

# Session Types for Message-Passing Concurrency

Dan Frumin and Jorge A. Pérez

University of Groningen, The Netherlands

[www.rug.nl/fse/fc](http://www.rug.nl/fse/fc) - d.frumin | j.a.perez [[at]] rug.nl



UNIFYING  
C•RRECTNESS FOR  
C•MMUNICATING  
S•FTWARE

IPA Formal Methods - June 2022  
(Part 1, v1.1)

# This Course

A bird's eye view on session types for message-passing concurrency, in two parts:

1. Session types as protocol abstractions (Jorge):  
Motivation, key ideas, binary and multiparty session types.
2. Session types as a discipline for communicating processes (Dan):  
The Curry-Howard correspondence between linear logic and session types  
(aka “propositions as sessions”).

# Outline

## Context

### Binary Session Types

Syntax

Example: Two-Buyer Protocol

Subtyping

### Multiparty Session Types

Motivation

Syntax

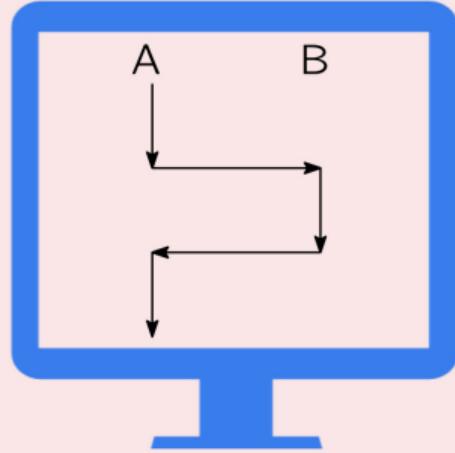
Example: Two-Buyer Protocol, Revisited

Projection

Projection, Revisited

# When is a Program Correct?

## Sequential Programs



“Programs produce outputs that are consistent with their input”

# Concurrent Programs?

# Message-Passing Concurrent Programs?

An (imperfect) analogy:



# Message-Passing Concurrent Programs

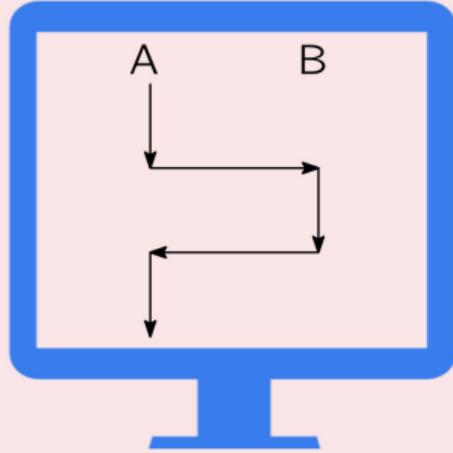
An (imperfect) analogy:



- ▶ Distributed, heterogeneous components or ( $\mu$ -)services
- ▶ Compatible message exchanges are crucial for correctness
- ▶ A single faulty exchange can cause system-wide bugs

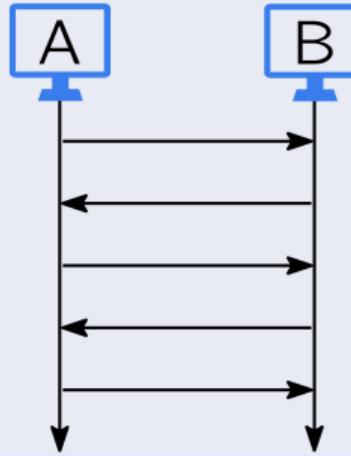
# When is a Program Correct?

## Sequential Programs



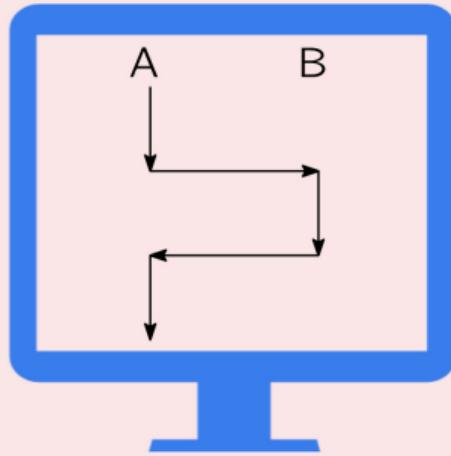
"Programs produce outputs that are consistent with their input"

## Concurrent Programs



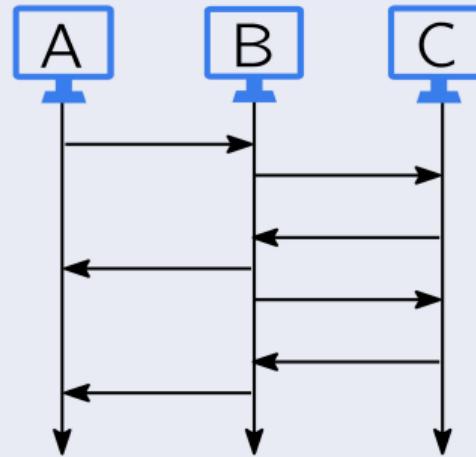
# When is a Program Correct?

## Sequential Programs



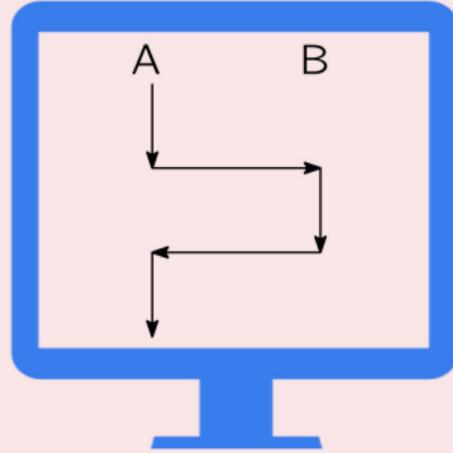
"Programs produce outputs that are consistent with their input"

## Concurrent Programs



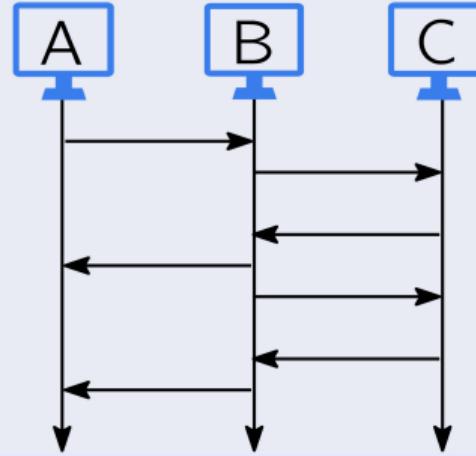
# When is a Program Correct?

## Sequential Programs



"Programs produce outputs that are consistent with their input"

## Concurrent Programs



"Programs always respect their intended protocols"

# Keywords and Slogans

Concurrency Theory, Message-Passing, Programming Languages, Verification

# Keywords and Slogans

Concurrency Theory, Message-Passing, Programming Languages, Verification

- **Type systems**

*Slogan:* Well-typed programs can't go wrong (Milner)

# Keywords and Slogans

Concurrency Theory, Message-Passing, Programming Languages, Verification

- **Type systems**

*Slogan:* Well-typed programs can't go wrong (Milner)

- **Session types** for communication correctness

*Slogan:* **What** and **when** should be sent through a channel

# Keywords and Slogans

Concurrency Theory, Message-Passing, Programming Languages, Verification

- **Type systems**

*Slogan:* Well-typed programs can't go wrong (Milner)

- **Session types** for communication correctness

*Slogan:* **What** and **when** should be sent through a channel

- **Process calculi**

*Slogan:* The  $\pi$ -calculus treats **processes** like the  $\lambda$ -calculus treats **functions**

# Keywords and Slogans

Concurrency Theory, Message-Passing, Programming Languages, Verification

- **Type systems**

*Slogan:* Well-typed programs can't go wrong (Milner)

- **Session types** for communication correctness

*Slogan:* **What** and **when** should be sent through a channel

- **Process calculi**

*Slogan:* The  $\pi$ -calculus treats **processes** like the  $\lambda$ -calculus treats **functions**

- **Propositions as sessions**

Linear logic propositions  $\leftrightarrow$  session types

Proofs  $\leftrightarrow$   $\pi$ -calculus processes

Cut elimination  $\leftrightarrow$  process communication

# Type Systems

- Can detect bugs before programs are run
- Attached to many programming languages
- Implement a specific notion of correctness  
A program is either correct or incorrect



# Type Systems

- Can detect bugs before programs are run
- Attached to many programming languages
- Implement a specific notion of correctness  
A program is either correct or incorrect



## Sequential Languages

- **Data type systems** classify values in a program
- Examples: Integers, strings of characters

# Type Systems

- Can detect bugs before programs are run
- Attached to many programming languages
- Implement a specific notion of correctness  
A program is either correct or incorrect



## Sequential Languages

- **Data type systems** classify values in a program
- Examples: Integers, strings of characters

## Concurrent Languages

- **Behavioral type systems** classify protocols in a program
- Example: “first send username, then receive true/false, finally close”
- A typical bug: sending messages in the wrong order

# Outline

Context

Binary Session Types

Syntax

Example: Two-Buyer Protocol

Subtyping

Multiparty Session Types

Motivation

Syntax

Example: Two-Buyer Protocol, Revisited

Projection

Projection, Revisited

# Protocols as Session Types

Session types uniformly describe protocols in terms of

- communication actions (input and output)
- labeled choices (offers and selections)
- sequential composition
- recursion



# Protocols as Session Types

Session types uniformly describe protocols in terms of

- communication actions (input and output)
- labeled choices (offers and selections)
- sequential composition
- recursion



Session protocols are attached to **interaction devices**:

- TCP-IP sockets
- channel endpoints in programming languages like Go and Rust
- $\pi$ -calculus names
- ...

Sequentiality in types goes **hand-in-hand** with sequentiality in processes

# Protocols as Session Types

$S ::= !U; S$	<b>output</b> value of type $U$ , continue as $S$
$?U; S$	<b>input</b> value of type $U$ , continue as $S$
$\&\{l_i : S_i\}_{i \in I}$	<b>branching</b> : offer a selection between $S_1, \dots, S_n$
$\oplus\{l_i : S_i\}_{i \in I}$	<b>select</b> one between $S_1, \dots, S_n$
$\mu t. S \mid t$	<b>recursion</b>
end	<b>terminated protocol</b>

(Labels  $l_1, \dots, l_n$  are pairwise different.)

# Protocols as Session Types

$S ::= !U; S$	<b>output</b> value of type $U$ , continue as $S$
$?U; S$	<b>input</b> value of type $U$ , continue as $S$
$\&\{l_i : S_i\}_{i \in I}$	<b>branching</b> : offer a selection between $S_1, \dots, S_n$
$\oplus\{l_i : S_i\}_{i \in I}$	<b>select</b> one between $S_1, \dots, S_n$
$\mu t.S \mid t$	<b>recursion</b>
end	<b>terminated protocol</b>

(Labels  $l_1, \dots, l_n$  are pairwise different.)

## Notice:

- Sequential communication patterns (no built-in concurrency)
- $U$  stands for basic values (e.g. int) but also sessions  $S$  (aka delegation)

# Session-Based Concurrency

Conceptually, two phases:

- I. Services advertise their session protocols along **channel names**.  
Agreements are realized by their point-to-point interaction,  
in an **unrestricted** and **non-deterministic** way.

# Session-Based Concurrency

Conceptually, two phases:

- I. Services advertise their session protocols along **channel names**.  
Agreements are realized by their point-to-point interaction,  
in an **unrestricted** and **non-deterministic** way.
- II. After agreement, services establish a session using **session names**.  
Intra-session interactions follow the intended protocol,  
in a **linear** and **deterministic** way.

# Session-Based Concurrency

Conceptually, two phases:

- I. Services advertise their session protocols along **channel names**.  
Agreements are realized by their point-to-point interaction,  
in an **unrestricted** and **non-deterministic** way.
- II. After agreement, services establish a session using **session names**.  
Intra-session interactions follow the intended protocol,  
in a **linear** and **deterministic** way.

## Notice:

- ‘Linear’ and ‘unrestricted’ in the sense of Girard’s **linear logic**.

## Example: A Two-Buyer Protocol



Alice and Bob cooperate in buying a book from Seller:



**bruna**

## Example: A Two-Buyer Protocol



Alice and Bob cooperate in buying a book from Seller:

1. Alice sends a book title to Seller, who sends a quote back.



**bruna**

## Example: A Two-Buyer Protocol



Alice and Bob cooperate in buying a book from Seller:

1. Alice sends a book title to Seller, who sends a quote back.
2. Alice checks whether Bob can contribute in buying the book.



**bruna**

## Example: A Two-Buyer Protocol



Alice and Bob cooperate in buying a book from Seller:

1. Alice sends a book title to Seller, who sends a quote back.
2. Alice checks whether Bob can contribute in buying the book.
3. Alice uses the answer from Bob to interact with Seller, either:
  - a) completing the payment and arranging delivery details
  - b) canceling the transaction

## Example: A Two-Buyer Protocol



Alice and Bob cooperate in buying a book from Seller:

1. Alice sends a book title to Seller, who sends a quote back.
2. Alice checks whether Bob can contribute in buying the book.
3. Alice uses the answer from Bob to interact with Seller, either:
  - a) completing the payment and arranging delivery details
  - b) canceling the transaction
4. In case 3(a) Alice contacts Bob to get his address, and forwards it to Seller.

# Example: A Two-Buyer Protocol



Alice and Bob cooperate in buying a book from Seller:

1. Alice sends a book title to Seller, who sends a quote back.
2. Alice checks whether Bob can contribute in buying the book.
3. Alice uses the answer from Bob to interact with Seller, either:
  - a) completing the payment and arranging delivery details
  - b) canceling the transaction
4. In case 3(a) Alice contacts Bob to get his address, and forwards it to Seller.
- 4'. In case 3(b) Alice is in charge of gracefully concluding the conversation.

# Example: A Two-Buyer Protocol

Desiderata for the implementations of Alice, Bob, and Seller:

- **Fidelity** – they **follow the intended protocol**.

- Alice doesn't continue the transaction if Bob can't contribute
- Alice chooses among the options provided by Seller



## Example: A Two-Buyer Protocol

Desiderata for the implementations of Alice, Bob, and Seller:

- **Fidelity** – they **follow the intended protocol**.
- **Safety** – they don't feature **communication errors**.
  - Seller always returns an integer when Alice requests a quote



## Example: A Two-Buyer Protocol

Desiderata for the implementations of Alice, Bob, and Seller:

- **Fidelity** – they **follow the intended protocol**.
- **Safety** – they don't feature **communication errors**.
- **Deadlock-Freedom** – they do not “**get stuck**” while running the protocol.
  - Alice eventually receives an answer from Bob on his contribution.



## Example: A Two-Buyer Protocol

Desiderata for the implementations of Alice, Bob, and Seller:

- **Fidelity** – they **follow the intended protocol**.
- **Safety** – they don't feature **communication errors**.
- **Deadlock-Freedom** – they do not “**get stuck**” while running the protocol.
- **Termination** – they do not engage in **infinite behavior** (that may prevent them from completing the protocol)



## Example: A Two-Buyer Protocol

Desiderata for the implementations of Alice, Bob, and Seller:

- **Fidelity** – they **follow the intended protocol**.
- **Safety** – they don't feature **communication errors**.
- **Deadlock-Freedom** – they do not “**get stuck**” while running the protocol.
- **Termination** – they do not engage in **infinite behavior** (that may prevent them from completing the protocol)



Correctness follows from the interplay of these properties.

**Hard to enforce**, especially when actions are “scattered around” in source programs.

## Example: A Two-Buyer Protocol

Two separate protocols, with Alice “leading” the interactions:

- A session type for Seller (in its interaction with Alice):

$$S_{SA} = ?book; !quote; \& \begin{cases} \text{buy : } & ?paym; ?address; !ok; \text{end} \\ \text{cancel : } & ?thanks; !bye; \text{end} \end{cases}$$



## Example: A Two-Buyer Protocol

Two separate protocols, with Alice “leading” the interactions:

- A session type for Seller (in its interaction with Alice):

$$S_{SA} = ?book; !quote; \& \begin{cases} \text{buy : } ?paym; ?address; !ok; \text{end} \\ \text{cancel : } ?thanks; !bye; \text{end} \end{cases}$$



- A session type for Alice (in its interaction with Bob):

$$S_{AB} = !cost; \& \begin{cases} \text{share : } ?address; !ok; \text{end} \\ \text{close : } !bye; \text{end} \end{cases}$$



## Example: A Two-Buyer Protocol

Implementations for Alice, Bob, Seller should be **compatible**.

## Example: A Two-Buyer Protocol

Implementations for Alice, Bob, Seller should be **compatible**.

- **Duality** relates session types with opposite behaviors.

Intuitively:

- the dual of input is output (and vice versa)
- branching is the dual of selection (and vice versa)

## Example: A Two-Buyer Protocol

Implementations for Alice, Bob, Seller should be **compatible**.

- **Duality** relates session types with opposite behaviors.

Intuitively:

- the dual of input is output (and vice versa)
- branching is the dual of selection (and vice versa)

- Recall that  $S_{AB}$  describes Alice's viewpoint in her interaction with Bob:

$$S_{AB} = !cost; \& \begin{cases} \text{share : ?address; !ok; end} \\ \text{close : !bye; end} \end{cases}$$

- Given this, Bob's implementation should conform to  $\overline{S_{AB}}$ , the dual of  $S_{AB}$ :

$$\overline{S_{AB}} = ?cost; \oplus \begin{cases} \text{share : !address; ?ok; end} \\ \text{close : ?bye; end} \end{cases}$$

## Example: A Two-Buyer Protocol

Implementations for Alice, Bob, Seller should be **compatible**.

- **Duality** relates session types with opposite behaviors.

Intuitively:

- the dual of input is output (and vice versa)
- branching is the dual of selection (and vice versa)

- Recall that  $S_{AB}$  describes Alice's viewpoint in her interaction with Bob:

$$S_{AB} = !cost; \& \begin{cases} \text{share : ?address; !ok; end} \\ \text{close : !bye; end} \end{cases}$$

- Given this, Bob's implementation should conform to  $\overline{S_{AB}}$ , the dual of  $S_{AB}$ :

$$\overline{S_{AB}} = ?cost; \oplus \begin{cases} \text{share : !address; ?ok; end} \\ \text{close : ?bye; end} \end{cases}$$

- Also, Alice's implementation should conform to both  $\overline{S_{SA}}$  and  $S_{AB}$ .

# Session Type Duality, Formally

Given a (finite) session type  $S$ , its dual type  $\overline{S}$  is inductively defined as follows:

$$\overline{!U; S} = ?U; \overline{S}$$

$$\overline{?U; S} = !U; \overline{S}$$

$$\overline{\&\{l_i : S_i\}_{i \in I}} = \oplus\{l_i : \overline{S}_i\}_{i \in I}$$

$$\overline{\oplus\{l_i : S_i\}_{i \in I}} = \&\{l_i : \overline{S}_i\}_{i \in I}$$

$$\overline{\text{end}} = \text{end}$$

## Notice:

- Duality for recursive session types is defined coinductively  
(the dual of  $\mu t.S$  is *not*  $\mu t.\overline{S}$ )

# Enhancing Compatibility via Subtyping

Consider a “mathematical server” and a candidate client

- The session type for the server:

$$S = \& \begin{cases} \text{eq} : ?\text{Real}; ?\text{Real}; !\text{Bool}; \text{end} \\ \text{add} : ?\text{Int}; ?\text{Int}; !\text{Int}; \text{end} \end{cases}$$

# Enhancing Compatibility via Subtyping

Consider a “mathematical server” and a candidate client

- The session type for the server:

$$S = \& \begin{cases} \text{eq} : ?\text{Real}; ?\text{Real}; !\text{Bool}; \text{end} \\ \text{add} : ?\text{Int}; ?\text{Int}; !\text{Int}; \text{end} \end{cases}$$

- The session type for the client:

$$T_1 = \oplus \begin{cases} \text{eq} : !\text{Int}; !\text{Int}; ?\text{Bool}; \text{end} \\ \text{add} : !\text{Int}; !\text{Int}; ?\text{Int}; \text{end} \end{cases}$$

# Enhancing Compatibility via Subtyping

Consider a “mathematical server” and a candidate client

- The session type for the server:

$$S = \& \begin{cases} \text{eq} : ?\text{Real}; ?\text{Real}; !\text{Bool}; \text{end} \\ \text{add} : ?\text{Int}; ?\text{Int}; !\text{Int}; \text{end} \end{cases}$$

- The session type for the client:

$$T_1 = \oplus \begin{cases} \text{eq} : !\text{Int}; !\text{Int}; ?\text{Bool}; \text{end} \\ \text{add} : !\text{Int}; !\text{Int}; ?\text{Int}; \text{end} \end{cases}$$

Server and client are formally **incompatible**:

- ▶  $S$  and  $T_1$  are not dual to each other, because of base types (Real vs Int)

# Enhancing Compatibility via Subtyping

Consider a “mathematical server” and a candidate client

- The session type for the server:

$$S = \& \begin{cases} \text{eq} : ?\text{Real}; ?\text{Real}; !\text{Bool}; \text{end} \\ \text{add} : ?\text{Int}; ?\text{Int}; !\text{Int}; \text{end} \end{cases}$$

- The session type for the client:

$$T_1 = \oplus \begin{cases} \text{eq} : !\text{Int}; !\text{Int}; ?\text{Bool}; \text{end} \\ \text{add} : !\text{Int}; !\text{Int}; ?\text{Int}; \text{end} \end{cases}$$

Server and client are formally **incompatible**:

- ▶  $S$  and  $T_1$  are not dual to each other, because of base types (Real vs Int)

Still, one may argue that server and client should be compatible.

# Enhancing Compatibility via Subtyping

Consider now an upgrade of the “mathematical server”

- The session type for the server known to clients:

$$S = \& \begin{cases} \text{eq} : ?\text{Real}; ?\text{Real}; !\text{Bool}; \text{end} \\ \text{add} : ?\text{Int}; ?\text{Int}; !\text{Int}; \text{end} \end{cases}$$

# Enhancing Compatibility via Subtyping

Consider now an upgrade of the “mathematical server”

- The session type for the server known to clients:

$$S = \& \begin{cases} \text{eq} : ?\text{Real}; ?\text{Real}; !\text{Bool}; \text{end} \\ \text{add} : ?\text{Int}; ?\text{Int}; !\text{Int}; \text{end} \end{cases}$$

- The session type for the upgraded server:

$$S' = \& \begin{cases} \text{eq} : ?\text{Real}; ?\text{Real}; !\text{Bool}; \text{end} \\ \text{add} : ?\text{Int}; ?\text{Int}; !\text{Int}; \text{end} \\ \text{mul} : ?\text{Int}; ?\text{Int}; !\text{Int}; \text{end} \end{cases}$$

# Enhancing Compatibility via Subtyping

Consider now an upgrade of the “mathematical server”

- The session type for the server known to clients:

$$S = \& \begin{cases} \text{eq} : ?\text{Real}; ?\text{Real}; !\text{Bool}; \text{end} \\ \text{add} : ?\text{Int}; ?\text{Int}; !\text{Int}; \text{end} \end{cases}$$

- The session type for the upgraded server:

$$S' = \& \begin{cases} \text{eq} : ?\text{Real}; ?\text{Real}; !\text{Bool}; \text{end} \\ \text{add} : ?\text{Int}; ?\text{Int}; !\text{Int}; \text{end} \\ \text{mul} : ?\text{Int}; ?\text{Int}; !\text{Int}; \text{end} \end{cases}$$

The upgraded server and existing clients are formally **incompatible**:

- ▶ The options (labels) of clients (such as  $T_1$ ) and  $S'$  do not match.

# Enhancing Compatibility via Subtyping

Consider now an upgrade of the “mathematical server”

- The session type for the server known to clients:

$$S = \& \begin{cases} \text{eq} : ?\text{Real}; ?\text{Real}; !\text{Bool}; \text{end} \\ \text{add} : ?\text{Int}; ?\text{Int}; !\text{Int}; \text{end} \end{cases}$$

- The session type for the upgraded server:

$$S' = \& \begin{cases} \text{eq} : ?\text{Real}; ?\text{Real}; !\text{Bool}; \text{end} \\ \text{add} : ?\text{Int}; ?\text{Int}; !\text{Int}; \text{end} \\ \text{mul} : ?\text{Int}; ?\text{Int}; !\text{Int}; \text{end} \end{cases}$$

The upgraded server and existing clients are formally **incompatible**:

- ▶ The options (labels) of clients (such as  $T_1$ ) and  $S'$  do not match.

Here again the upgraded server and old clients should be seen as compatible.

## Enhancing Compatibility via Subtyping

We may relate  $T_1$  with  $S$  and  $S'$  using a **subtyping** relation.

# Enhancing Compatibility via Subtyping

We may relate  $T_1$  with  $S$  and  $S'$  using a **subtyping** relation.

- Notation:  $S_1 \leq S_2$  (read:  $S_1$  is a subtype of  $S_2$ )
- Intuitively, if  $S_1 \leq S_2$  then a name of type  $S_1$  can safely be used where a name of type  $S_2$  is expected (**safe substitutability**)

# Enhancing Compatibility via Subtyping

- On the one hand, we have:

$$\frac{\& \left\{ \begin{array}{l} \text{eq : } ?\text{Int}; ?\text{Int}; !\text{Bool}; \text{end} \\ \text{add : } ?\text{Int}; ?\text{Int}; !\text{Int}; \text{end} \end{array} \right.}{\overline{T_1}} \leq \& \left\{ \begin{array}{l} \text{eq : } ?\text{Real}; ?\text{Real}; !\text{Bool}; \text{end} \\ \text{add : } ?\text{Int}; ?\text{Int}; !\text{Int}; \text{end} \end{array} \right. \overline{S}$$

$\overline{T_1} \leq S$ : It is safe for the server to receive integers if reals are supported.

# Enhancing Compatibility via Subtyping

- On the one hand, we have:

$$\frac{\& \left\{ \begin{array}{l} \text{eq : } ?\text{Int}; ?\text{Int}; !\text{Bool}; \text{end} \\ \text{add : } ?\text{Int}; ?\text{Int}; !\text{Int}; \text{end} \end{array} \right.}{\overline{T_1}} \leq \& \left\{ \begin{array}{l} \text{eq : } ?\text{Real}; ?\text{Real}; !\text{Bool}; \text{end} \\ \text{add : } ?\text{Int}; ?\text{Int}; !\text{Int}; \text{end} \end{array} \right. \overline{S}$$

$\overline{T_1} \leq S$ : It is safe for the server to receive integers if reals are supported.

- We also have:

$$\frac{\& \left\{ \begin{array}{l} \text{eq : } ?\text{Int}; ?\text{Int}; !\text{Bool}; \text{end} \\ \text{add : } ?\text{Int}; ?\text{Int}; !\text{Int}; \text{end} \end{array} \right.}{\overline{S}} \leq \& \left\{ \begin{array}{l} \text{eq : } ?\text{Int}; ?\text{Int}; !\text{Bool}; \text{end} \\ \text{add : } ?\text{Int}; ?\text{Int}; !\text{Int}; \text{end} \\ \text{mul : } ?\text{Int}; ?\text{Int}; !\text{Int}; \text{end} \end{array} \right. \overline{S'}$$

$S \leq S'$ : It is safe to serve clients who know only some of the options.

# Subtyping, Formally

We assume expected relations for base types, e.g.,  $\text{Int} \leq \text{Real}$ .

$$\frac{}{\text{end} \leq \text{end}}$$
$$\frac{U_1 \leq U_2 \quad S_1 \leq S_2}{!U_2; S_1 \leq !U_1; S_2}$$
$$\frac{I \subseteq J \quad \forall i \in I. S_i \leq T_i}{\&\{l_i : S_i\}_{i \in I} \leq \&\{l_j : T_j\}_{j \in J}}$$
$$\frac{U_1 \leq U_2 \quad S_1 \leq S_2}{?U_1; S_1 \leq ?U_2; S_2}$$
$$\frac{J \subseteq I \quad \forall j \in J. S_j \leq T_j}{\oplus\{l_j : S_j\}_{j \in J} \leq \oplus\{l_i : T_i\}_{i \in I}}$$

# Subtyping, Formally

We assume expected relations for base types, e.g.,  $\text{Int} \leq \text{Real}$ .

$$\frac{}{\text{end} \leq \text{end}}$$
$$\frac{U_1 \leq U_2 \quad S_1 \leq S_2}{!U_2; S_1 \leq !U_1; S_2}$$
$$\frac{I \subseteq J \quad \forall i \in I. S_i \leq T_i}{\&\{l_i : S_i\}_{i \in I} \leq \&\{l_j : T_j\}_{j \in J}}$$
$$\frac{U_1 \leq U_2 \quad S_1 \leq S_2}{?U_1; S_1 \leq ?U_2; S_2}$$
$$\frac{J \subseteq I \quad \forall j \in J. S_j \leq T_j}{\oplus\{l_j : S_j\}_{j \in J} \leq \oplus\{l_i : T_i\}_{i \in I}}$$

Notice:

- $\leq$  is co-variant for inputs and contra-variant for outputs  
Example:  $?Int; S \leq ?Real; S$  is sound, but  $!Int; S \leq !Real; S$  is not.
- $\leq$  is co-variant for branching and contra-variant for choices.

# Subtyping, Formally

We assume expected relations for base types, e.g.,  $\text{Int} \leq \text{Real}$ .

$$\frac{}{\text{end} \leq \text{end}} \quad \frac{U_1 \leq U_2 \quad S_1 \leq S_2}{!U_2; S_1 \leq !U_1; S_2} \quad \frac{U_1 \leq U_2 \quad S_1 \leq S_2}{?U_1; S_1 \leq ?U_2; S_2}$$
$$\frac{I \subseteq J \quad \forall i \in I. S_i \leq T_i}{\&\{l_i : S_i\}_{i \in I} \leq \&\{l_j : T_j\}_{j \in J}}$$
$$\frac{J \subseteq I \quad \forall j \in J. S_j \leq T_j}{\oplus\{l_j : S_j\}_{j \in J} \leq \oplus\{l_i : T_i\}_{i \in I}}$$

Notice:

- $\leq$  is co-variant for inputs and contra-variant for outputs  
Example:  $?Int; S \leq ?Real; S$  is sound, but  $!Int; S \leq !Real; S$  is not.
- $\leq$  is co-variant for branching and contra-variant for choices.

Also:

- $\leq$  concerns substitutability of names implementing protocols.  
Safe substitutability of processes (programs) also possible.
- A general definition of  $\leq$  is coinductive

# Outline

Context

Binary Session Types

Syntax

Example: Two-Buyer Protocol

Subtyping

Multiparty Session Types

Motivation

Syntax

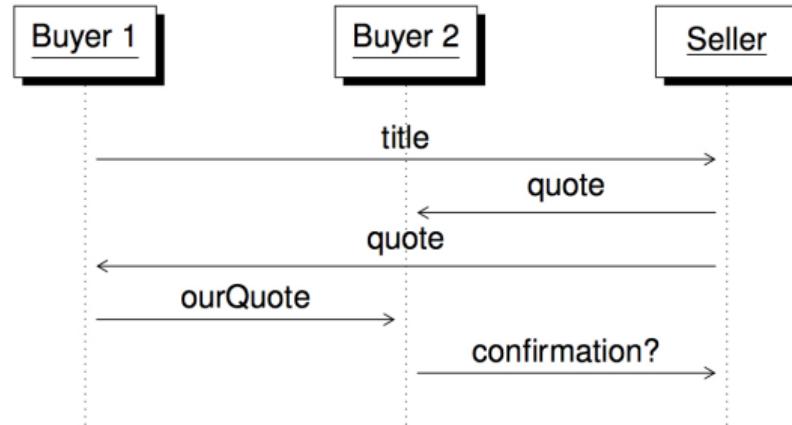
Example: Two-Buyer Protocol, Revisited

Projection

Projection, Revisited

# From Binary to Multiparty Protocols

A two-buyer protocol, similar to the one discussed earlier:



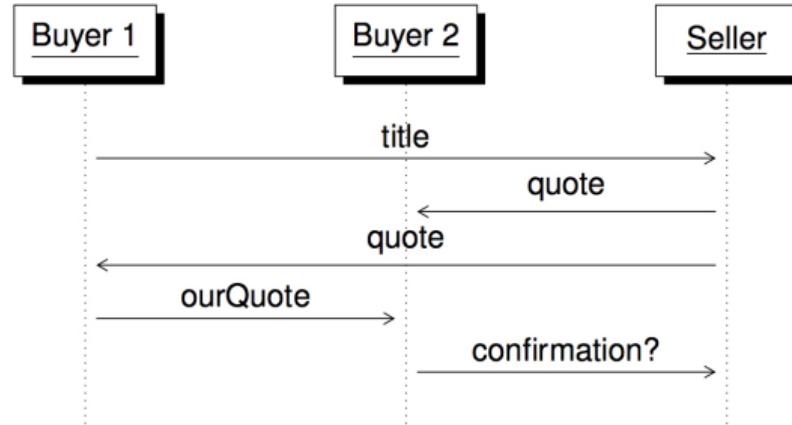
# From Binary to Multiparty Protocols

- Binary session types organize interactions between exactly two partners.
- If there are three or more partners involved, binary protocols between them are unavoidably **disjoint**.

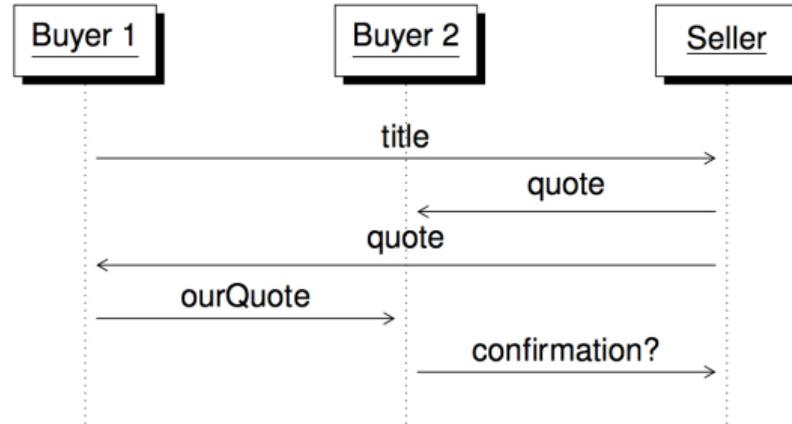
# From Binary to Multiparty Protocols

- Binary session types organize interactions between exactly two partners.
- If there are three or more partners involved, binary protocols between them are unavoidably **disjoint**.
- In realistic scenarios, we find **multiparty protocols**, in which multiple partners are expected to interact along the **same session protocol**.
- Decomposing such multiparty protocols into binary session types is not always possible — essential **sequencing information** may be lost.

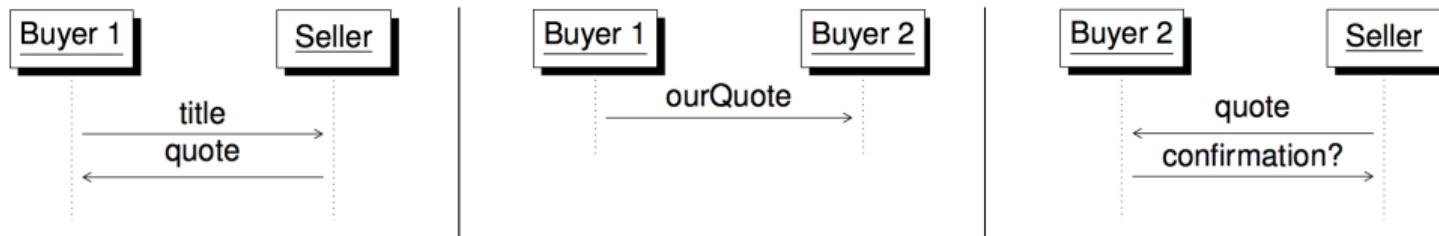
# The Need for Sequencing Information



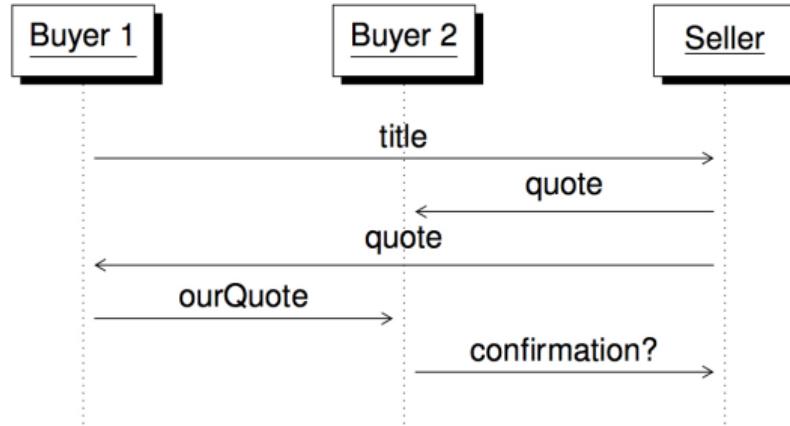
# The Need for Sequencing Information



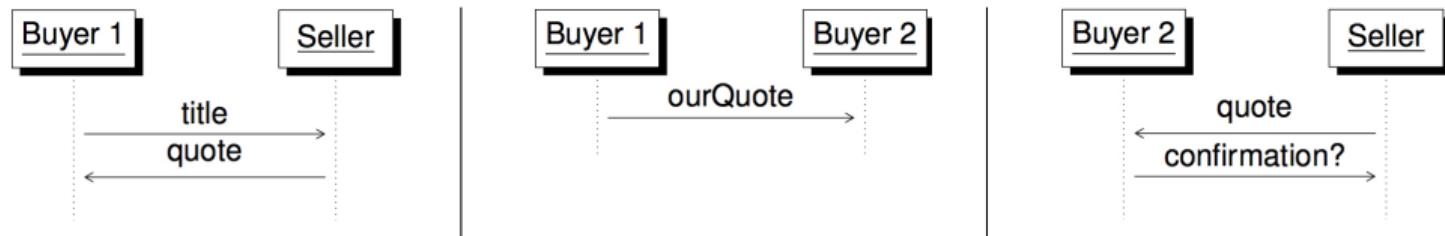
A decomposition as binary protocols may appear plausible...



# The Need for Sequencing Information



A decomposition as binary protocols may appear plausible...



... but misses key sequencing between unrelated partners.

# Binary and Multiparty Session Types

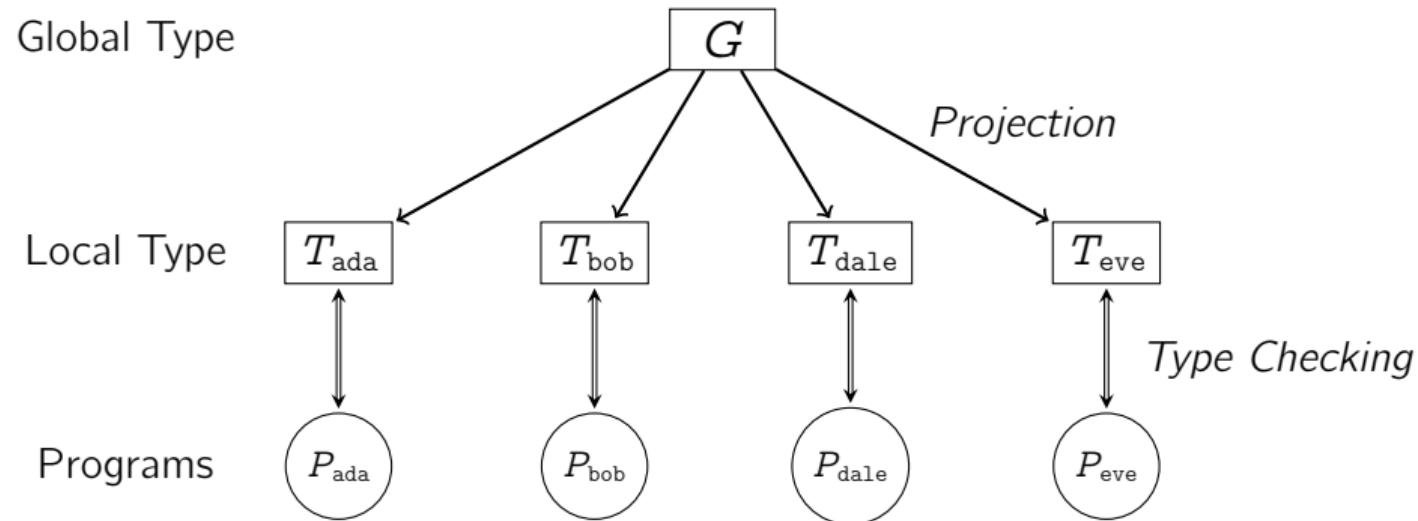
- In **binary** session types protocols involve exactly two partners
- **Multiparty** session types lift this limitation.

Two levels:

- a **global type** offers a high-level perspective of the protocol
- **local types** abstract each partner's contribution to the protocol

# Multiparty Session Types

A methodology for **decentralized** specification, development, and validation of protocols between participants that communicate asynchronously:



# The Syntax of Multiparty Session Types

We use  $p, q, \dots$  to denote participants.

Let  $U$  denote the type of transmittable values.

## 1. Global types:

$G ::=$	$p \rightarrow q : \langle U \rangle.G$	Exchange of value of type $U$
	$p \rightarrow q : \{l_i : G_i\}_{i \in I}$	Branching
	$\mu t.G \quad   \quad t$	Recursion
	end	Terminated global protocol

## 2. Local types

## The Two-Buyer Protocol, Revisited (1/2)

Alice and Bob cooperate in buying a book from Seller:

1. Alice sends a book title to Seller, who sends a quote back.
2. Alice checks with Bob whether he can contribute in buying the book.
3. Alice uses the answer from Bob to interact with Seller, either
  - a) completing the payment and arranging delivery details
  - b) canceling the transaction

## The Two-Buyer Protocol, Revisited (2/2)

A **single** global protocol  $G$  between three participants: Alice, Bob, and Seller.

```
 $G = \text{Alice} \rightarrow \text{Seller} : \langle \text{book} \rangle.$ 
           $\text{Seller} \rightarrow \text{Alice} : \langle \text{quote} \rangle.$ 
           $\text{Alice} \rightarrow \text{Bob} : \langle \text{cost} \rangle.$ 
           $\text{Bob} \rightarrow \text{Alice} : \{ \text{share} : \text{Bob} \rightarrow \text{Alice} : \langle \text{address} \rangle.$ 
                            $\text{Alice} \rightarrow \text{Bob} : \langle \text{ok} \rangle.$ 
                            $\text{Seller} \rightarrow \text{Alice} : \langle \text{invoice} \rangle.$ 
                            $\text{Alice} \rightarrow \text{Seller} : \langle \text{paym} \rangle.\text{end}$ 
                            $\text{close} : \text{Alice} \rightarrow \text{Bob} : \langle \text{bye} \rangle.\text{end}$ 
          }
```

(with base types ‘book’, ‘quote’, ‘cost’, ‘address’, ‘ok’, ‘invoice’, ‘paym’, ‘bye’)

## The Two-Buyer Protocol, Revisited (2/2)

A **single** global protocol  $G$  between three participants: Alice, Bob, and Seller.

```
 $G = \text{Alice} \rightarrow \text{Seller} : \langle \text{book} \rangle.$ 
    \text{Seller} \rightarrow \text{Alice} : \langle \text{quote} \rangle.
    \text{Alice} \rightarrow \text{Bob} : \langle \text{cost} \rangle.
    \text{Bob} \rightarrow \text{Alice} : \{ \text{share} : \text{Bob} \rightarrow \text{Alice} : \langle \text{address} \rangle.
                                \text{Alice} \rightarrow \text{Bob} : \langle \text{ok} \rangle.
                                \text{Seller} \rightarrow \text{Alice} : \langle \text{invoice} \rangle.
                                \text{Alice} \rightarrow \text{Seller} : \langle \text{paym} \rangle.\text{end}
                                \text{close} : \text{Alice} \rightarrow \text{Bob} : \langle \text{bye} \rangle.\text{end}
                            \}
```

(with base types ‘book’, ‘quote’, ‘cost’, ‘address’, ‘ok’, ‘invoice’, ‘paym’, ‘bye’)

(Note: There is a problem with this protocol - can you spot it?)

# The Syntax of Multiparty Session Types

We use  $p, q, \dots$  to denote participants.  $U$  is the type of transmittable values.

## 1. Global types:

$G ::=$	$p \rightarrow q : \langle U \rangle.G$	Exchange of value of type $U$
	$p \rightarrow q : \{l_i : G_i\}_{i \in I}$	Branching
	$\mu t.G \quad   \quad t$	Recursion
	end	Terminated global protocol

# The Syntax of Multiparty Session Types

We use  $p, q, \dots$  to denote participants.  $U$  is the type of transmittable values.

## 1. Global types:

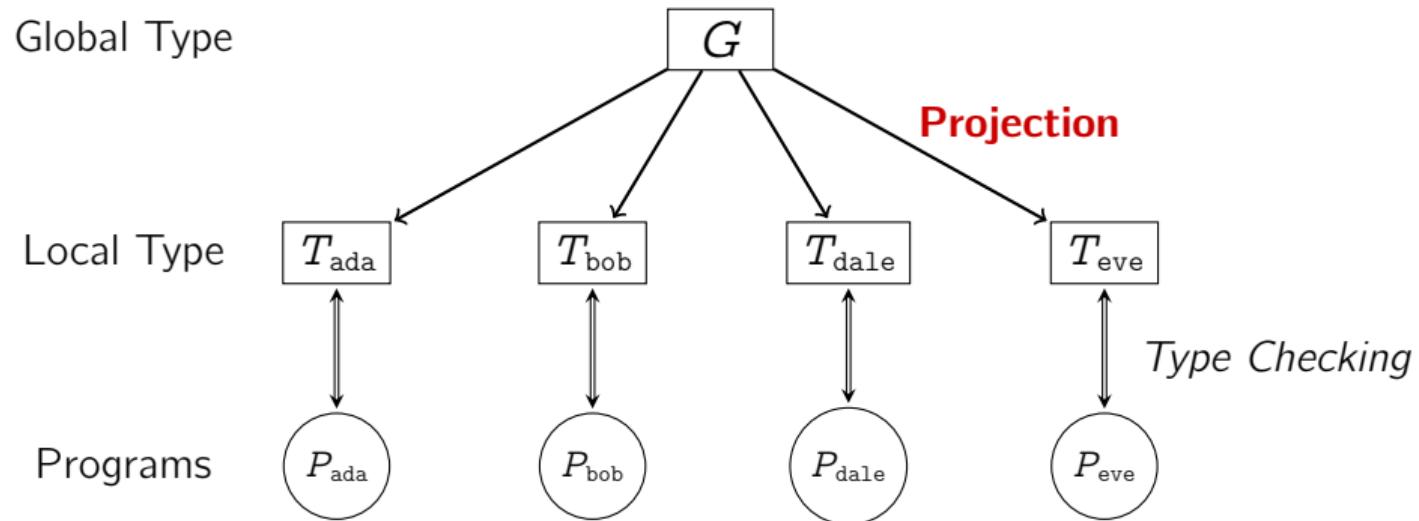
$G ::=$	$p \rightarrow q : \langle U \rangle . G$	Exchange of value of type $U$
	$p \rightarrow q : \{l_i : G_i\}_{i \in I}$	Branching
	$\mu t. G \quad   \quad t$	Recursion
	end	Terminated global protocol

## 2. Local types:

$T ::=$	$! \langle p, U \rangle . T$	Send value to $p$
	$? \langle p, U \rangle . T$	Receive value from $p$
	$\& \langle p, \{l_i : T_i\}_{i \in I} \rangle$	Offer labeled options $l_1, l_2, \dots$ to $p$
	$\oplus \langle p, \{l_i : T_i\}_{i \in I} \rangle$	Select from $l_1, l_2, \dots$ offered by $p$
	$\mu t. T \quad   \quad t \quad   \quad \text{end}$	Recursion / Terminated Protocol

# Multiparty Session Types

A methodology for **decentralized** specification, development, and validation of protocols between participants that communicate asynchronously:



# From Global to Local Types via Projection

The **projection** of  $G$  onto participant  $r$ , denoted  $G \upharpoonright r$ , is defined as follows:

- $(p \rightarrow q : \langle U \rangle . G') \upharpoonright r = \begin{cases} !\langle q, U \rangle . (G' \upharpoonright r) & \text{if } r = p \\ ?\langle p, U \rangle . (G' \upharpoonright r) & \text{if } r = q \\ G' \upharpoonright r & \text{otherwise} \end{cases}$
- $(p \rightarrow q : \{l_i : G_i\}_{i \in I}) \upharpoonright r = \begin{cases} \oplus \langle q, \{l_i : (G_i \upharpoonright r)\}_{i \in I} \rangle & \text{if } r = p \\ \& \langle p, \{l_i : (G_i \upharpoonright r)\}_{i \in I} \rangle & \text{if } r = q \\ G_j \upharpoonright r & \text{if } r \neq p, r \neq q, j \in I \text{ and } G_k \upharpoonright r = G_l \upharpoonright r, \text{ for all } k, l \in I \end{cases}$

# From Global to Local Types via Projection

The **projection** of  $G$  onto participant  $r$ , denoted  $G \upharpoonright r$ , is defined as follows:

- $(p \rightarrow q : \langle U \rangle . G') \upharpoonright r = \begin{cases} !\langle q, U \rangle . (G' \upharpoonright r) & \text{if } r = p \\ ?\langle p, U \rangle . (G' \upharpoonright r) & \text{if } r = q \\ G' \upharpoonright r & \text{otherwise} \end{cases}$
- $(p \rightarrow q : \{l_i : G_i\}_{i \in I}) \upharpoonright r = \begin{cases} \oplus \langle q, \{l_i : (G_i \upharpoonright r)\}_{i \in I} \rangle & \text{if } r = p \\ \& \langle p, \{l_i : (G_i \upharpoonright r)\}_{i \in I} \rangle & \text{if } r = q \\ G_j \upharpoonright r & \text{if } r \neq p, r \neq q, j \in I \text{ and } G_k \upharpoonright r = G_l \upharpoonright r, \text{ for all } k, l \in I \end{cases}$
- $(\mu t . G) \upharpoonright r = \begin{cases} \mu t . (G \upharpoonright r) & \text{if } G \upharpoonright r \neq t \\ \text{end} & \text{otherwise} \end{cases}$
- Also:  $t \upharpoonright r = t$  and  $\text{end} \upharpoonright r = \text{end}$

## Well-Formed Global Types

- ▶ **Fact:** Not all global types generated by the syntax make sense and/or can be implemented in a distributed setting.

## Well-Formed Global Types

- ▶ **Fact:** Not all global types generated by the syntax are make sense and/or can be implemented in a distributed setting.
- ▶ Example:

$$G = p \rightarrow q : \{ l_1 : q \rightarrow r \langle \text{Int} \rangle . \text{end} , \quad l_2 : r \rightarrow q \langle \text{Str} \rangle . \text{end} \}$$

In  $G$ , participant  $p$  communicates a choice (label  $l_1$  or  $l_2$ ) to  $q$ .  
This decision determines two different behaviors for  $r$ , which remains unaware.

# Well-Formed Global Types

- ▶ **Fact:** Not all global types generated by the syntax are make sense and/or can be implemented in a distributed setting.
- ▶ Example:

$$G = p \rightarrow q : \{l_1 : q \rightarrow r \langle \text{Int} \rangle.\text{end} , \quad l_2 : r \rightarrow q \langle \text{Str} \rangle.\text{end}\}$$

In  $G$ , participant  $p$  communicates a choice (label  $l_1$  or  $l_2$ ) to  $q$ .  
This decision determines two different behaviors for  $r$ , which remains unaware.

- ▶ In theories of MPSTs, non-sensical protocols are ruled out by defining **well-formed** global types — those that can be projected onto all participants.
- ▶  $G$  above is not well formed:  
projecting  $G$  onto  $r$  fails, because the behaviors under  $l_1$  and  $l_2$  are not equal.

# The Role of Projection

In MPSTs, projection addresses two intertwined issues:

1. Whether participants can be implemented (and how)
2. The class of well-formed global types

Projectability determines expressivity, measured by the “size” of the class of well-formed global types.

We explore this aspect by continuing our example.

## An Amended Protocol

Consider again  $G = p \rightarrow q : \{l_1 : q \rightarrow r \langle \text{Int} \rangle.\text{end} , l_2 : r \rightarrow q \langle \text{Str} \rangle.\text{end}\}$ .

We amend  $G$  by making  $p$ 's decision known to  $r$  by sending a message:

$$G' = p \rightarrow q : \{l_1 : \underbrace{q \rightarrow r : \{ \text{rcv} : q \rightarrow r \langle \text{Int} \rangle.\text{end} \}}_{G_1} , \\ l_2 : \underbrace{q \rightarrow r : \{ \text{snd} : r \rightarrow q \langle \text{Str} \rangle.\text{end} \}}_{G_2} \}$$

Is this enough? Let's consider the projections  $G_1|_r$  and  $G_2|_r$ . We have:

$$G_1|_r = \&\langle q, \{ \text{rcv} : ?\langle q, \text{Int} \rangle.\text{end} \} \rangle$$

$$G_2|_r = \&\langle q, \{ \text{snd} : !\langle q, \text{Str} \rangle.\text{end} \} \rangle$$

Because local types  $G_1|_r$  and  $G_2|_r$  are not identical,  $G'$  is still not well-formed.

# An Amended Protocol

The amended protocol:

$$G' = p \rightarrow q : \{ l_1 : \underbrace{q \rightarrow r : \{ \text{rcv} : q \rightarrow r \langle \text{Int} \rangle . \text{end} \}}_{G_1}, \\ l_2 : \underbrace{q \rightarrow r : \{ \text{snd} : r \rightarrow q \langle \text{Str} \rangle . \text{end} \}}_{G_2} \}$$

The non-identical projections  $G_1 \upharpoonright r$  and  $G_2 \upharpoonright r$ :

$$G_1 \upharpoonright r = \& \langle q, \{ \text{rcv} : ? \langle q, \text{Int} \rangle . \text{end} \} \rangle \quad G_2 \upharpoonright r = \& \langle q, \{ \text{snd} : ! \langle q, \text{Str} \rangle . \text{end} \} \rangle$$

Morally,  $G'$  should be projectable, though:

$G_1 \upharpoonright r$  and  $G_2 \upharpoonright r$  define the same behavior ( $r$  expecting a choice by  $q$ ) while determining separate branches.

## An Amended Protocol

The non-identical projections  $G_1 \upharpoonright r$  and  $G_2 \upharpoonright r$ :

$$G_1 \upharpoonright r = \&\langle q, \{ \text{rcv} : ?\langle q, \text{Int} \rangle.\text{end} \} \rangle \quad G_2 \upharpoonright r = \&\langle q, \{ \text{snd} : !\langle q, \text{Str} \rangle.\text{end} \} \rangle$$

Morally,  $G'$  should be projectable, though:

$G_1 \upharpoonright r$  and  $G_2 \upharpoonright r$  define the same behavior ( $r$  expecting a choice by  $q$ ) while determining separate branches.

What we need is a **more flexible** notion of projection, in which non-identical local types like  $G_1 \upharpoonright r$  and  $G_2 \upharpoonright r$  can be **merged** in a more general specification:

$$\&\langle q, \{ \text{rcv} : ?\langle q, \text{Int} \rangle.\text{end} , \text{snd} : !\langle q, \text{Str} \rangle.\text{end} \} \rangle$$

# Merging Local Types

[see, e.g., Deni  lou and Yoshida - ICALP'13]

Given local types  $T_1$  and  $T_2$ , their **merge**  $T_1 \sqcup T_2$  is defined inductively as:

$$\begin{aligned} \text{end} \sqcup \text{end} &= \text{end} \\ !\langle q, U \rangle . T \sqcup !\langle q, U \rangle . T &= !\langle q, U \rangle . T \\ ?\langle q, U \rangle . T \sqcup ?\langle q, U \rangle . T &= ?\langle q, U \rangle . T \\ \oplus \langle q, \{l_i : T_i\}_{i \in I} \rangle \sqcup \oplus \langle q, \{l_i : T_i\}_{i \in I} \rangle &= \oplus \langle q, \{l_i : T_i\} \rangle \\ \&\langle q, \{l_i : T_i\}_{i \in I} \rangle \sqcup \&\langle q, \{l'_j : T'_j\}_{j \in J} \rangle &= \&\langle q, \{l_i : T_i\}_{i \in I \setminus J} \cup \{l'_j : T'_j\}_{j \in J \setminus I} \\ &&&\cup \{l_k : T_k \sqcup T'_k\}_{k \in I \cap J} \rangle \end{aligned}$$

## Intuition:

Merge is equality on local types **except** for branching types ( $\&$ ):  
when  $l_k \notin I \cap J$  we combine branches, and when  $l_k \in I \cap J$  we merge further.

# Projection, Revisited

The **merge-based projection** of  $G$  onto participant  $r$ , denoted  $G \upharpoonright r$ , is defined as:

- $(p \rightarrow q : \langle U \rangle . G') \upharpoonright r = \begin{cases} !\langle q, U \rangle . (G' \upharpoonright r) & \text{if } r = p \\ ?\langle p, U \rangle . (G' \upharpoonright r) & \text{if } r = q \\ G' \upharpoonright r & \text{otherwise} \end{cases}$
- $(p \rightarrow q : \{l_i : G_i\}_{i \in I}) \upharpoonright r = \begin{cases} \oplus \langle q, \{l_i : (G_i \upharpoonright r)\}_{i \in I} \rangle & \text{if } r = p \\ \& \langle p, \{l_i : (G_i \upharpoonright r)\}_{i \in I} \rangle & \text{if } r = q \\ \sqcup_{i \in I} G_i \upharpoonright r & \text{otherwise} \end{cases}$
- $(\mu t. G) \upharpoonright r = \begin{cases} \mu t. (G \upharpoonright r) & \text{if } G \upharpoonright r \neq t \\ \text{end} & \text{otherwise} \end{cases}$
- Also:  $t \upharpoonright r = t$  and  $\text{end} \upharpoonright r = \text{end}$

# The Two-Buyer Protocol, Once Again

The global type  $G$  between Alice, Bob, and Seller:

$$\begin{aligned} G = & \text{Alice} \rightarrow \text{Seller} : \langle \text{book} \rangle. \\ & \text{Seller} \rightarrow \text{Alice} : \langle \text{quote} \rangle. \\ & \text{Alice} \rightarrow \text{Bob} : \langle \text{cost} \rangle. \\ & \text{Bob} \rightarrow \text{Alice} : \{ \text{share} : \text{Bob} \rightarrow \text{Alice} : \langle \text{address} \rangle. \\ & \qquad \qquad \qquad \text{Alice} \rightarrow \text{Bob} : \langle \text{ok} \rangle. \\ & \qquad \qquad \qquad \text{Seller} \rightarrow \text{Alice} : \langle \text{invoice} \rangle. \\ & \qquad \qquad \qquad \text{Alice} \rightarrow \text{Seller} : \langle \text{paym} \rangle. \text{end} \\ & \qquad \qquad \qquad \text{close} : \text{Alice} \rightarrow \text{Bob} : \langle \text{bye} \rangle. \text{end} \\ & \} \end{aligned}$$

$G$  is problematic: if Bob decides to close the transaction, Seller is not notified!  
We can amend the protocol by adding some explicit messages.

# The Two-Buyer Protocol, Once Again

A revised global type  $G'$  in which Seller is notified of the exchange between Alice and Bob:

$$\begin{aligned} G = & \text{Alice} \rightarrow \text{Seller} : \langle \text{book} \rangle. \\ & \text{Seller} \rightarrow \text{Alice} : \langle \text{quote} \rangle. \\ & \text{Alice} \rightarrow \text{Bob} : \langle \text{cost} \rangle. \\ & \text{Bob} \rightarrow \text{Alice} : \{ \text{share} : \text{Bob} \rightarrow \text{Alice} : \langle \text{address} \rangle. \\ & \qquad\qquad\qquad \text{Alice} \rightarrow \text{Bob} : \langle \text{ok} \rangle. \\ & \qquad\qquad\qquad \text{Bob} \rightarrow \text{Seller} : \{ \text{order} : \text{Seller} \rightarrow \text{Alice} : \langle \text{invoice} \rangle. \\ & \qquad\qquad\qquad \text{Alice} \rightarrow \text{Seller} : \langle \text{paym} \rangle.\text{end} \} \\ & \text{close} : \text{Alice} \rightarrow \text{Bob} : \langle \text{bye} \rangle.\text{end}. \\ & \text{Bob} \rightarrow \text{Seller} : \{ \text{cancel} : \text{Seller} \rightarrow \text{Alice} : \langle \text{bye} \rangle.\text{end} \} \\ & \} \end{aligned}$$

# The Two-Buyer Protocol, Once Again

The projections of  $G'$  onto Alice, Bob, and Seller:

$$\begin{aligned} G' \upharpoonright \text{Alice} = & \quad !\langle \text{Seller}, \text{book} \rangle . ?\langle \text{Seller}, \text{quote} \rangle . !\langle \text{Bob}, \text{cost} \rangle . \\ & \quad \& \langle \text{Bob}, \{\text{share} : ?\langle \text{Bob}, \text{address} \rangle , \\ & \quad \quad !\langle \text{Bob}, \text{ok} \rangle . ?\langle \text{Seller}, \text{invoice} \rangle . !\langle \text{Seller}, \text{paym} \rangle . \text{end} \\ & \quad \quad \text{close} : !\langle \text{Bob}, \text{bye} \rangle . !\langle \text{Seller}, \text{bye} \rangle . \text{end} \\ & \quad \} \rangle \end{aligned}$$
$$\begin{aligned} G' \upharpoonright \text{Bob} = & \quad ?\langle \text{Alice}, \text{cost} \rangle . \\ & \quad \oplus \langle \text{Alice}, \{\text{share} : !\langle \text{Alice}, \text{address} \rangle . ?\langle \text{Alice}, \text{ok} \rangle . \oplus \langle \text{Seller}, \{\text{order} : \text{end} \} \rangle \\ & \quad \quad \text{close} : ?\langle \text{Alice}, \text{bye} \rangle . \oplus \langle \text{Seller}, \{\text{cancel} : \text{end} \} \rangle \} \rangle \end{aligned}$$
$$\begin{aligned} G' \upharpoonright \text{Seller} = & \quad ?\langle \text{Alice}, \text{book} \rangle . !\langle \text{Alice}, \text{quote} \rangle . \\ & \quad \& \langle \text{Bob}, \{\text{share} : !\langle \text{Alice}, \text{invoice} \rangle . ?\langle \text{Alice}, \text{paym} \rangle . \text{end} , \\ & \quad \quad \text{cancel} : !\langle \text{Alice}, \text{bye} \rangle . \text{end} \} \rangle \end{aligned}$$

# Taking Stock

**Binary session types**

**Multiparty session types**

# Taking Stock

## Binary session types

- Describe protocols between **exactly** two partners

## Multiparty session types

# Taking Stock

## Binary session types

- Describe protocols between **exactly** two partners

## Multiparty session types

- Describe protocols between **more than** two partners

# Taking Stock

## Binary session types

- Describe protocols between **exactly** two partners
- A session type specifies the sequence of actions by some given participant

## Multiparty session types

- Describe protocols between **more than** two partners

# Taking Stock

## Binary session types

- Describe protocols between **exactly** two partners
- A session type specifies the sequence of actions by some given participant

## Multiparty session types

- Describe protocols between **more than** two partners
- A global type describes the overall interaction scenario.  
Local types: binary session types + participant identities.

# Taking Stock

## Binary session types

- Describe protocols between **exactly** two partners
- A session type specifies the sequence of actions by some given participant
- Compatibility defined in terms of duality

## Multiparty session types

- Describe protocols between **more than** two partners
- A global type describes the overall interaction scenario.  
Local types: binary session types + participant identities.

# Taking Stock

## Binary session types

- Describe protocols between **exactly** two partners
- A session type specifies the sequence of actions by some given participant
- Compatibility defined in terms of duality

## Multiparty session types

- Describe protocols between **more than** two partners
- A global type describes the overall interaction scenario.  
Local types: binary session types + participant identities.
- Compatibility enforced by projection into local types.  
Not all global types are well-formed (i.e., implementable).

# Taking Stock

## Binary session types

- Describe protocols between **exactly** two partners
- A session type specifies the sequence of actions by some given participant
- Compatibility defined in terms of duality
- Enhancements of compatibility via subtyping

## Multiparty session types

- Describe protocols between **more than** two partners
- A global type describes the overall interaction scenario.  
Local types: binary session types + participant identities.
- Compatibility enforced by projection into local types.  
Not all global types are well-formed (i.e., implementable).

# Taking Stock

## Binary session types

- Describe protocols between **exactly** two partners
- A session type specifies the sequence of actions by some given participant
- Compatibility defined in terms of duality
- Enhancements of compatibility via subtyping

## Multiparty session types

- Describe protocols between **more than** two partners
- A global type describes the overall interaction scenario.  
Local types: binary session types + participant identities.
- Compatibility enforced by projection into local types.  
Not all global types are well-formed (i.e., implementable).
- Enhancements via merge-based projectability (a form of subtyping)

# Session Types for Message-Passing Concurrency

Dan Frumin and Jorge A. Pérez

University of Groningen, The Netherlands

[www.rug.nl/fse/fc](http://www.rug.nl/fse/fc) - d.frumin | j.a.perez [[at]] rug.nl



UNIFYING  
C•RRECTNESS FOR  
C•MMUNICATING  
S•FTWARE

IPA Formal Methods - June 2022  
(Part 1, v1.1)