Choreographic Quick Changes: First-Class Location (Set) Polymorphism

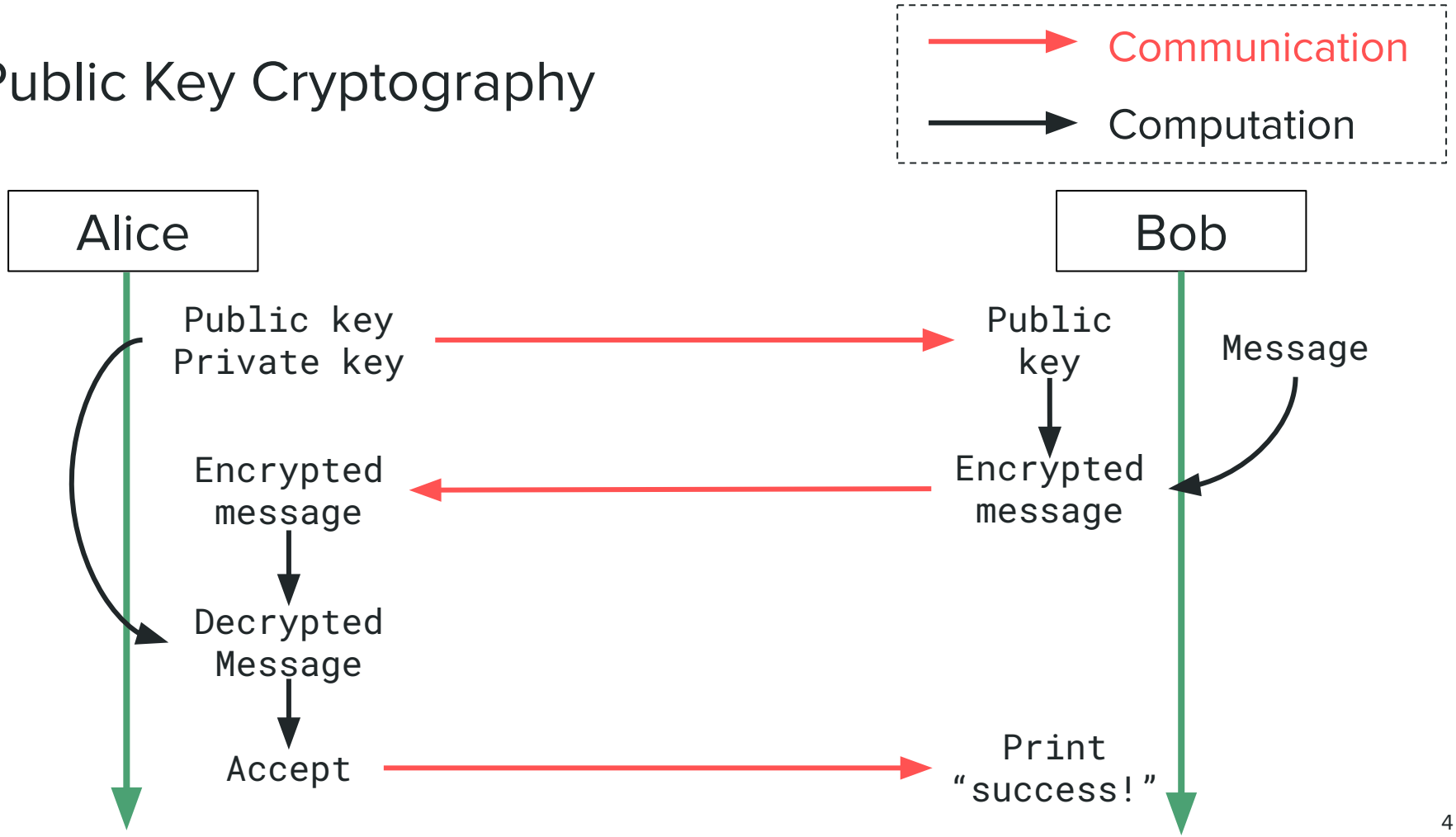
Ashley Samuelson, Ethan Cecchetti, Andrew K. Hirsch
OOPSLA'25

Choreographic Programming

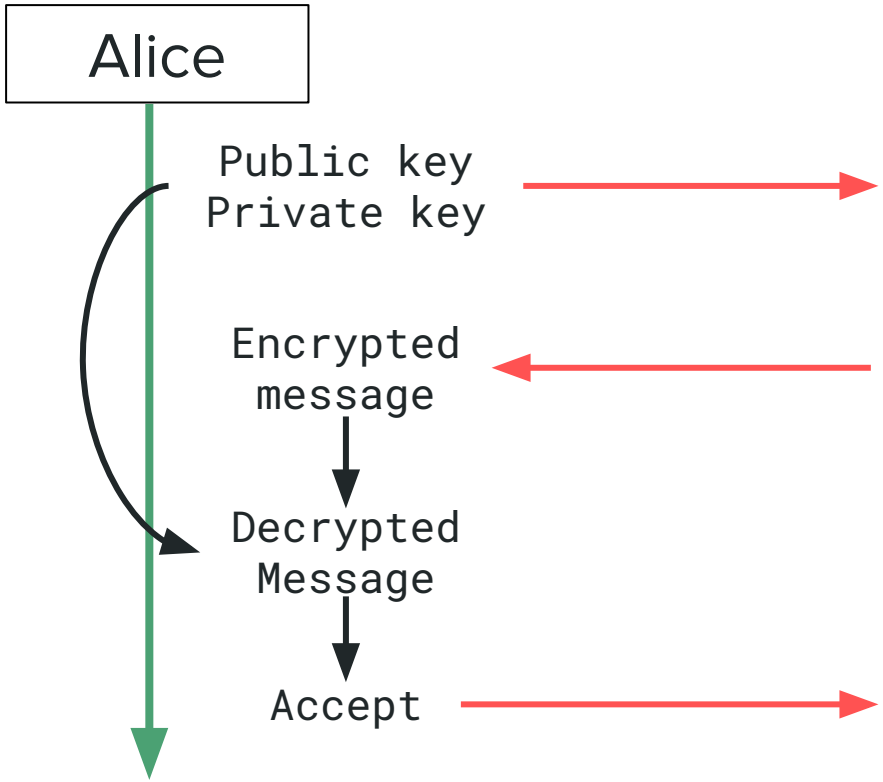
An emerging paradigm for safe concurrent programming

What does concurrent programming typically look like?

Public Key Cryptography



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



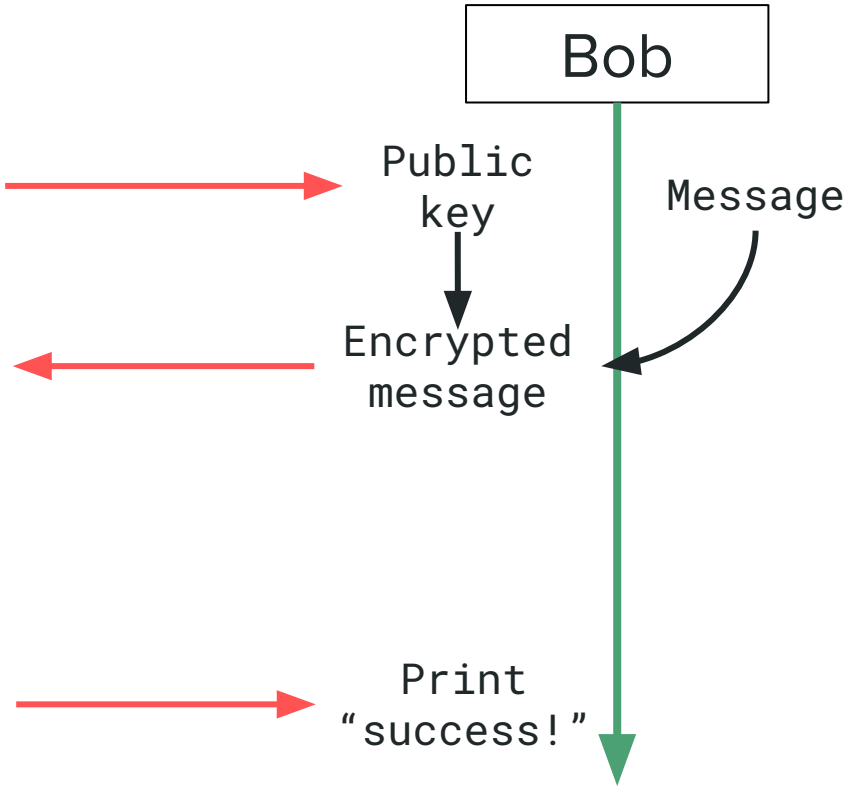
Alice

```
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
```

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

Bob

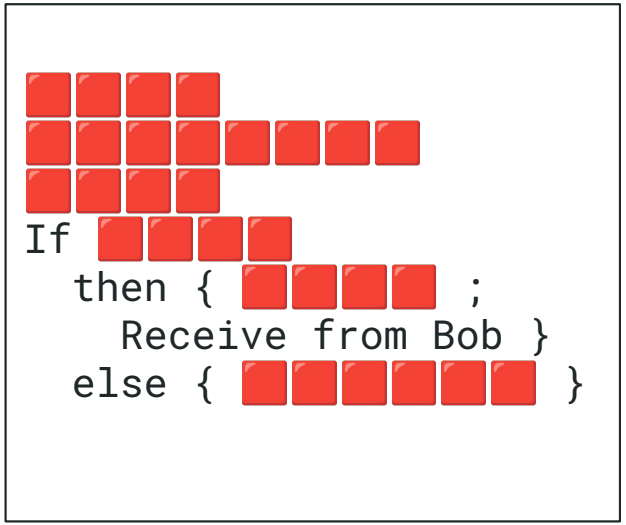
```
Receive public key from Alice
Encrypt message
Send encrypted message to Alice
Receive choice from Alice
    If True print “success”
    Else print “failure”
```



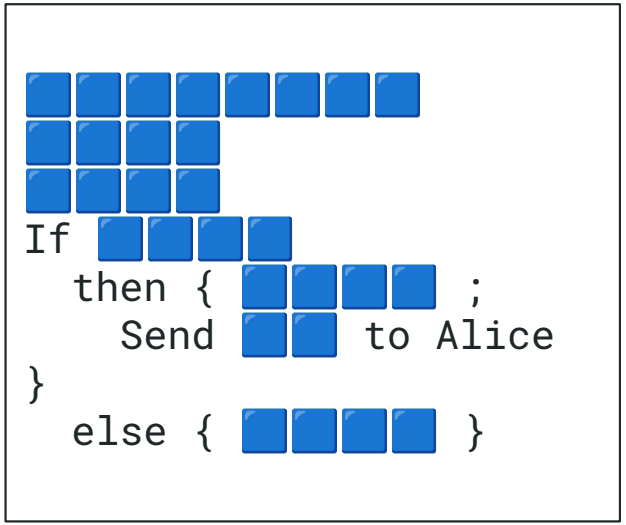
Why is concurrent programming difficult?

Changes to **one participant's code** must be **reflected in** the code of **other participants**

Alice

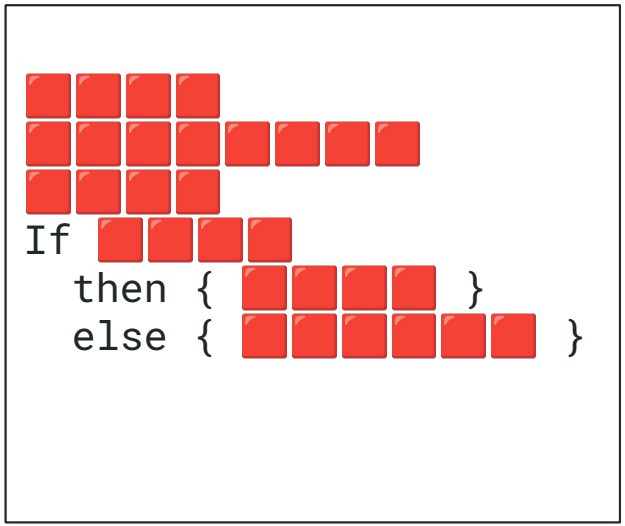


Bob

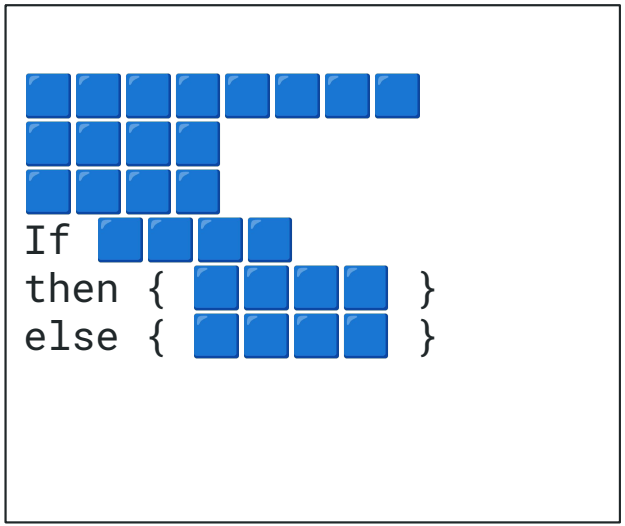


It can be easy to mistakenly introduce **deadlocks**

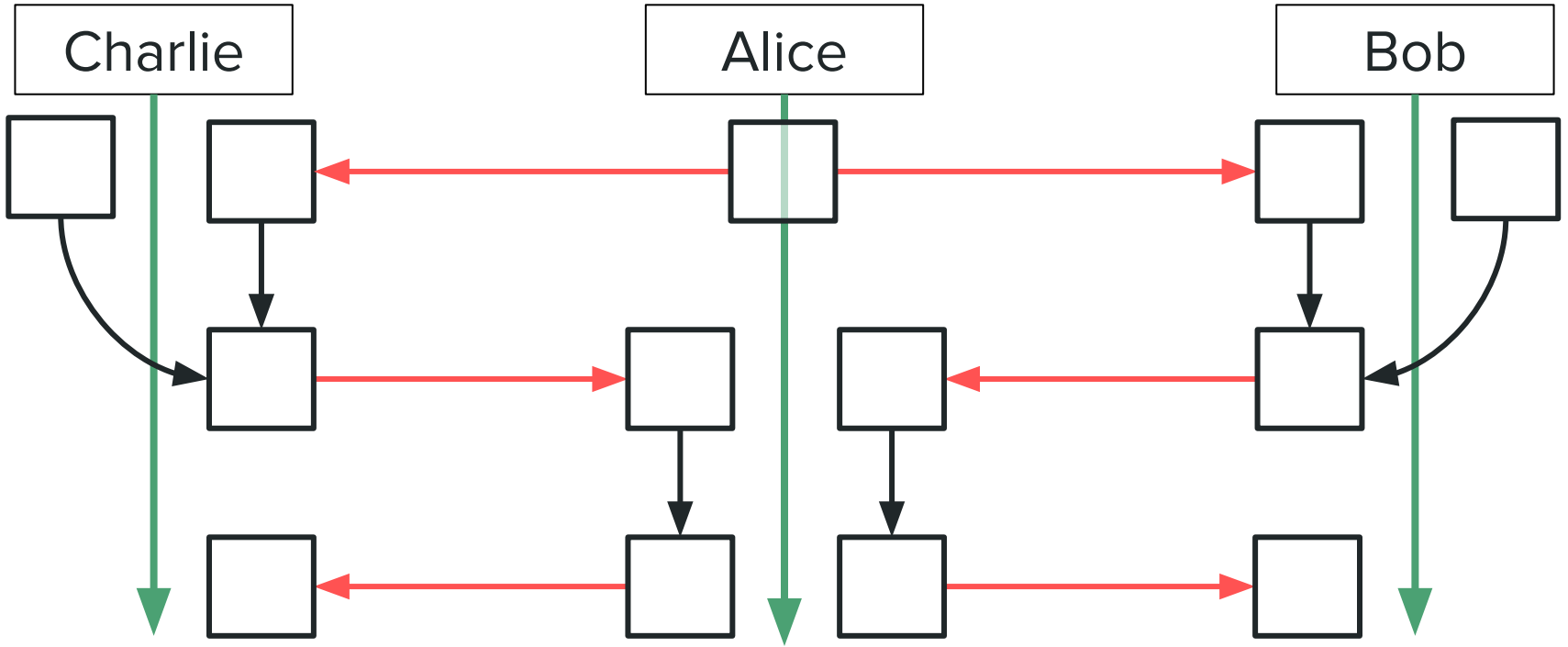
Alice ⏱️ 🤔



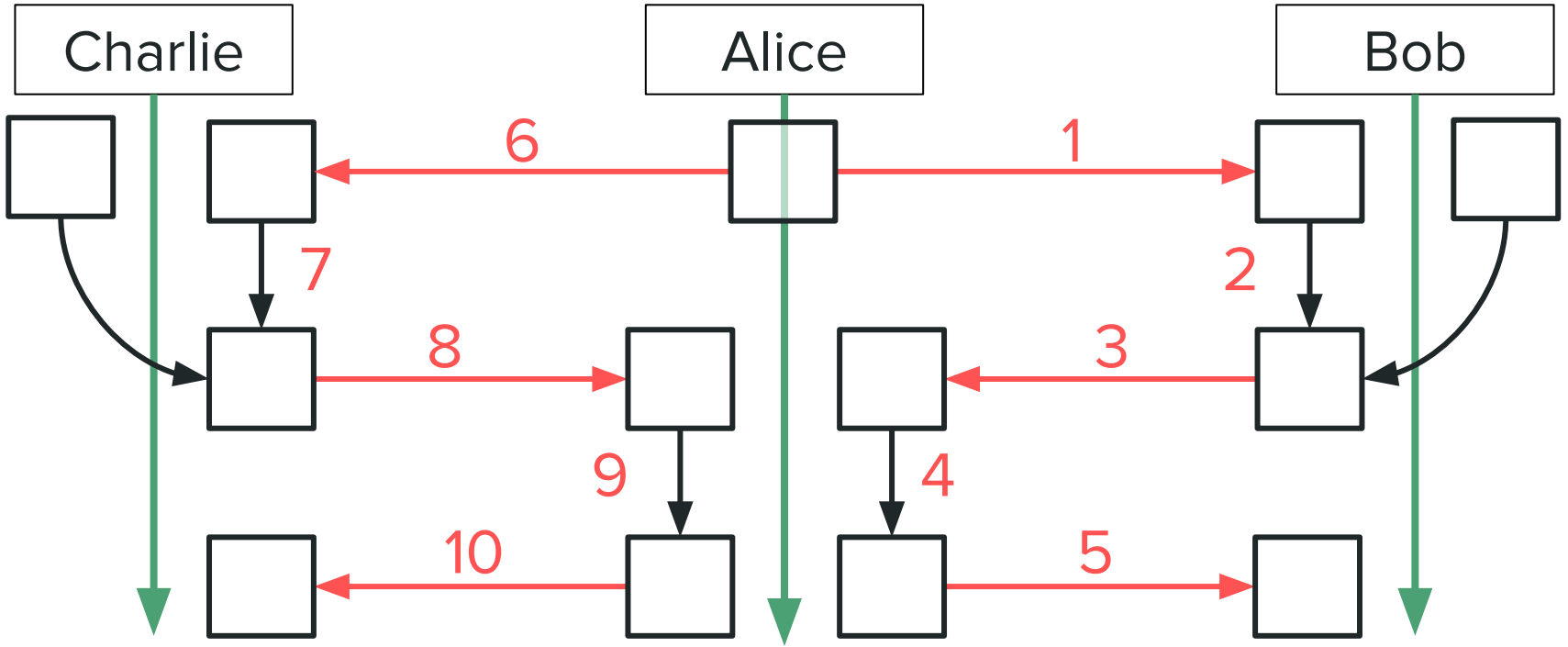
Bob 😕



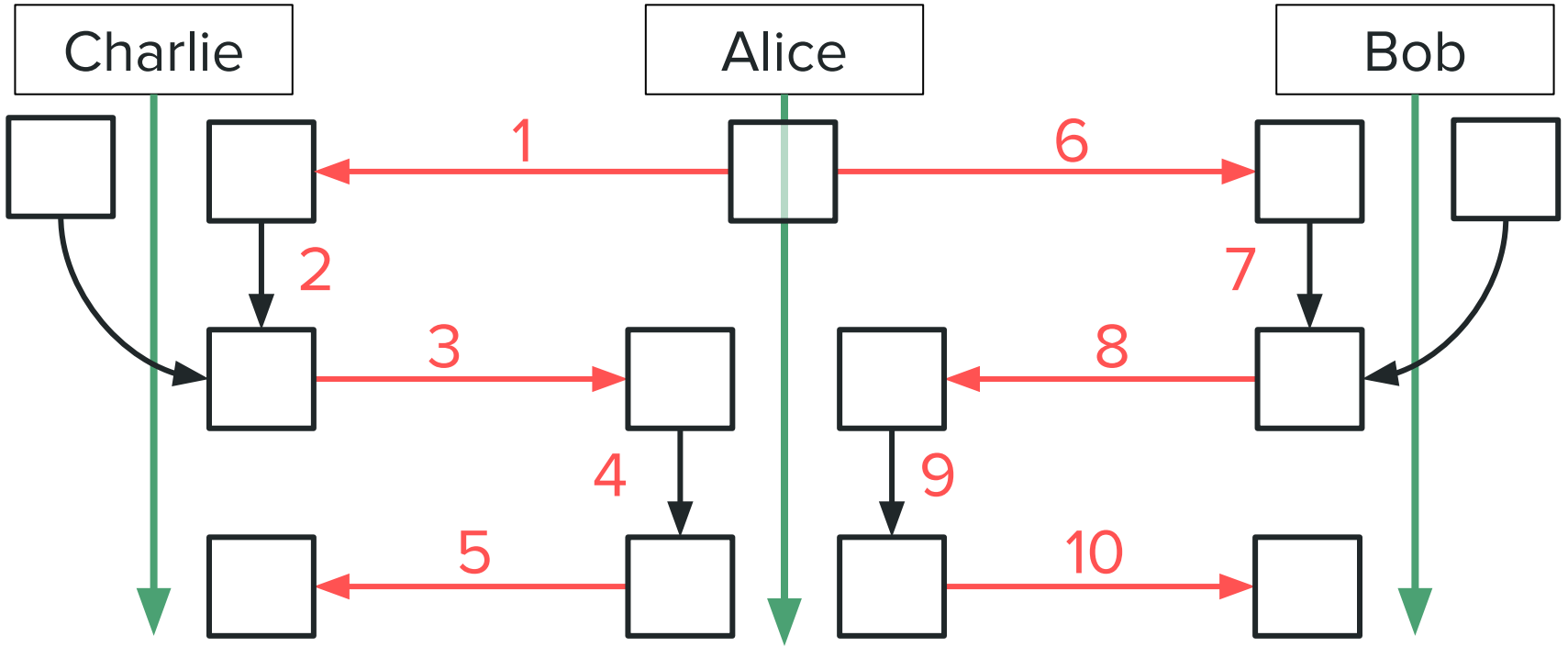
More participants makes analysis more **complex**



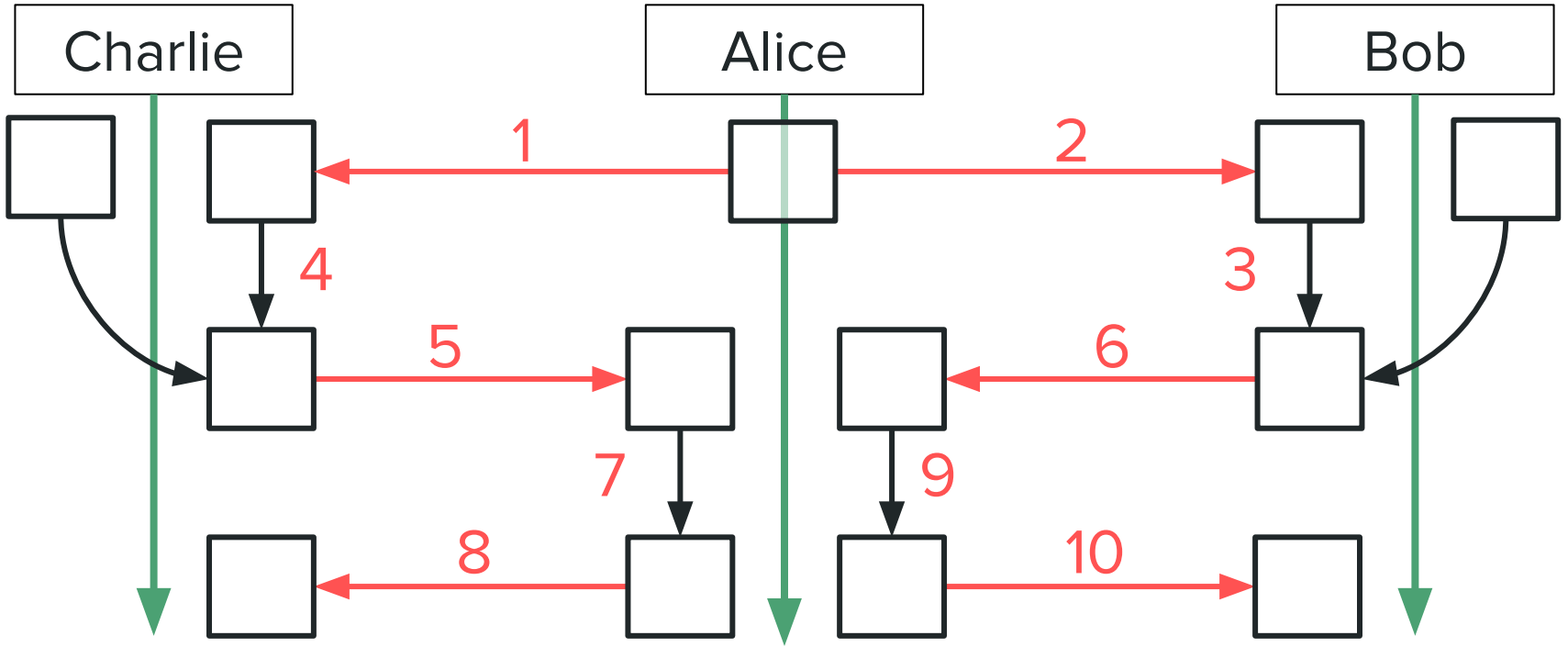
To prevent issues like **deadlocks**, we need to be able to reason about **all possible event orders**



To prevent issues like **deadlocks**, we need to be able to reason about **all possible event orders**

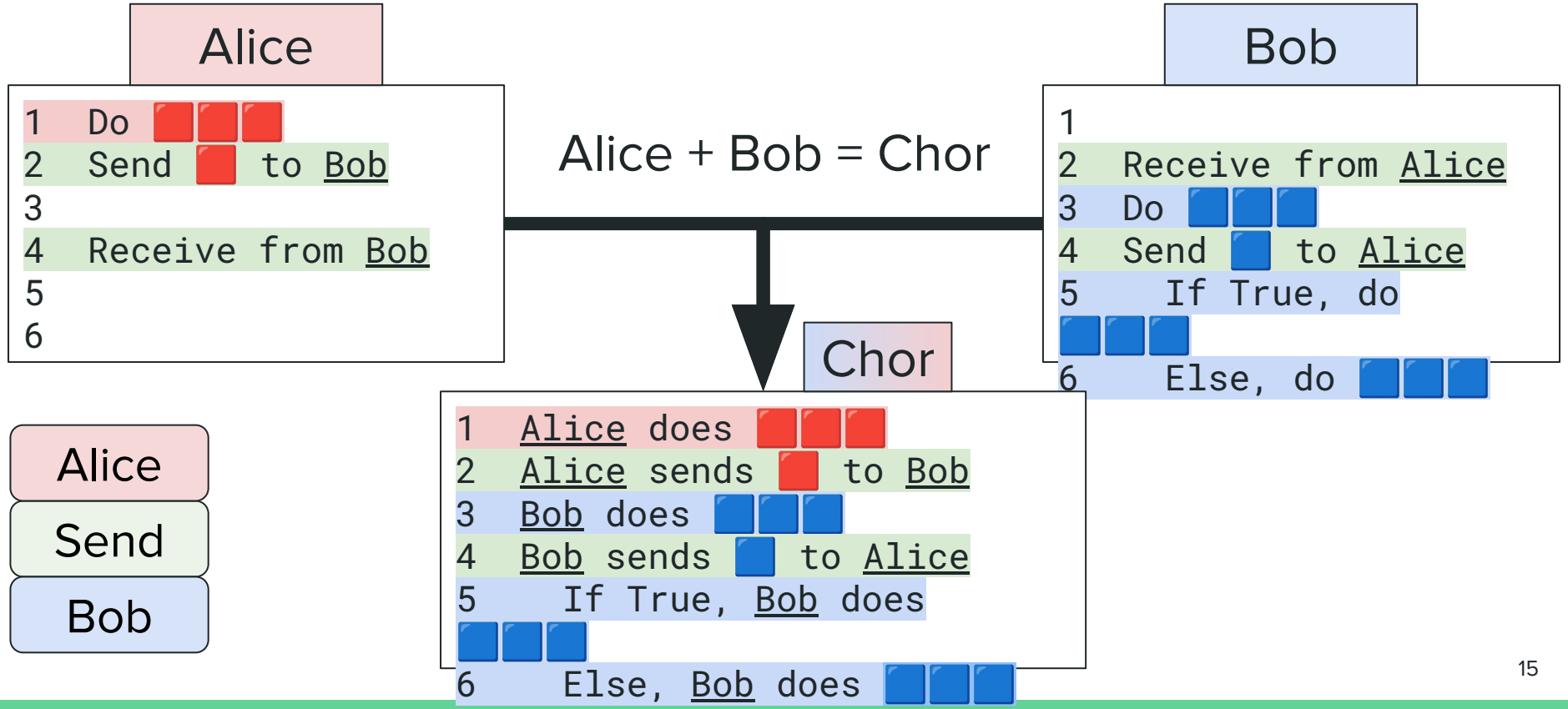


To prevent issues like **deadlocks**, we need to be able to reason about **all possible event orders**

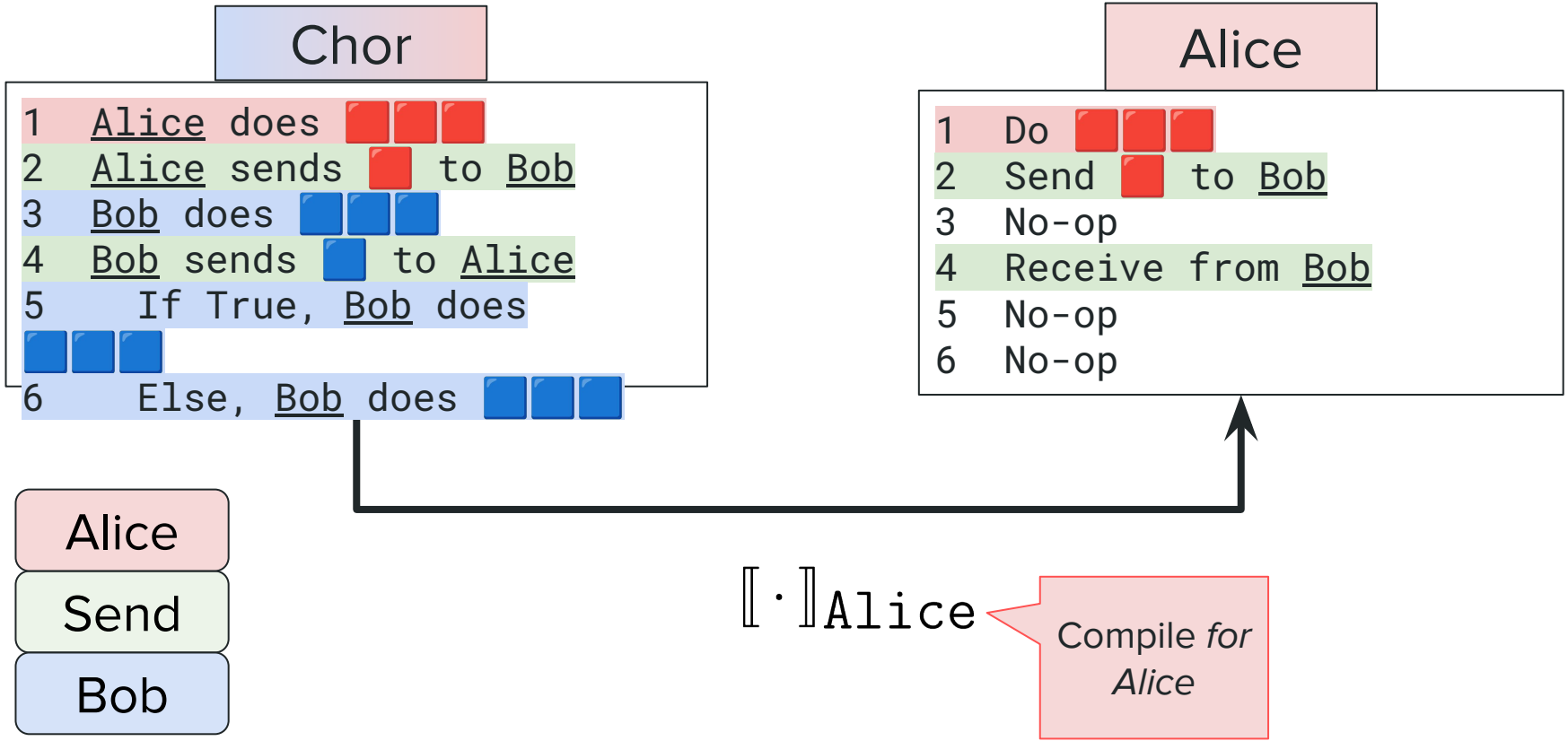


How does choreographic programming solve this?

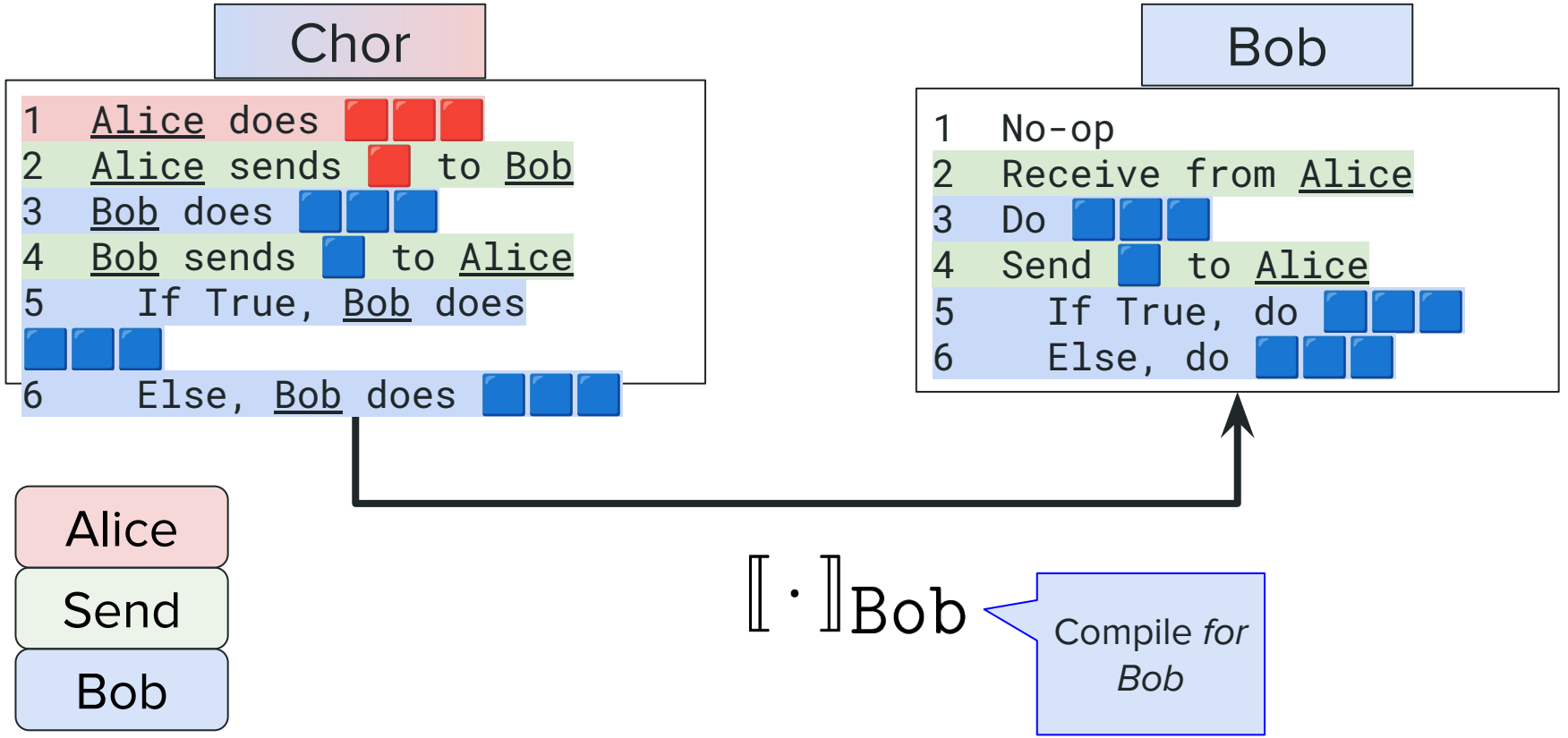
Choreographies are a **single program** which specifies the actions of **all participants**



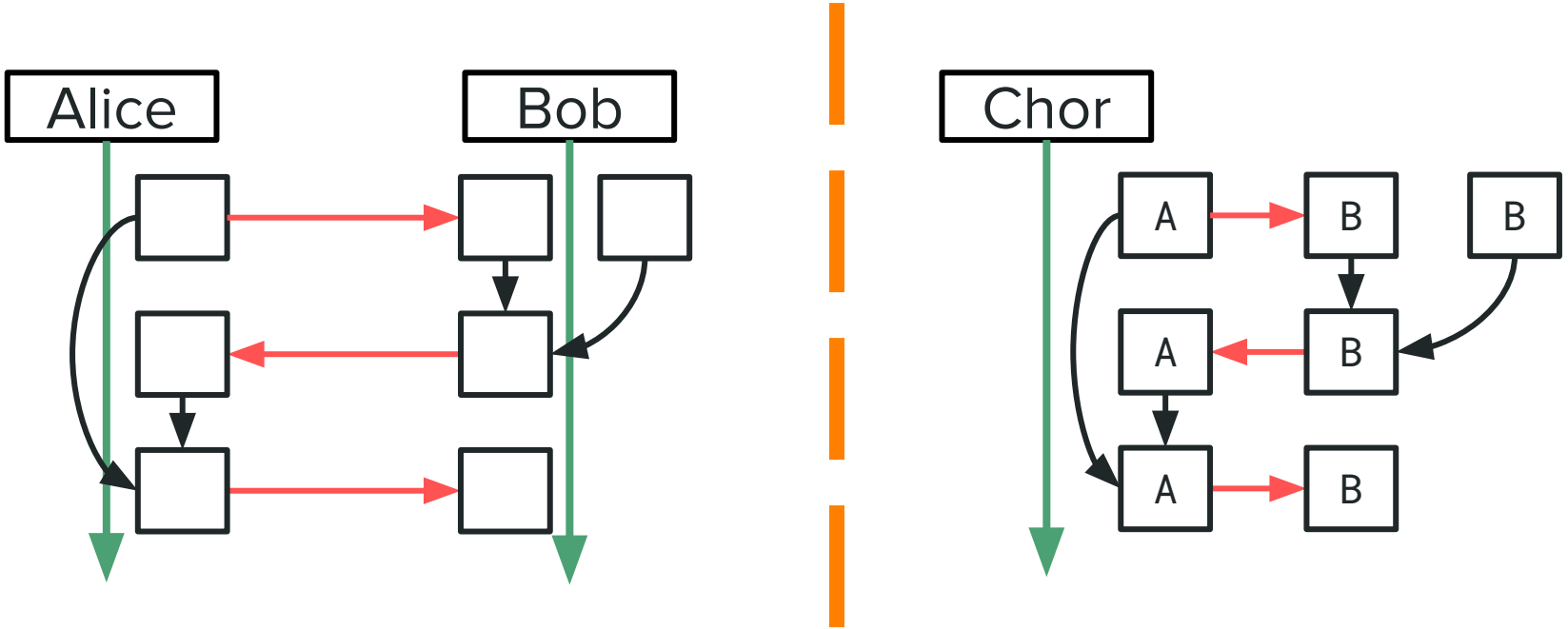
We can determine the actions of each participant with a **compilation procedure** called **endpoint projection**



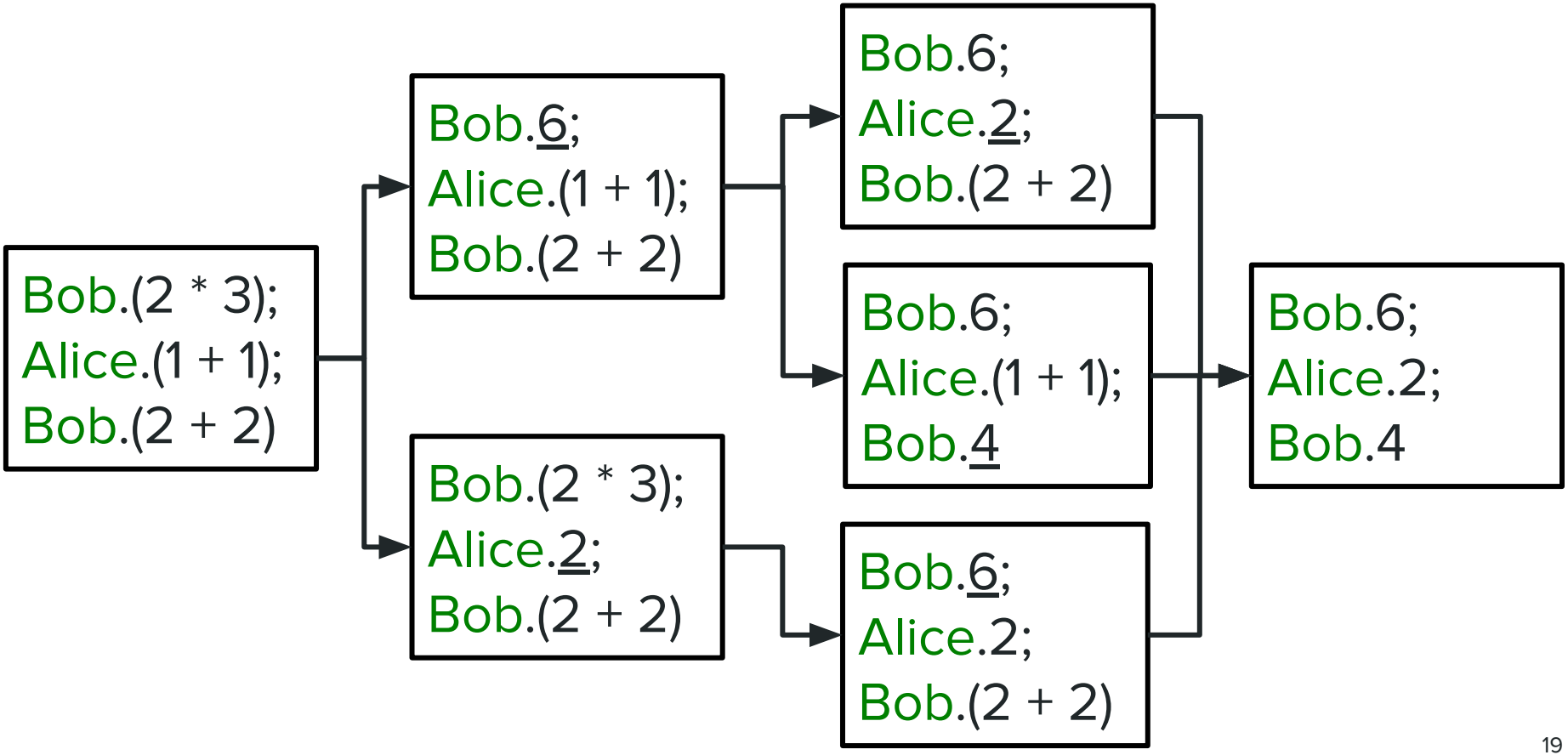
We can determine the actions of each participant with a **compilation procedure** called **endpoint projection**



We can define a **semantics for choreographies** which **simulates the behavior** of a **concurrent system** and **captures all possible event orders**

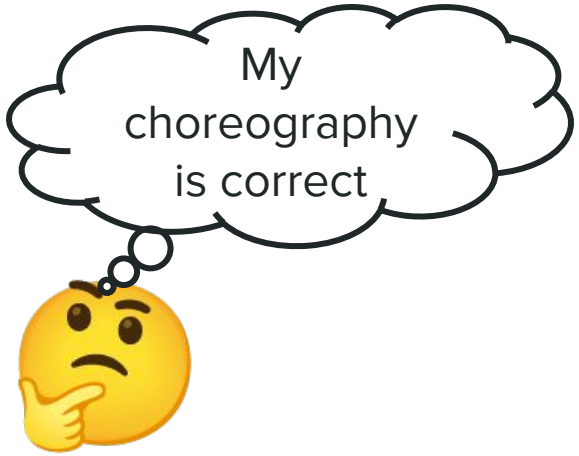


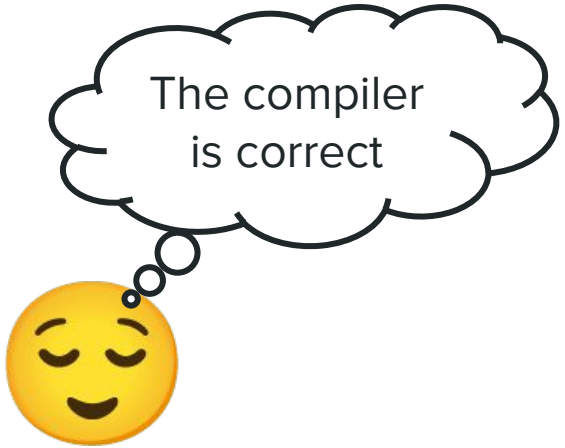
How can we capture all possible event orders?

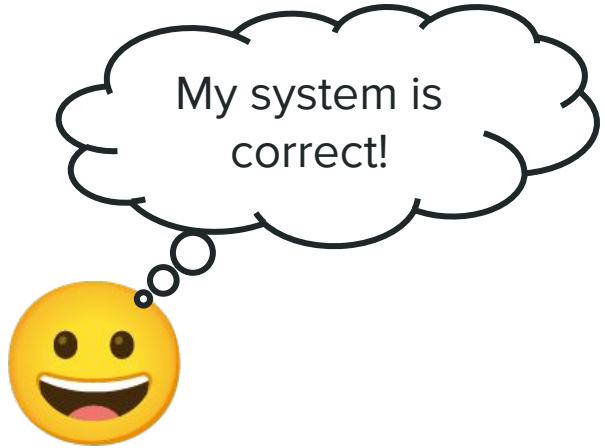


Using a separate **choreographic semantics** allows us to **analyze the behavior** of **concurrent systems**

$$\{\varphi\}C\{\psi\} \quad + \quad C \sim [\![C]\!] \quad = \quad \{\varphi\}[\![C]\!]\{\psi\}$$

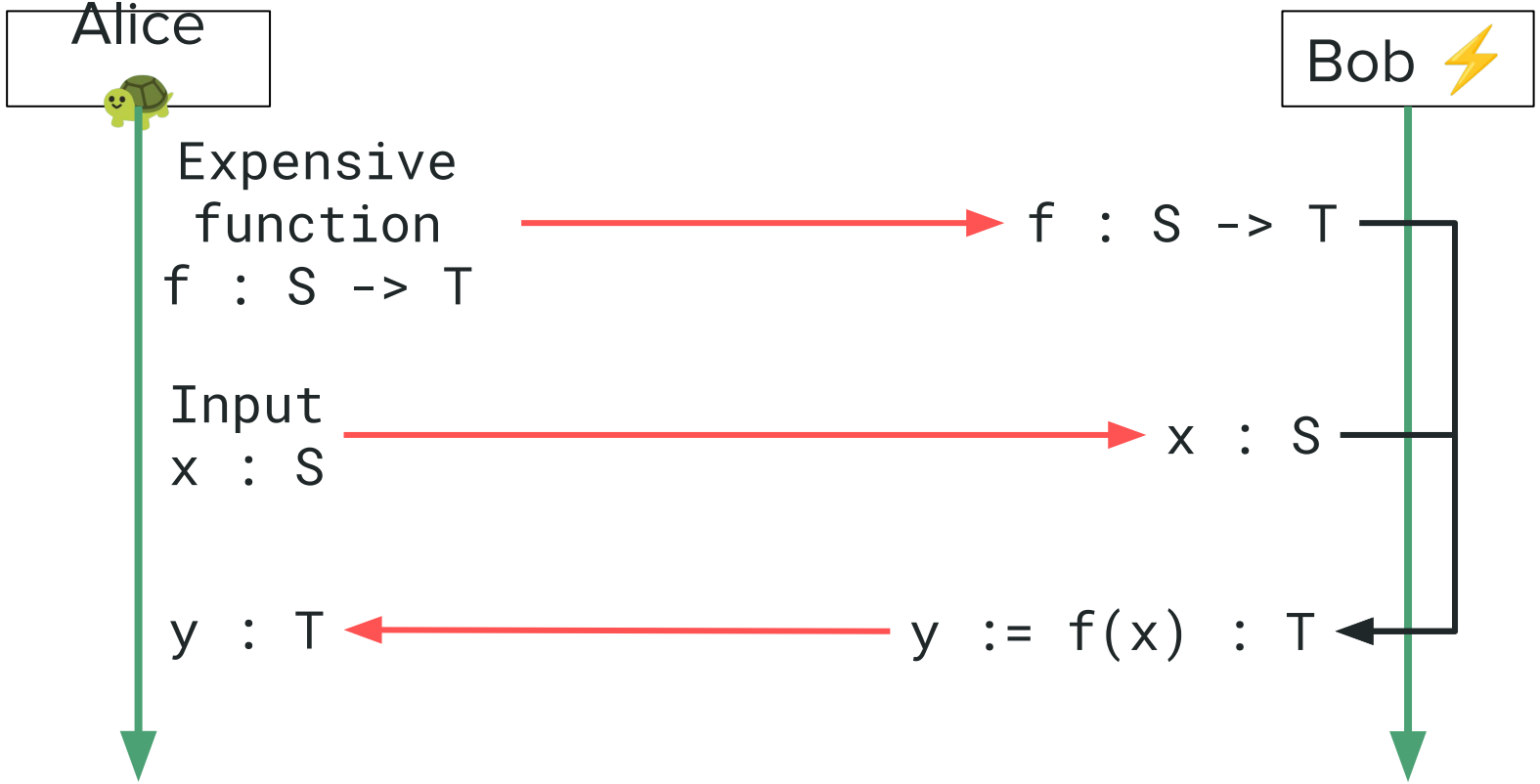






First-Class Location Names

Prior choreographies can handle a **static client-worker pattern**



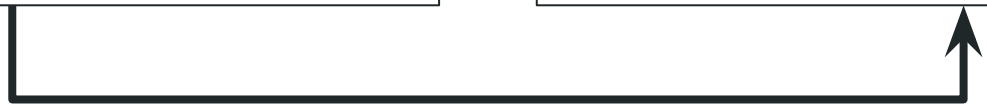
Prior choreographies can handle a **static client-worker pattern**

Chor

```
fun runAtWorker(
  F : (T1 → T2) @ Alice,
  X : T1 @ Alice) :=
 let Bob.f := F ~> Bob
     Bob.x := X ~> Bob
 in Bob.f(x) ~> Alice
```

Alice

```
fun runAtWorker(
  F : T1 → T2,
  X : T1) :=
 send F to Bob;
 send X to Bob;
 recv from Bob
```



$[\![\cdot]\!]_{\mathtt{Alice}}$

Prior choreographies can handle a **static client-worker pattern**

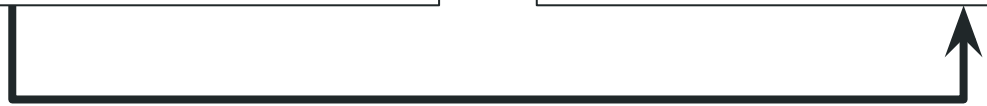
Chor

```
fun runAtWorker(
  F : (T1 → T2) @ Alice,
  X : T1 @ Alice) :=
 let Bob.f := F ~> Bob
     Bob.x := X ~> Bob
 in Bob.f(x) ~> Alice
```

Bob

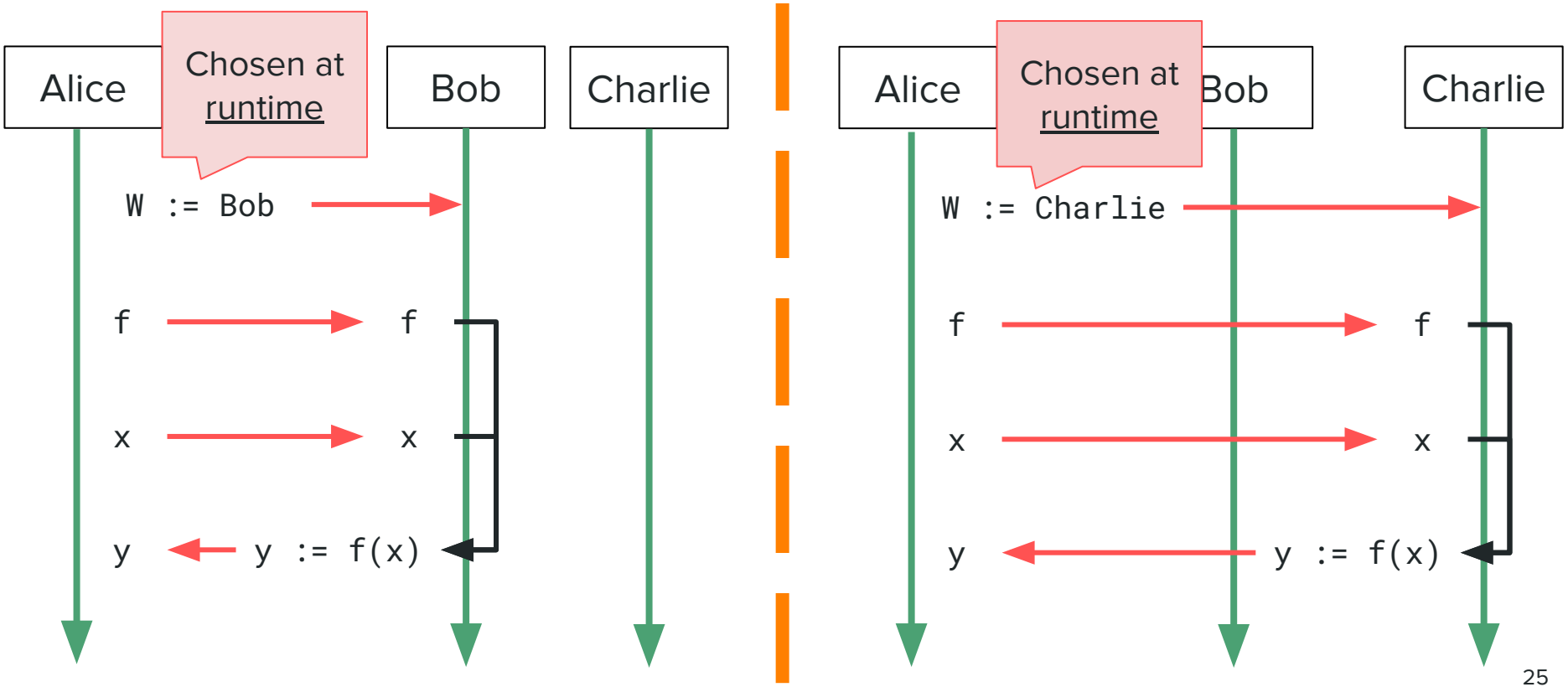
```
fun runAtWorker(

  ) :=
 let f := recv from Alice
     x := recv from Alice
 in send f(x) to Alice
```



$[\![\cdot]\!]_{\text{Bob}}$

But what if we want to **dynamically select a worker**?



Location names need to be **first-class** to accomplish this

This means that we can:

- Generate and inspect them at runtime
- Send them as messages
- Use them as type-level values

```
Alice computes W = select();
W     computes f(x) : T @ W
```

How do we enable **first-class location names** in a **type-safe manner**?

Chor

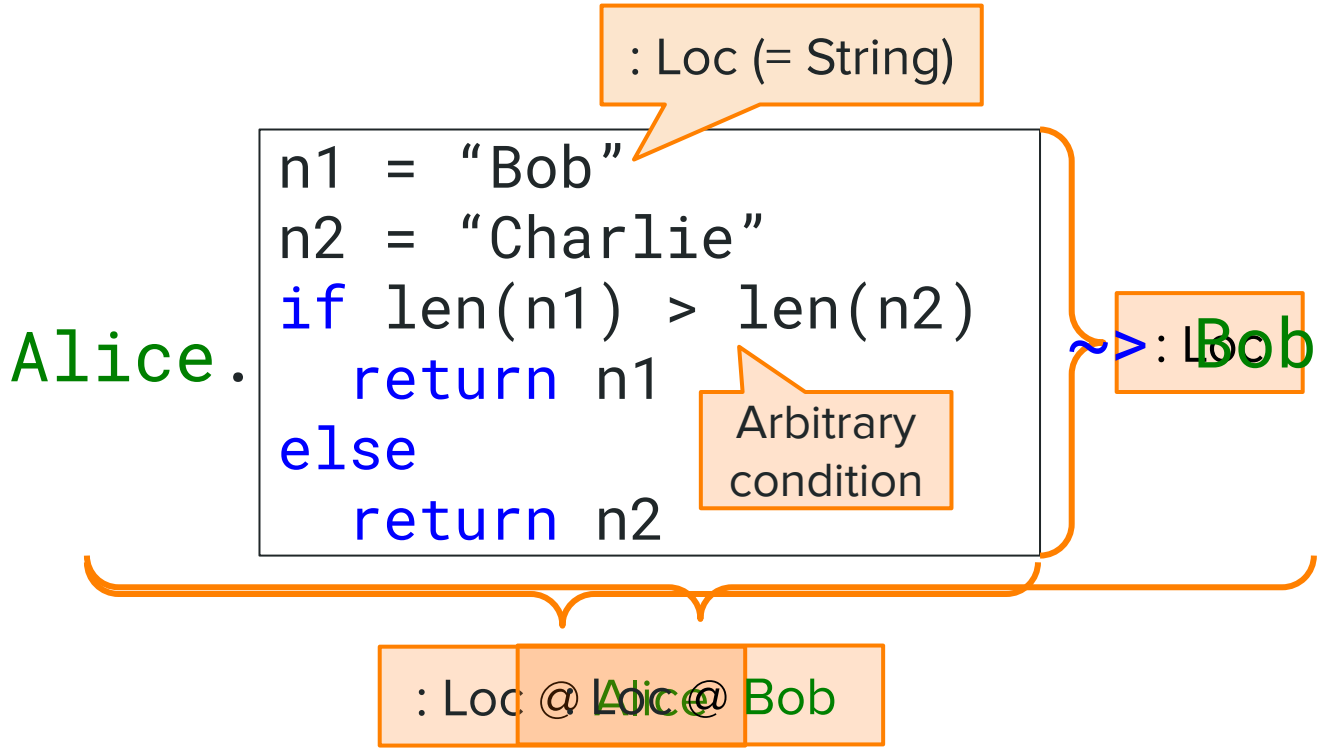
```
let W := Alice.choose()
    W.f := F ~> W
    W.x := X ~> W
in W.f(x) ~> Alice
```

Alice

```
let W := choose()
in send F to W;
   send X to W;
   recv from W
```

$[\![\cdot]\!]_{\texttt{Alice}}$

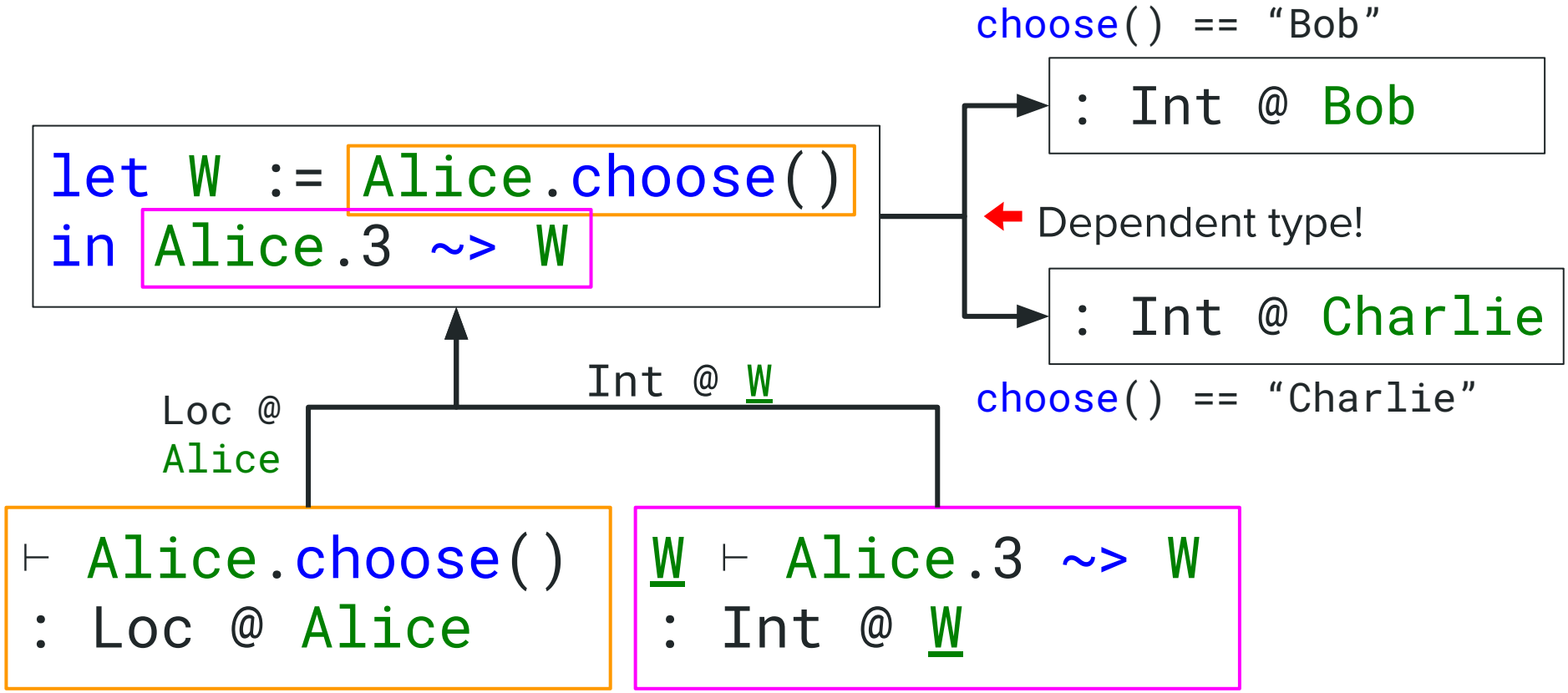
Solution Part 1: Have some way of **representing location names** in a first-class manner



Solution Part 2: Allow for **reifying** a “representation” to a **real location** using a **let-expression**

```
let
    Bob.f := F ~> Bob
    Bob.x := X ~> Bob
in Bob.f(x) ~> Alice
```

Problem: We might introduce **dependent types**



What's wrong with using dependent types?

- Slower and possibly undecidable type-checking
  - Can implement a semi-decision procedure
- Complex for programmers to understand
- Other issues related to their use in choreographies:

```
T1, T2 : Type @ Alice

Alice.x = Alice.(if b then e1 else e2)
  : (if b then T1 else T2) @ Alice

Alice.x ~> Bob : ??? @ Bob
```

Some programs **don't have dependency**...

Chor

```
let W := Alice.choose()
    W.f := F ~> W
    W.x := X ~> W
in W.f(x) ~> Alice
```

: T2 @ Alice

😔 Intermediate types depend on W

```
W        : Loc @ Alice
W.f      : (T1 → T2) @ W
W.x      : T1 @ W
W.f(x)   : T2 @ W

W.f(x) ~> Alice : T2 @ Alice
```

🤔 💡 But the output type does not depend on W!

Limit ourselves: Ensure the **type** of a **let's body does not depend** on the **chosen location**

```
let W := Alice.choose()
    W.f := F ~> W
    W.x := X ~> W
in W.f(x) ~> Alice
```

T2 @ Alice ✅

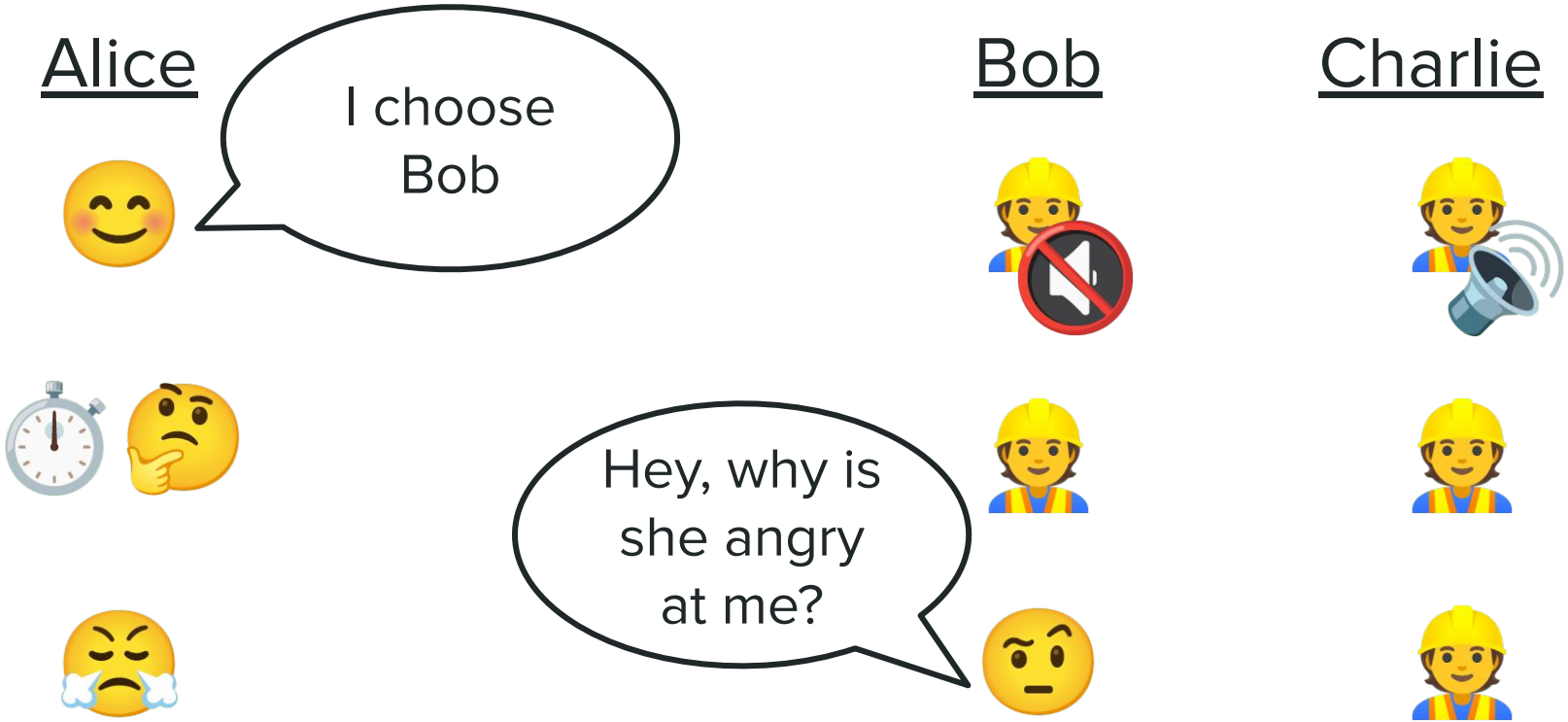
```
let W := Alice.choose()
in Alice.3 ~> W
```

Int @ W ❌

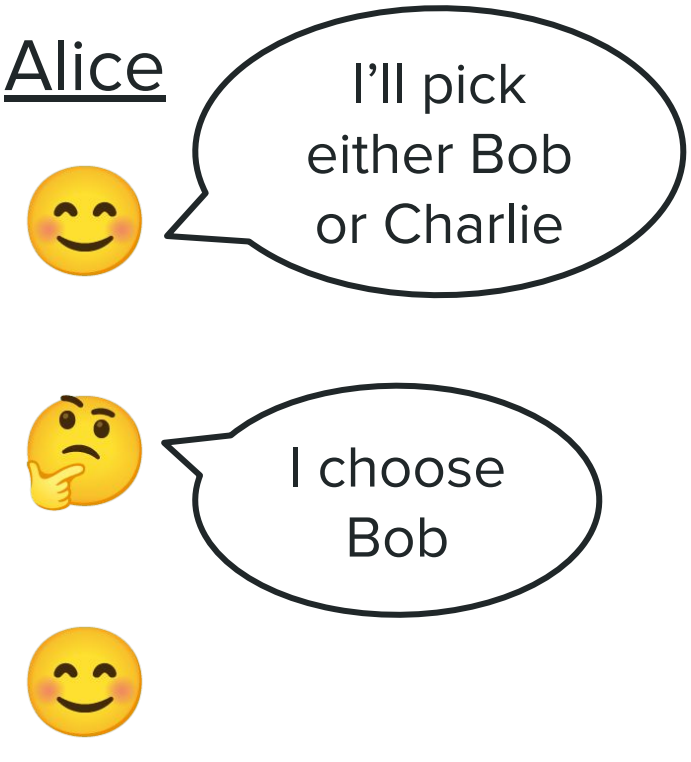
```
let W := Alice.choose()
in λ(Y : Int @ Alice).
      Y ~> W
```

Int @ Alice → Int @ W ❌

Problem: Dynamically chosen locations can **cause deadlocks when a worker doesn't know that they've been selected**



Solution Part 3: **Bound the set of possible workers** and **ensure that they are all aware of the selected worker**

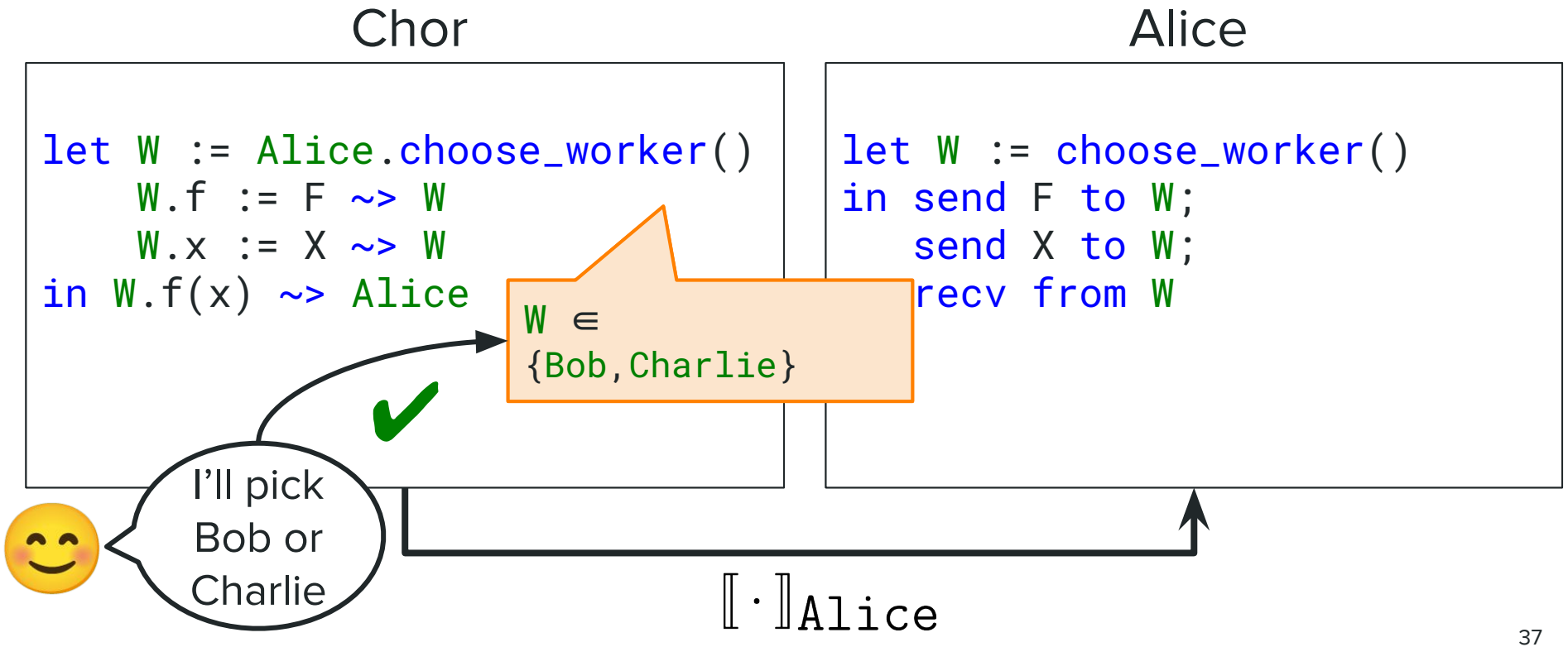


We bound the set of workers **with the local type system**

```
n1 = “Bob”
   : Loc{Bob} <: Loc{Bob,Charlie}
n2 = “Charlie”
   : Loc{Charlie} <: Loc{Bob,Charlie}
if len(n1) > len(n2)
  return n1 : Loc{Bob,Charlie}
else
  return n2 : Loc{Bob,Charlie}
```

Loc{Bob,Charlie}

We then require the **chosen worker to be shared with the entire worker pool** before reifying it



We then require the **chosen worker to be shared with the entire worker pool** before reifying it

Chor

```
let W := Alice.choose_worker()
         ~> {Bob,Charlie}
    W.f := F ~> W
    W.x := X ~> W
in W.f(x) ~> Alice
```

Bob

```
let W := recv from Alice in
  if (W == Bob) then
    let f := recv from Alice
        x := recv from Alice
    in send f(x) to Alice
  else return
```

$[\![\cdot]\!]_{\text{Bob}}$

Comparison: Session Types/Concurrent λ-calculus

Alice

```
let W := Alice.choose_worker()
in choose W for {Bob,Charlie}
   | “Bob” =>
     send F to Bob;
     send X to Bob;
     recv from Bob
   | “Charlie” =>
     send F to Charlie;
     send X to Charlie;
     recv from Charlie
   | else => Fail
end
```

Bob

```
choice from Alice of
   | “Bob” =>
     let f := recv from Alice;
         x := recv from Alice;
     in send (f x) to Alice
   | “Charlie” => return
   | else => Fail
end
```

Choreography Type

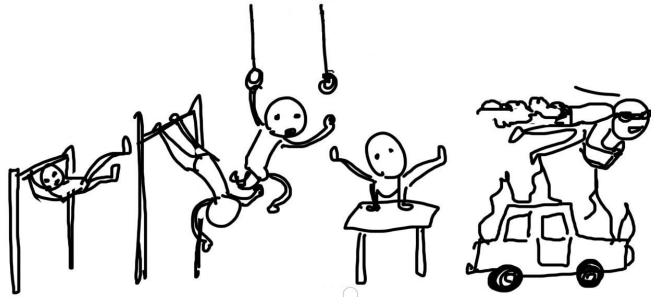
```
runAtWorker :
  (T1 → T2) @ Alice →
  T1 @ Alice →
  T2 @ Alice
```

Session Types

```
Alice.runAtWorker :
  choose for {Bob,Charlie}
   | “Bob” =>
     send (T1 → T2) to Bob;
     send T1 to Bob;
     recv T2 from Bob
   | “Charlie” =>
     send (T1 → T2) to Charlie;
     send T1 to Charlie;
     recv T2 from Charlie
   | else => done
  end
```

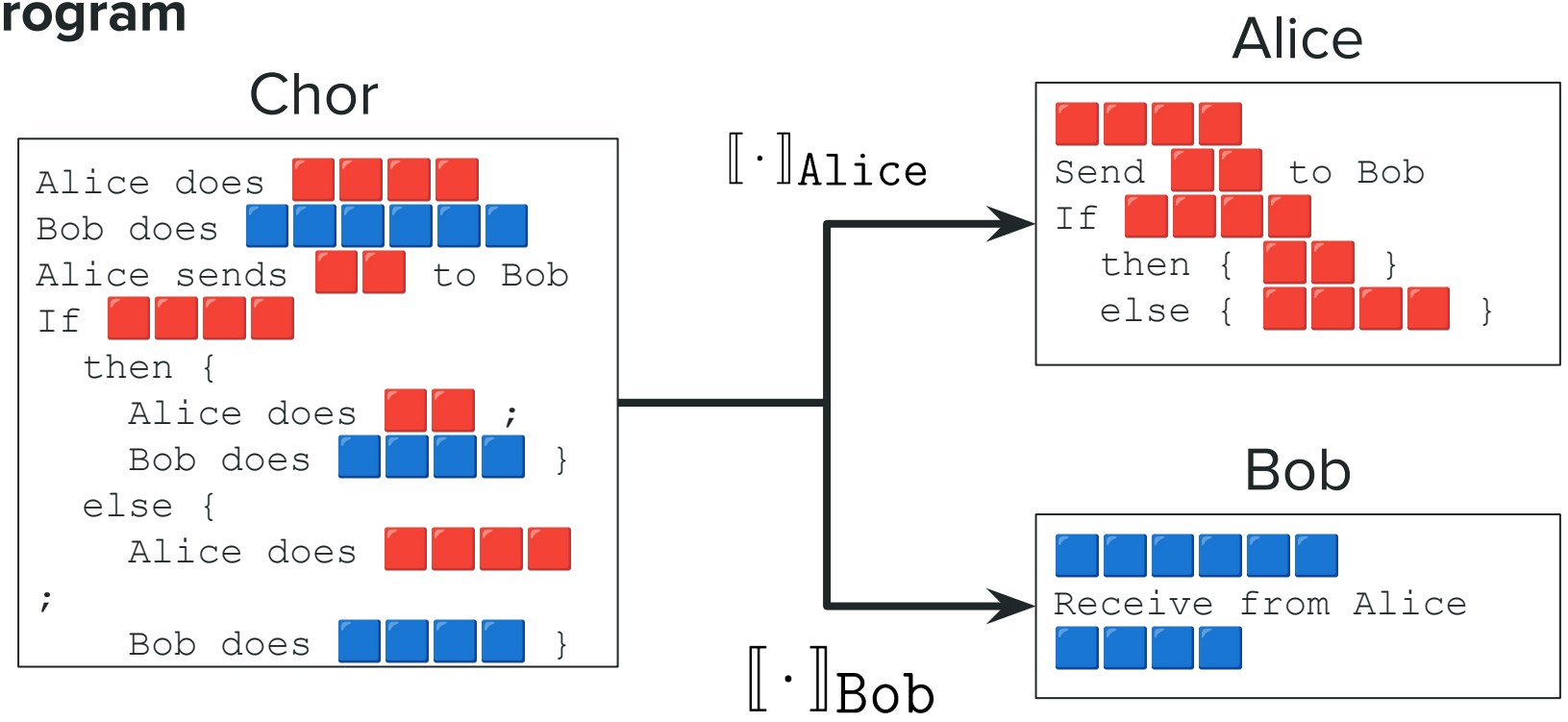
```
Bob.runAtWorker :
  choice from Alice
   | “Bob” =>
     recv (T1 → T2) Alice;
     recv T1 Bob;
     send T2 to Alice
   | “Charlie” => done
   | else => done
  end
```

This whole thing is a type 😨



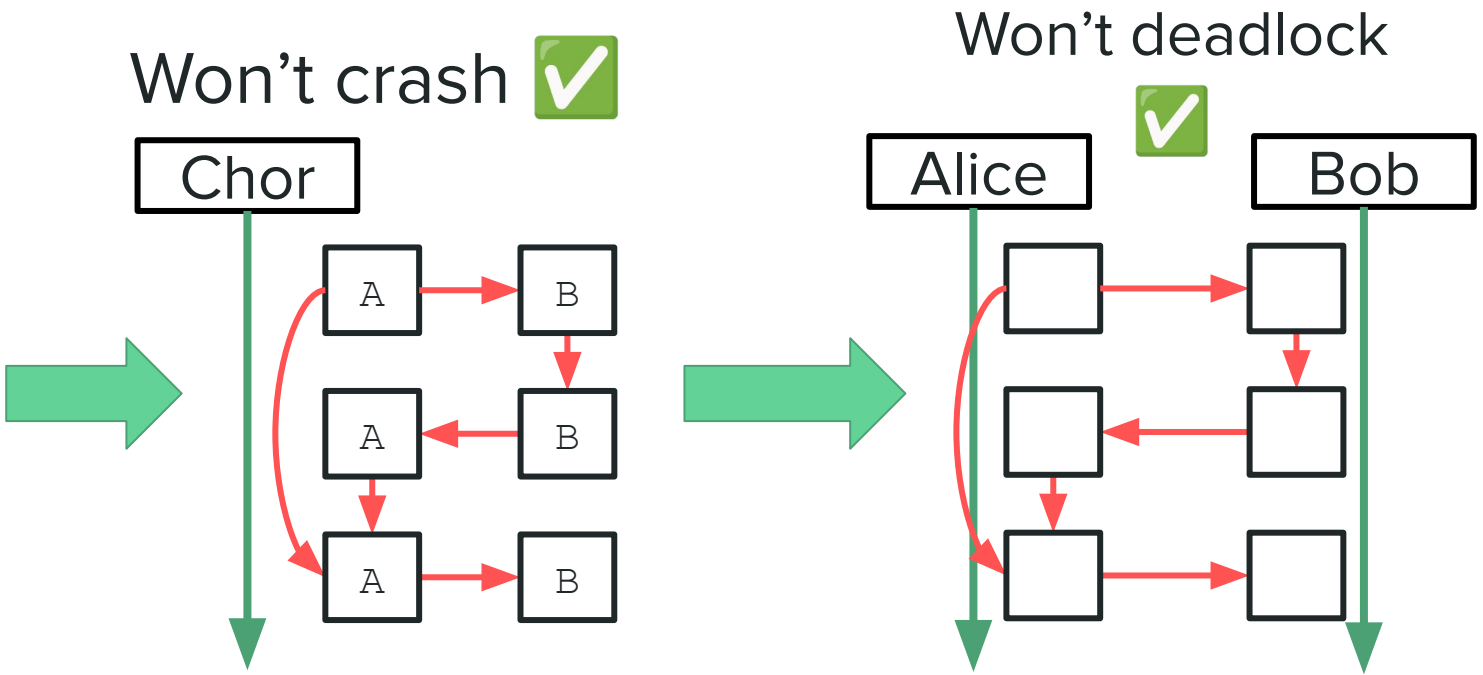
To summarize...

Choreographies are an **emerging paradigm** for **distributed programming** which combines the **actions of all nodes into one program**

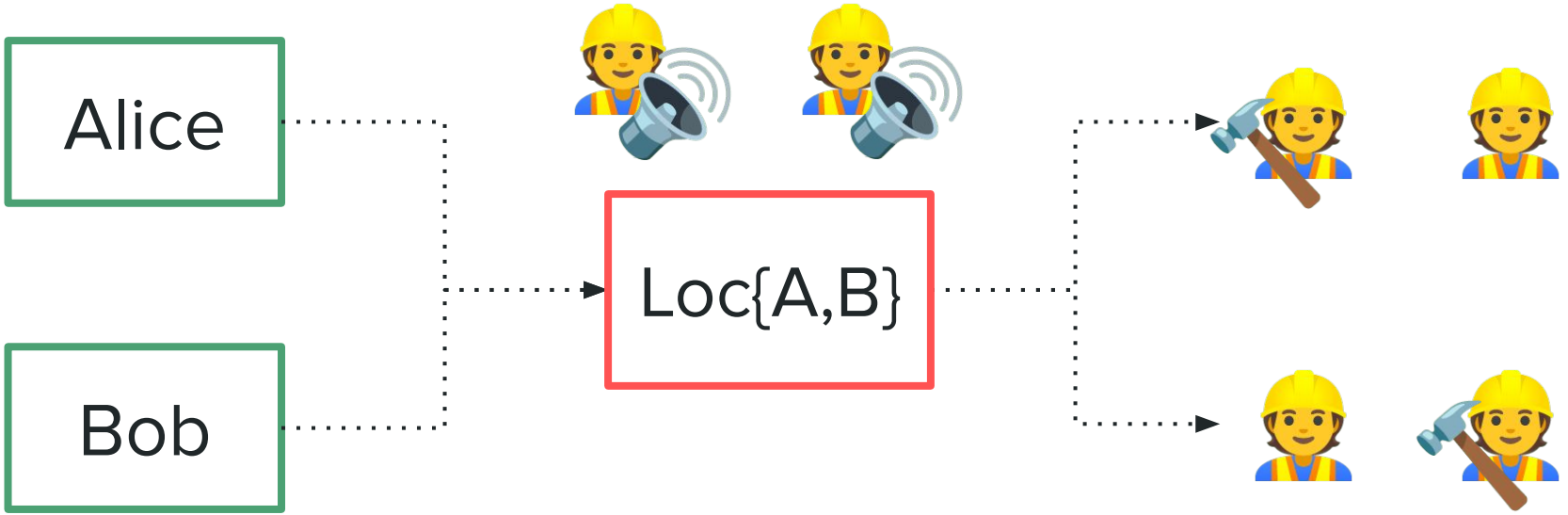


By using a **semantics** and **type system for choreographies**, we can guarantee that the **distributed system won't deadlock**

$\vdash C : T$



We allow for **well-typed, first-class location names** by **preventing type dependency** and ensuring that the **worker pool is notified of the worker selection**



Thanks for listening!