Session Types for Message-Passing Concurrency

Jorge A. Pérez
University of Groningen, The Netherlands
www.jperez.nl - j.a.perez [[at]] rug.nl

ISR - July 2021
(Part 1, v1.1)

This Course

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

1. Session types before 2010:
   Motivation, key ideas, essential notions of binary and multiparty session types.
2. Session types after 2010:
   The Curry-Howard correspondence between linear logic and session types (aka "propositions as sessions").

This Course

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

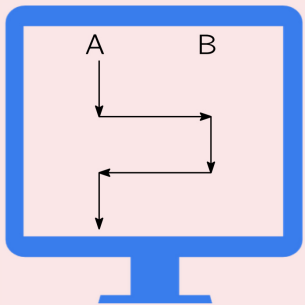
1. Session types before 2010:
   Motivation, key ideas, essential notions of binary and multiparty session types.
2. Session types after 2010:
   The Curry-Howard correspondence between linear logic and session types (aka "propositions as sessions").

My proposal: Part 1 $\rightarrow$ Q&A $\rightarrow$ Break $\rightarrow$ Part 2 $\rightarrow$ Q&A

Outline

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:

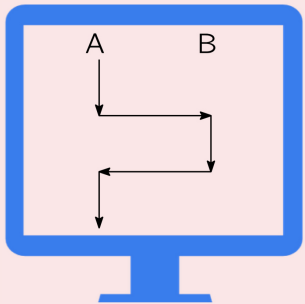
Message-Passing Concurrent Programs

An (imperfect) analogy:

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

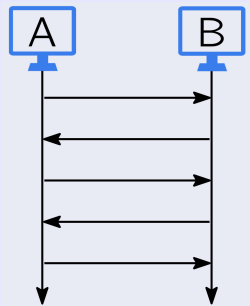
When is a Program Correct?

Sequential Programs



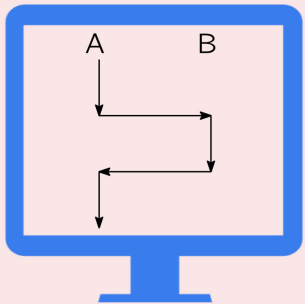
"Programs produce outputs that are consistent with their input"

Concurrent Programs



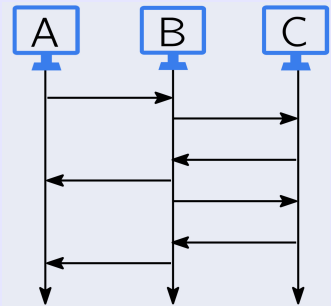
When is a Program Correct?

Sequential Programs



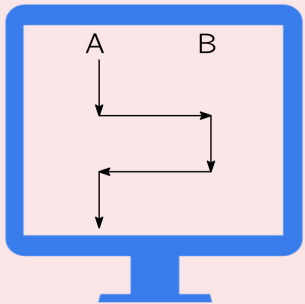
"Programs produce outputs that are consistent with their input"

Concurrent Programs



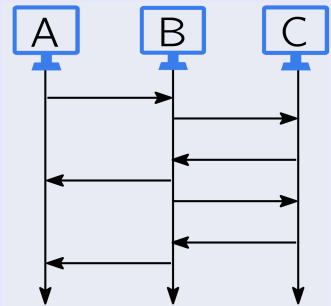
When is a Program Correct?

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 π-calculus treats **processes** like the λ-calculus treats **functions**

Keywords and Slogans

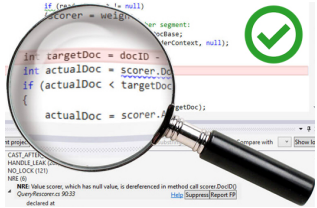
Concurrency Theory, Message-Passing, Programming Languages, Verification

- **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 π-calculus treats **processes** like the λ-calculus treats **functions**
- **Propositions as sessions**

$$\begin{array}{rcl} \text{Linear logic propositions} & \leftrightarrow & \text{session types} \\ \text{Proofs} & \leftrightarrow & \pi\text{-calculus processes} \\ \text{Cut elimination} & \leftrightarrow & \text{process communication} \end{array}$$

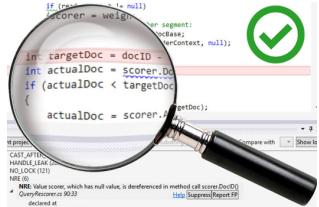
Type Systems

- 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

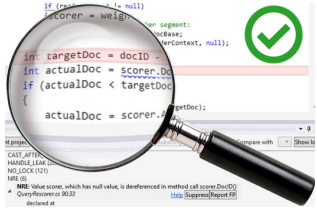


Sequential Languages

- **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

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**:

- channel endpoints
- channels in programming languages like Go
- π-calculus names
- ...

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

Protocols as Session Types

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

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

Protocols as Session Types

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

(Labels $l_1, \ldots, l_n$ 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:

1. 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:

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.

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.

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):

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

Example: A Two-Buyer Protocol

Two separate protocols, with Alice "leading" the interactions:

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

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

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

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

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:

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

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

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

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:

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

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

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

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

Session Type Duality, Formally

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

$$\begin{aligned}
\overline{!U;S} &= ?U;\overline{S} \\
\overline{?U;S} &= !U;\overline{S} \\
\overline{\&\{l_i : S_i\}_{i\in I}} &= \oplus\{l_i : \overline{S_i}\}_{i\in I} \\
\overline{\oplus\{l_i : S_i\}_{i\in I}} &= \&\{l_i : \overline{S_i}\}_{i\in I} \\
\overline{\mathsf{end}} &= \mathsf{end}
\end{aligned}$$

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:

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

Enhancing Compatibility via Subtyping

Consider a "mathematical server" and a candidate client

- The session type for the server:

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

- The session type for the client:

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

Enhancing Compatibility via Subtyping

Consider a "mathematical server" and a candidate client

- The session type for the server:

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

- The session type for the client:

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

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:

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

- The session type for the client:

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

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:

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

Enhancing Compatibility via Subtyping

Consider now an upgrade of the "mathematical server"

- The session type for the server known to clients:

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

- The session type for the upgraded server:

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

Enhancing Compatibility via Subtyping

Consider now an upgrade of the "mathematical server"

- The session type for the server known to clients:

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

- The session type for the upgraded server:

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

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:

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

- The session type for the upgraded server:

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

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 sustitutability**)

Enhancing Compatibility via Subtyping

- On the one hand, we have:

$$\underbrace{\&\begin{cases}\texttt{eq}: & \texttt{?Int};\ \texttt{?Int};\ \texttt{!Bool};\ \texttt{end}\\ \texttt{add}: & \texttt{?Int};\ \texttt{?Int};\ \texttt{!Int};\ \texttt{end}\end{cases}}_{\overline{T_1}} \leq \underbrace{\&\begin{cases}\texttt{eq}: & \texttt{?Real};\ \texttt{?Real};\ \texttt{!Bool};\ \texttt{end}\\ \texttt{add}: & \texttt{?Int};\ \texttt{?Int};\ \texttt{!Int};\ \texttt{end}\end{cases}}_{S}$$

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

Enhancing Compatibility via Subtyping

- On the one hand, we have:

$$\underbrace{\&\begin{cases}\texttt{eq}: & \texttt{?Int};\ \texttt{?Int};\ \texttt{!Bool};\ \texttt{end}\\ \texttt{add}: & \texttt{?Int};\ \texttt{?Int};\ \texttt{!Int};\ \texttt{end}\end{cases}}_{\overline{T_1}} \leq \underbrace{\&\begin{cases}\texttt{eq}: & \texttt{?Real};\ \texttt{?Real};\ \texttt{!Bool};\ \texttt{end}\\ \texttt{add}: & \texttt{?Int};\ \texttt{?Int};\ \texttt{!Int};\ \texttt{end}\end{cases}}_{S}$$

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

- We also have:

$$\underbrace{\&\begin{cases}\texttt{eq}: & \texttt{?Int};\ \texttt{?Int};\ \texttt{!Bool};\ \texttt{end}\\ \texttt{add}: & \texttt{?Int};\ \texttt{?Int};\ \texttt{!Int};\ \texttt{end}\end{cases}}_{S} \leq \underbrace{\&\begin{cases}\texttt{eq}: & \texttt{?Int};\ \texttt{?Int};\ \texttt{!Bool};\ \texttt{end}\\ \texttt{add}: & \texttt{?Int};\ \texttt{?Int};\ \texttt{!Int};\ \texttt{end}\\ \texttt{mul}: & \texttt{?Int};\ \texttt{?Int};\ \texttt{!Int};\ \texttt{end}\end{cases}}_{S'}$$

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

Subtyping, Formally

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

$$\frac{}{\mathsf{end} \leq \mathsf{end}} \qquad \frac{U_1 \leq U_2 \qquad S_1 \leq S_2}{!\,U_2;\ S_1 \ \leq\ !\,U_1;\ S_2} \qquad \frac{U_1 \leq U_2 \qquad S_1 \leq S_2}{?\,U_1;\ S_1 \ \leq\ ?\,U_2;\ S_2}$$

$$\frac{I \subseteq J \quad \forall i \in I.\, S_i \leq T_i}{\&\{l_i : S_i\}_{i \in I} \ \leq\ \&\{l_j : T_j\}_{j \in J}} \qquad \frac{J \subseteq I \quad \forall j \in J.\, S_j \leq T_j}{\oplus\{l_j : S_j\}_{j \in J} \ \leq\ \oplus\{l_i : T_i\}_{i \in I}}$$

Subtyping, Formally

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

$$\frac{}{\text{end} \leq \text{end}} \qquad \frac{U_1 \leq U_2 \qquad S_1 \leq S_2}{!\,U_2;\, S_1 \;\leq\; !\,U_1;\, S_2} \qquad \frac{U_1 \leq U_2 \qquad S_1 \leq S_2}{?\,U_1;\, S_1 \;\leq\; ?\,U_2;\, S_2}$$

$$\frac{I \subseteq J \quad \forall i \in I.\, S_i \leq T_i}{\&\{l_i : S_i\}_{i \in I} \;\leq\; \&\{l_j : T_j\}_{j \in J}} \qquad \frac{J \subseteq I \quad \forall j \in J.\, S_j \leq T_j}{\oplus\{l_j : S_j\}_{j \in J} \;\leq\; \oplus\{l_i : T_i\}_{i \in I}}$$

Notice:

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

Subtyping, Formally

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

$$\frac{}{\mathsf{end} \leq \mathsf{end}} \qquad \frac{U_1 \leq U_2 \qquad S_1 \leq S_2}{!\,U_2;\, S_1 \;\leq\; !\,U_1;\, S_2} \qquad \frac{U_1 \leq U_2 \qquad S_1 \leq S_2}{?\,U_1;\, S_1 \;\leq\; ?\,U_2;\, S_2}$$

$$\frac{I \subseteq J \quad \forall i \in I.\, S_i \leq T_i}{\&\{l_i : S_i\}_{i \in I} \;\leq\; \&\{l_j : T_j\}_{j \in J}} \qquad \frac{J \subseteq I \quad \forall j \in J.\, S_j \leq T_j}{\oplus\{l_j : S_j\}_{j \in J} \;\leq\; \oplus\{l_i : T_i\}_{i \in I}}$$

Notice:

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

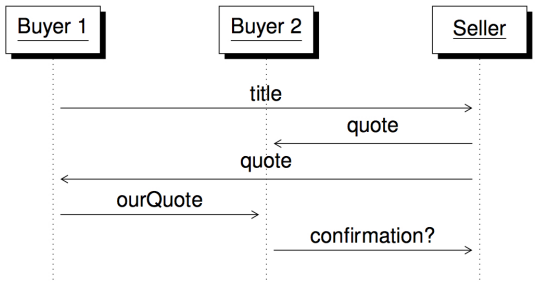
Also:

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

Outline

From Binary to Multiparty Protocols

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



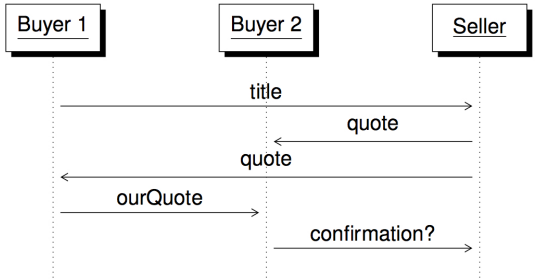
From Binary to Multiparty Protocols

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

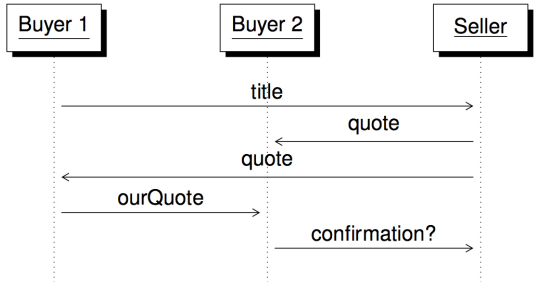
From Binary to Multiparty Protocols

- 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.

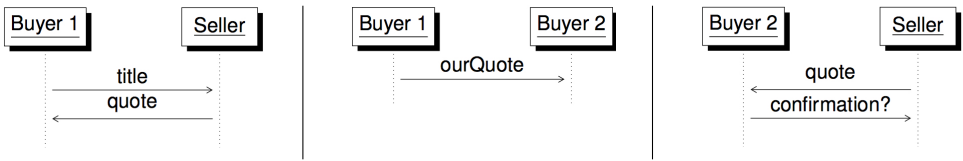
The Need for Sequencing Information



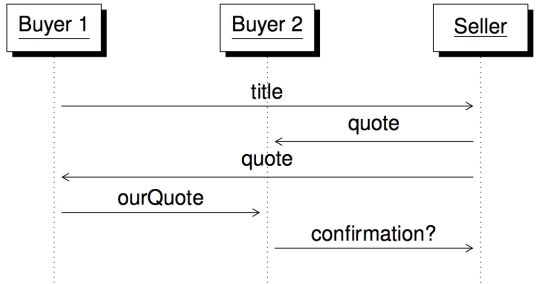
The Need for Sequencing Information



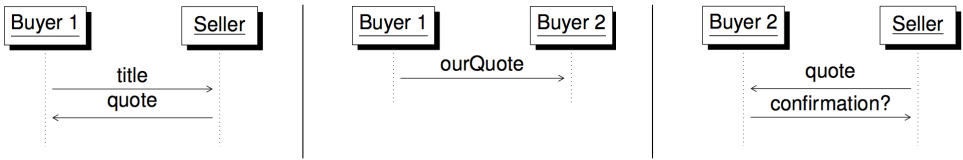
A decomposition as binary protocols may appear plausible...



The Need for Sequencing Information



A decomposition as binary protocols may appear plausible...



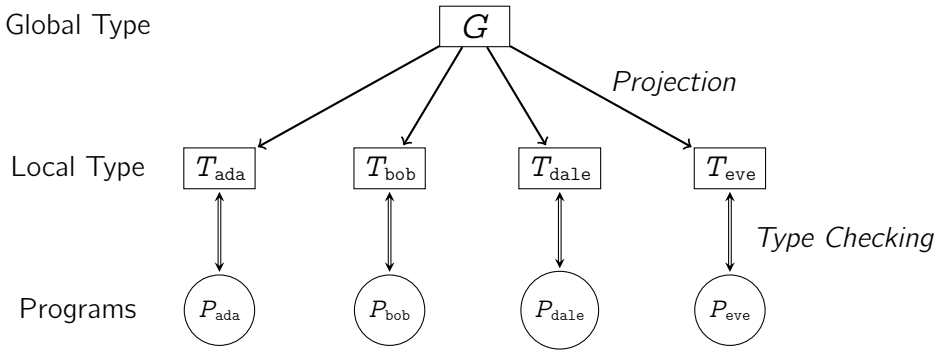
... but misses key sequencing between unrelated partners.

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 multiple participants:



The Syntax of Multiparty Session Types

We use $\mathtt{p}, \mathtt{q}, \ldots$ to denote participants. U is the type of transmittable values.

1. Global types:

$$
\begin{array}{rlll}
G ::= & \mathtt{p} \to \mathtt{q} : \langle U \rangle . G & & \text{Exchange of value of type } U \\
\mid & \mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I} & & \text{Branching} \\
\mid & \mu t . G \quad \mid \quad t & & \text{Recursion} \\
\mid & \text{end} & & \text{Terminated global protocol}
\end{array}
$$

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.

$$
\begin{array}{rll}
G = \mathsf{Alice} \to \mathsf{Seller} : & \langle \texttt{book} \rangle . & \\
\mathsf{Seller} \to \mathsf{Alice} : & \langle \texttt{quote} \rangle . & \\
\mathsf{Alice} \to \mathsf{Bob} : & \langle \texttt{cost} \rangle . & \\
\mathsf{Bob} \to \mathsf{Alice} : & \{ \text{ share} : & \mathsf{Bob} \to \mathsf{Alice} : \langle \texttt{address} \rangle . \\
 & & \mathsf{Alice} \to \mathsf{Bob} : \langle \texttt{ok} \rangle . \\
 & & \mathsf{Seller} \to \mathsf{Alice} : \langle \texttt{invoice} \rangle . \\
 & & \mathsf{Alice} \to \mathsf{Seller} : \langle \texttt{paym} \rangle . \mathsf{end} \\
 & \text{close} : & \mathsf{Alice} \to \mathsf{Bob} : \langle \texttt{bye} \rangle . \mathsf{end} \\
 & \} &
\end{array}
$$

(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 $\mathtt{p}, \mathtt{q}, \ldots$ to denote participants. U is the type of transmittable values.

1. Global types:

$$
\begin{array}{rcll}
G ::= & \mathtt{p} \to \mathtt{q} : \langle U \rangle . G & & \text{Exchange of value of type } U \\
\mid & \mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I} & & \text{Branching} \\
\mid & \mu t . G & \mid \quad t & \text{Recursion} \\
\mid & \text{end} & & \text{Terminated global protocol}
\end{array}
$$

The Syntax of Multiparty Session Types

We use $\mathtt{p}, \mathtt{q}, \ldots$ to denote participants. U is the type of transmittable values.

1. Global types:

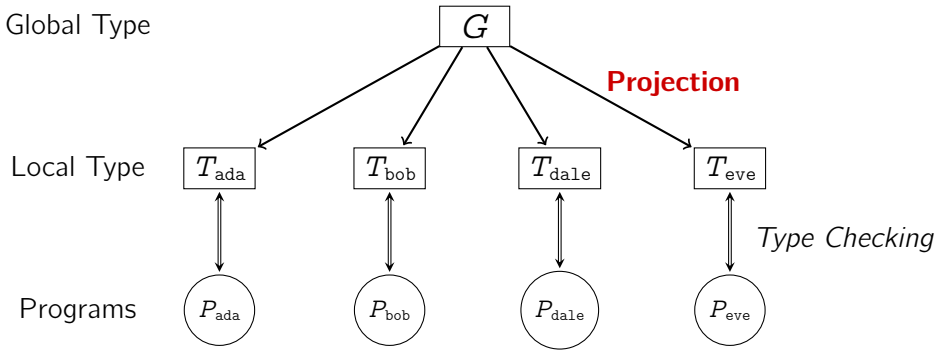
$G ::=$	$\mathtt{p} \to \mathtt{q} : \langle U \rangle . G$		Exchange of value of type U
$\mid$	$\mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I}$		Branching
$\mid$	$\mu t . G$	$\mid \; t$	Recursion
$\mid$	$\mathtt{end}$		Terminated global protocol

2. Local types:

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

Multiparty Session Types

A methodology for **decentralized** specification, development, and validation of protocols between multiple participants:



From Global to Local Types via Projection

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

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

From Global to Local Types via Projection

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

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

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.

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:

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

In G, participant $\mathtt{p}$ communicates a choice (label l_1 or l_2) to $\mathtt{q}$.
This decision determines two different behaviors for $\mathtt{r}$, 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:

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

  In G, participant $\mathtt{p}$ communicates a choice (label l_1 or l_2) to $\mathtt{q}$.
  This decision determines two different behaviors for $\mathtt{r}$, 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 is not well formed:
  projecting G onto $\mathtt{r}$ 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 $G = \mathtt{p} \to \mathtt{q} : \{l_1 : \mathtt{q} \to \mathtt{r}\langle \mathsf{Int}\rangle.\mathsf{end} \;\;,\;\; l_2 : \mathtt{r} \to \mathtt{q}\langle \mathsf{Str}\rangle.\mathsf{end}\}$.

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

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

Is this enough? Let's consider the projections $G_1{\upharpoonright}\mathtt{r}$ and $G_2{\upharpoonright}\mathtt{r}$. We have:

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

Because local types $G_1{\upharpoonright}\mathtt{r}$ and $G_2{\upharpoonright}\mathtt{r}$ are not identical, G' is still not well-formed.

An Amended Protocol

The amended protocol:

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

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

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

Morally, G' should be projectable, though:

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

An Amended Protocol

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

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

Morally, G' should be projectable, though:

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

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

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

Merging Local Types [see, e.g., Deniélou and Yoshida - ICALP'13]

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

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

Intuition:

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

Projection, Revisited

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

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

The Two-Buyer Protocol, Once Again

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

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

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:

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

The Two-Buyer Protocol, Once Again

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

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

$$
\begin{aligned}
G'\upharpoonright \mathsf{Bob} = \; & ?\langle \mathsf{Alice}, \mathtt{cost}\rangle. \\
& \oplus\langle \mathsf{Alice}, \{\mathsf{share} : !\langle \mathsf{Alice}, \mathtt{address}\rangle.?\langle \mathsf{Alice}, \mathtt{ok}\rangle. \oplus \langle \mathsf{Seller}, \{\mathsf{order} : \mathtt{end}\ \}\rangle \\
& \qquad \mathsf{close} : ?\langle \mathsf{Alice}, \mathtt{bye}\rangle. \oplus \langle \mathsf{Seller}, \{\mathsf{cancel} : \mathtt{end}\ \}\rangle\ \}\rangle
\end{aligned}
$$

$$
\begin{aligned}
G'\upharpoonright \mathsf{Seller} = \; & ?\langle \mathsf{Alice}, \mathtt{book}\rangle.!\langle \mathsf{Alice}, \mathtt{quote}\rangle. \\
& \&\langle \mathsf{Bob}, \{\mathsf{share} : !\langle \mathsf{Alice}, \mathtt{invoice}\rangle.?\langle \mathsf{Alice}, \mathtt{paym}\rangle.\mathtt{end}\ , \\
& \qquad \mathsf{cancel} : !\langle \mathsf{Alice}, \mathtt{bye}\rangle.\mathtt{end}\}\rangle
\end{aligned}
$$

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 session type 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.
- Global type projection into local types enforces compatibility.
  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

Jorge A. Pérez
University of Groningen, The Netherlands
www.jperez.nl - j.a.perez [[at]] rug.nl

ISR - July 2021
(Part 1, v1.1)