\}$$

How to formalize executions?

# How to formalize executions?

Traditional approach



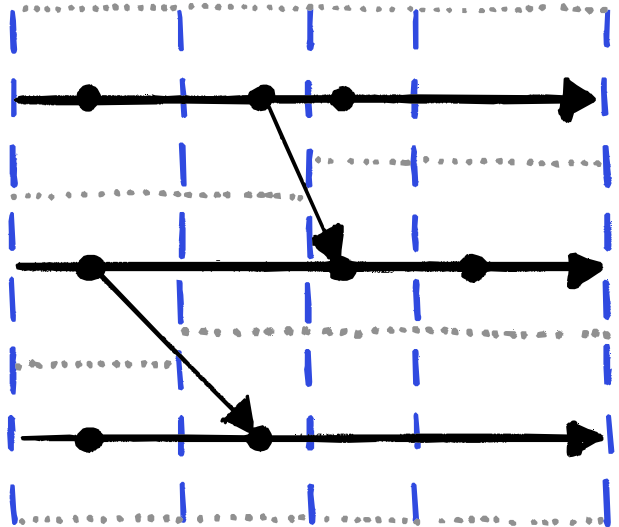
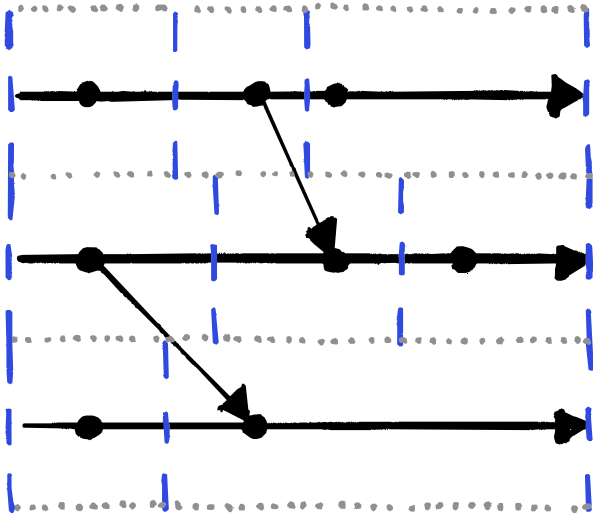
— — — — = temporal boundary  
..... = spatial boundary

# How to formalize executions?

our approach (oopsla '24):

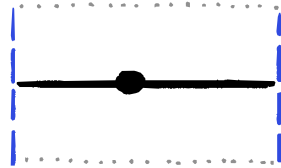
## Causal separation diagrams

## Traditional approach



— — — — = temporal boundary  
..... = spatial boundary

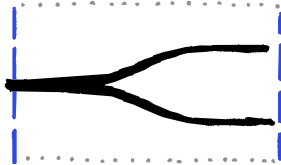
# CSDs are made of Composable "tiles"



tick



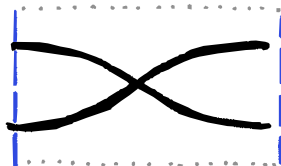
noop



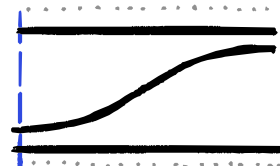
fork



join



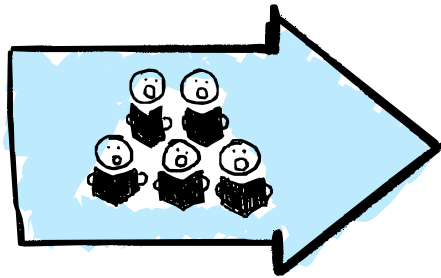
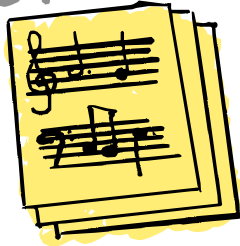
swap



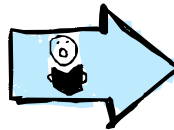
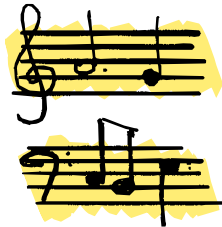
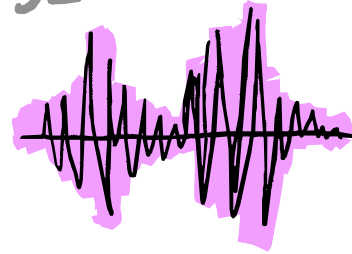
assoc

# inductively defined data

SYNTAX

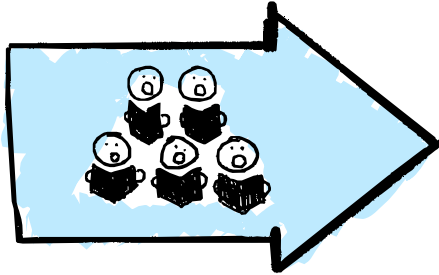


SEMANTICS

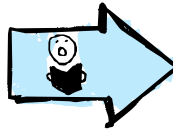
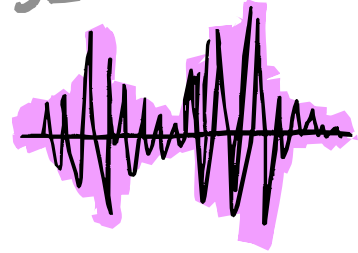


# inductively defined data

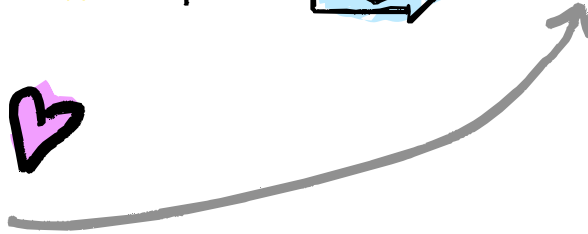
SYNTAX



SEMANTICS

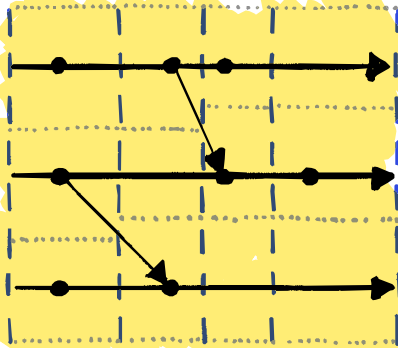


Structural  
induction  
principle



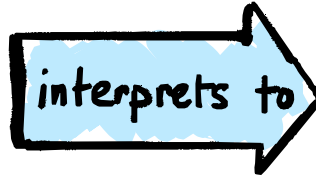
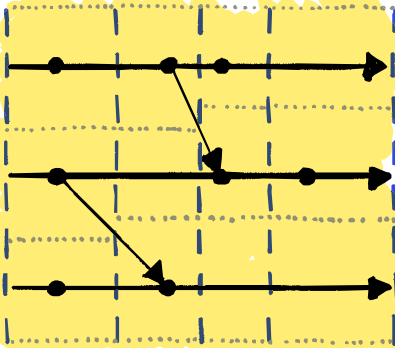
CSDs are interpretable

SYNTAX



CSDs are interpretable

SYNTAX



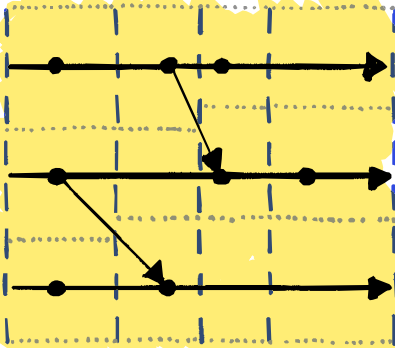
SEMANTICS:

causal paths

proof-relevant happens-before!

# CSDs are interpretable

SYNTAX



interprets to

SEMANTICS:

causal paths

proof-relevant happens-before!

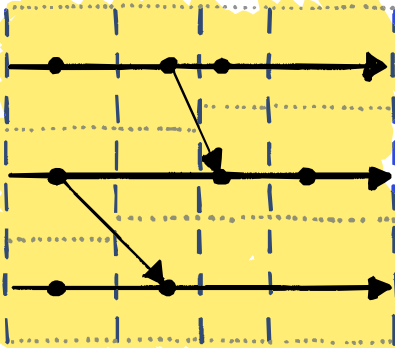
... and, to

(logical) clocks

e.g., vector clocks,  
matrix clocks

# CSDs are interpretable

SYNTAX



interprets to

SEMANTICS:

causal paths

*proof-relevant happens-before!*

... and, to

(logical) clocks

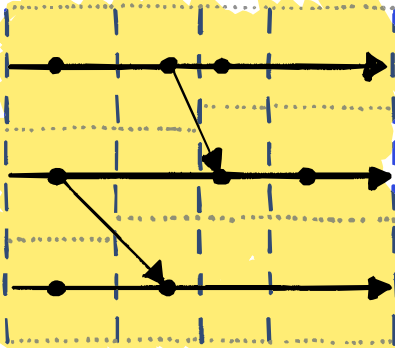
*e.g., vector clocks,  
matrix clocks*

... and, to

proofs that  
clocks respect  
causal paths

# CSDs are interpretable

SYNTAX



interprets to

SEMANTICS:

causal paths

*proof-relevant happens-before!*

... and, to

(logical) clocks

*e.g., vector clocks,  
matrix clocks*

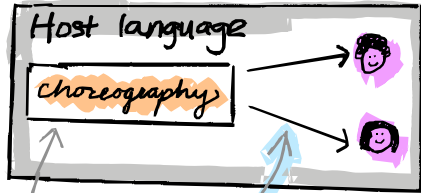
... and, to

proofs that  
clocks respect  
causal paths

[github.com/lsd-ucsc/csd](https://github.com/lsd-ucsc/csd)

# it was all interpreters all along!

library-level choreographic programming



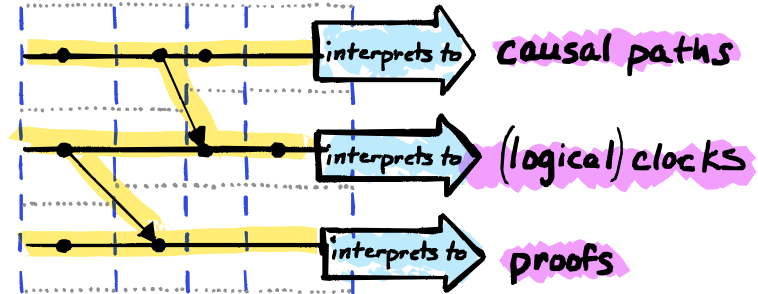
embedded DSL ♡

endpoint projection via

interpretation

ICFP '23,  
PLDI '25

causal separation diagrams



OOPSLA '24

Thank you, Dan-aspora ♡