On the Expressiveness of Asynchronous Multiparty Session Types

Romain Demangeon - Nobuko Yoshida

UPMC (Paris) - Imperial College (London)

Séminaire APR / GdT Prog. - 10/12/2015
Motivation

Background

- Asynchronous networks of distributed applications,
  - existence of buffers storing exchanged messages,
- Verification of multiparty protocols.
- Sessions as behavioural types for applications.
- Rich formalism:
  - parallel composition,
  - sequence subtyping (flexibility),
  - interruptible blocks, ... 

- Expressiveness of asynchronous multiparty sessions.
  - How to give a denotational semantics to sessions ?
  - How buffers affects semantics ?
  - Are flexible and interruptible sessions more expressive ?
Non-centralised Protocols

- Three independent applications (client, agent, instrument):
  - written in different languages,
  - with local compilers and libraries,
  - message-passing communication.

- No global control.

- Goal: enforcing interaction success.
  - Message layer soundness.
  - Method: session types.
Non-centralised Protocols

- Three independent applications (client, agent, instrument):
  - written in different languages,
  - with local compilers and libraries,
  - message-passing communication.

- No global control.

- Goal: enforcing interaction success.
  - Message layer soundness.
  - Method: session types.
Non-centralised Protocols

- Three independent applications (client, agent, instrument):
  - written in different languages,
  - with local compilers and libraries,
  - message-passing communication.
- No global control.
- Goal: enforcing interaction success.
  - Message layer soundness.
  - Method: session types.
Non-centralised Protocols

- Three independent applications (client, agent, instrument):
  - written in different languages,
  - with local compilers and libraries,
  - message-passing communication.

- No global control.
- Goal: enforcing interaction success.
  - Message layer soundness.
  - Method: session types.
Session Types

- **Behavioural Types.**
  - characterise operational semantics properties.

- Historically: **binary sessions**, *Languages Primitives and Type Discipline for Structured Communication-Based Programming*, Honda, Kubo, Vasconcelos, ESOP 1998
  - Domain: process algebras (π-calculi): messages-passing agents communicating on channels.
  - Motivation: build types to guide interactions between two agents on a same channel.

- **Principles:**
  - Formally describing interactions between two agents (a *session*) on a single channel *s*.
    - Using communication (directed choice, label), choice, recursion, session end.
  - Dividing the session in two endpoint types (similar to CCS processes).
  - Validation, (type system) of each participant w.r.t. its type.

- Use sequence inside π types:
  - simple types for π: $a : \#^i((\text{Nat}, \#^o(\text{Bool})))$.
  - session types: $s : ?(\text{Nat}); !(\text{Bool})$. 
Binary Sessions - Example

- **Global type / session:**
  \[ G = p \rightarrow q : \text{price} \cdot q \rightarrow p \begin{cases} \text{KO.end} \\ \text{OK} \cdot p \rightarrow q : \text{order.end} \end{cases} \]

- **Local types / end points:**
  \[ T_p : \text{!price.} \begin{cases} \text{KO.end} \\ \text{OK.} \text{!order.end} \end{cases} \]
  \[ T_q : \text{?price.} \begin{cases} \text{KO.end} \\ \text{OK.} \text{?order.end} \end{cases} \]

- **Candidate processes (π):**
  - \( s_{\text{price}}(x) \cdot (\overline{\text{OK}} \cdot s_{\text{order}}(o) + \overline{\text{KO}}) : \)
Binary Sessions - Example

- **Global type / session:**
  \[ G = p \rightarrow q : \text{price}.q \rightarrow p \begin{cases} 
  \text{KO}.\text{end} \\
  \text{OK}.p \rightarrow q : \text{order}.\text{end} 
\end{cases} \]

- **Local types / end points:**
  \[ T_p : !\text{price}.? \begin{cases} 
  \text{KO}.\text{end} \\
  \text{OK}.!\text{order}.\text{end} 
\end{cases} \]
  \[ T_q : ?\text{price}.! \begin{cases} 
  \text{KO}.\text{end} \\
  \text{OK}.?\text{order}.\text{end} 
\end{cases} \]

- **Candidate processes (\(\pi\)):**
  - \(s_{\text{price}}(x).(\overline{s}_{\text{OK}}.s_{\text{order}}(o) + \overline{s}_{\text{KO}}) : \text{good } q\).
  - \(s_{\text{price}}(x).\overline{s}_{\text{KO}} : \)
 Binary Sessions - Example

- **Global type / session:**
  \[ G = p \rightarrow q : \text{price} . q \rightarrow p \begin{cases} \text{KO.end} \\ \text{OK.p} \rightarrow q : \text{order.end} \end{cases} \]

- **Local types / end points:**
  \[ T_p : !\text{price} . ? \begin{cases} \text{KO.end} \\ \text{OK!order.end} \end{cases} \]
  \[ T_q : ?\text{price} . ! \begin{cases} \text{KO.end} \\ \text{OK.?order.end} \end{cases} \]

- **Candidate processes (\( \pi \)):**
  - \( s_{\text{price}}(x) . (\bar{s}_{\text{OK}} . s_{\text{order}}(o) + \bar{s}_{\text{KO}}) : \text{good q.} \)
  - \( s_{\text{price}}(x) . \bar{s}_{\text{KO}} : \text{good q.} \)
  - \( \bar{s}_{\text{price}}(100 \text{ Fr}) . s_{\text{KO}} : \)
Global type / session:

\[ G = p \rightarrow q : \text{price.q} \rightarrow p \begin{cases} \text{KO.end} \\ \text{OK.p} \rightarrow q : \text{order.end} \end{cases} \]

Local types / end points:

\[ T_p : \text{!!price.?} \begin{cases} \text{KO.end} \\ \text{OK.!order.end} \end{cases} \]
\[ T_q : \text{?!price.!} \begin{cases} \text{KO.end} \\ \text{OK.?order.end} \end{cases} \]

Candidate processes (\(\pi\)):

- \( s_{\text{price}}(x).(\overline{s}_{\text{OK}}.s_{\text{order}}(o) + \overline{s}_{\text{KO}}) : \text{good q.} \)
- \( s_{\text{price}}(x).\overline{s}_{\text{KO}} : \text{good q.} \)
- \( \overline{s}_{\text{price}}(100 \text{ Fr}).s_{\text{KO}} : \text{bad p.} \)
Multiparty Session Types

- Sessions with (at least) 3 participants [Honda Y. Carbone 08].
- Same principles (projection).
- Symmetry is lost.

Example

| G = r → q : m, q → p : m₁, r → p : m₂.end |
| Tₚ : q?m₁.r?m₂.end |
| Tₗ : r?m.p!m₁.end |
| Tᵣ : q!m.p!m₂.end |

Semantics:

- Let $A, B$ be applications s.t. $\vdash A : Tᵣ$ and $\vdash B : Tₗ$
- $A$ can send message $m$ and $B$ can receive it (giving $A', B'$).
- $\vdash A' : p!m₂.end$ and $\vdash B' : p!m₁.end$
- At type level, reduction semantics:
  
  $\vdash q!m.p!m₂.end \mid r?m.p!m₁.end \to p!m₂.end \mid p!m₁.end$
MPST as a Verification Method

- Verification of **networks** of services and applications:
  - **non-centralised** networks
    - message-passing communication,
    - no global control.
  - **specification**: global interaction choreographies between several participants.
  - **Theorem**: local type enforcement
    \[ \Rightarrow \text{global progress} \] (according to the specification).
  - **Session refinement**: enforcing other **properties** (security, state).

- **Endpoint verification**:
  - **validation**: static analysis of the program (typechecker).
  - **monitoring**: runtime analysis of I/O.

*Scribble* language: algorithm for projection and monitor generation.
MPST as a Verification Method (II)

(from Monitoring Networks through Multiparty Session Types)
Asynchronous Networks

$.ajax({
    dataType: "jsonp",
    jsonpCallback: "callback",
    success: "store_price"}
}

$.ajax({
    dataType: "jsonp",
    jsonpCallback: "callback",
    success: "store_price"}
}

- **Asynchronous** calls through the web.
- **Verification:**
  - monitors intercepting HTTP requests and responses.
  - local type: p!price.p?answer.q!price.q?answer.end
Asynchronous Networks

$.ajax({
    dataType: "jsonp",
    jsonpCallback: "callback",
    success: "store_price" }
})

$.ajax({
    dataType: "jsonp",
    jsonpCallback: "callback",
    success: "store_price" }
})

- **Asynchronous** calls through the web.
- **Verification:**
  - monitors intercepting HTTP requests and responses.
  - local type: p!price.p?answer.q!price.q?answer.end
- **Asynchrony:** answer from q can arrive before answer from p.
Asynchronous Networks

$.ajax({
    dataType: "jsonp",
    jsonpCallback: "callback",
    success: "store_price"
})

$.ajax({
    dataType: "jsonp",
    jsonpCallback: "callback",
    success: "store_price"
})

▶ Asynchronous calls through the web.
▶ Verification:
  ▶ monitors intercepting HTTP requests and responses.
  ▶ local type: p!price.p?answer.q!price.q?answer.end
▶ Asynchrony: answer from q can arrive before answer from p.
▶ p!price.p?answer || q!price.q?answer ?
  ▶ sending order is lost,
  ▶ implementation of ||.
Asynchronous Multiparty Session Types

- Messages "take time" to reach their destination.
- Queues are used to model travelling messages.
  - input queues: inbox storing arriving messages.
  - output queues: buffer storing messages to be sent.
- Order of arriving messages can change.
  - order between messages with same source and same target is preserved.

Example

\[ G = r \rightarrow q : m, q \rightarrow p : m_1, r \rightarrow p : m_2.\text{end} \]

- with asynchronous semantics \( m_2 \) can arrive before \( m_1 \).
AMST Basic Syntax

\[ G ::= \text{end} \mid \mu t. G \mid t \mid r_1 \rightarrow r_2 \{ m_i \cdot G_i \}_{i \in I} \]
\[ T ::= \text{end} \mid \mu t. T \mid t \mid p? \{ m_i \cdot T_i \}_{i \in I} \mid p! \{ m_i \cdot T_i \}_{i \in I} \]

- Simple presentation:
  - directed choice inside communication,
  - recursion.
- Projection divides communication into input and output.

AMST Semantics

How to give an asynchronous operational semantics to types?

- usage of queues (store) at different places in the network.
- queues are order-preserving.
We compare the expressiveness of several type systems:

- Models with input and/or output queues,
- Sequence subtyping (switching interaction order at type level),
- Parallel composition.
- Interruptible sessions (encoding ?).

### Queue models

Different models used in literature:

- **input** queues storing arriving messages:
  - none: participants consume messages from the network,
  - one: each participant has one inbox for all incoming messages,
  - several: each participant has one inbox for each other participant,
- same choices for **output** queue design.
- yields 9 different queue policy \((0, 0), (1, 0), (M, 1), \ldots\)
Example: One input queue

<table>
<thead>
<tr>
<th>Initial</th>
<th>Ongoing</th>
<th>Deadlock</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r : q!m.p!m_2.end$</td>
<td>$r : end$</td>
<td>$r : end$</td>
</tr>
<tr>
<td>$p : q?m_1.r?m_2.end$</td>
<td>$p : q?m_1.r?m_2.end$</td>
<td>$p : q?m_1.r?m_2.end$</td>
</tr>
</tbody>
</table>

- **Single input queue**: one inbox per participant.
- **Asynchrony** let $m_2$ arrive before $m_1$.
  - $p$ expects to receive $m_1$ first.
- System is **deadlocked**.
Example: One input queue

<table>
<thead>
<tr>
<th>Initial</th>
<th>Ongoing</th>
<th>Deadlock</th>
</tr>
</thead>
<tbody>
<tr>
<td>r: q!m.p!m_2.end</td>
<td>r: end</td>
<td>r: end</td>
</tr>
<tr>
<td>rIMENTH</td>
<td>ε</td>
<td>ε</td>
</tr>
<tr>
<td>q: r?m.p!m_1.end</td>
<td>q: r?m!m_1end</td>
<td>q: end</td>
</tr>
<tr>
<td>qIMENTH</td>
<td>ε</td>
<td>ε</td>
</tr>
<tr>
<td>p: q?m_1.r?m_2.end</td>
<td>p: q?m_1.r?m_2.end</td>
<td>p: q?m_1.r?m_2.end</td>
</tr>
<tr>
<td>pIMENTH</td>
<td>ε</td>
<td>⟨m⟩</td>
</tr>
<tr>
<td>⟨m⟩</td>
<td>⟨m_2⟩</td>
<td>⟨m_2⟩ ⟨m_1⟩</td>
</tr>
</tbody>
</table>

- Single input queue: one inbox per participant.
- Asynchrony let $m_2$ arrive before $m_1$.
  - p expects to receive $m_1$ first.
- System is deadlocked.
- Single input queues → wrong semantics.
Example: Multiple input queues

- **Initial**
  - \( r : q!m.p.m_2.\text{end} \)
  - \( q : r?m.p!m_1.\text{end} \)
  - \( p : q?m_1.r?m_2.\text{end} \)

- **Ongoing**
  - \( r : \text{end} \)
  - \( q : r?m.p!m_1.\text{end} \)
  - \( p : q?m_1.r?m_2.\text{end} \)

- **No Deadlock**
  - \( r : \text{end} \)
  - \( q : \text{end} \)
  - \( p : q?m_1.r?m_2.\text{end} \)

- **Multiple input queues**: one inbox per pair of participants.
- **Asynchrony** let \( m_2 \) arrive before \( m_1 \).
  - \( p \) expects to receive \( m_1 \) first.
- **System can progress** because \( m_1 \) is available in one input queue.
- **Multiple input queues semantics is safe.**
Configuration Semantics

\[ \Delta ::= \emptyset \mid p : T, \Delta \mid Q, \Delta \]
\[ Q ::= (p \leftarrow q : h) \mid (p \rightarrow q : h) \mid (p \leftarrow h) \mid (p \rightarrow h) \]

- Configuration: composition of participants (local types) and queues,
- Input \( \leftarrow \) and output \( \rightarrow \) queues.
- Single \( p \leftarrow \) and multiple \( p \leftarrow q \) queues.
- Global type \( G \rightarrow \) initial configuration (projection and empty queues).
- Rules enforces message transfer.

**Rule (OutOut)**

- \( q : p?m. T_q \)
- \( q \rightarrow \ldots \)
- \( q \leftarrow r \ldots q \leftarrow p \ldots \)
- \( p : q!m. T_p \)
- \( p \rightarrow \ldots \)
- \( p \leftarrow r \ldots p \leftarrow q \ldots \)

**Rule (Transit)**

- \( q : p?m. T_q \)
- \( q \rightarrow \ldots \)
- \( q \leftarrow r \ldots q \leftarrow p \ldots \)
- \( p : T_p \)
- \( p \rightarrow \ldots \)
- \( p \leftarrow r \ldots p \leftarrow q \ldots \)

**Rule (InIn)**

- \( q : p?m. T_q \)
- \( q \rightarrow \ldots \)
- \( q \leftarrow r \ldots q \leftarrow p \langle m \rangle \ldots \)
- \( p : T_p \)
- \( p \rightarrow \ldots \)
- \( p \leftarrow r \ldots p \leftarrow q \ldots \)
Queue policy guides the semantics rules.

(p ≪ q : h) stands for either (p ⩾ q : h) or (p ⩾: h).
Trace Denotations

- **Configuration traces** as a measure for expressiveness.

- A trace \( \sigma \) is a **mapping** from participants to **sequence of events**.
  - An **event** is either sending or receiving a message.
  - There is **no order** between events of different participants.
  - Order is **kept** between events of a same participant.
  - Transit of messages (from queue to queue) is **not observable**.

- A trace \( \sigma \) is **terminated** w.r.t. a type \( G \) if the initial configuration of \( G \) cannot progress after \( \sigma \).
  - Captures **deadlocks**.
  - Depends on the queue **policy**.

- A trace \( \sigma \) is **completed** w.r.t. a type \( G \) if the initial configuration of \( G \) reaches **happy termination** after \( \sigma \).
  - After \( \sigma \) participants reaches end and queues are empty.
Example

- **Global type:**
  \[ r \to q : m, q \to p : m_1, r \to p : m_2. \text{end} \]

- **Initial configuration for \((0, 0)\):**
  \[ r : q!m.p!m_2, \ q : r?m.p!m_1, \ p : q?m_1.r?m_2, \]

- **Initial configuration for \((M, 1)\):**
  \[ r : q!m.p!m_2, \ q : r?m.p!m_1, \ p : q?m_1.r?m_2, \]
  \( (p \rhd: \epsilon), (q \rhd: \epsilon), (r \rhd: \epsilon), (p \blacklozenge q : \epsilon), (p \blacklozenge r : \epsilon), \)
  \( (q \blacklozenge p : \epsilon), (q \blacklozenge r : \epsilon), (r \blacklozenge p : \epsilon), (r \blacklozenge q : \epsilon). \]

- **Trace \(\sigma_e\):**
  \[
  \begin{align*}
  r & \mapsto q!m.p!m_2 \\
  q & \mapsto r?m.p!m_1 \\
  p & \mapsto q?m_1.r!m_2 \\
  \end{align*}
  \]
  is completed for both semantics.

- **Trace \(\sigma_t\):**
  \[
  \begin{align*}
  p & \mapsto q!m \\
  q & \mapsto \epsilon \\
  r & \mapsto \epsilon \\
  \end{align*}
  \]
  is a valid (uncompleted) trace for \((M, 1)\) and not for \((0, 0)\).

- **Trace \(\sigma_d\):**
  \[
  \begin{align*}
  r & \mapsto q!m.p!m_2 \\
  q & \mapsto r?m.p!m_1 \\
  p & \mapsto \epsilon \\
  \end{align*}
  \]
  is terminated for \((1, 0)\) but not for \((0, M)\).
Expressiveness Results

- \( \mathbf{D}(G, \phi) \), the **denotation** of \( G \) under semantics (queue policy) \( \phi \) is the set of all terminated traces from \( G \) according to \( \phi \).
- the **expressive power** of a session calculus (syntax + semantics) is defined as the **language** of all completed traces for all well-formed types.

**Results**

- **Single input** queue policy \((1, 0), (1, 1), (1, M)\) are **unsafe**.
  - they do not ensure **progress**.
  - all other semantics are **safe**.
- All safe semantics yield the same **denotations**.
- The expressive power of safe semantics is **regular**.

**Intuition**: Local actions are constrained by type.
Flexibility

- Real applications often have mechanisms to accept messages in different order.
  - unordered data structures, threads, . . .
- At the level of local type, modeled with flexibility subtyping:
  - exists in literature,
  - rules allow to switch consecutive actions.
  - 6 subtyping policies ($\emptyset, II, OO, IO, IO, IO/OI$)

Example ($II$)-flexibility

- $p?m_1.q?m_2.p!m_3.end$ switches to $q?m_2.p?m_1.p!m_3.end$. 
Flexibility

- Real applications often have mechanisms to accept messages in different order.
  - unordered data structures, threads, ... 
- At the level of local type, modeled with flexibility subtyping:
  - exists in literature,
  - rules allow to switch consecutive actions.
  - 6 subtyping policies ($\emptyset$, $II$, $OO$, $IO$, $OI$, $IO/OI$)

### Example ($II$)-flexibility

- $p?m_1.q?m_2.p!m_3.end$ switches to $q?m_2.p?m_1.p!m_3.end$.
- $p? \begin{cases} m_{11}.q?m_2.p!m_3.end \\ m_{12}.q?m_2.p!m_4.end \end{cases}$ switches to $q?m_2.p? \begin{cases} m_{11}.p!m_3.end \\ m_{12}.p!m_4.end \end{cases}$
Subtyping Rules

\[ C_i^q ::= [ ] \mid p?\{m_i.C_i^q\}_{i \in I} \ (p \neq q) \]
\[ C_{1O}^q ::= [ ] \mid p?\{m_i.C_{1O}^q\}_{i \in I} \mid r!\{m_i.C_{1O}^q\}_{i \in I} \ (r \neq q) \]
\[ C_0^q ::= [ ] \mid q!\{m_i.C_0^q\}_{i \in I} \ (p \neq q) \]
\[ C_{0I}^q ::= [ ] \mid p!\{m_i.C_{0I}^q\}_{i \in I} \ (p \neq q) \mid r?\{m_i.C_{0I}^q\}_{i \in I} \]

\[ \forall (i, k), T_i \leq q?m_k.C_i^p[T_i'] \quad q \neq p \]
\[ p?\{m_i.T_i\}_{i \in I} \leq q\{m_k.C_i^p[p?\{T_i'\}_{i \in I}]\}_{k \in K} \]  

(II)

\[ \forall (i, k), T_i \leq q!m_k.C_0^p[T_i'] \quad q \neq p \]
\[ p!\{m_i.T_i\}_{i \in I} \leq q\{m_k.C_0^p[p!\{T_i'\}_{i \in I}]\}_{k \in K} \]  

(OO)

\[ \forall (i, k), T_i \leq q!m_k.C_{1O}^p[T_i'] \quad q \neq p \]
\[ p!\{m_i.T_i\}_{i \in I} \leq q\{m_k.C_{1O}^p[p!\{T_i'\}_{i \in I}]\}_{k \in K} \]  

(IO)

\[ \forall (i, k), T_i \leq q!m_k.C_{0I}^p[T_i'] \quad q \neq p \]
\[ p?\{m_i.T_i\}_{i \in I} \leq q\{m_k.C_{0I}^p[p?\{T_i'\}_{i \in I}]\}_{k \in K} \]  

(OI)

▶ Formal definition of flexibility through subtyping.
Subtyping Rules

\[ C^q_i ::= \left[ \right] \mid p?\{m_i.C^q_i\}_{i \in I} \quad (p \neq q) \]

\[ C^q_{IO} ::= \left[ \right] \mid p?\{m_i.C^q_{IO}\}_{i \in I} \mid r!\{m_i.C^q_{IO}\}_{i \in I} \quad (r \neq q) \]

\[ C^q_o ::= \left[ \right] \mid q!\{m_i.C^q_o\}_{i \in I} \quad (p \neq q) \]

\[ C^q_{OI} ::= \left[ \right] \mid p!\{m_i.C^q_{OI}\}_{i \in I} \quad (p \neq q) \mid r?\{m_i.C^q_{OI}\}_{i \in I} \]

\[ \forall (i, k), T_i \leq q?m_k.C^q_i[T'_i] \quad q \neq p \]

\[ p?\{m_i.T_i\}_{i \in I} \leq q?\{m_k.C^q_i[p?\{T'_i\}_{i \in I}]\}_{k \in K} \quad \text{(II)} \]

\[ \forall (i, k), T_i \leq q!m_k.C^p_o[T'_i] \quad q \neq p \]

\[ p!\{m_i.T_i\}_{i \in I} \leq q!\{m_k.C^q_o[p!\{T'_i\}_{i \in I}]\}_{k \in K} \quad \text{(OO)} \]

\[ \forall (i, k), T_i \leq q!m_k.C^p_{IO}[T'_i] \quad q \neq p \]

\[ p!\{m_i.T_i\}_{i \in I} \leq q!\{m_k.C^q_{IO}[p!\{T'_i\}_{i \in I}]\}_{k \in K} \quad \text{(IO)} \]

\[ \forall (i, k), T_i \leq q!m_k.C^p_{OI}[T'_i] \quad q \neq p \]

\[ p?\{m_i.T_i\}_{i \in I} \leq q!\{m_k.C^q_{OI}[p?\{T'_i\}_{i \in I}]\}_{k \in K} \quad \text{(OI)} \]

- Formal definition of flexibility through subtyping.
- An input action bypassing an output action can create deadlocks.
  - binary interaction: !price.?OK \leq_{oi} ?OK.!price
Expressiveness of Flexibility

Results

- **Safe** flexible semantics (queue policy + subtyping policy) are given below.
- The **expressive power** of flexible session is **strictly greater** than the one of standard session.
  - **Intuition**: local type $\mu t. q! \{ m_1.r!m.t \begin{cases} m_2.end \\ \mbox{yields the shuffling of } (q!m_1)^n \mbox{ and } (r!m)^n \mbox{ for all } n.} \end{cases}$

<table>
<thead>
<tr>
<th></th>
<th>(0, 0)</th>
<th>(0, 1)</th>
<th>(0, D)</th>
<th>(1, 0)</th>
<th>(1, 1)</th>
<th>(1, D)</th>
<th>(D, 0)</th>
<th>(D, 1)</th>
<th>(D, D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$\Pi$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$\Omega\Omega$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$\Pi\Omega$</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$\Omega\Pi$</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>$\Pi, \Omega\Pi$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Some session language in literature uses **parallel composition**.

Parallel composition makes explicit **unordered set** of actions:

\[ q!m_1.r!m_2 \text{ compared to } q!m_1 || r!m_2 \]

- introduces flexibility at **type level**.

**Result**

Parallel sessions have a **strictly greater** expressive power than flexible sessions.

- **Intuition**: Parallel composition can be used to **simulate** subtyping rules.
Expressiveness of Interruptible Sessions

- **Interruptions**: describe interactions involving exceptional behaviours [D. Honda Hu Neykova Y. 2015].
- Adds scope constructions:  ${G}^c(\langle l \text{ by } r \rangle; G')$
- Notification of interruption (broadcast) is handled via messages.
  - interactions from an interrupted scope proceed until notification is received.
-  ${[r \rightarrow p : m. (\mu t.p \rightarrow q : m_1.q \rightarrow p : m_2.t)]}^c(\langle i \text{ by } r \rangle; q \rightarrow r : a. \text{end}}$
  - loop of messages between p and q,
  - scope c can be interrupted anytime by r.
  - after being notified of the interruption, q continues by sending a message to r.
- Can interruptions be encoded using standard sessions constructs?

**Result**

**Interruptible sessions** have different expressive power compared to parallel and flexible sessions.

- **Intuition**: nested scopes with recursion yield $q!^n.q?^k$ with $k \leq n$
Conclusion

- **Trace-based (denotational) models of session types to compare expressiveness of sessions.**
- **Safety results for different asychrony policies.**
- **No encoding from interruptible to ”standard sessions”.**
- **Comparison of expressive power:**

```
Parallel ← Interruptible + Parallel

Flexible

Interruptible

Standard
```