A Synthetic Reconstruction of Multiparty Session Types

Castro-Perez et al., POPL ‘26

A.K.A.: A Crash-Course in Session Types

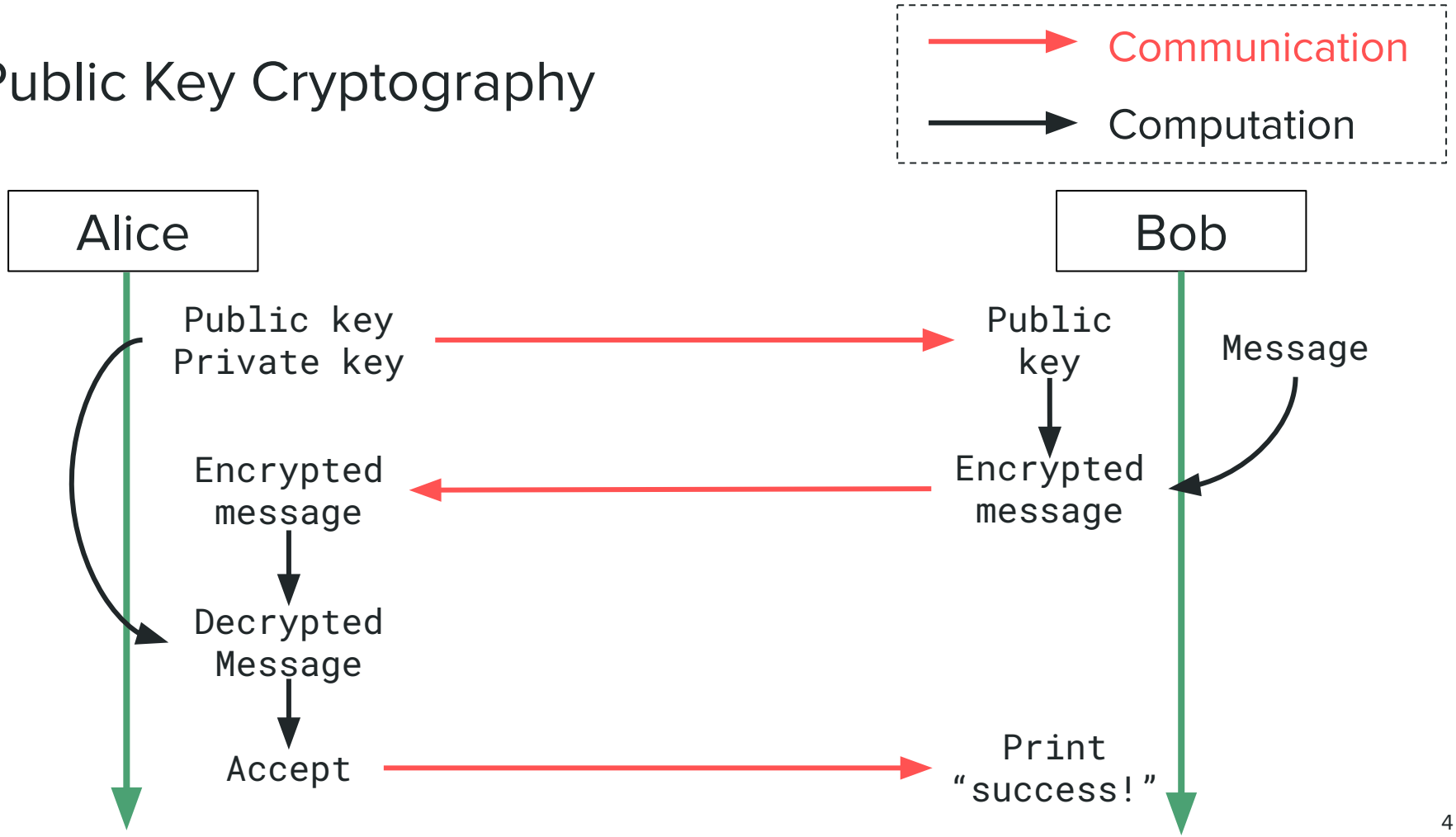
Presented by: Ashley Samuelson

Session Types

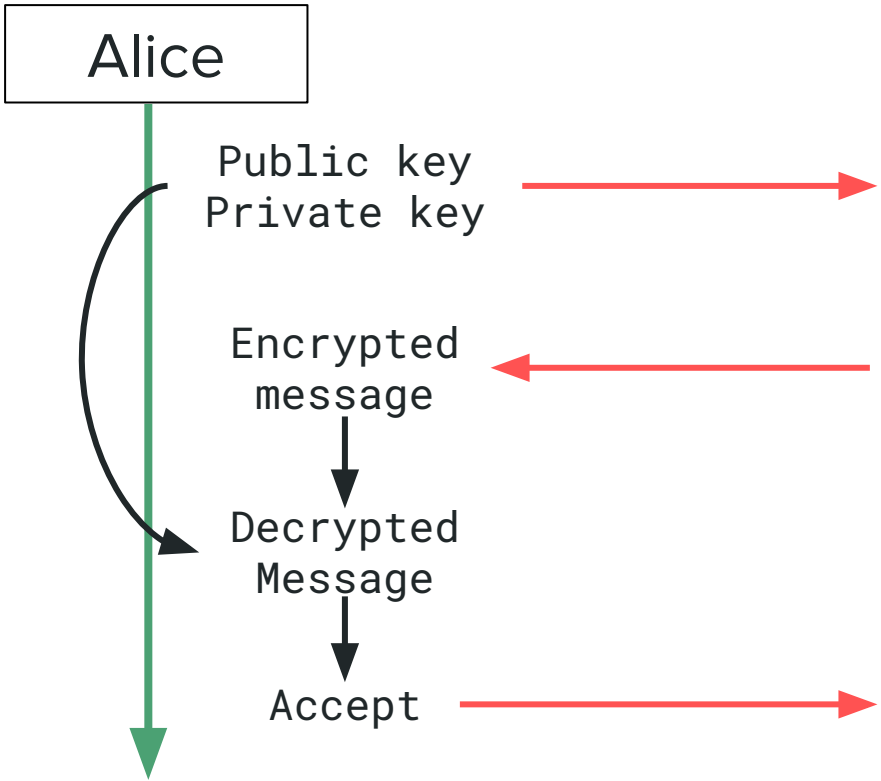
A type-systems approach to guaranteeing that concurrent systems are deadlock-free

What does concurrent programming look like?

Public Key Cryptography



A **separate program** is used to describe the actions of **each participant** in a concurrent computation



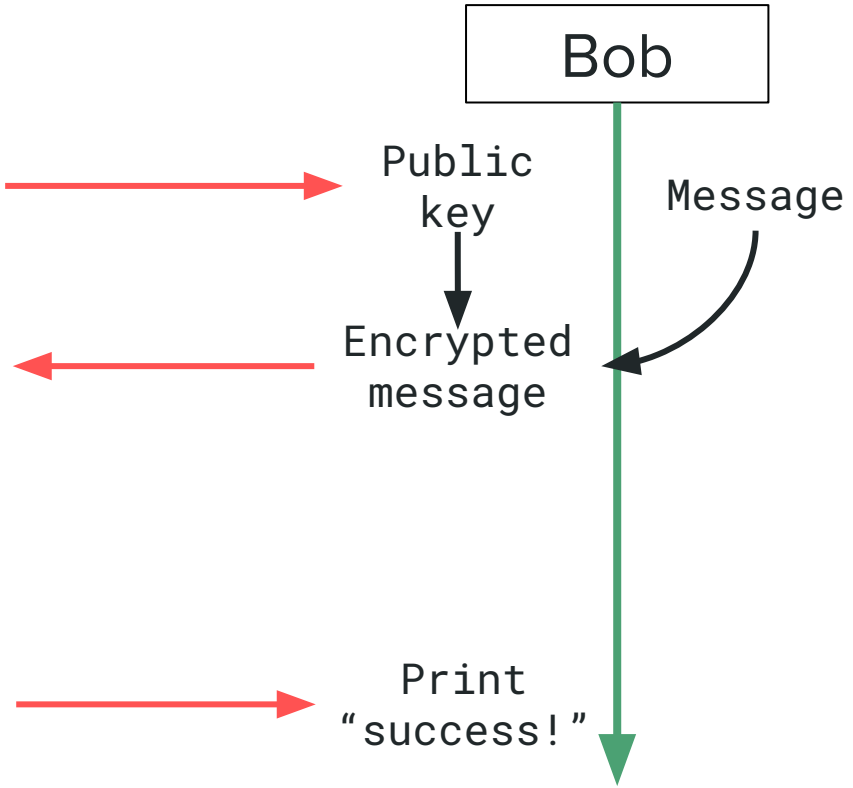
Alice

```
Generate public and secret keys
Send public key to Bob
Receive message from Bob
Decrypt message
Verify message:
    If accept send True to Bob
    Else send False to Bob
```

A **separate program** is used to describe the actions of **each participant** in a concurrent computation

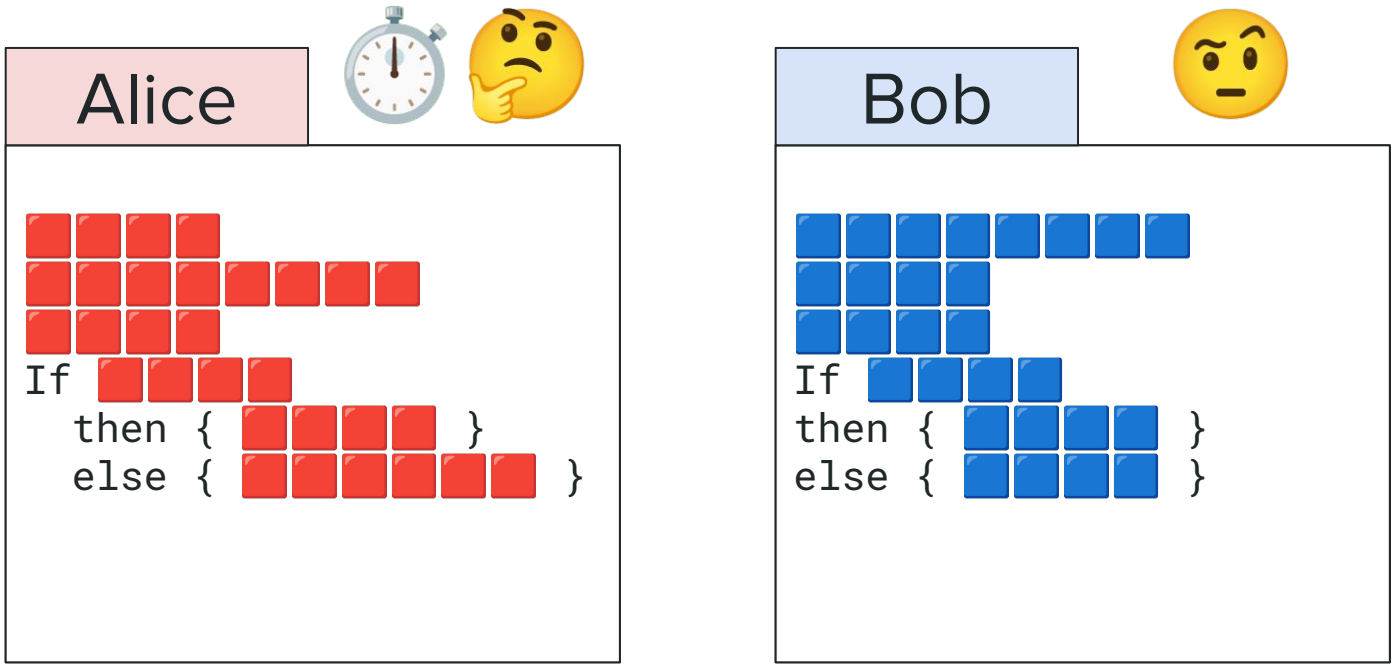
Bob

```
Receive public key from Alice
Encrypt message
Send encrypted message to Alice
Receive choice from Alice
    If True print “success”
    Else print “failure”
```

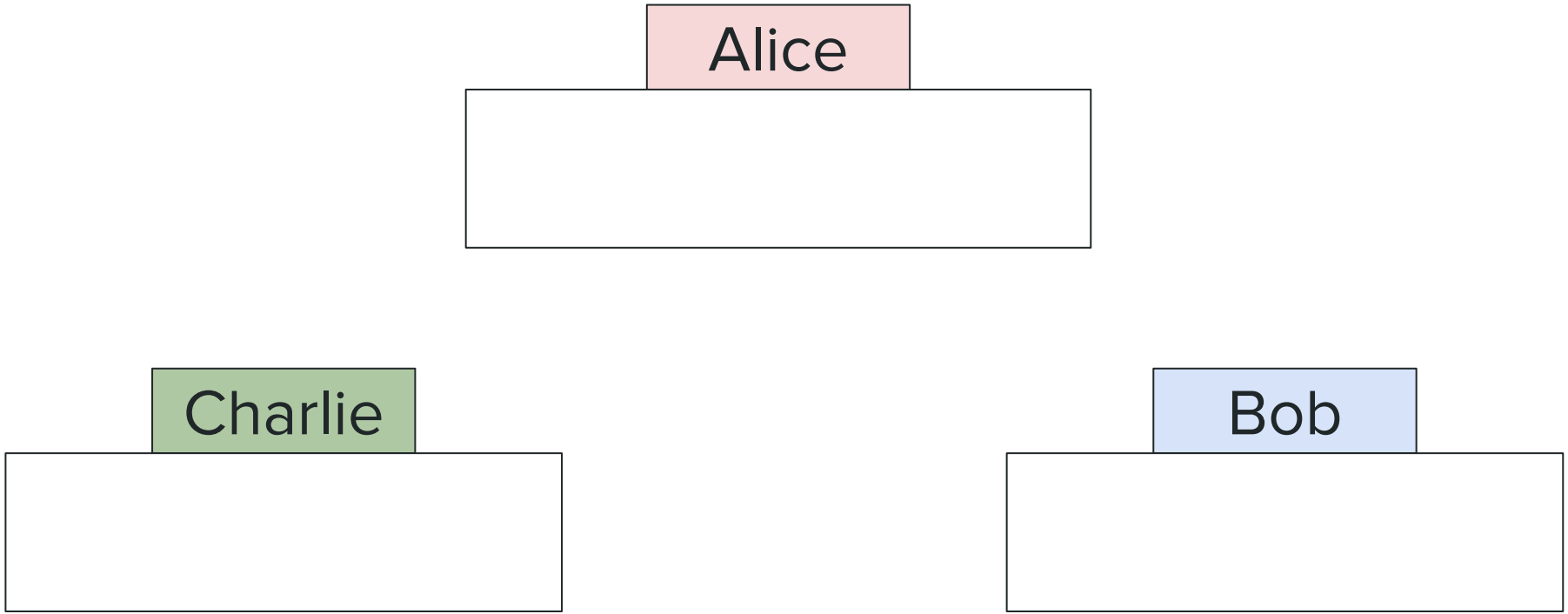


Why is concurrent programming difficult?

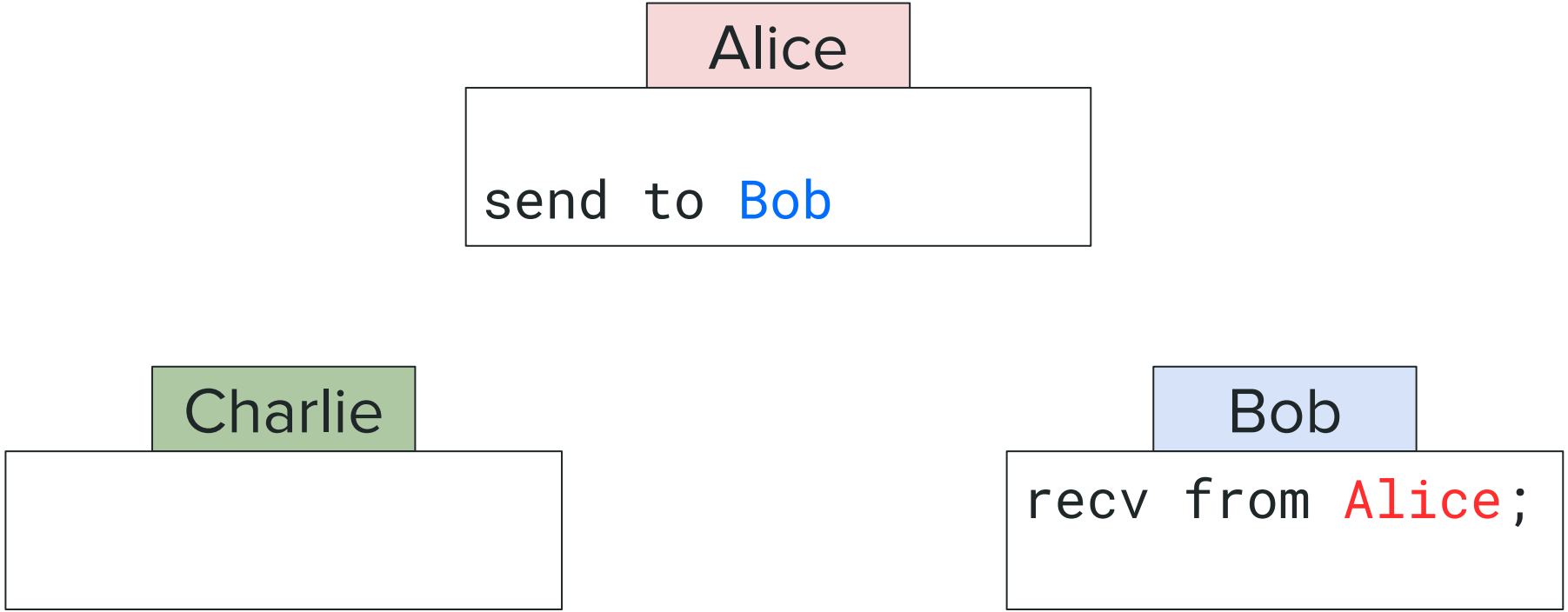
It can be easy to mistakenly introduce **deadlocks**



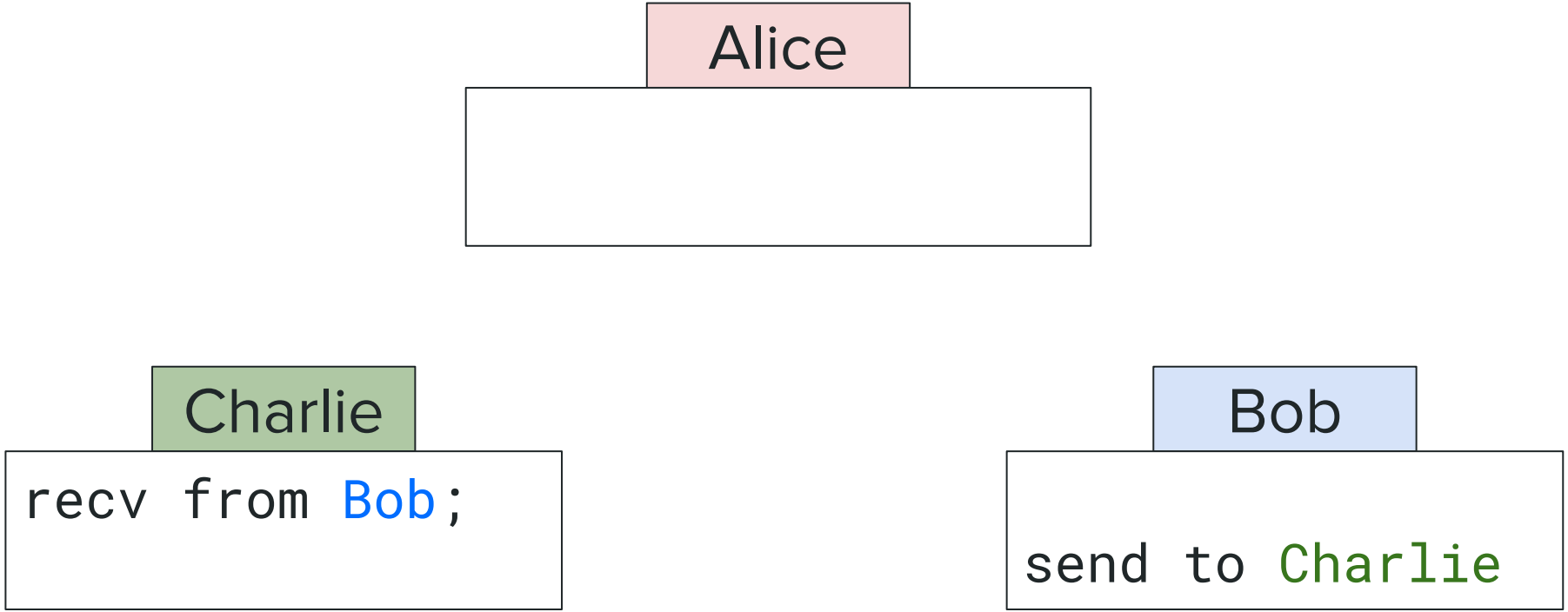
Deadlocks can be **difficult to detect**



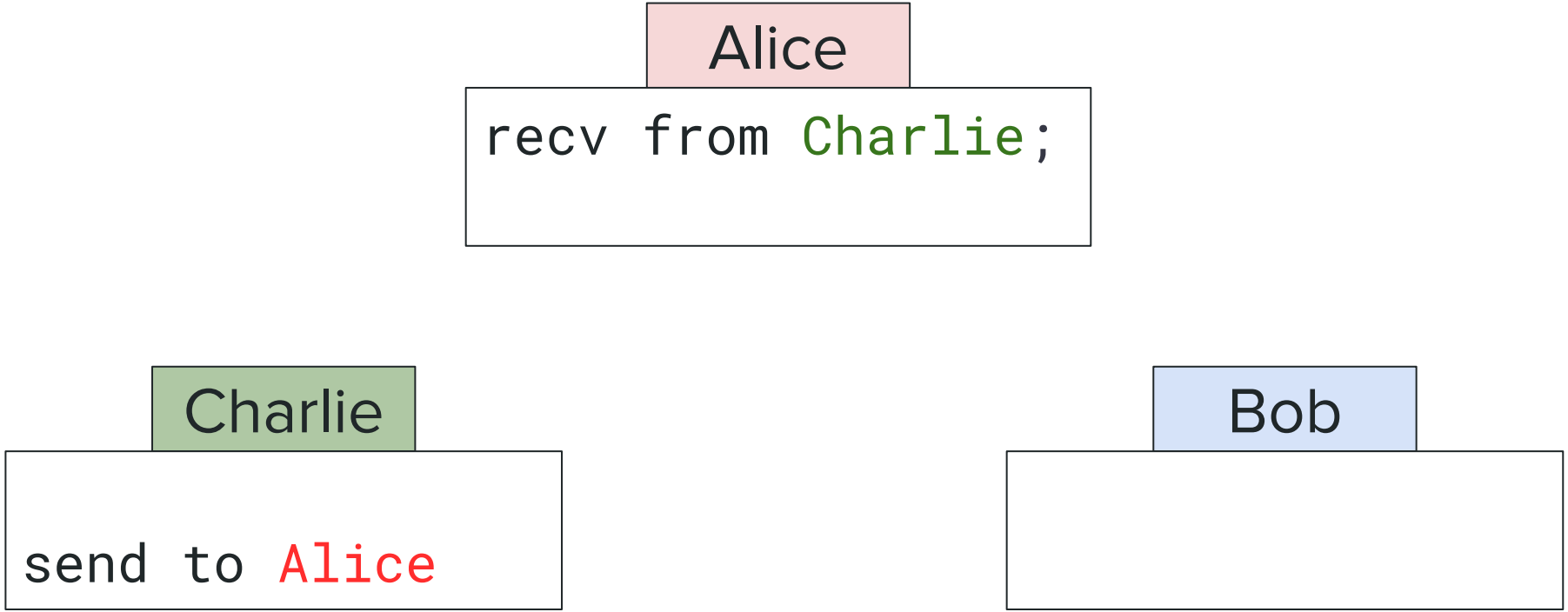
Deadlocks can be **difficult to detect**



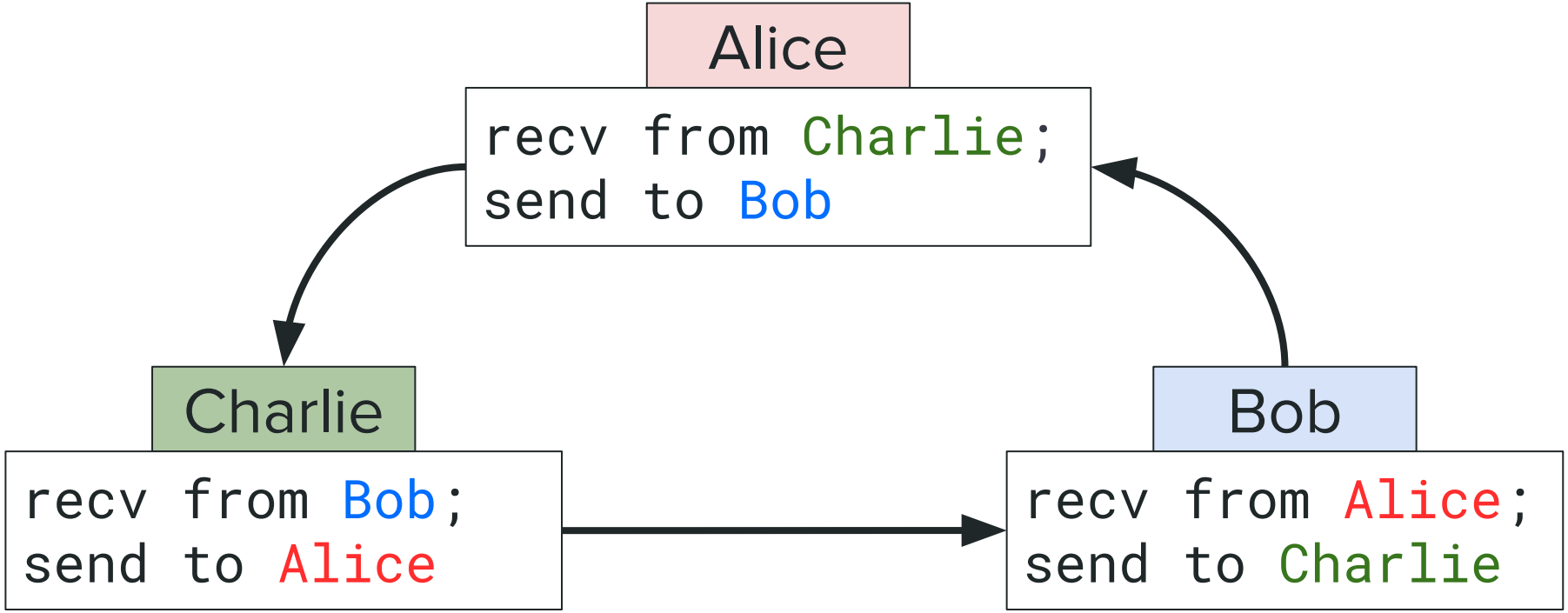
Deadlocks can be **difficult to detect**



Deadlocks can be **difficult to detect**



Deadlocks can be **difficult to detect**



How do session types solve this?

Session types are a **behavioral type system**: “treat **types as simple processes** that reduce and evolve along with a computation”

Standard Type System

```
let x = read(A0)
in if x == 0
   then A0 := 1;
        A1 := 2;        : Bool
        True
   else A1 := 1 + x;
        False
```

Type and Effect System

```
let x = read(A0)
in if x == 0
    then A0 := 1;                          { R(A0),
         A1 := 2;        : Bool ▷            W(A0),
         True                                W(A1) }
    else A1 := 1 + x;
         False
```

Behavioral Type System

```
let x = read(A0)                                  R(A0);
in if x == 0                                      branch
   then A0 := 1;                                  | L =>
        A1 := 2;          : Bool ▷                    W(A0);
        True                                          W(A1)
   else A1 := 1 + x;                              | R =>
        False                                         W(A1)
```

Session types describe the pattern of **message sends and receives** in a program

Alice

```
let x = 1 + 1 in
send x to Bob;
recv y from Bob;
print(y)
```

▷

```
send Int to Bob;
recv String from Bob;
End
```

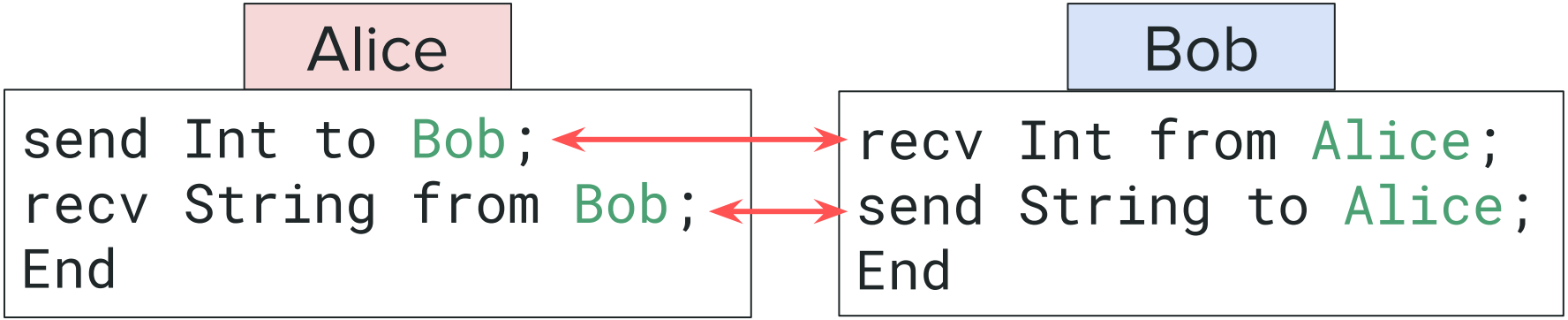
Session types describe the pattern of **message sends and receives** in a program

Bob

```
recv x from Alice;
let y =
  (x == 0 ? “Yes” : “No”)
in send y to Alice
```

```
recv Int from Alice;
send String to Alice;
End
```

We then **compare the session types** assigned to each individual program



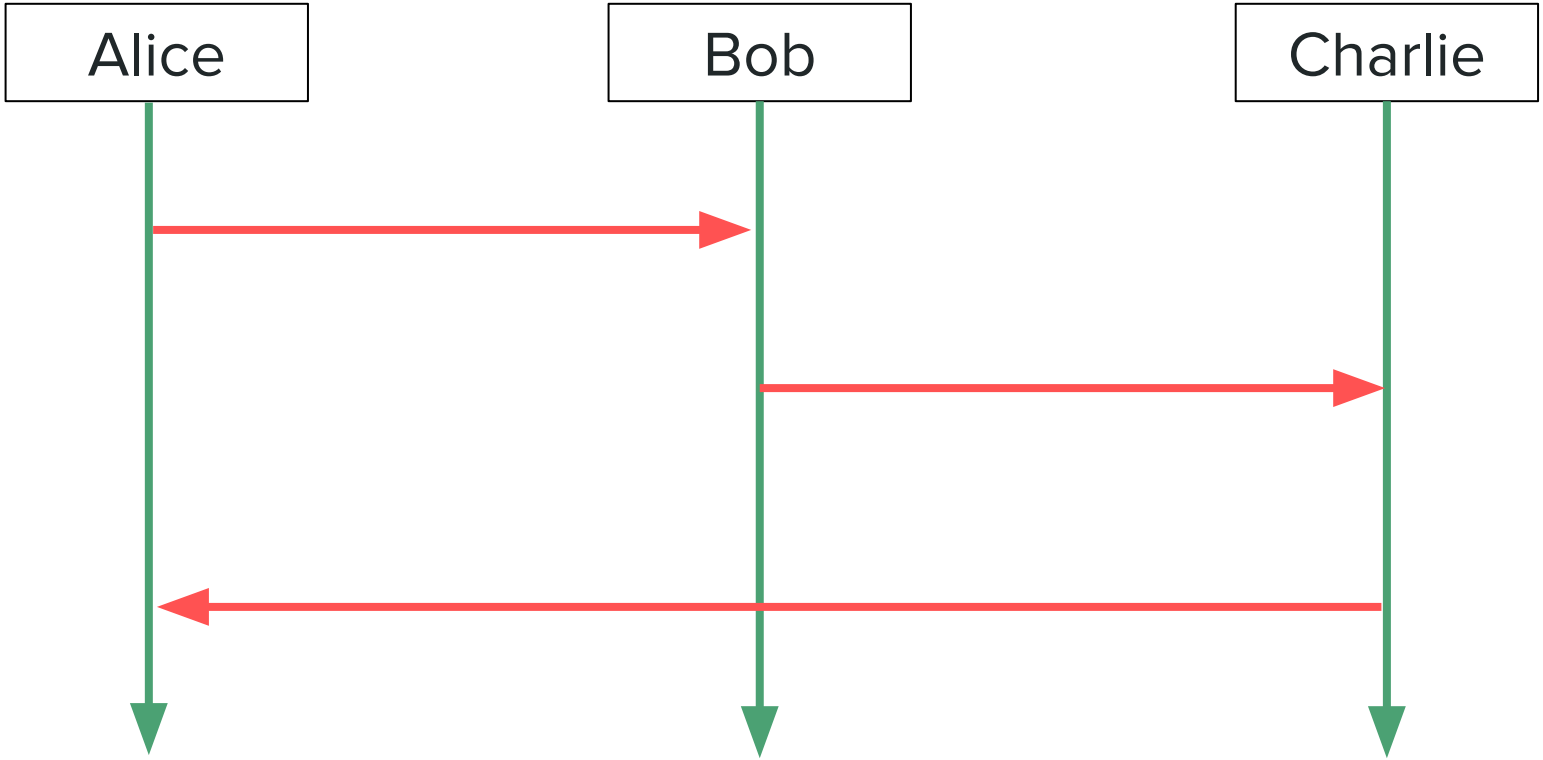
If the **sends and receives match**, the system is deadlock-free!

(Bonus) Binary Session Types and Duality

```
S ::= X | end | send T; S | recv T; S
    | μX.S | choose S1 S2 | choice S1 S2
```

```
dual(X)              = X
dual(end)            = end
dual(send T; S)      = recv T; dual(S)
dual(recv T; S)      = send T; dual(S)
dual(μX.S)           = μX.dual(S)
dual(choose S1 S2) = choice dual(S1) dual(S2)
dual(choice S1 S2) = choose dual(S1) dual(S2)
```

Surprisingly, this idea **does not work for more than 2 parties!**



Alice

```
recv from Charlie;
send to Bob
```

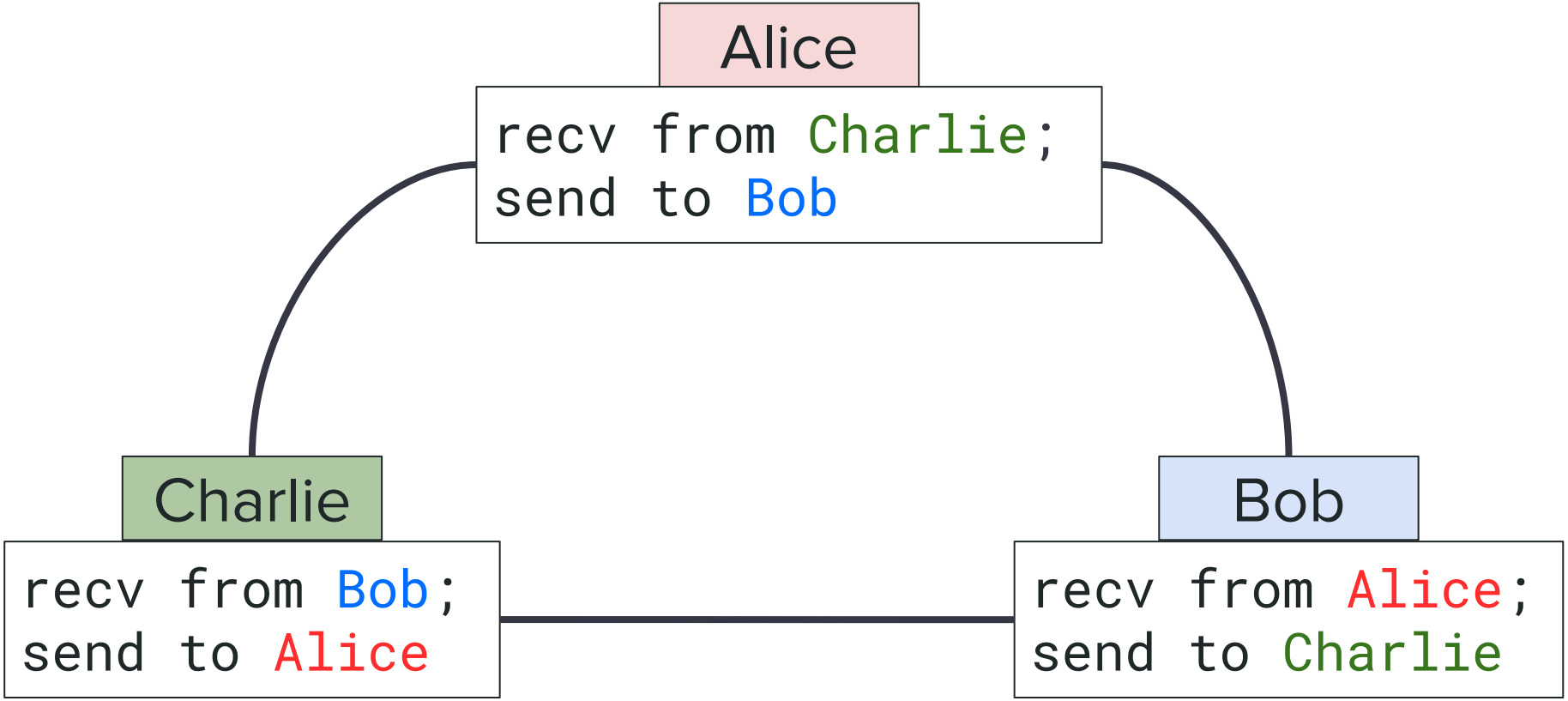
Deadlock!

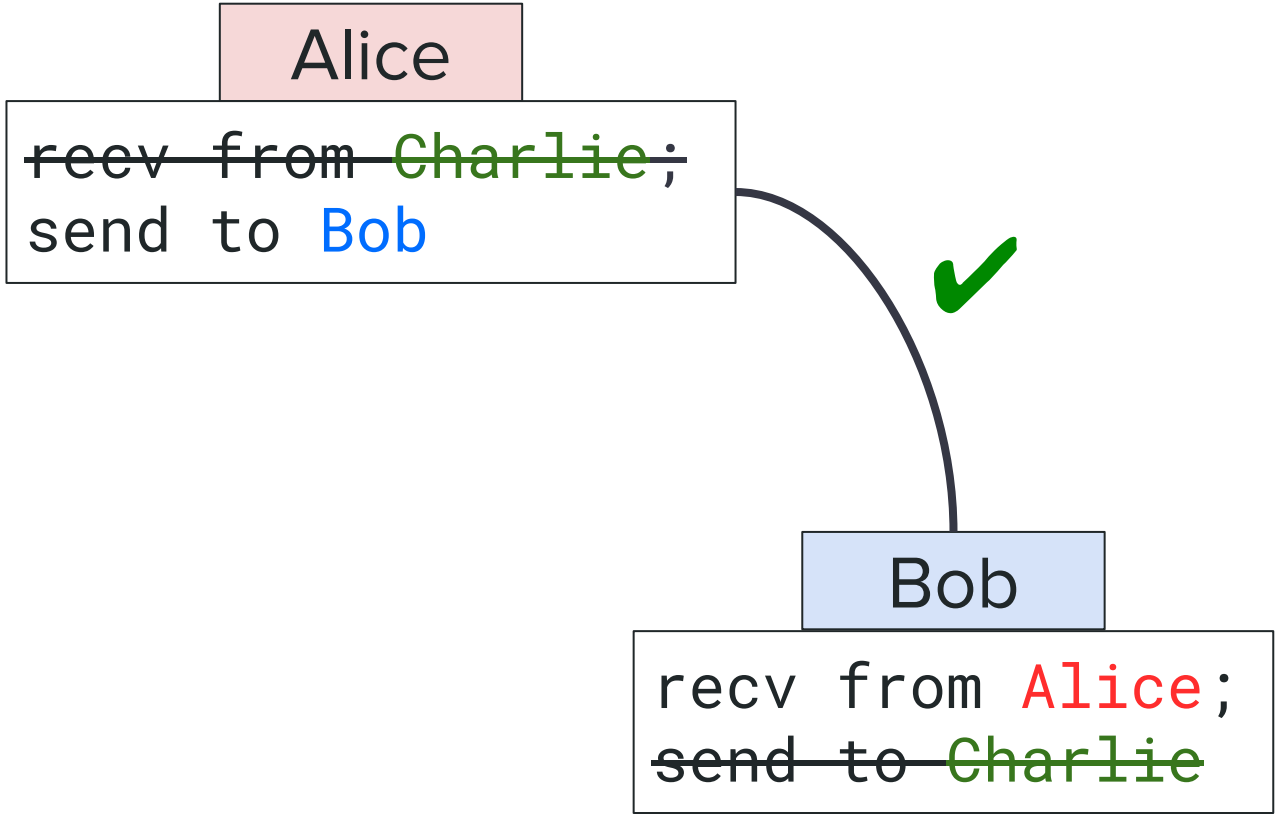
Charlie

```
recv from Bob;
send to Alice
```

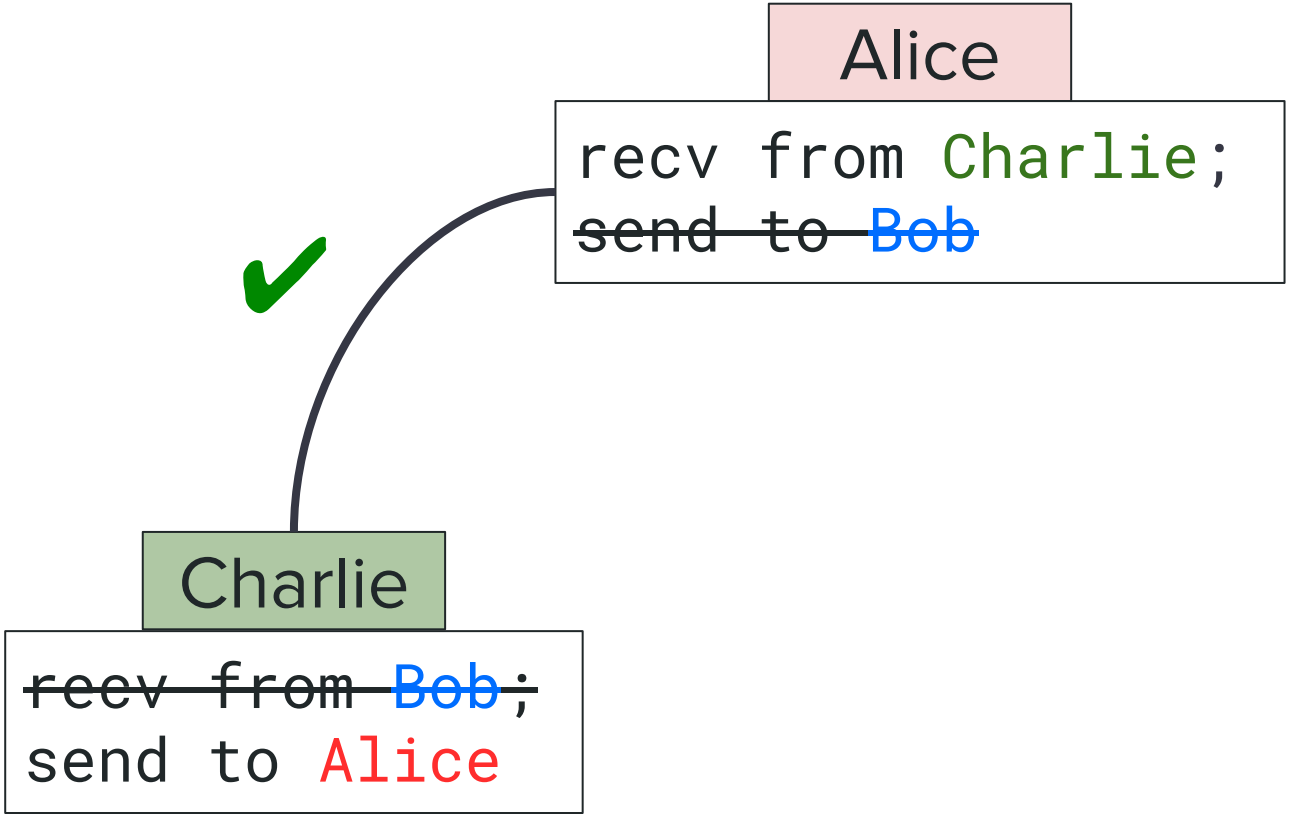
Bob

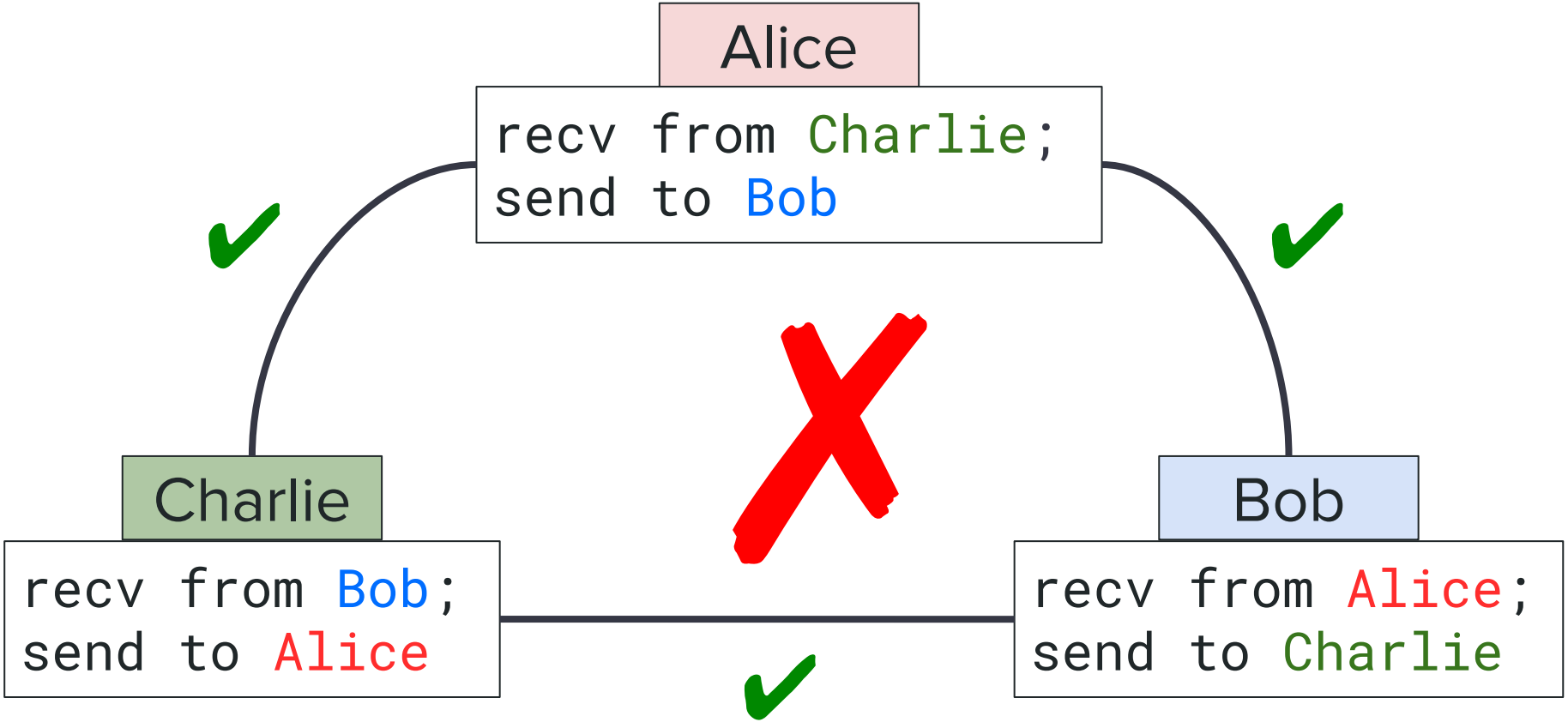
```
recv from Alice;
send to Charlie
```


Alice
recv from Charlie;
send to Bob
Charlie
recv from Bob;
send to Alice
Bob
recv from Alice;
send to Charlie


Alice
recv from Charlie;
send to Bob
✔
Bob
recv from Alice;
send to Charlie


Charlie
recv from Bob;
send to Alice
Bob
recv from Alice;
send to Charlie

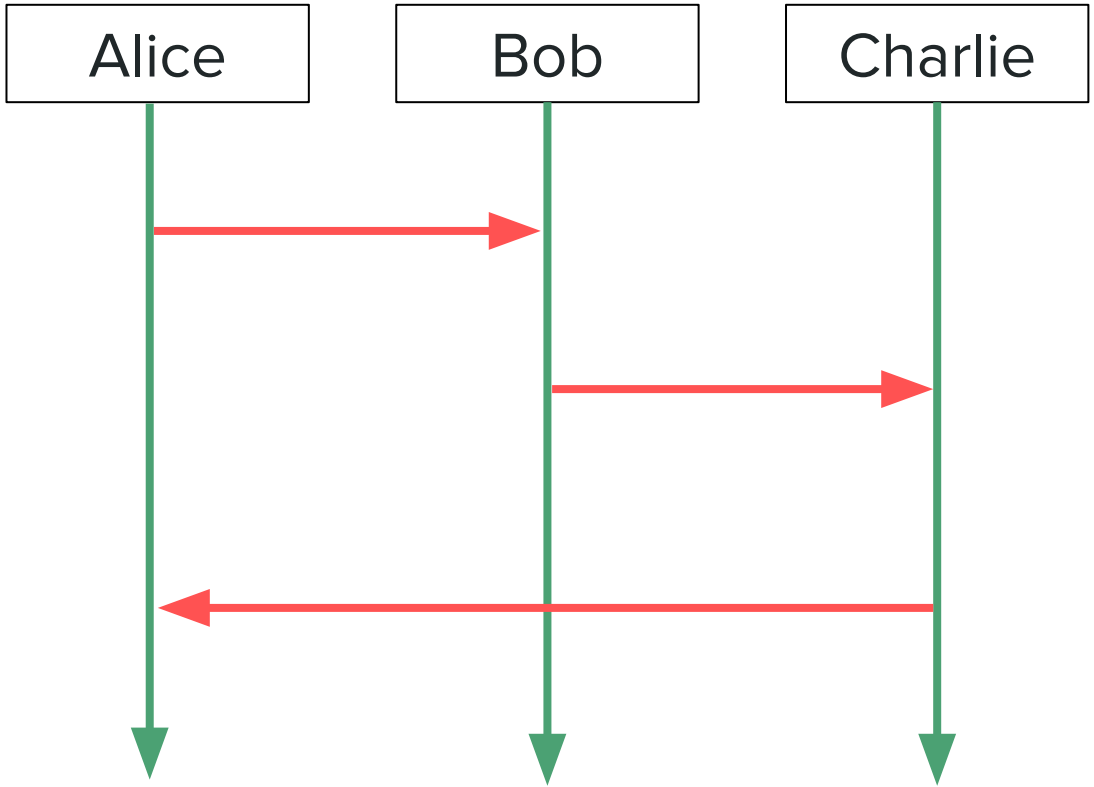

Alice
recv from Charlie;
send to Bob
Charlie
recv from Bob;
send to Alice


Alice
recv from Charlie;
send to Bob
Charlie
recv from Bob;
send to Alice
Bob
recv from Alice;
send to Charlie

If we **only compare pairs** of interacting parties, we **lose** critical **global ordering information**!

Multiparty Session Types

Construct a **global session type** (also called a choreography 👀) that describes the **actions of all participants** and the intended **global order of events**



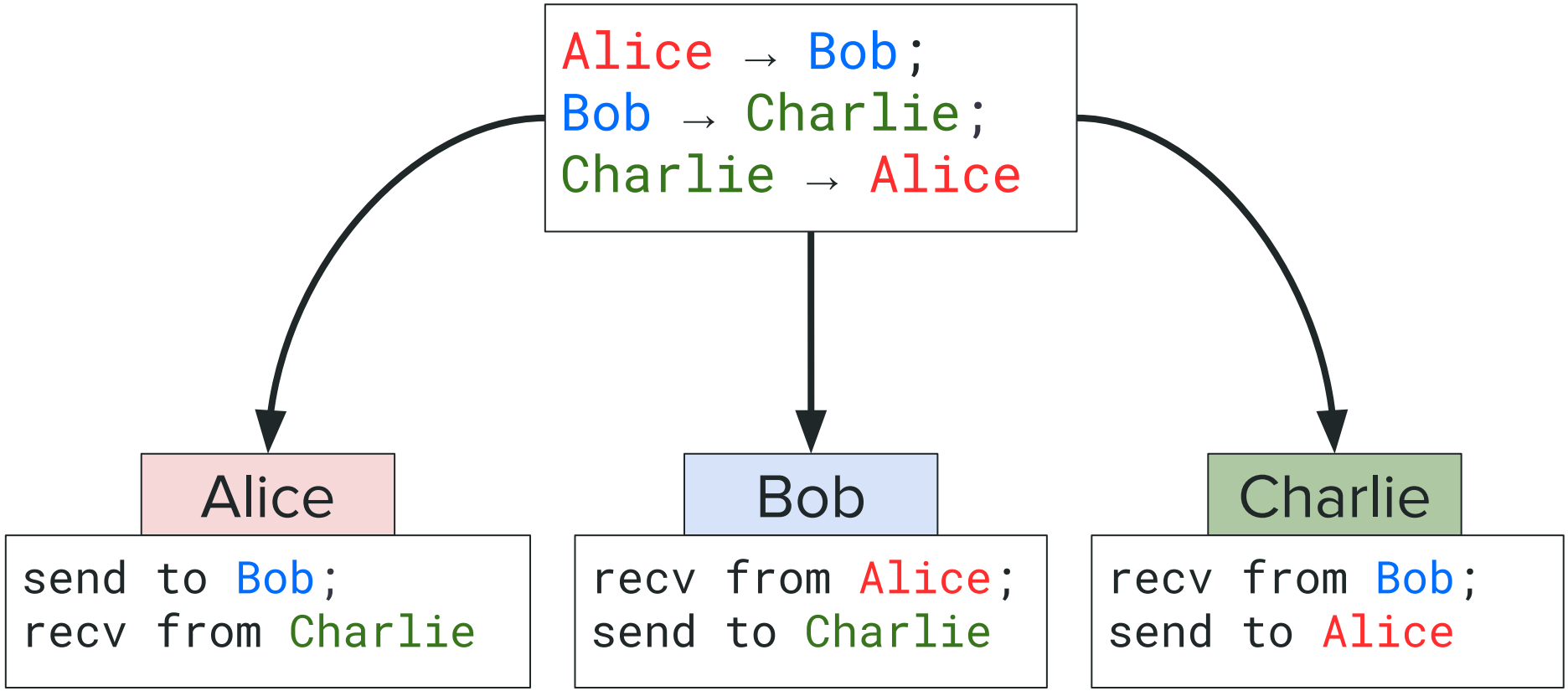

```
Alice → Bob;
Bob → Charlie;
Charlie → Alice
```

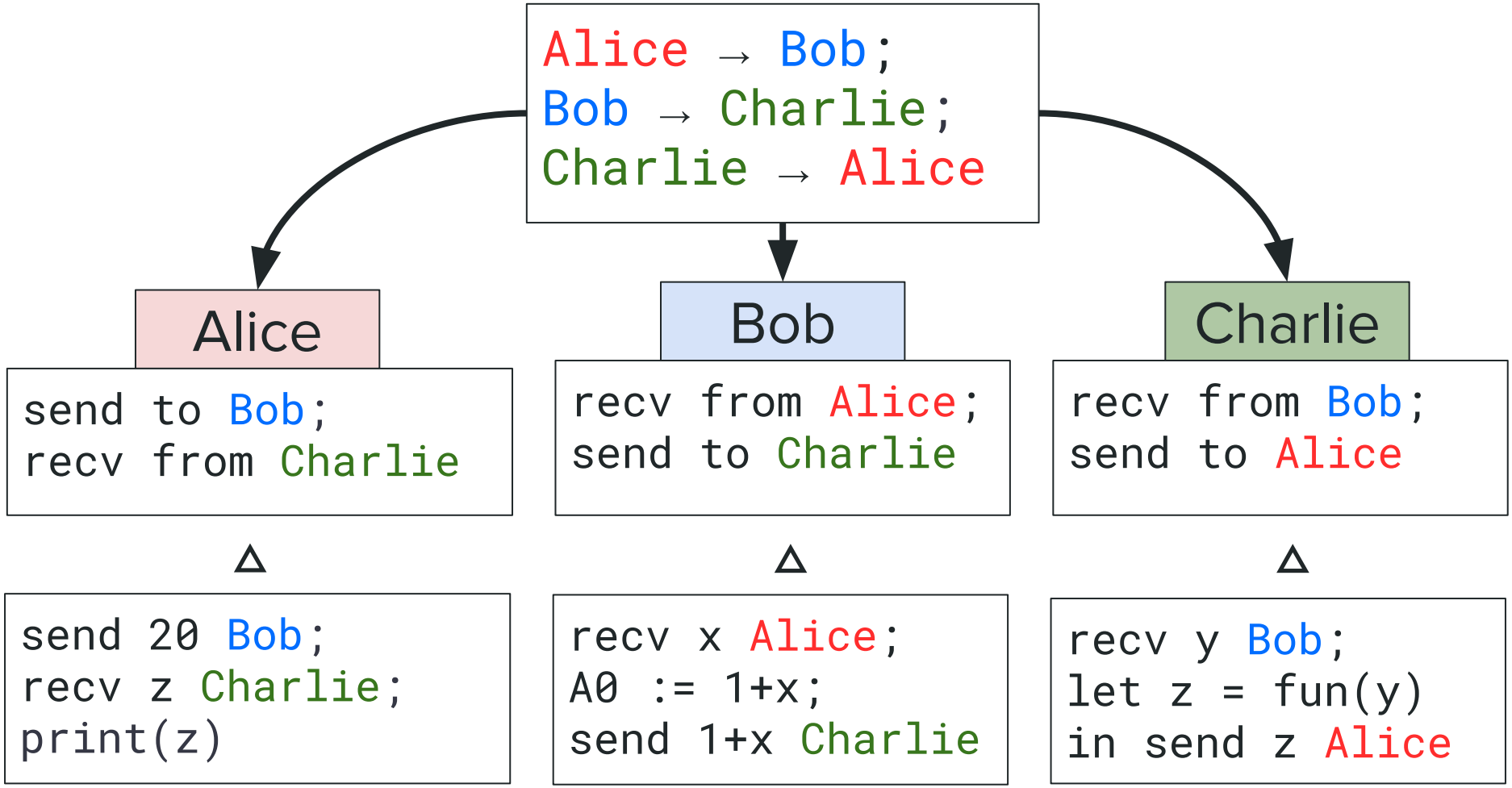
From this global session, **project** 👀 the actions of a specified participant to get their individual session type, and **check their program against it**

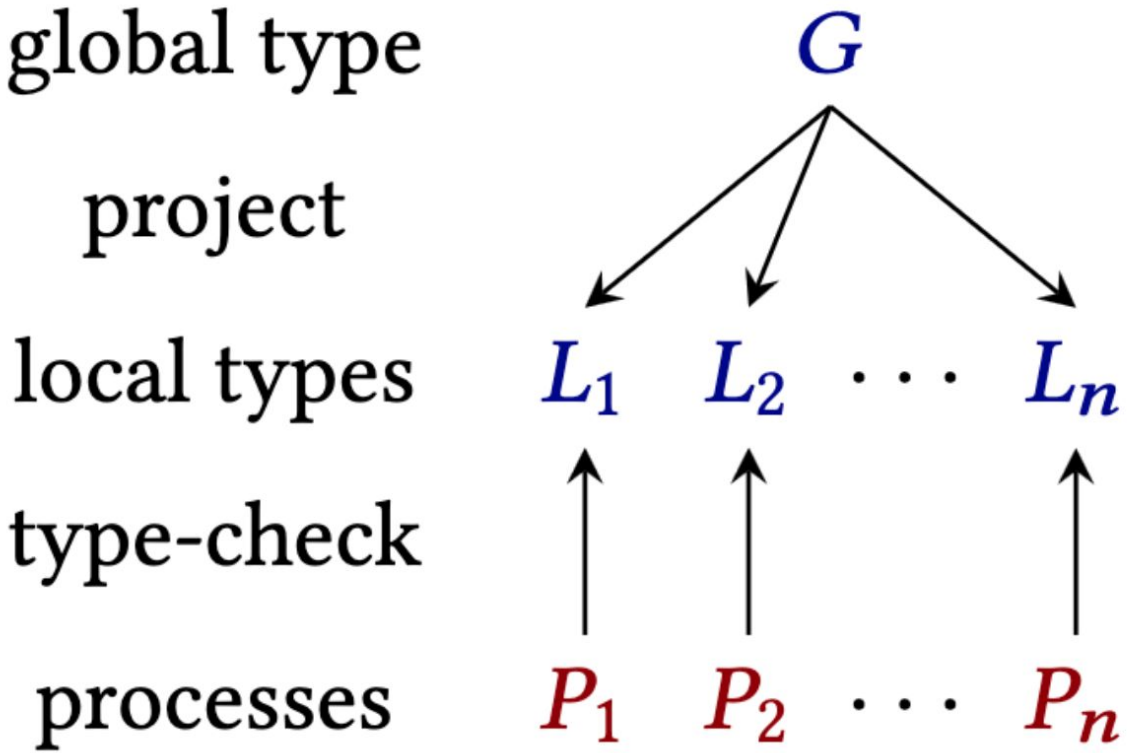
(Bonus) Global Session Types and Projection

```
G ::= X | end | p[T] → q; G
    | μX.G | p → q{ G1, G2 }
```

```
proj(X, p)                 = X
proj(end, p)               = end
proj(μX.G, p)              = μX.proj(G, p)
proj(p[T] → q; G, p) = send T q; proj(G, p)
proj(p[T] → q; G, q) = recv T p; proj(G, q)
proj(p[T] → q; G, r) = proj(G, r)
…
```


Alice → Bob;
Bob → Charlie;
Charlie → Alice
Alice
send to Bob;
recv from Charlie
Bob
recv from Alice;
send to Charlie
Charlie
recv from Bob;
send to Alice


Alice → Bob;
Bob → Charlie;
Charlie → Alice
Alice
send to Bob;
recv from Charlie
Bob
recv from Alice;
send to Charlie
Charlie
recv from Bob;
send to Alice
send 20 Bob;
recv z Charlie;
print(z)
recv x Alice;
A0 := 1+x;
send 1+x Charlie
recv y Bob;
let z = fun(y)
in send z Alice



(a) Classical [17]

Drawback: For some deadlock-free systems, there is no corresponding global type

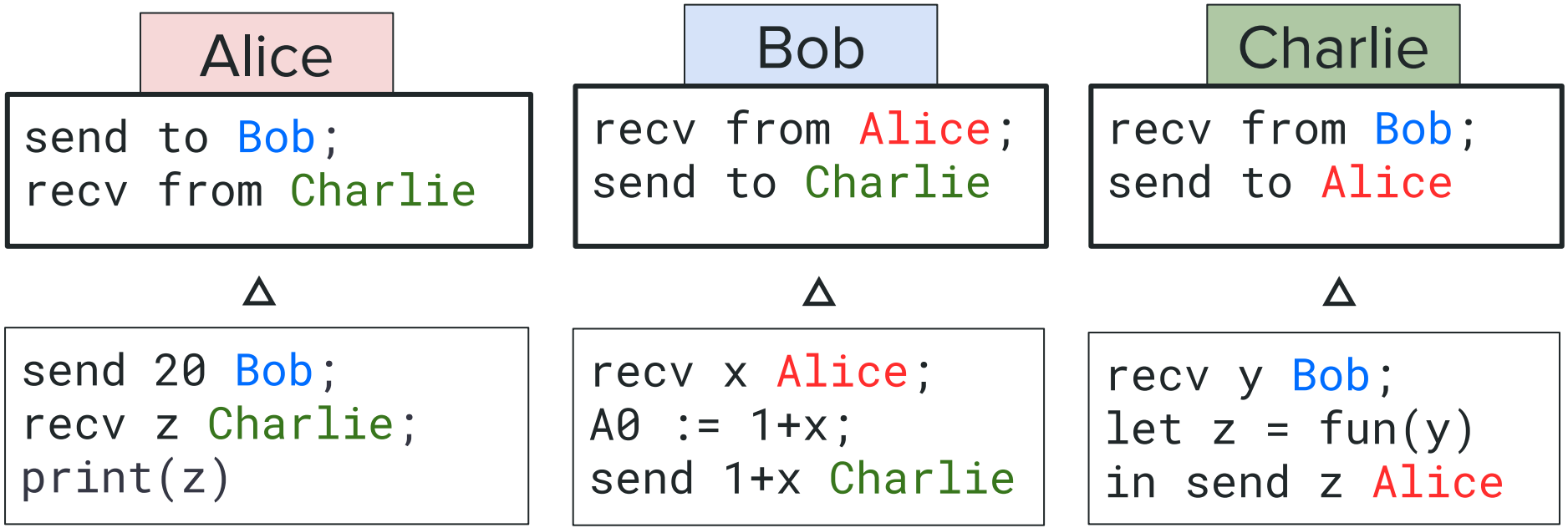
Fix: Use a **semantic** description of “consistency” between local sessions

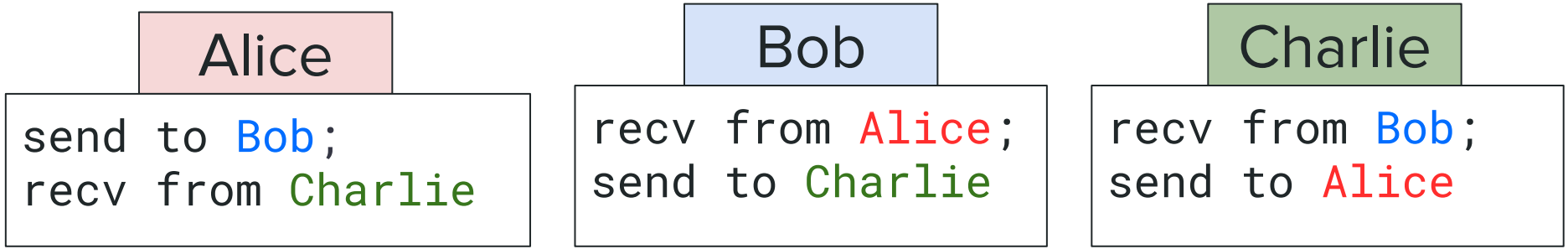
“Less is More”

Scalas & Yoshida

POPL ‘19

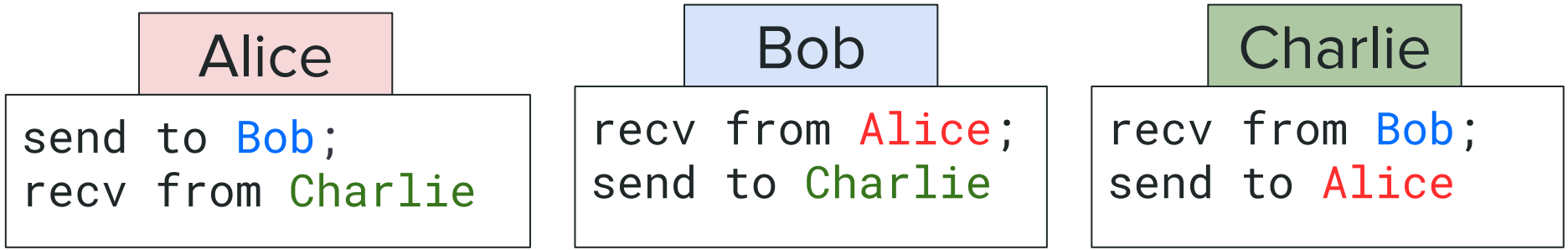
First: **abstract** the behavior of the programs as the corresponding (local) session types




Alice
send to Bob;
recv from Charlie
Bob
recv from Alice;
send to Charlie
Charlie
recv from Bob;
send to Alice

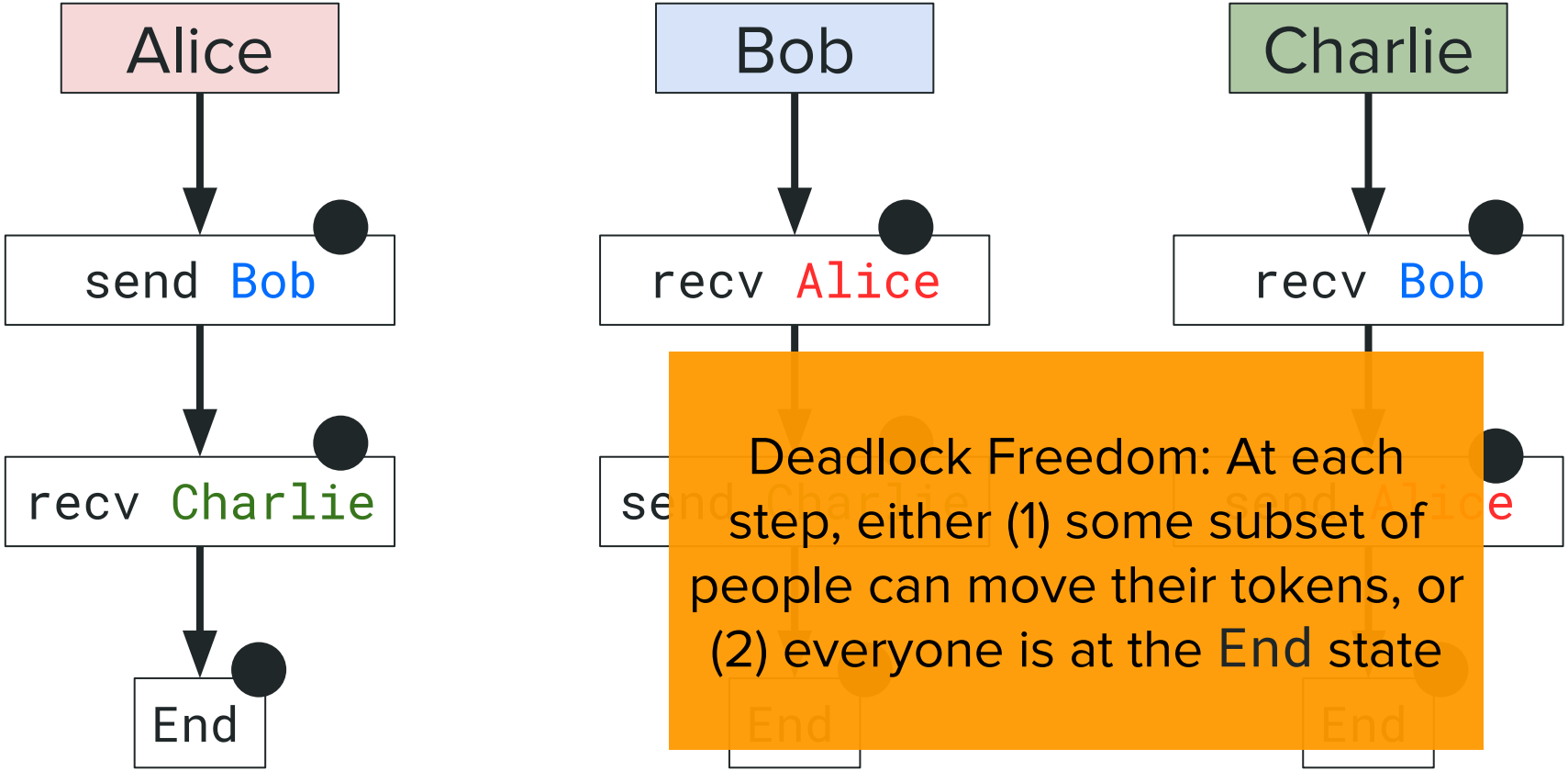
Second: Prove/check that this **abstracted system is deadlock-free**

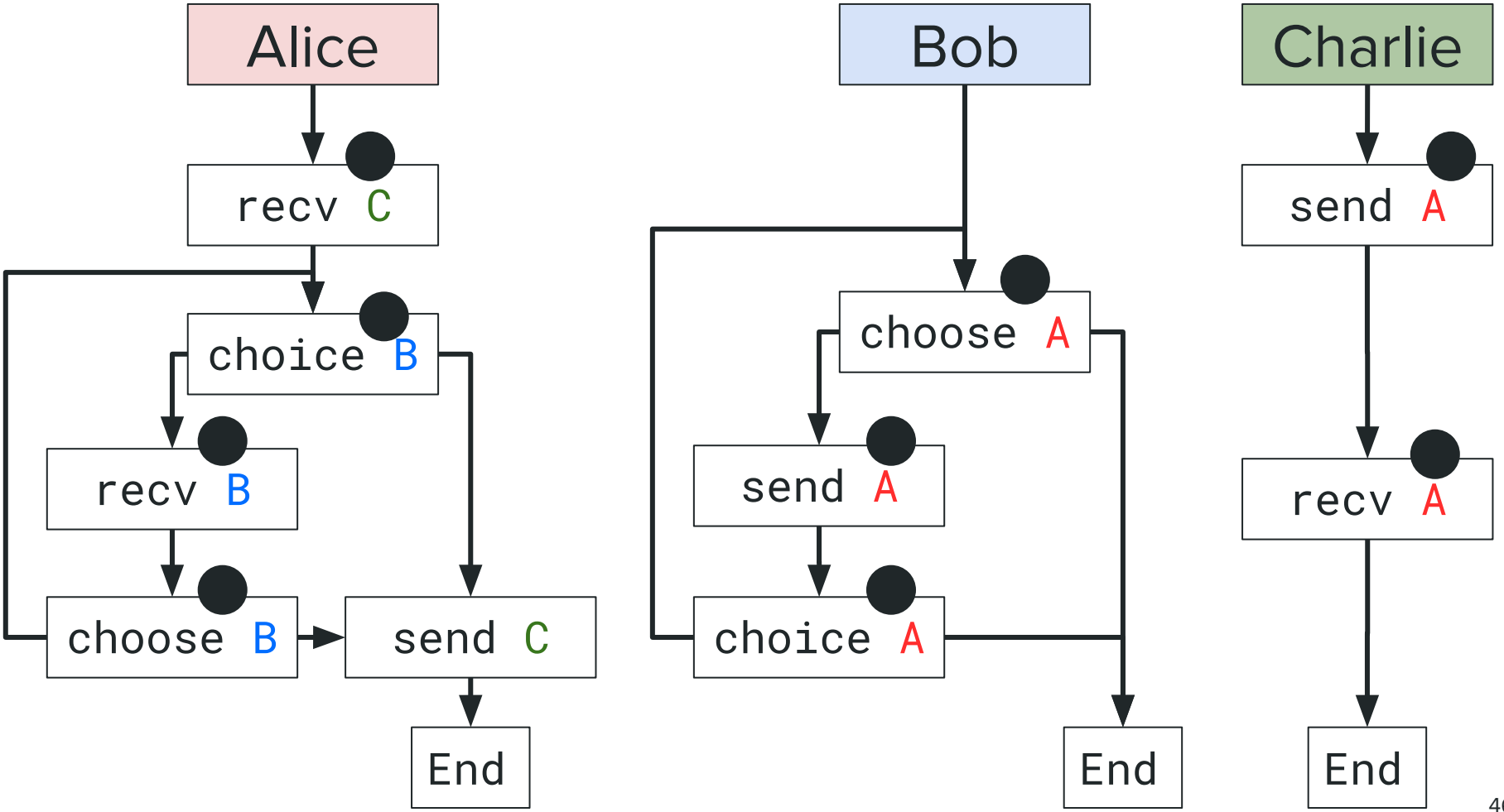
Q: How can we do this?
A: Model-checking!



Use **model-checking** to determine whether the abstracted system is deadlock-free (or has other properties like liveness, termination, etc)

Specifically, describe deadlock freedom as a **modal μ-calculus formula**, for which **model-checking is decidable**!


Alice
send Bob
recv Charlie
End
Bob
recv Alice
Charlie
recv Bob
Deadlock Freedom: At each step, either (1) some subset of people can move their tokens, or (2) everyone is at the End state


Alice
recv C
choice B
recv B
choose B
send C
End
Bob
choose A
send A
choice A
End
Charlie
send A
recv A
End

(Bonus) Deadlock Freedom in μ-calculus

```
φ ::= X | ⊥ | φ1 ∧ φ2 | ¬ φ | [a]φ | νX.φ
a ::= p → ?q | ?p → q
```

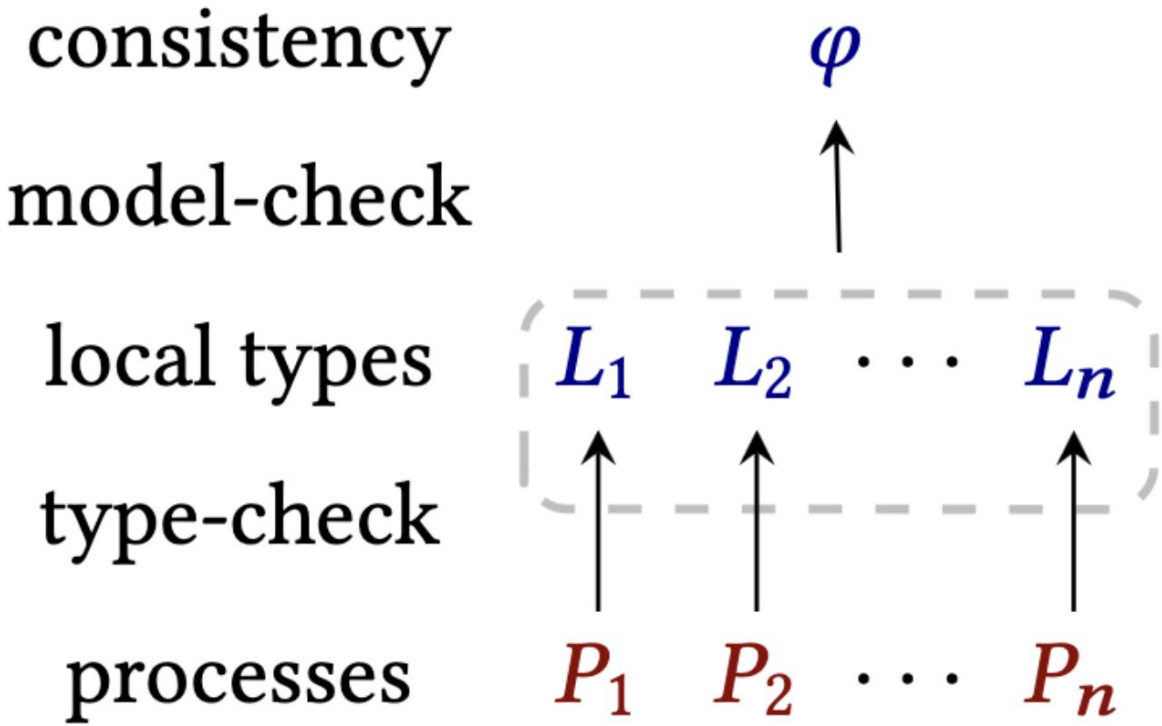
If no m… can b…

Then no one is waiting to send or receive a message

```
φDF = νX.
    ((∀p,q.[p → q]⊥)
      ∀p,q.([p → ?q]⊥ ⇒ ⟨?p → q⟩⊥))
   ∧ ∀p,q.([p → q]X)
```

The system transitions via message sends

```
DeadlockFree(S1,...,Sn) := S1,...,Sn ⊨ φDF
```



(b) “Less Is More” [31]

Drawback: The LTS to model-check is the **product of the LTS** for each participant — non compositional

Fix: *Synthetic* behavioral typing

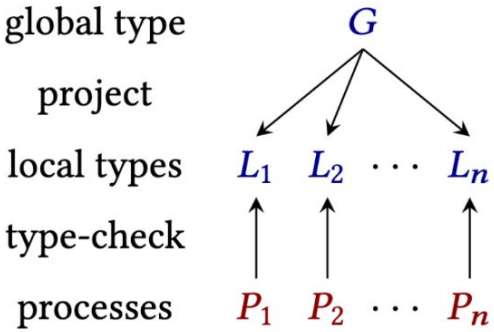
A Synthetic Reconstruction of Multiparty Session Types

Castro-Perez et al.

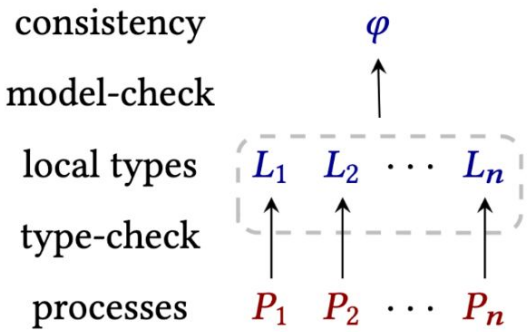
POPL ‘26

Combine the two approaches:

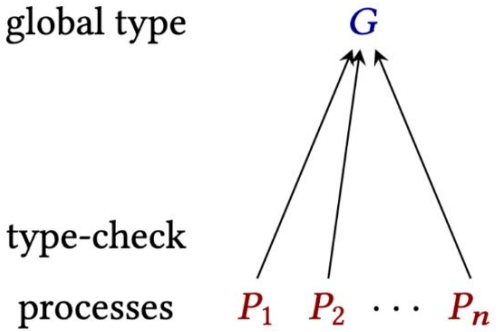
1) Define a **semantics** for global types (i.e., the overall system), and **prove it's deadlock-free**
2) Type-check each **local program** against a **global session type**



(a) Classical [17]

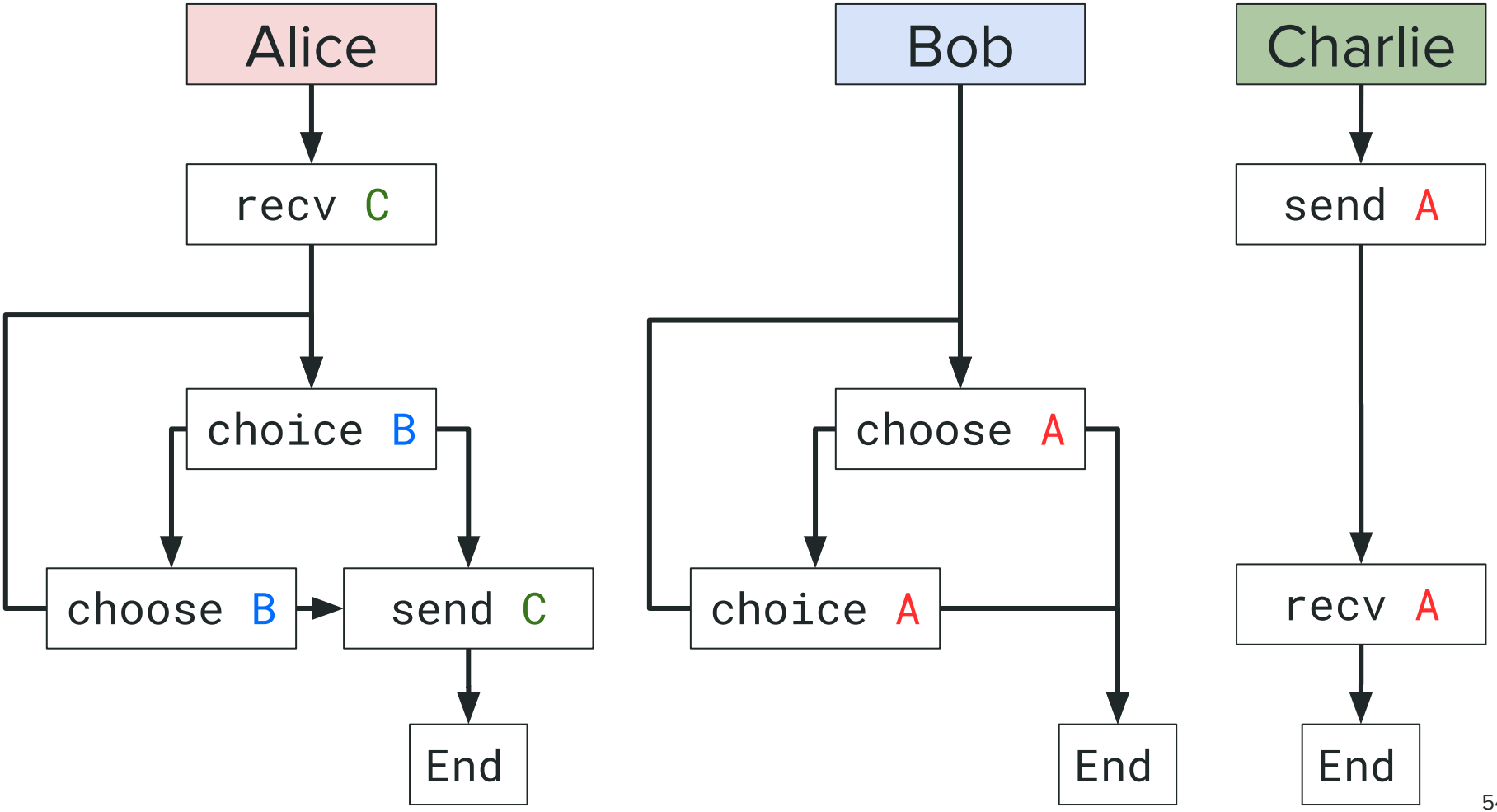


(b) "Less Is More" [31]



(c) Synthetic [this paper]

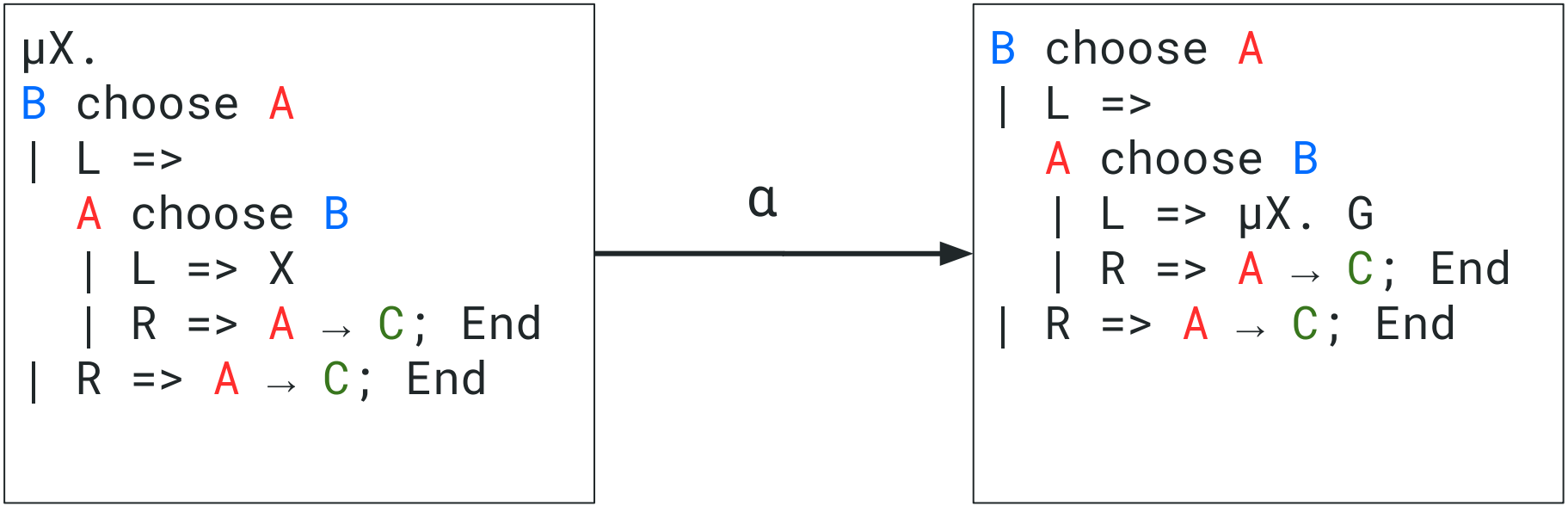
Global Type Semantics

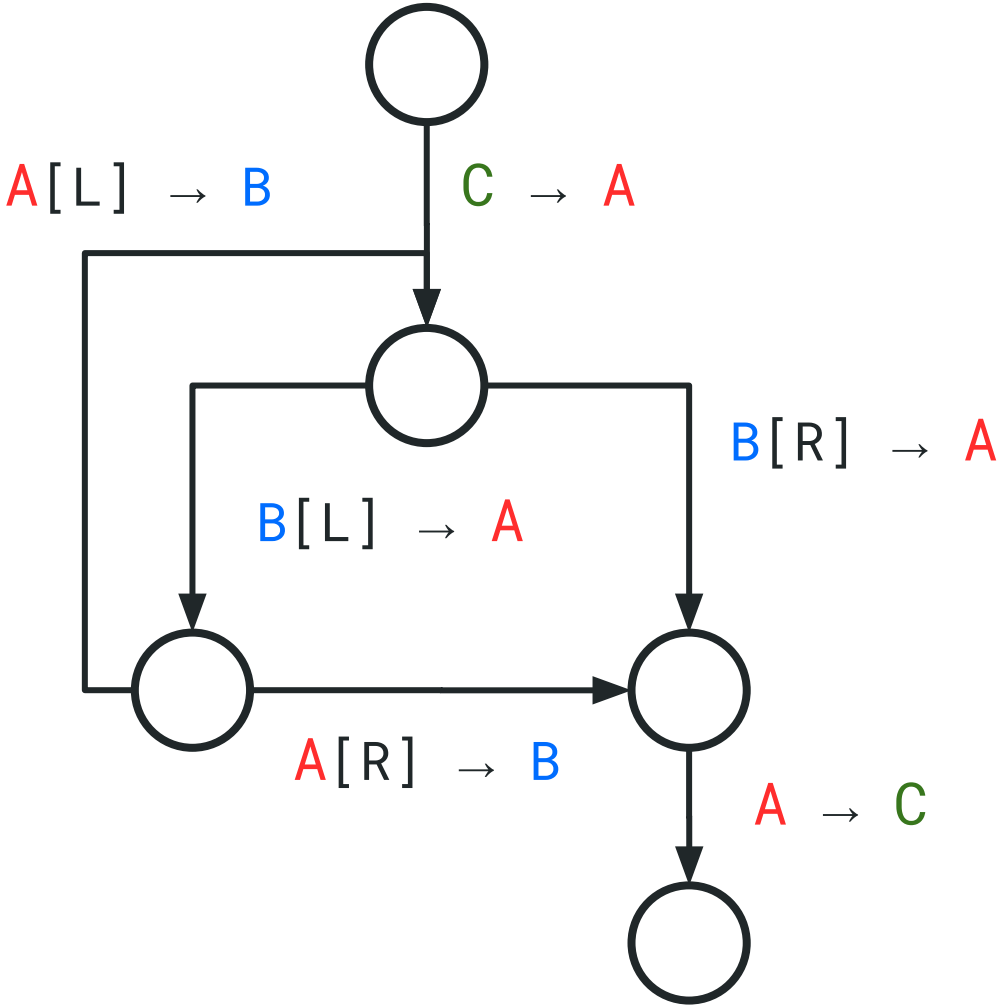

Alice
recv C
choice B
choose B
send C
End
Bob
choose A
choice A
End
Charlie
send A
recv A
End

```
C → A;
μX.
B choose A
| L =>
   A choose B
   | L => X
   | R => A → C; End
| R => A → C; End
```

```
C → A;
μX.
B choose A
| L =>
   A choose B
   | L => X
   | R => A → C; End
| R => A → C; End
```


```
μX.
B choose A
| L =>
   A choose B
   | L => X
   | R => A → C; End
| R => A → C; End
```


μX.
B choose A
| L =>
A choose B
| L => X
| R => A → C; End
| R => A → C; End
α
B choose A
| L =>
A choose B
| L => μX. G
| R => A → C; End
| R => A → C; End




```
C → A;
μX.
B choose A
| L =>
   A choose B
   | L => X
   | R => A → C; End
| R => A → C; End
```

Type-Checking

```
recv x C;
loop():
  choice B:
  | L =>
      choose (x < 0) B:
      | L => loop()
      | R => send 2*x C
  | R => send 2*x+1 C
```

$\triangleright_A$

```
C → A;
μX.
B choose A
| L =>
    A choose B
    | L => X
    | R => A → C; End
| R => A → C; End
```

```
loop():
  choice B:
  | L =>
     choose (x < 0) B:
     | L => loop()
     | R => send 2*x C
  | R => send 2*x+1 C
```

$\triangleright_A$

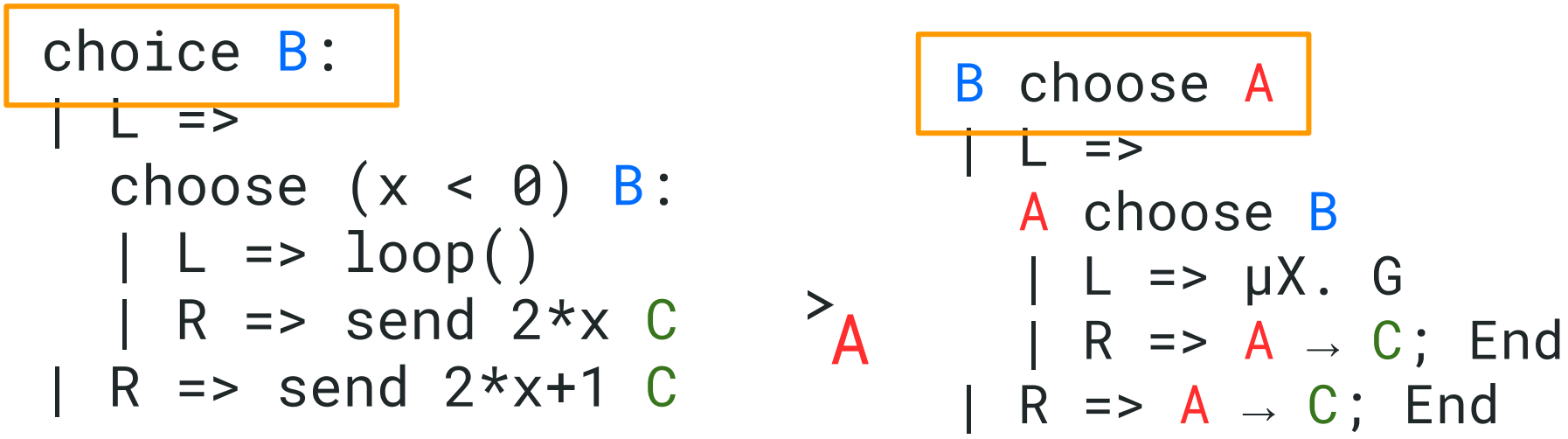
```
μX.
B choose A
| L =>
   A choose B
   | L => X
   | R => A → C; End
| R => A → C; End
```

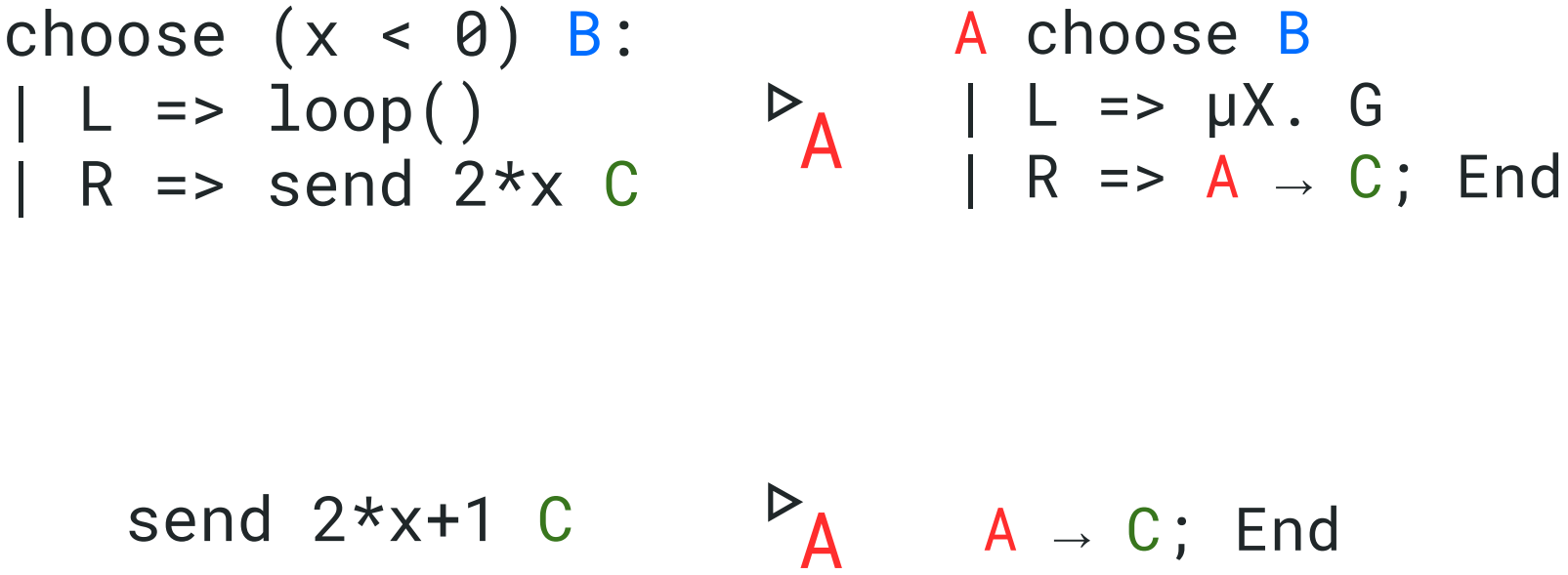
```
choice B:                    ?
| L =>
    choose (x < 0) B:
    | L => loop()
    | R => send 2*x C
| R => send 2*x+1 C
```

$\not>$ A

```
μX.
B choose A
| L =>
    A choose B
    | L => X
    | R => A → C; End
| R => A → C; End
```

Allowed to step semantics in type-checking!


choice B:
| L =>
choose (x < 0) B:
| L => loop()
| R => send 2*x C
| R => send 2*x+1 C
>A
B choose A
| L =>
A choose B
| L => μX. G
| R => A → C; End
| R => A → C; End


choose (x < 0) B:
| L => loop()
| R => send 2*x C
▷A
A choose B
| L => μX. G
| R => A → C; End
send 2*x+1 C
▷A
A → C; End

```
send 20 A;
recv y A;
print(y)
```

$\triangleright C$

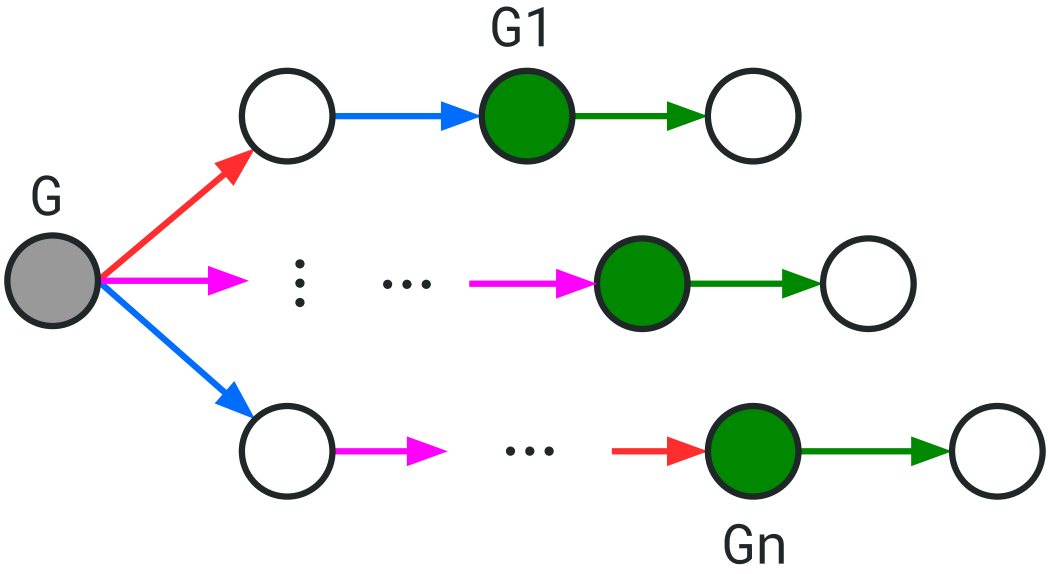
```
C → A;
μX.
B choose A
| L =>
    A choose B
    | L => X
    | R => A → C; End
| R => A → C; End
```

```
recv y A;
print(y)
```

$\triangleright C$

```
μX.
B choose A
| L =>
    A choose B
    | L => X
    | R => A → C; End
| R => A → C; End
```

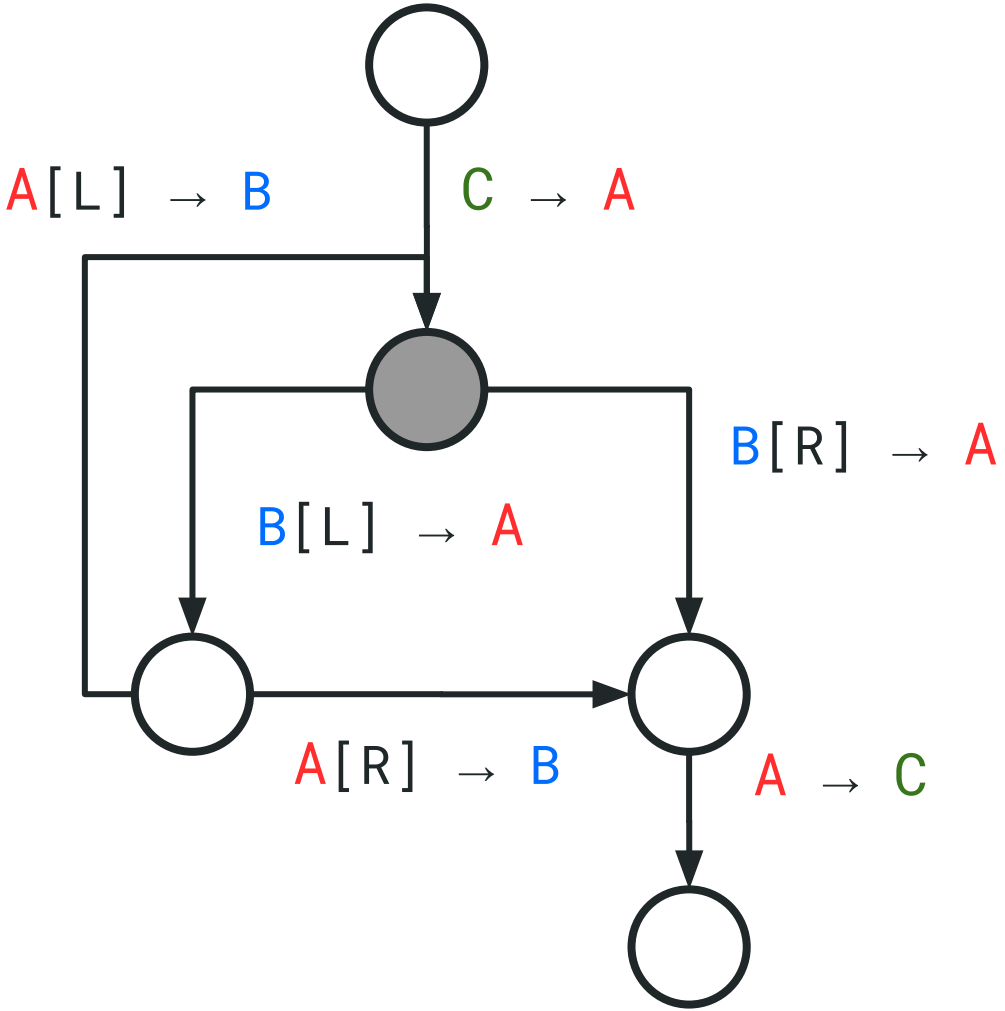
Unfold all possibilities
until C participates



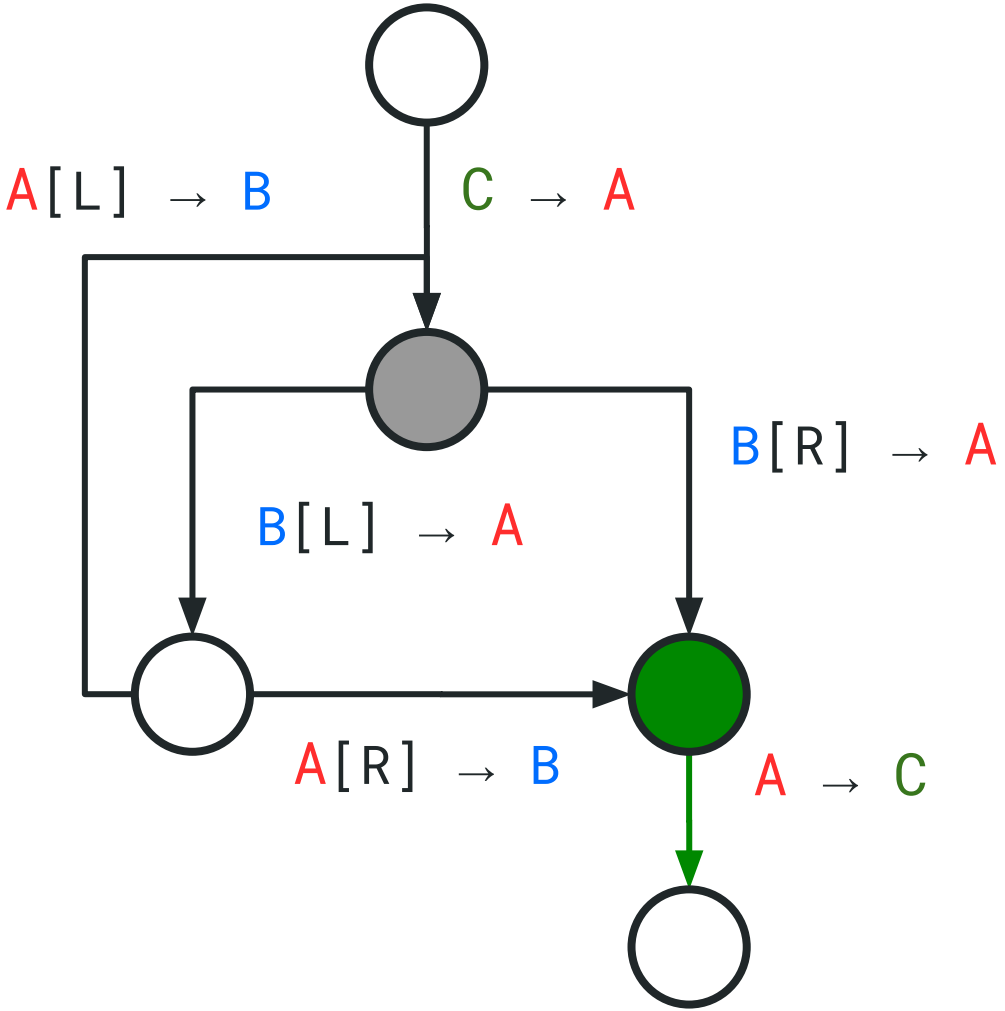
To type-check program P at state G for process ■ (where *no next-step* of G involves ■), we need to type-check P at *each* Gi for ■

$$\frac{(1)\ G \xrightarrow{\{r\}} \not\to \qquad (2)\ \forall G \xrightarrow{\overline{\{r\}}}{}^{*} G' .\ [\exists G'' .\ [G' \overset{\overline{\{r\}}}{\Longrightarrow}{}^{*} G'' \xrightarrow{\{r\}}]] \qquad (3)\ \forall G = G' \overset{\overline{\{r\}}}{\Longrightarrow}{}^{*} G'' \xrightarrow{\{r\}} .\ [\Gamma; \Delta \vdash \mathsf{r} \triangleleft P : G''] \qquad (4)\ \forall G \xrightarrow{\overline{\{r\}}}{}^{*} G' .\ [G' \xrightarrow{\{r\}} \vee\ G' \xrightarrow{\overline{\{r,\mathrm{obj}(P)\}}}{}^{*} \xrightarrow{\{r,\mathrm{obj}(P)\}}]}{\Gamma; \Delta \vdash \mathsf{r} \triangleleft P : G}\ [\vdash\text{-Skip}]$$

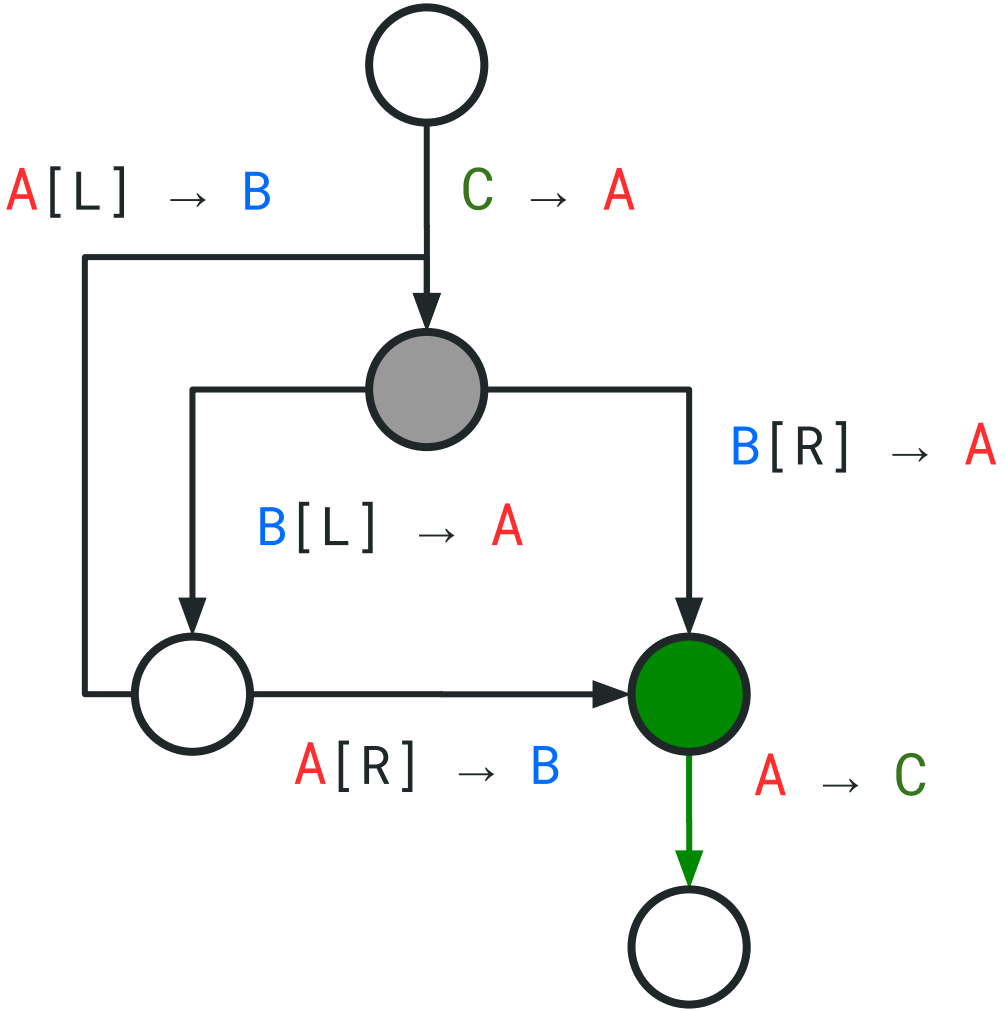
1) G *currently* doesn't specify an action for r
2) An action by r is *eventually* specified by G
3) P must ignore all actions it isn't a part of, and type-check at *all possible future states* of G involving r
4) Communication cannot happen *spontaneously*




```
C → A;
μX.
B choose A
| L =>
   A choose B
   | L => X
   | R => A → C; End
| R => A → C; End
```




```
C → A;
μX.
B choose A
| L =>
   A choose B
   | L => X
   | R => A → C; End
| R => A → C; End
```




```
C → A;
μX.
B choose A
| L =>
   A choose B
   | L => X
   | R => A → C; End
| R => A → C; End
```

```
recv y A;
print(y)
```

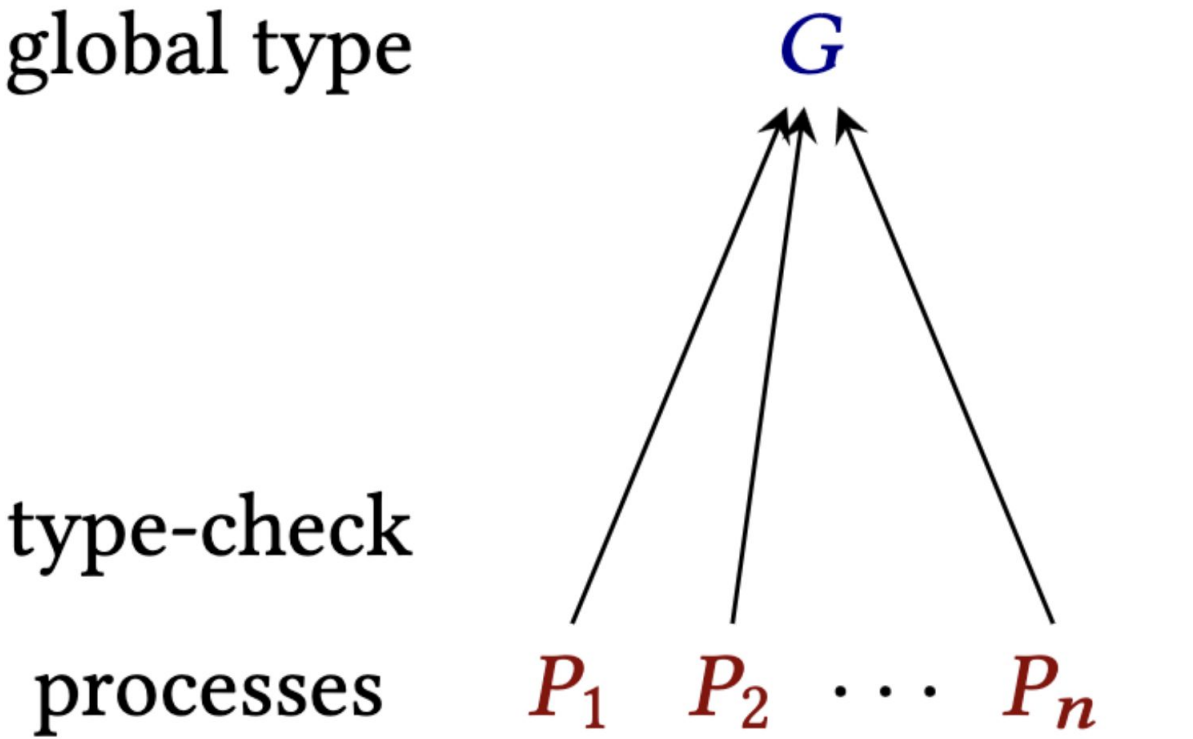
$\triangleright C$

```
μX.
B choose A
| L =>
    A choose B
    | L => X
    | R => A → C; End
| R => A → C; End
```

```
recv y A;
print(y)
```

$\triangleright C$

$A \rightarrow C; \mathrm{End}$



(c) Synthetic [this paper]

In fact, the type system will be sound if we replace “G” with **any LTS** with the appropriate **semantic properties**!

Fixed the state-explosion problem by allowing **“optimized” LTSs** (e.g., global types), rather than the product LTS!

Drawback: To decide type-checking, need to be able to compute the next time (if at all) a process will participate in the LTS; **complex protocols have slow type-checking**, and it **cannot support Turing-complete protocols**

Fix: Choreographic programming 😉

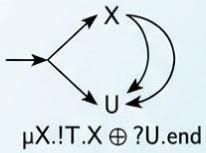
Thoughts?

Questions?

STOP DOING SESSION TYPES

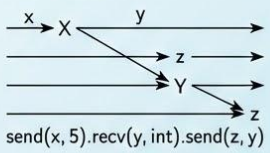
- INTERACTIONS WERE NOT SUPPOSED TO BE GIVEN NAMES
- YEARS OF CONCURRENCY THEORY yet NO REAL-WORLD USE FOUND for going higher than basic SEND/RECEIVE
- Wanted to go higher anyway for a laugh? We had a tool for that: It was called "GUESSING (ASYNC)"
- "Yes please give me a SESSION OF ZERO DURATION. Please give me an INFINITE SESSION OF INFINITE BRANCHES" - Statements dreamed up by the utterly Deranged

LOOK at what Researchers have been demanding your Respect for all this time, with all the type checkers & protocols we built for them **(This is REAL Session Types, done by REAL Concurrency Theorists)**:



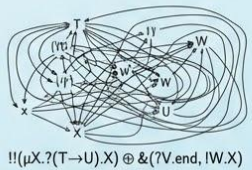
μX.!T.X ⊕ ?U.end

???



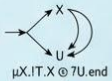
send(x, 5).recv(y, int).send(z, y)

???????



!!(μX.?(T→U).X) ⊕ &(?V.end, !W.X)

??????????????????

"Hello I would like a  interaction please"

μX.!T.X ⊕ ?U.end

They have played us for absolute fools

Generated with Gemini