On the Expressiveness of Asynchronous Multiparty Session Types

Romain Demangeon - Nobuko Yoshida

UPMC (Paris) - Imperial College (London)

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

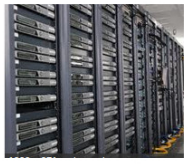
Motivation

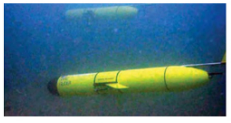
Background

- 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 ?

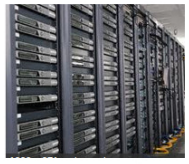
Non-centralised Protocols

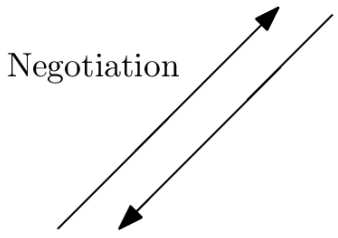


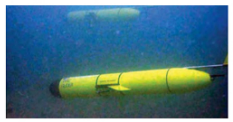


- Three independent applications (client, agent, instrument):
  - written in different languages,
  - with local compilers and libraries,
  - message-passing communication.
- No global control.
- Goal: enforcing interaction success.
  - Message layer soundness.
  - Method: session types.

Non-centralised Protocols

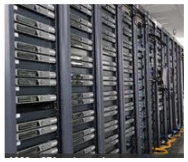


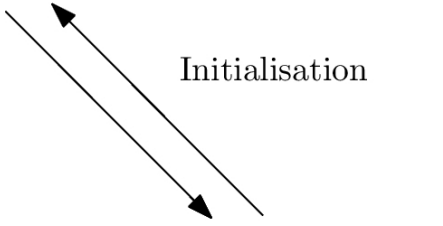


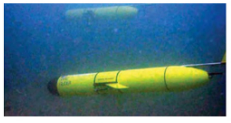


- Three independent applications (client, agent, instrument):
  - written in different languages,
  - with local compilers and libraries,
  - message-passing communication.
- No global control.
- Goal: enforcing interaction success.
  - Message layer soundness.
  - Method: session types.

Non-centralised Protocols

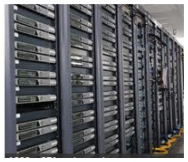


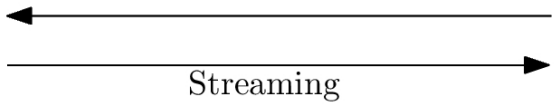


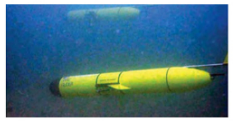


- Three independent applications (client, agent, instrument):
  - written in different languages,
  - with local compilers and libraries,
  - message-passing communication.
- No global control.
- Goal: enforcing interaction success.
  - Message layer soundness.
  - Method: session types.

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: $a : \#^{i}((\mathbf{Nat}, \#^{o}(\mathbf{Bool})))$.
  - session types: $s : ?(\mathbf{Nat}); !(\mathbf{Bool})$.

Binary Sessions - Example

- Global type / session:

$$G = \mathtt{p} \to \mathtt{q} : \mathtt{price}.\mathtt{q} \to \mathtt{p} \begin{cases} \mathtt{KO}.\mathtt{end} \\ \mathtt{OK}.\mathtt{p} \to \mathtt{q} : \mathtt{order}.\mathtt{end} \end{cases}$$

- Local types / end points:

$$T_{\mathtt{p}} : \ !\mathtt{price}.? \begin{cases} \mathtt{KO}.\mathtt{end} \\ \mathtt{OK}.!\mathtt{order}.\mathtt{end} \end{cases}$$

$$T_{\mathtt{q}} : \ ?\mathtt{price}.! \begin{cases} \mathtt{KO}.\mathtt{end} \\ \mathtt{OK}.?\mathtt{order}.\mathtt{end} \end{cases}$$

- Candidate processes (π):
  - $s_{\mathtt{price}}(x).(\overline{s}_{\mathtt{OK}}.s_{\mathtt{order}}(o) + \overline{s}_{\mathtt{KO}})$:

Binary Sessions - Example

- Global type / session:

$$G = \mathtt{p} \to \mathtt{q} : \mathtt{price}.\mathtt{q} \to \mathtt{p} \begin{cases} \mathtt{KO}.\mathtt{end} \\ \mathtt{OK}.\mathtt{p} \to \mathtt{q} : \mathtt{order}.\mathtt{end} \end{cases}$$

- Local types / end points:

$$T_{\mathtt{p}} : \ !\mathtt{price}.? \begin{cases} \mathtt{KO}.\mathtt{end} \\ \mathtt{OK}.!\mathtt{order}.\mathtt{end} \end{cases}$$

$$T_{\mathtt{q}} : \ ?\mathtt{price}.! \begin{cases} \mathtt{KO}.\mathtt{end} \\ \mathtt{OK}.?\mathtt{order}.\mathtt{end} \end{cases}$$

- Candidate processes (π):
  - $s_{\mathtt{price}}(x).(\overline{s}_{\mathtt{OK}}.s_{\mathtt{order}}(o) + \overline{s}_{\mathtt{KO}})$: good $\mathtt{q}$.
  - $s_{\mathtt{price}}(x).\overline{s}_{\mathtt{KO}}$:

Binary Sessions - Example

- Global type / session:

$$G = \mathtt{p} \to \mathtt{q} : \mathtt{price}.\mathtt{q} \to \mathtt{p} \begin{cases} \mathtt{KO}.\mathtt{end} \\ \mathtt{OK}.\mathtt{p} \to \mathtt{q} : \mathtt{order}.\mathtt{end} \end{cases}$$

- Local types / end points:

$$T_{\mathtt{p}} : \ !\mathtt{price}.? \begin{cases} \mathtt{KO}.\mathtt{end} \\ \mathtt{OK}.!\mathtt{order}.\mathtt{end} \end{cases}$$

$$T_{\mathtt{q}} : \ ?\mathtt{price}.! \begin{cases} \mathtt{KO}.\mathtt{end} \\ \mathtt{OK}.?\mathtt{order}.\mathtt{end} \end{cases}$$

- Candidate processes (π):
  - $s_{\mathtt{price}}(x).(\overline{s}_{\mathtt{OK}}.s_{\mathtt{order}}(o) + \overline{s}_{\mathtt{KO}})$: good $\mathtt{q}$.
  - $s_{\mathtt{price}}(x).\overline{s}_{\mathtt{KO}}$: good $\mathtt{q}$.
  - $\overline{s}_{\mathtt{price}}\langle 100\ \mathtt{Fr}\rangle.s_{\mathtt{KO}}$:

Binary Sessions - Example

- Global type / session:

$$G = \mathtt{p} \to \mathtt{q} : \mathtt{price}.\mathtt{q} \to \mathtt{p} \begin{cases} \mathtt{KO}.\mathtt{end} \\ \mathtt{OK}.\mathtt{p} \to \mathtt{q} : \mathtt{order}.\mathtt{end} \end{cases}$$

- Local types / end points:

$$T_{\mathtt{p}} : \ !\mathtt{price}.? \begin{cases} \mathtt{KO}.\mathtt{end} \\ \mathtt{OK}.!\mathtt{order}.\mathtt{end} \end{cases}$$

$$T_{\mathtt{q}} : \ ?\mathtt{price}.! \begin{cases} \mathtt{KO}.\mathtt{end} \\ \mathtt{OK}.?\mathtt{order}.\mathtt{end} \end{cases}$$

- Candidate processes (π):
  - $s_{\mathtt{price}}(x).(\overline{s}_{\mathtt{OK}}.s_{\mathtt{order}}(o) + \overline{s}_{\mathtt{KO}})$: good $\mathtt{q}$.
  - $s_{\mathtt{price}}(x).\overline{s}_{\mathtt{KO}}$: good $\mathtt{q}$.
  - $\overline{s}_{\mathtt{price}}\langle 100\ \mathtt{Fr}\rangle.s_{\mathtt{KO}}$: bad $\mathtt{p}$.

Multiparty Session Types

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

Example

- $G = \mathtt{r} \to \mathtt{q} : m, \mathtt{q} \to \mathtt{p} : m_1, \mathtt{r} \to \mathtt{p} : m_2.\mathtt{end}$
- $T_{\mathtt{p}} : \mathtt{q}?m_1.\mathtt{r}?m_2.\mathtt{end}$
- $T_{\mathtt{q}} : \mathtt{r}?m.\mathtt{p}!m_1.\mathtt{end}$
- $T_{\mathtt{r}} : \mathtt{q}!m.\mathtt{p}!m_2.\mathtt{end}$

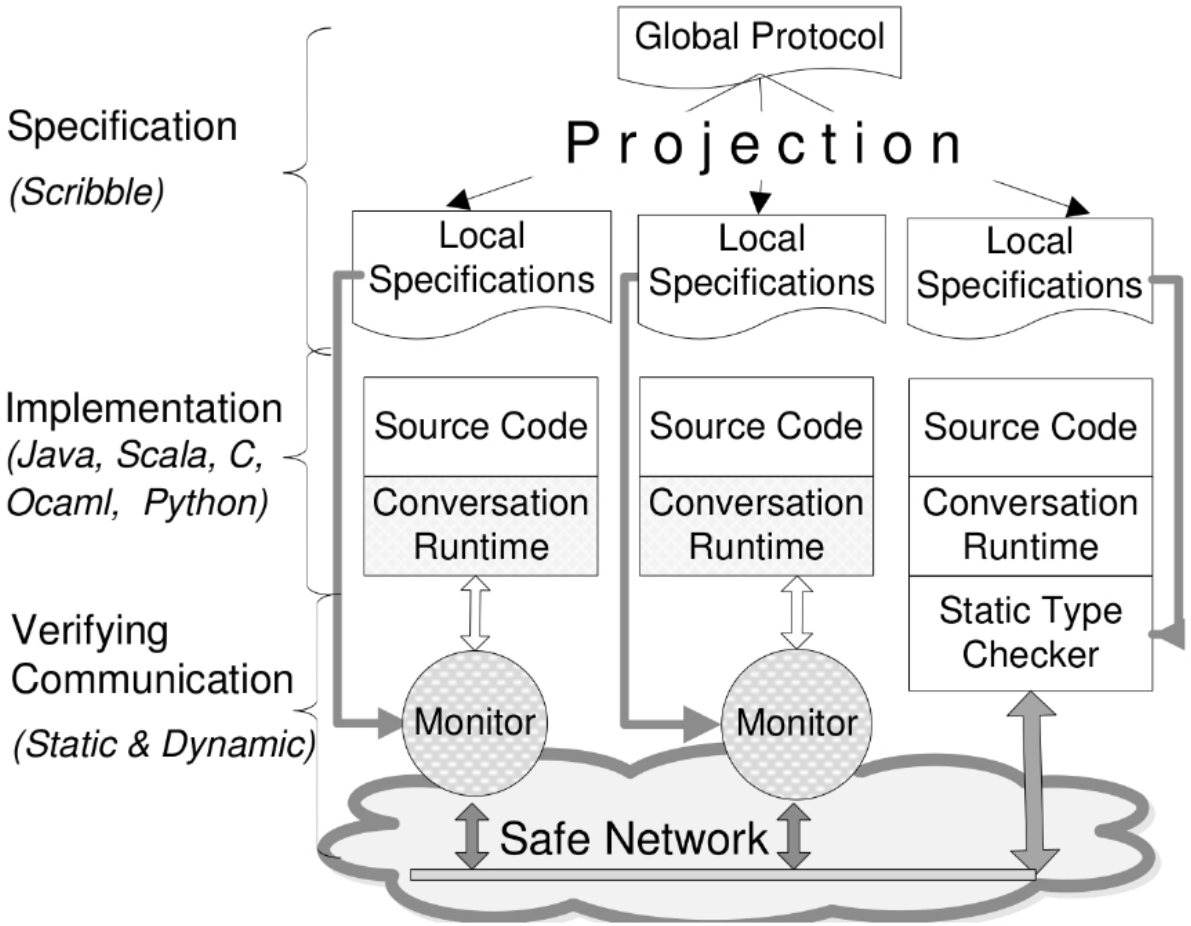
- Semantics:
  - Let A, B be applications s.t. $\vdash A : T_{\mathtt{r}}$ and $\vdash B : T_{\mathtt{q}}$
  - A can send message m and B can receive it (giving A', B').
  - $\vdash A' : \mathtt{p}!m_2.\mathtt{end}$ and $\vdash B' : \mathtt{p}!m_1.\mathtt{end}$
  - At type level, reduction semantics:
    $\mathtt{q}!m.\mathtt{p}!m_2.\mathtt{end} \mid \mathtt{r}?m.\mathtt{p}!m_1.\mathtt{end} \to \mathtt{p}!m_2.\mathtt{end} \mid \mathtt{p}!m_1.\mathtt{end}$

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
    $\Rightarrow$ 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({
   url: "http://www.p.com?price",
   dataType: "jsonp",
   jsonpCallback: "callback",
   success: "store_price"})

$.ajax({
   url: "http://www.q.net?price",
   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({
   url: "http://www.p.com?price",
   dataType: "jsonp",
   jsonpCallback: "callback",
   success: "store_price"})

$.ajax({
   url: "http://www.q.net?price",
   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({
   url: "http://www.p.com?price",
   dataType: "jsonp",
   jsonpCallback: "callback",
   success: "store_price"})

$.ajax({
   url: "http://www.q.net?price",
   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

- Models for real-life networks. [Bocchi Chen D. Honda Y. 13]
- 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

- $G = \mathtt{r} \to \mathtt{q} : m, \mathtt{q} \to \mathtt{p} : m_1, \mathtt{r} \to \mathtt{p} : m_2.\mathtt{end}$
- with asynchronous semantics m_2 can arrive before m_1.

AMST Basic Syntax

$$
\begin{array}{rcl}
G & ::= & \mathtt{end} \mid \mu \mathtt{t}.G \mid \mathtt{t} \mid \mathtt{r}_1 \to \mathtt{r}_2\{m_i.G_i\}_{i\in I} \\
T & ::= & \mathtt{end} \mid \mu \mathtt{t}.T \mid \mathtt{t} \mid \mathtt{p}?\{m_i.T_i\}_{i\in I} \mid \mathtt{p}!\{m_i.T_i\}_{i\in I}
\end{array}
$$

- 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.

Type Expressiveness

We compare the expressiveness of several type system:

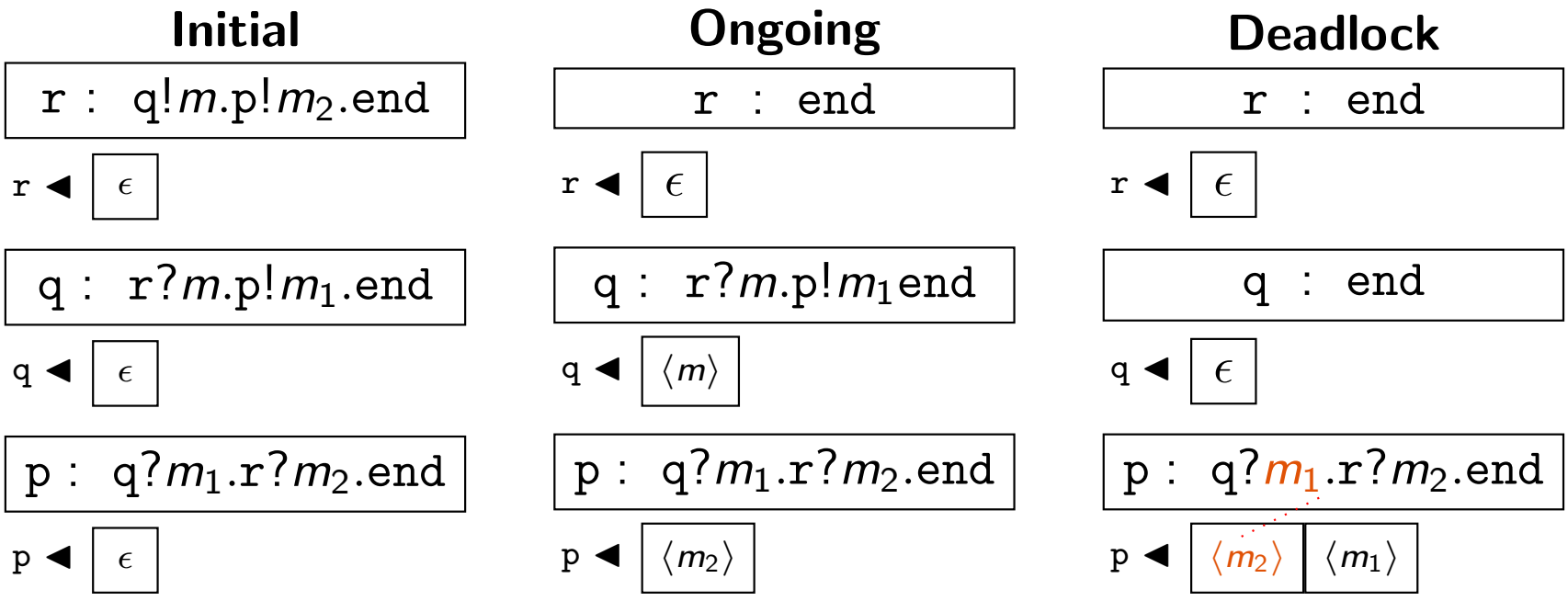
- Models with input and/or output queues,
- Sequence subtyping (switching interaction order at type level),
- Parallel composition.
- Interruptible sessions (encoding ?).

Queue models

Different models used in literature:

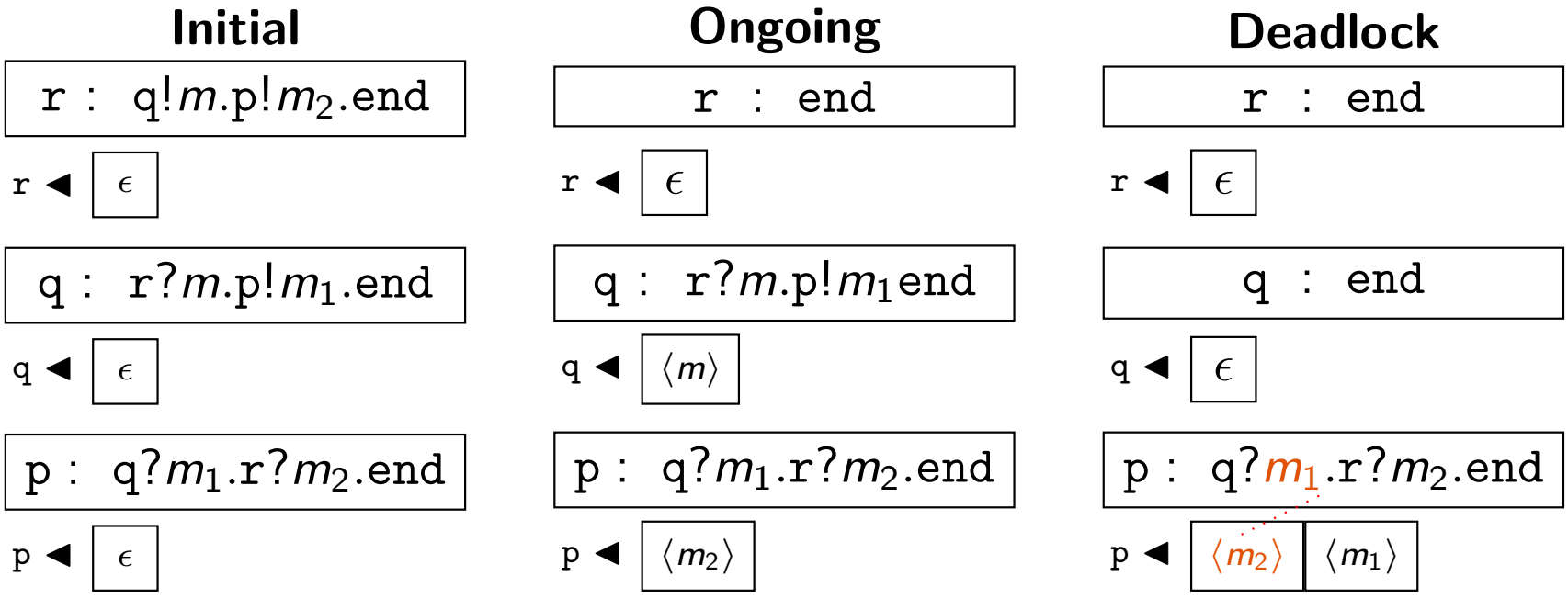
- input queues storing arriving messages:
  - none: participants consume messages from the network,
  - one: each participant has one inbox for all incoming messages,
  - several: each participant has one inbox for each other participant,
- same choices for output queue design.
- yields 9 different queue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

Example: One input queue



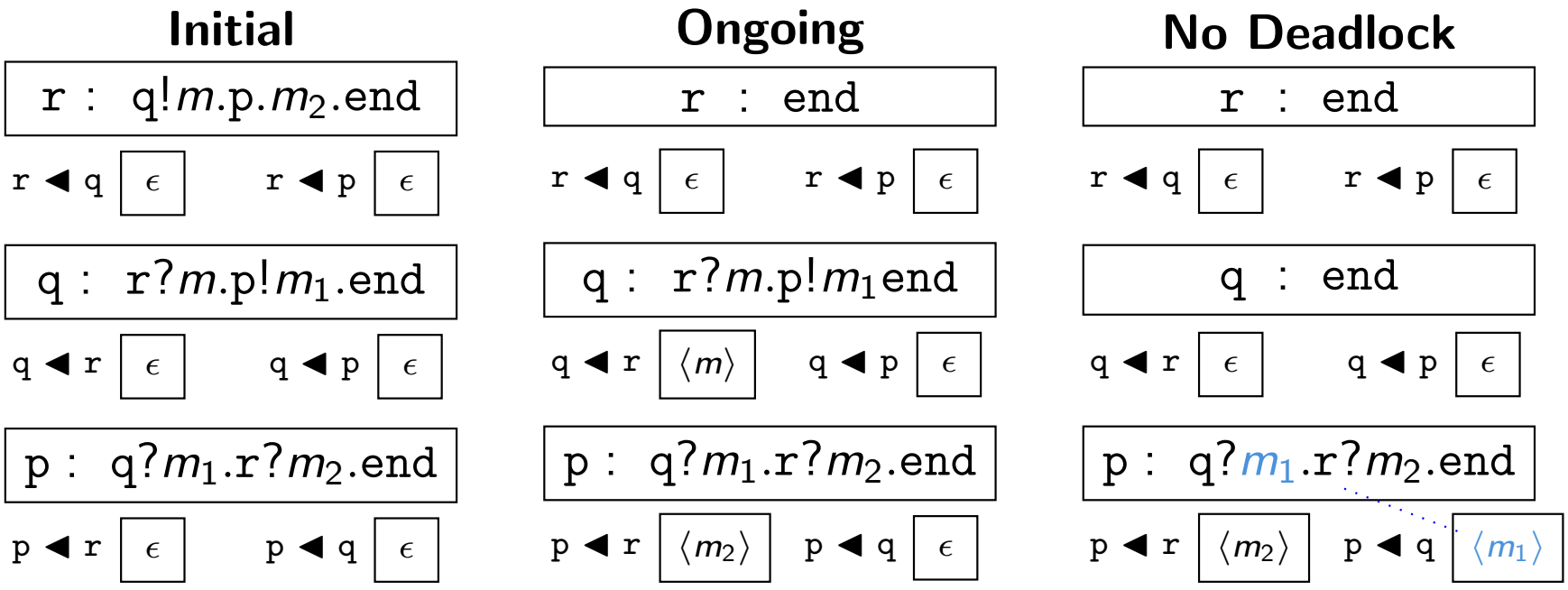
- Single input queue: one inbox per participant.
- Asynchrony let m_2 arrive before m_1.
  - p expects to receive m_1 first.
- System is deadlocked.

Example: One input queue



- 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 $\rightarrow$ wrong semantics.

Example: Multiple input queues



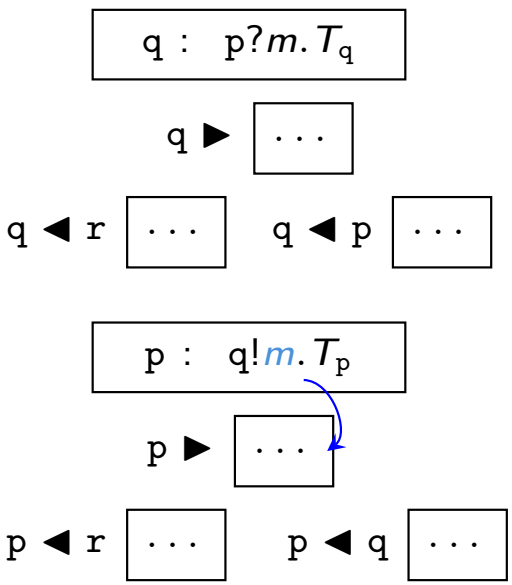
- Multiple input queues: one inbox per pair of participants.
- Asynchrony let m_2 arrive before m_1.
  - p expects to receive m_1 first.
- System can progress because m_1 is available in one input queue.
- Multiple input queues semantics is safe.

Configuration Semantics

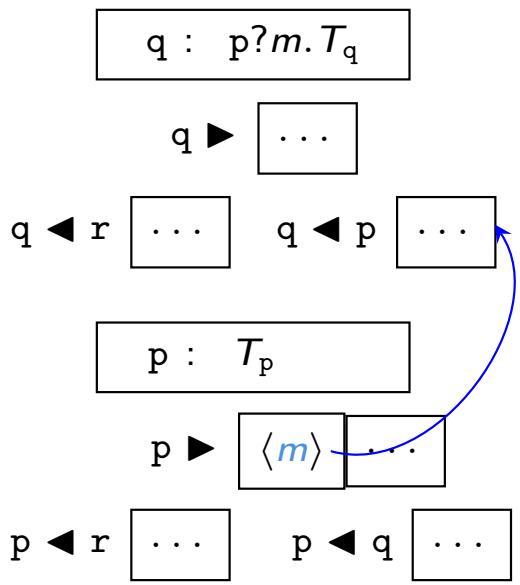
$$\Delta ::= \emptyset \mid \mathtt{p} : T, \Delta \mid Q, \Delta \qquad h ::= \epsilon \mid \langle \mathtt{p}, \mathtt{q}, m \rangle . h$$
$$Q ::= (\mathtt{p} \blacktriangleleft \mathtt{q} : h) \mid (\mathtt{p} \blacktriangleright \mathtt{q} : h) \mid (\mathtt{p} \blacktriangleleft : h) \mid (\mathtt{p} \blacktriangleright : h)$$

- Configuration: composition of participants (local types) and queues,
- Input $\blacktriangleleft$ and output $\blacktriangleright$ queues.
- Single $\mathtt{p} \blacktriangleleft$ and multiple $\mathtt{p} \blacktriangleleft \mathtt{q}$ queues.
- Global type $G \to$ initial configuration (projection and empty queues).
- Rules enforces message transfer.

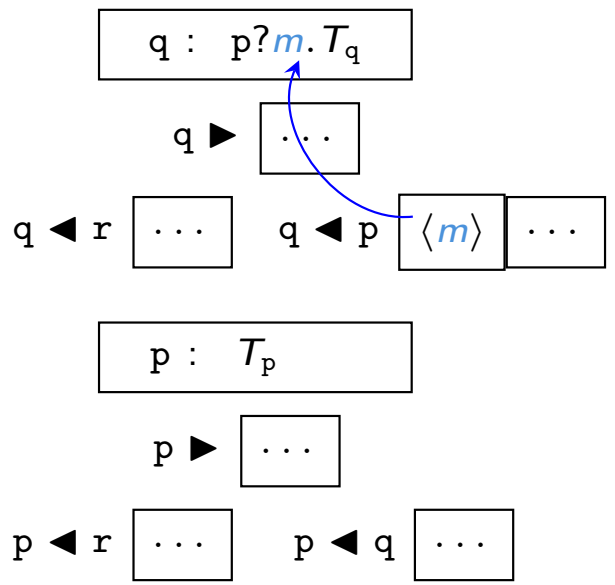
Rule (OutOut)



Rule (Transit)



Rule (InIn)



Configuration Semantics Rules

$$
\begin{array}{rrcll}
\text{(Com)} & \mathtt{p} : \mathtt{q}!\{m_i.T_i\}_{i\in I}, \mathtt{q} : \mathtt{p}?\{m_i.T_i\}_{i\in I} & \xrightarrow{\mathtt{pq}:m_j} & \mathtt{p} : T_j, \mathtt{q} : T_j & j \in I \\
\text{(InIn)} & \mathtt{q} : \mathtt{p}?\{m_i.T_i\}_{i\in I}, (\mathtt{q} \triangleleft \mathtt{p} : \langle \mathtt{p}, \mathtt{q}, m_j\rangle.h) & \xrightarrow{\mathtt{p}?\mathtt{q}:m_j} & \mathtt{p} : T_j, (\mathtt{q} \triangleleft \mathtt{p} : h) & j \in I \\
\text{(OutIn)} & \mathtt{p} : \mathtt{q}!\{m_j.T_i\}_{i\in I}, (\mathtt{q} \triangleleft \mathtt{p} : h) & \xrightarrow{\mathtt{p}!\mathtt{q}:m_j} & \mathtt{p} : T_j, (\mathtt{q} \triangleleft \mathtt{p} : h.\langle \mathtt{p}, \mathtt{q}, m_j\rangle) & j \in I \\
\text{(InOut)} & \mathtt{q} : \mathtt{p}?\{m_i.T_i\}_{i\in I}, (\mathtt{p} \triangleright \mathtt{q} : h.\langle \mathtt{p}, \mathtt{q}, m_j\rangle) & \xrightarrow{\mathtt{p}?\mathtt{q}:m_j} & \mathtt{p} : T_j, (\mathtt{p} \triangleright \mathtt{q} : h) & j \in I \\
\text{(OutOut)} & \mathtt{p} : \mathtt{q}!\{m_i.T_i\}_{i\in I}, (\mathtt{p} \triangleright \mathtt{q} : h) & \xrightarrow{\mathtt{p}!\mathtt{q}:m_j} & \mathtt{p} : T_j, (\mathtt{p} \triangleright \mathtt{q} : \langle \mathtt{p}, \mathtt{q}, m_j\rangle.h) & j \in I \\
\text{(Transit)} & (\mathtt{p} \triangleright \mathtt{q} : h.\langle \mathtt{p}, \mathtt{q}, m\rangle), (\mathtt{q} \triangleleft \mathtt{p} : h) & \xrightarrow{\tau} & (\mathtt{p} \triangleright \mathtt{q} : h), (\mathtt{q} \triangleleft \mathtt{p} : \langle \mathtt{p}, \mathtt{q}, m\rangle.h) & \\
\text{(Par)} & \Delta_1 \xrightarrow{\ell} \Delta_1' & \Longrightarrow & \Delta_1, \Delta_2 \xrightarrow{\ell} \Delta_1', \Delta_2 &
\end{array}
$$

	$(0,0)$	$(0,1)$	$(0,M)$	$(1,0)$	$(1,1)$	$(1,M)$	$(M,0)$	$(M,1)$	(M,M)
(Com)	√								
(InIn)				√	√	√	√	√	√
(OutIn)				√			√		
(InOut)		√	√						
(OutOut)		√	√		√	√		√	√
(Transit)					√	√		√	√

- Queue policy guides the semantics rules.
- $(\mathtt{p} \triangleleft \mathtt{q} : h)$ stands for either $(\mathtt{p} \blacktriangleleft \mathtt{q} : h)$ or $(\mathtt{p} \blacktriangleleft : h)$.

UPMC SORBONNE UNIVERSITÉS

Trace Denotations

- Configuration traces as a measure for expressiveness.
- A trace σ 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 σ is terminated w.r.t. a type G if the initial configuration of G cannot progress after σ.
  - captures deadlocks.
  - depends on the queue policy.
- A trace σ is completed w.r.t. a type G if the initial configuration of G reaches happy termination after σ.
  - after σ participants reaches `end` and queues are empty.

Example

- Global type:
  $\mathtt{r} \to \mathtt{q} : m, \mathtt{q} \to \mathtt{p} : m_1, \mathtt{r} \to \mathtt{p} : m_2.\mathtt{end}$
- Initial configuration for $(0, 0)$:
  $\mathtt{r} : \mathtt{q}!m.\mathtt{p}!m_2,\ \mathtt{q} : \mathtt{r}?m.\mathtt{p}!m_1,\ \mathtt{p} : \mathtt{q}?m_1.\mathtt{r}?m_2$,
- Initial configuration for $(M, 1)$:
  $\mathtt{r} : \mathtt{q}!m.\mathtt{p}!m_2,\ \mathtt{q} : \mathtt{r}?m.\mathtt{p}!m_1,\ \mathtt{p} : \mathtt{q}?m_1.\mathtt{r}?m_2,$
  $(\mathtt{p} \blacktriangleright : \epsilon), (\mathtt{q} \blacktriangleright : \epsilon), (\mathtt{r} \blacktriangleright : \epsilon), (\mathtt{p} \blacktriangleleft \mathtt{q} : \epsilon),\ (\mathtt{p} \blacktriangleleft \mathtt{r} : \epsilon),$
  $(\mathtt{q} \blacktriangleleft \mathtt{p} : \epsilon),\ (\mathtt{q} \blacktriangleleft \mathtt{r} : \epsilon),\ (\mathtt{r} \blacktriangleleft \mathtt{p} : \epsilon),\ (\mathtt{r} \blacktriangleleft \mathtt{q} : \epsilon).$
- Trace $\sigma_e : \begin{cases} \mathtt{r} & \mapsto & \mathtt{q}!m.\mathtt{p}!m_2 \\ \mathtt{q} & \mapsto & \mathtt{r}?m.\mathtt{p}!m_1 \\ \mathtt{p} & \mapsto & \mathtt{q}?m_1.\mathtt{r}!m_2 \end{cases}$
  is completed for both semantics.
- Trace $\sigma_t : \begin{cases} \mathtt{p} & \mapsto & \mathtt{q}!m \\ \mathtt{q} & \mapsto & \epsilon \\ \mathtt{r} & \mapsto & \epsilon \end{cases}$
  is a valid (uncompleted) trace for $(M, 1)$ and not for $(0, 0)$.
- Trace $\sigma_d : \begin{cases} \mathtt{r} & \mapsto & \mathtt{q}!m.\mathtt{p}!m_2 \\ \mathtt{q} & \mapsto & \mathtt{r}?m.\mathtt{p}!m_1 \\ \mathtt{p} & \mapsto & \epsilon \end{cases}$
  is terminated for $(1, 0)$ but not for $(0, M)$.

Expressiveness Results

- $\mathbf{D}(G, \phi)$, the denotation of G under semantics (queue policy) ϕ is the set of all terminated traces from G according to ϕ.
- 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 ($\emptyset, II, OO, IO, OI, IO/OI$)

Example (II)-flexibility

- $\mathtt{p}?m_1.\mathtt{q}?m_2.\mathtt{p}!m_3.\mathtt{end}$ switches to $\mathtt{q}?m_2.\mathtt{p}?m_1.\mathtt{p}!m_3.\mathtt{end}$.

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

- $\mathtt{p}?m_1.\mathtt{q}?m_2.\mathtt{p}!m_3.\mathtt{end}$ switches to $\mathtt{q}?m_2.\mathtt{p}?m_1.\mathtt{p}!m_3.\mathtt{end}$.
- $\mathtt{p}?\left\{\begin{array}{l} m_{11}.\mathtt{q}?m_2.\mathtt{p}!m_3.\mathtt{end} \\ m_{12}.\mathtt{q}?m_2.\mathtt{p}!m_4.\mathtt{end} \end{array}\right.$ switches to $\mathtt{q}?m_2.\mathtt{p}?\left\{\begin{array}{l} m_{11}.\mathtt{p}!m_3.\mathtt{end} \\ m_{12}.\mathtt{p}!m_4.\mathtt{end} \end{array}\right.$

Subtyping Rules

$$\mathbb{C}^{\mathtt{q}}_{\mathsf{I}} ::= [\,] \mid \mathtt{p}?\{m_i.\mathbb{C}^{\mathtt{q}}_{\mathsf{I}}\}_{i\in I}\ (\mathtt{p}\neq\mathtt{q})$$

$$\mathbb{C}^{\mathtt{q}}_{\mathsf{IO}} ::= [\,] \mid \mathtt{p}?\{m_i.\mathbb{C}^{\mathtt{q}}_{\mathsf{IO}}\}_{i\in I} \mid \mathtt{r}!\{m_i.\mathbb{C}^{\mathtt{q}}_{\mathsf{IO}}\}_{i\in I}\ (\mathtt{r}\neq\mathtt{q})$$

$$\mathbb{C}^{\mathtt{q}}_{\mathsf{O}} ::= [\,] \mid \mathtt{q}!\{m_i.\mathbb{C}^{\mathtt{q}}_{\mathsf{O}}\}_{i\in I}\ (\mathtt{p}\neq\mathtt{q})$$

$$\mathbb{C}^{\mathtt{q}}_{\mathsf{OI}} ::= [\,] \mid \mathtt{p}!\{m_i.\mathbb{C}^{\mathtt{q}}_{\mathsf{OI}}\}_{i\in I}\ (\mathtt{p}\neq\mathtt{q}) \mid \mathtt{r}?\{m_i.\mathbb{C}^{\mathtt{q}}_{\mathsf{OI}}\}_{i\in I}$$

$$(\mathsf{II})\ \frac{\forall(i,k),\ T_i \leq \mathtt{q}?m_k.\mathbb{C}^{\mathtt{p}}_{\mathsf{I}}[T'_i] \quad \mathtt{q}\neq\mathtt{p}}{\mathtt{p}?\{m_i.T_i\}_{i\in I} \leq \mathtt{q}?\{m_k.\mathbb{C}^{\mathtt{q}}_{\mathsf{I}}[\mathtt{p}?\{T'_i\}_{i\in I}]\}_{k\in K}}$$

$$(\mathsf{OO})\ \frac{\forall(i,k),\ T_i \leq \mathtt{q}!m_k.\mathbb{C}^{\mathtt{p}}_{\mathsf{O}}[T'_i] \quad \mathtt{q}\neq\mathtt{p}}{\mathtt{p}!\{m_i.T_i\}_{i\in I} \leq \mathtt{q}!\{m_k.\mathbb{C}^{\mathtt{q}}_{\mathsf{O}}[\mathtt{p}!\{T'_i\}_{i\in I}]\}_{k\in K}}$$

$$(\mathsf{IO})\ \frac{\forall(i,k),\ T_i \leq \mathtt{q}!m_k.\mathbb{C}^{\mathtt{p}}_{\mathsf{IO}}[T'_i] \quad \mathtt{q}\neq\mathtt{p}}{\mathtt{p}!\{m_i.T_i\}_{i\in I} \leq \mathtt{q}?\{m_k.\mathbb{C}^{\mathtt{q}}_{\mathsf{IO}}[\mathtt{p}!\{T'_i\}_{i\in I}]\}_{k\in K}}$$

$$(\mathsf{OI})\ \frac{\forall(i,k),\ T_i \leq \mathtt{q}!m_k.\mathbb{C}^{\mathtt{p}}_{\mathsf{OI}}[T'_i] \quad \mathtt{q}\neq\mathtt{p}}{\mathtt{p}?\{m_i.T_i\}_{i\in I} \leq \mathtt{q}!\{m_k.\mathbb{C}^{\mathtt{q}}_{\mathsf{IO}}[\mathtt{p}?\{T'_i\}_{i\in I}]\}_{k\in K}}$$

- Formal definition of flexibility through subtyping.

Subtyping Rules

$$\mathbb{C}^{\mathtt{q}}_{\mathsf{I}} ::= [\,] \mid \mathtt{p}?\{m_i.\mathbb{C}^{\mathtt{q}}_{\mathsf{I}}\}_{i\in I}\ (\mathtt{p}\neq\mathtt{q})$$

$$\mathbb{C}^{\mathtt{q}}_{\mathsf{IO}} ::= [\,] \mid \mathtt{p}?\{m_i.\mathbb{C}^{\mathtt{q}}_{\mathsf{IO}}\}_{i\in I} \mid \mathtt{r}!\{m_i.\mathbb{C}^{\mathtt{q}}_{\mathsf{IO}}\}_{i\in I}\ (\mathtt{r}\neq\mathtt{q})$$

$$\mathbb{C}^{\mathtt{q}}_{\mathsf{O}} ::= [\,] \mid \mathtt{q}!\{m_i.\mathbb{C}^{\mathtt{q}}_{\mathsf{O}}\}_{i\in I}\ (\mathtt{p}\neq\mathtt{q})$$

$$\mathbb{C}^{\mathtt{q}}_{\mathsf{OI}} ::= [\,] \mid \mathtt{p}!\{m_i.\mathbb{C}^{\mathtt{q}}_{\mathsf{OI}}\}_{i\in I}\ (\mathtt{p}\neq\mathtt{q}) \mid \mathtt{r}?\{m_i.\mathbb{C}^{\mathtt{q}}_{\mathsf{OI}}\}_{i\in I}$$

$$(\mathsf{II})\ \frac{\forall(i,k),\ T_i \leq \mathtt{q}?m_k.\mathbb{C}^{\mathtt{p}}_{\mathsf{I}}[T'_i] \quad \mathtt{q}\neq\mathtt{p}}{\mathtt{p}?\{m_i.T_i\}_{i\in I} \leq \mathtt{q}?\{m_k.\mathbb{C}^{\mathtt{q}}_{\mathsf{I}}[\mathtt{p}?\{T'_i\}_{i\in I}]\}_{k\in K}}$$

$$(\mathsf{OO})\ \frac{\forall(i,k),\ T_i \leq \mathtt{q}!m_k.\mathbb{C}^{\mathtt{p}}_{\mathsf{O}}[T'_i] \quad \mathtt{q}\neq\mathtt{p}}{\mathtt{p}!\{m_i.T_i\}_{i\in I} \leq \mathtt{q}!\{m_k.\mathbb{C}^{\mathtt{q}}_{\mathsf{O}}[\mathtt{p}!\{T'_i\}_{i\in I}]\}_{k\in K}}$$

$$(\mathsf{IO})\ \frac{\forall(i,k),\ T_i \leq \mathtt{q}!m_k.\mathbb{C}^{\mathtt{p}}_{\mathsf{IO}}[T'_i] \quad \mathtt{q}\neq\mathtt{p}}{\mathtt{p}!\{m_i.T_i\}_{i\in I} \leq \mathtt{q}?\{m_k.\mathbb{C}^{\mathtt{q}}_{\mathsf{IO}}[\mathtt{p}!\{T'_i\}_{i\in I}]\}_{k\in K}}$$

$$(\mathsf{OI})\ \frac{\forall(i,k),\ T_i \leq \mathtt{q}!m_k.\mathbb{C}^{\mathtt{p}}_{\mathsf{OI}}[T'_i] \quad \mathtt{q}\neq\mathtt{p}}{\mathtt{p}?\{m_i.T_i\}_{i\in I} \leq \mathtt{q}!\{m_k.\mathbb{C}^{\mathtt{q}}_{\mathsf{IO}}[\mathtt{p}?\{T'_i\}_{i\in I}]\}_{k\in K}}$$

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

UPMC SORBONNE UNIVERSITÉS

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 $\mu \mathtt{t}.\mathtt{q}! \begin{cases} m_1.\mathtt{r}!m.\mathtt{t} \\ m_2.\mathtt{end} \end{cases}$ yields the shuffling of $(\mathtt{q}!m_1)^n$ and $(\mathtt{r}!m)^n$ for all n.

	$(0,0)$	$(0,1)$	$(0,D)$	$(1,0)$	$(1,1)$	$(1,D)$	$(D,0)$	$(D,1)$	(D,D)
$\emptyset$	$\surd$	$\surd$	$\surd$	$\times$	$\times$	$\times$	$\surd$	$\surd$	$\surd$
II	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
OO	$\surd$	$\surd$	$\surd$	$\times$	$\times$	$\times$	$\surd$	$\surd$	$\surd$
IO	$\times$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
OI	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
IO, OI	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$

- Flexibility allows the safe use of single input queues.

Expressiveness of Parallel Composition

- Some session language in literature uses parallel composition.
- Parallel composition makes explicit unordered set of actions: $\mathtt{q!m_1.r!m_2}$ compared to $\mathtt{q!m_1 || r!m_2}$
  - 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: $\{\!|\, G \,|\!\}^{\mathtt{c}}\langle I \text{ by } \mathtt{r}\rangle; G'$
- Notification of interruption (broadcast) is handled via messages.
  - interactions from an interrupted scope proceed until notification is received.
- $\{\!|\mathtt{r} \to \mathtt{p} : m.(\mu \mathtt{t}.\mathtt{p} \to \mathtt{q} : m_1.\mathtt{q} \to \mathtt{p} : m_2.\mathtt{t})|\!\}^{\mathtt{c}}\langle i \text{ by } \mathtt{r}\rangle; \mathtt{q} \to \mathtt{r} : a.\mathtt{end}$
  - loop of messages between $\mathtt{p}$ and $\mathtt{q}$,
  - scope $\mathtt{c}$ can be interupted anytime by $\mathtt{r}$.
  - after being notified of the interruption, $\mathtt{q}$ continues by sending a message to $\mathtt{r}$.
- 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 $\mathtt{q}!^n.\mathtt{q}?^k$ with $k \leq n$

UPMC SORBONNE UNIVERSITÉS

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: