On the Expressiveness of Multiparty Sessions

Romain Demangeon[1] and Nobuko Yoshida[2]

1 **Université Pierre et Marie Curie, Paris 6**
2 **Imperial College London**

Abstract

This paper explores expressiveness of asynchronous multiparty sessions. We model the behaviours of endpoint implementations in several ways: (i) by the existence of different buffers and queues used to store messages exchanged asynchronously, (ii) by the ability for an endpoint to lightly reconfigure his behaviour at runtime (flexibility), (iii) by the presence of explicit parallelism or interruptions (exceptional actions) in endpoint behaviour. For a given protocol we define several denotations, based on traces of events, corresponding to the different implementations and compare them.

Keywords and phrases concurrency, message-passing, session, asynchrony, expressiveness

1 Introduction

Asynchronous Multiparty Sessions In large-scale distributed infrastructures, most interactions are based upon the production of interleaving flows of messages between independent participants. Verification of such distributed protocols is challenging: participants are executing applications written in different languages and the way messages are treated between production and consumption may vary. The presence of intermediate layers where on-transit messages are stored and transferred via, *e.g.* buffers or queues, makes the analyses difficult, even with the guarantee that the order of messages is preserved for each intermediate structure.

The approach of *multiparty session types* [11, 7] (extended from the binary [10]) introduced a flexible formal method for verification of message-passing protocols without central control: the desired interactions at the scale of the network itself are specified into a session (called *global type*). These formal objects describe interactions between all participants through simple syntax including send and receive operations, choice and recursion. Global types are then projected onto several *local types* (one for each participant), which describe the protocol from a local point of view. These local types are used to validate an application through type-checking or monitoring. Theory of session types guarantees that local conformance of all participants induces global conformance of the network to the initial global type. Sessions type theory is well-studied and gave birth to languages such as *Scribble* [22], directly inspired by formal session types, letting developers specify and verify (through automatically generated monitors) distributed protocols and applications, *e.g.* for large cyberinfrastructures [8] and business protocols [15].

Although various extensions of multiparty sessions [11] are studied, several fundamental open problems remain, such as *expressiveness* questions: whether *permutations* of types (used to compensate the order of arrival of messages from different sources) [16, 6, 17] and *interruptible* sessions [8] are more expressive than standard sessions or not. We require a canonical methodology to compare these extensions systematically.

Session type expressiveness This paper explores and compares expressiveness of different semantics for asynchronous multiparty sessions in the literature, based on message traces.

We first study the effect of buffers – order-preserving stores for in-transit messages – on the expressiveness. For instance, adding buffers on the sender side (messages are stored in a queue after being produced and before being transferred to the receiver) is innocuous, whereas receiver-side buffers can produce deadlocks. As an example, consider global type $G = \mathtt{r} \to \mathtt{q} : m, \mathtt{q} \to \mathtt{p} :$

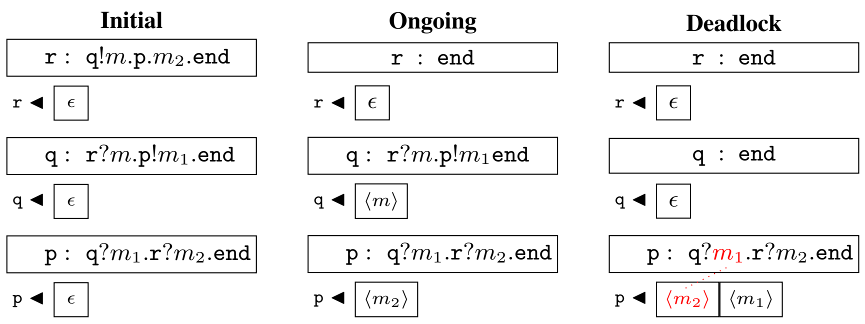

Figure 1 Configurations with single input queues for G

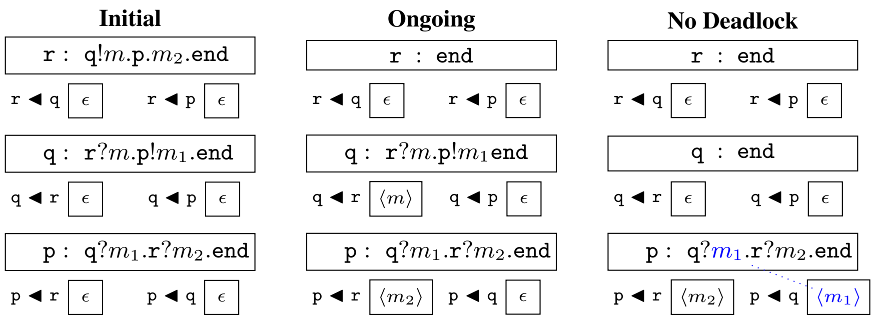

Figure 2 Configurations with multiple input queues for G

$m_1, \mathtt{r} \to \mathtt{p} : m_2.\mathtt{end}$ which consists of a sequence of three messages exchanged between three participants. It is projected to local types $\mathtt{p} : \mathtt{q}?m_1.\mathtt{r}?m_2.\mathtt{end}$, $\mathtt{q} : \mathtt{r}?m.\mathtt{p}!m_1.\mathtt{end}$ and $\mathtt{r} : \mathtt{q}!m_1.\mathtt{p}!m_2.\mathtt{end}$, in which each participant $(\mathtt{p}, \mathtt{q}, \mathtt{r})$ is expected to perform two consecutive actions. $\mathtt{q}!m$ is the output of message m to $\mathtt{q}$ and $\mathtt{r}?m$ is the input of message m from $\mathtt{r}$. $\mathtt{end}$ denotes termination. If each participant uses one buffer on the receiver-side (called *input queue*), message m_2 can be arrived and be enqueued in the structure of $\mathtt{p}$ before m_1, leading to a deadlock (as $\mathtt{p}$ expects to consume m_1 first), as described in Figure 1. There exist several ways to allow usage of buffers on the receiving side without risking deadlocks. First, one can separate the input queue into several input queues (as in [7]), one for each possible sender, a program being allowed to consume messages from any queues. In our example, m_2 (coming from $\mathtt{r}$) and m_1 (coming from $\mathtt{q}$) would be stored in different input queues at $\mathtt{p}$, allowing m_1 not to be blocked by an early arrival of m_2. This situation is described in Figure 2. Alternatively, one can introduce flexibility in the program running at $\mathtt{p}$ to make it able to accept m_2 before m_1. Formally, it boils down to a permutation of $\mathtt{p} : \mathtt{q}?m_1.\mathtt{r}?m_2.\mathtt{end}$ to $\mathtt{p} : \mathtt{r}?m_2.\mathtt{q}?m_1.\mathtt{end}$, adapting it to the order of arrival of m_1 and m_2.

Configurations and traces The common framework we use to describe session networks is *configurations* (drawn from [2]), which are collections of local types – the remaining expected actions for all participants – and queues – the order-preserving structures storing in-transit messages. For instance, the deadlocked situation explained above is described by configuration $\mathtt{p} : \mathtt{q}?m_1.\mathtt{r}?m_2.\mathtt{end}, \mathtt{q} : \mathtt{end}, \mathtt{r} : \mathtt{end}, (\mathtt{p} \blacktriangleleft: \langle \mathtt{q}, \mathtt{p}, m_1\rangle.\langle \mathtt{r}, \mathtt{p}, m_2\rangle)$ where $\mathtt{q}$ and $\mathtt{r}$ are finished, $\mathtt{p}$ expects to first receive m_1 then m_2 and the (only) input queue at $\mathtt{p}$ is ready to deliver m_2 then m_1.

From these configurations, we extract *traces* to compare semantics. They are mappings from participants to sequences of actions, send or receive *events*, ordered *locally*: two events at the same location are ordered, but two events performed by different participants are not. As an example, a trace σ leading to the configuration above from the initial configuration is s.t. $\sigma(\mathtt{p}) = \epsilon$, $\sigma(\mathtt{q}) = \mathtt{r}?m.\mathtt{p}!m_1$, $\sigma(\mathtt{r}) = \mathtt{q}!m.\mathtt{p}!m_2$, describing the fact that m_1 and m_2 have been sent by $\mathtt{r}$ and $\mathtt{q}$, but not yet received by $\mathtt{p}$. The traces contains no information on whether m_1 was sent before of after m_2.

A protocol can then be given a *denotation*, w.r.t. a given semantics, as a set of completed local traces, that is, traces of configurations which cannot progress further. Different semantics yield different denotations for the same type; for instance, the denotation of G under a semantics with simple input queues contains traces stopped at deadlocked configurations (such as σ), whereas the denotation of G under a semantics with multiple input queues (as described above) will only contain completed traces (traces reaching a configurations where all local types are $\mathtt{end}$).

Parallel and interruptible sessions Next we study the impact on expressiveness of two different constructs: the parallel composition, explicitly notifying that two actions can appear in any order and interruptions. Interruptible sessions have been studied in [12, 8] through the use of *scopes* describing sessions in which a participant can, at any time, raise a interruption to stop the current block of interactions. Suppose $\{\!|\mathtt{p} \to \mathtt{q} : m_1.\mathtt{q} \to \mathtt{r} : m_2.\mathtt{end}|\!\}^{\mathtt{c}}\langle i \text{ by } \mathtt{p}\rangle; \mathtt{p} \to \mathtt{r} : m_3$. In $\{\!|G|\!\}$, m_2 is supposed to be sent by $\mathtt{q}$ after receiving m_1. Participant $\mathtt{p}$ can interrupt the session at any time, as specified in $\langle i \text{ by } \mathtt{p}\rangle$, for instance after sending m_1, by broadcasting the message i. If i reaches $\mathtt{q}$ after m_1 is received and before m_2 is sent, $\mathtt{q}$ will not send m_2 and the session continues with message m_3 from $\mathtt{p}$ to $\mathtt{r}$.

Contributions This paper systematically compares the expressiveness of different semantics of multiparty session types based on: (i) the presence and the nature of different data structures used to store messages on either side of communications, (ii) the flexibility of the local types – defined as a subtyping relation, and (iii) the presence of parallel and interruptions.

For the first time, we use sets of languages of local traces to compare expressiveness. We prove, for (i) that the introduction of universal input queues (buffer storing incoming messages regardless of their provenance) leads to deadlock but that in absence of such structure, the denotation of any session G stays the same, regardless of the structures used. We then introduce *flexible subtyping* (ii) which permutes the order of local actions in a limited way. We explain how the combination of flexibility and queues can lead to deadlocks and prove that using flexibility yields greater expressive power. Finally (iii), we claim that session parallelism and interruption have greater expressiveness using our local trace formalism.

2 Multiparty Session Types

Sessions, seen as protocol specifications, are described by *global types* G [11, 7], the main objects being compared in this work. A global type specifies the interactions expected to happen in a session, between several participants (denoted by $\mathtt{p}, \mathtt{q}, \mathtt{r}$), seen from an omniscient point of view. Syntax of the global and local types is given by:

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

We call the different $(m_i)_{i\in I}$ *sets of messages*. Type $\mathtt{end}$ is a termination of session, which we sometimes omit. $\mu\mathtt{t}.G$ and $\mathtt{t}$ are the recursion operators. We manipulate equirecursive types, not distinguishing between $\mu\mathtt{t}.G$ and $G[\mu\mathtt{t}.G/\mathtt{t}]$. We assume recursion variables $\mathtt{t}$ are guarded, i.e. they appear only under some prefix. $\mathtt{r}_1 \to \mathtt{r}_2\{m_i.G_i\}_{i\in I}$ is the basic interaction inside global types: participant $\mathtt{r}_1$ is expected to send message m_j to participant $\mathtt{r}_2$ – we assume $\mathtt{r}_1 \neq \mathtt{r}_2$; according to the j chosen by the sender, the protocol will continue as global type G_j. We write $\mathtt{p} \to \mathtt{q} : m_1.G_1$ when $|I| = 1$. We sometimes write $\mathtt{q}?$ or $\mathtt{r}!$ when the message is not relevant.

Local types describe these protocols from the point of view of a participant and are considered as local guidelines distributed processes must follow. Interactions are decomposed into two sides: input $\mathtt{p}?\{m_i.T_i\}_{i\in I}$ and output $\mathtt{p}!\{m_i.T_i\}_{i\in I}$. Local types are effectively (potentially infinite) trees of input and output actions. We often write $\mathtt{p}!m.T$ or $\mathtt{p}?m.T$ for a singleton and $\mathtt{p}!$ if the message is not important.

Projections Local types are obtained from global types through projection $G\lceil(\mathtt{r})$ (the projection of a global type G onto a participant $\mathtt{r}$). Projection is given by the following rules:

$$\begin{array}{rclll}
 & \mathtt{end}\lceil(\mathtt{r}) = \mathtt{end} & & \mathtt{t}\lceil(\mathtt{r}) = \mathtt{t} & \\
\mu\mathtt{t}.G\lceil(\mathtt{r}) = \mu\mathtt{t}.G\lceil(\mathtt{r}) & (\text{if } G\lceil(\mathtt{r}) \neq \mathtt{t}) & & \mu\mathtt{t}.G\lceil(\mathtt{r}) = \mathtt{end} & (\text{otherwise}) \\
 & \mathtt{r}_1 \to \mathtt{r}_2\{m_i.G_i\}_{i\in I}\lceil(\mathtt{r}) & = & \mathtt{r}!\{m_i.G_i\lceil(\mathtt{r})\}_{i\in I} & (\text{if } \mathtt{r} = \mathtt{r}_1) \\
 & \mathtt{r}_1 \to \mathtt{r}_2\{m_i.G_i\}_{i\in I}\lceil(\mathtt{r}) & = & \mathtt{r}?\{m_i.G_i\lceil(\mathtt{r})\}_{i\in I} & (\text{if } \mathtt{r} = \mathtt{r}_2) \\
 & \mathtt{r}_1 \to \mathtt{r}_2\{m_i.G_i\}_{i\in I}\lceil(\mathtt{r}) & = & G_1\lceil(\mathtt{r}) & (\text{otherwise, and } \forall i,j \in I.G_i\lceil(\mathtt{r}) = G_j\lceil(\mathtt{r}))
\end{array}$$

Recursive global types are projected into recursive local types except when projection name $\mathtt{r}$ does not appear in a recursion block, i.e. $\mathtt{r}$ is not involved in the recursion, thus projection is $\mathtt{end}$. When projecting an communication, if the projection name is the sender (resp. the receiver), the result will be a send (resp. receive) action. If the name is not involved in the communication, the first branch is chosen to continue projection. In the last rule, a choice made during a communication is unobservable to other participants, hence projections in all branches are the same (see [11]). We call projectable global types *well-formed* and assume all types are well-formed in the following.

► **Example 1** (Projection)**.** Consider $G = \mathtt{p} \to \mathtt{q} : m.\mathtt{q} \to \mathtt{p} : m_1.\mathtt{r} \to \mathtt{p} : m_2.\mathtt{end}$, described above. This global type describes a session composed of three interactions: $\mathtt{r}$ sends a message m to $\mathtt{q}$ which then sends a message m_1 to $\mathtt{p}$ and finally $\mathtt{r}$ sends a message m_2 to $\mathtt{p}$. Projection of G onto its three participants gives: $\{\mathtt{r} : \mathtt{q}!m.\mathtt{p}!m_2,\quad \mathtt{q} : \mathtt{r}?m.\mathtt{p}!m_1,\quad \mathtt{p} : \mathtt{q}?m_1.\mathtt{r}?m_2\}$.

As seen above, local types do not represent a direct causality between sending m_1 and m_2 as the actions are done by different participants. There is however causality between the reception of m_1 and m_2 from the point-of-view of $\mathtt{p}$ – should the semantics be synchronous, this causality would be propagated to send operations.

3 Expressiveness of Multiparty Session Configurations

This section first defines the operational semantics of multiparty session types as *session configurations*. Then we define our notion of expressiveness, introducing the denotational semantics. Finally we show that (without asynchronous subtyping), expressive powers of all semantics are equivalent.

Semantics for sessions are transitions between *configurations* Δ: models of the state of a system through (i) a set of local types describing remaining actions to be performed by the participants and (ii) queues describing messages currently travelling in the networks. Semantics presented below are parametric w.r.t. the existence (and usage) of such queues.

3.1 Configuration semantics

The syntax of configurations (Δ) and queues (Q) is given below:

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

Queues can be *output queues* $(\mathtt{p} \blacktriangleright \mathtt{q} : h)$, $(\mathtt{p} \blacktriangleright : h)$ and store messages after they are produced by a participant – before they travel through the network – or *input queues* $(\mathtt{p} \blacktriangleleft \mathtt{q} : h)$, $(\mathtt{p} \blacktriangleleft : h)$ and store messages before they are consumed by a participant – after they arrived from the network.

Queues can be linked to a single endpoint, the endpoint consuming messages for input queues, and the endpoint producing messages for output queues. They are written $(\mathtt{p} \blacktriangleleft : h)$ and $(\mathtt{p} \blacktriangleright : h)$ and are called *single queues*. Queues can also be labelled by two endpoints (source and destination of the message) and are in this case called *multiple queues* and written $(\mathtt{p} \blacktriangleleft \mathtt{q} : h)$ and $(\mathtt{p} \blacktriangleright \mathtt{q} : h)$.

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

$(\mathsf{p} \triangleleft \mathsf{q} : h)$ (resp. $(\mathsf{p} \triangleright \mathsf{q} : h)$) stands for either $(\mathsf{p} \blacktriangleleft \mathsf{q} : h)$ (resp. $(\mathsf{p} \blacktriangleright \mathsf{q} : h)$) or $(\mathsf{p} \blacktriangleleft : h)$ (resp. $(\mathsf{p} \blacktriangleright : h)$)

Figure 3 Operational semantics of session cofigurations

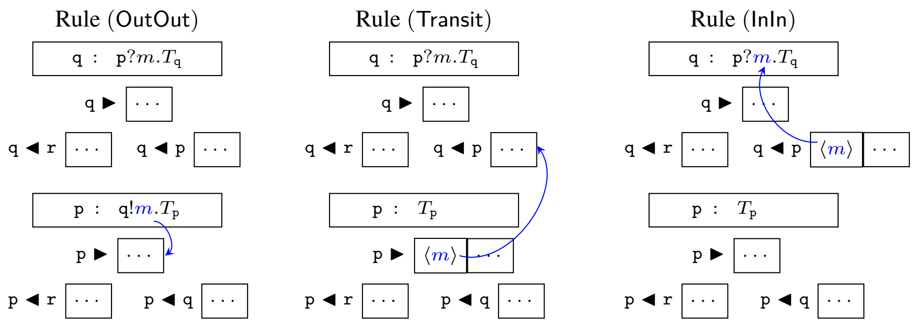

Figure 4 Illustration of several rules in the semantics $(M, 1)$

The transition rules are given in Figure 3. In the following, the system will either (i) have no input (resp. output) queues, or (ii) one single input (resp. output) queue per participant and no multiple input (resp. output) queues, or (iii) one multiple input (resp. output) queues per pair of participants and no multiple input (resp. output) queues. A system with n participants with single input (resp. output) queues will have n input (resp. output) queues. A system with n participants with multiple input (resp. output) will have n^2 input (resp. output) queues. In the last rule, ℓ denotes a label which is either input $(\mathsf{p}?\mathsf{q} : m_j)$, output $(\mathsf{p}!\mathsf{q} : m_j)$, internal action (τ) or synchronisation $(\mathsf{pq} : m)$.

A semantics ϕ is defined by a pair (I, O) representing the nature of the input and output queues of the system. I (resp. O) can be 0 (no input (resp. output) queues), 1 (single input (resp. output) queues), or M (multiple input (resp. output) queues). This effectively defines 9 different semantics using different sets of rules. They are summarised in Table 1.

Figure 4 illustrates the three rules used by semantics $(M, 1)$, i.e. a semantics with single output queues and multiple input queues. Rule (OutOut) consumes the action $\mathsf{q}!m$ of participant p to produce message $\langle \mathsf{p}, \mathsf{q}, m\rangle$ (noted as only $\langle m\rangle$ in the picture) in the output queue $(\mathsf{p} \blacktriangleright)$. When the message reaches the end of the queue, it is dispatched to the input queue $(\mathsf{p} \blacktriangleleft \mathsf{q})$ through rule $(\mathsf{Transit})$. Eventually, the message will be ready to be consumed by the action $\mathsf{p}?m$ of participant q by rule (InIn).

The $(0, 0)$ semantics is the *synchronous semantics*. The three semantics $(1, \_)$ are called the *single-input semantics* or *unsafe semantics* and the six other ones are called *safe semantics*. These names come from Proposition 11. We say that $\Delta_1 \xrightarrow{\ell} \Delta_2$ through semantics ϕ when $\Delta_1 \xrightarrow{\ell} \Delta_2$ is derived with rules belonging to ϕ (according to Table 1).

In the following, we consider that two systems are different if the possible sequences of actions for *one* participant differ. We only consider the order of actions happening locally. We will compare

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

Table 1 Rules used by the different semantics ϕ

configuration traces, which are collections of local traces.

An event e is either a send event $\mathtt{p}!m$ or a receive event $\mathtt{p}?m$. For a participant, sending corresponds either to a communication (synchronous semantics), or putting a message in its own output queue ($(\_,1)$ and $(\_,M)$), or putting a message in the target input queue $(\_,0)$.

▶ **Definition 2** (Configuration traces). A configuration trace σ is a mapping from participants to finite sequences of events: $\sigma(\mathtt{r}) = (e)_{n \leq N}$ for $N \in \mathbb{N}$. We use ϵ for the empty sequence. A participant $\mathtt{r}$ is in the *domain* of σ if $\sigma(\mathtt{r}) \neq \epsilon$. The length of a trace σ is the sum of the length of the sequences $\sigma(\mathtt{r})$ for all $\mathtt{r}$ in its domain. We say $\sigma \leq \sigma'$ when $\forall \mathtt{r}$, $\sigma(\mathtt{r})$ is a sequence prefix of $\sigma'(\mathtt{r})$.

The relation between traces and configuration is given by the relation $\Delta \rightsquigarrow^{\sigma}_{\phi} \Delta'$ meaning Δ *executes trace* σ_0 *to* Δ *for semantics* ϕ defined with:

1. For any configuration Δ and semantics ϕ, $\Delta \rightsquigarrow^{\sigma_0}_{\phi} \Delta$ where σ_0 is defined by: for all roles $\mathtt{r}$, $\sigma_0(\mathtt{r}) = \epsilon$.
2. For any configurations $\Delta, \Delta_1, \Delta_2$, any trace σ, any label ℓ, and any semantics ϕ, if $\Delta \rightsquigarrow^{\sigma}_{\phi} \Delta_1$ and if $\Delta_1 \xrightarrow{\ell} \Delta_2$ through ϕ, then we define $\Delta \rightsquigarrow^{\sigma'}_{\phi} \Delta_2$ as follows:
   a. if $\ell = \mathtt{p}!\mathtt{q} : m_j$, then σ' is defined by: $\sigma'(\mathtt{p}) = \sigma(\mathtt{p}).\mathtt{q}!m_j$ and $\sigma'(\mathtt{r}) = \sigma(\mathtt{r})$ for $\mathtt{r} \neq \mathtt{p}$.
   b. if $\ell = \mathtt{p}?\mathtt{q} : m_j$, then σ' is defined by: $\sigma'(\mathtt{q}) = \sigma(\mathtt{q}).\mathtt{p}?m_j$ and $\sigma'(\mathtt{r}) = \sigma(\mathtt{r})$ for $\mathtt{r} \neq \mathtt{p}$.
   c. if $\ell = \mathtt{pq} : m_j$, then σ' is defined by: $\sigma'(\mathtt{p}) = \sigma(\mathtt{p}).\mathtt{q}!m_j$, $\sigma'(\mathtt{q}) = \sigma(\mathtt{q}).\mathtt{p}?m_j$ and $\sigma'(\mathtt{r}) = \sigma(\mathtt{r})$ for $\mathtt{r} \notin \{\mathtt{p},\mathtt{q}\}$.
   d. if $\ell = \tau$, then $\sigma' = \sigma$.

A trace σ is in the *trace set* of a configuration Δ for a semantic ϕ, written $\sigma \in \mathcal{T}_{\phi}(\Delta)$, (we sometimes write Δ *has trace* σ *for semantics* ϕ) whenever there exist Δ' s.t. $\Delta \rightsquigarrow^{\sigma}_{\phi} \Delta'$. The trace set of a global type G for the semantics ϕ is the trace set for semantics ϕ of configuration $\delta(G)$ defined by $\delta(G) = \mathtt{r}_1 : T_1 \ldots, \mathtt{r}_n : T_n, Q_1, \ldots, Q_n$ where $\mathtt{r}_1, \ldots, \mathtt{r}_n$ are the roles involved in G, $T_i = G\lceil(\mathtt{r}_i)$, and Q_i are all empty ϵ and correspond to ϕ. A *terminated trace* of a global type G for the semantics ϕ is a trace σ s.t. $\delta(G) \rightsquigarrow^{\sigma}_{\phi} \Delta$ where $\Delta \not\rightarrow$. A *completed trace* of a global type G for the semantics ϕ is a trace σ s.t. $\delta(G) \rightsquigarrow^{\sigma}_{\phi} 0$ where $0 = \mathtt{r}_1 : \mathtt{end}, \ldots, \mathtt{r}_n : \mathtt{end}, Q_1, \ldots, Q_n$ where Q_i are all ϵ. A completed trace is terminated.

▶ **Example 3** (Configuration). Let $\Delta_e = \{\mathtt{p} : \mathtt{q}!m_1.\mathtt{r}!m_3, \mathtt{q} : \mathtt{p}?m_1.\mathtt{r}!m_2, \mathtt{r} : \mathtt{q}?m_2.\mathtt{p}?m_3\}$ (cf. Example 1). The initial configuration from G_e for $(0,0)$ is Δ_e, the one for $(M,1)$ is $\Delta_e, (\mathtt{p} \blacktriangleright: \epsilon), (\mathtt{q} \blacktriangleright: \epsilon), (\mathtt{r} \blacktriangleright: \epsilon), (\mathtt{p} \blacktriangleleft \mathtt{q} : \epsilon), (\mathtt{p} \blacktriangleleft \mathtt{r} : \epsilon), (\mathtt{q} \blacktriangleleft \mathtt{p} : \epsilon), (\mathtt{q} \blacktriangleleft \mathtt{r} : \epsilon), (\mathtt{r} \blacktriangleleft \mathtt{p} : \epsilon), (\mathtt{r} \blacktriangleleft \mathtt{q} : \epsilon)$. Both configurations can evolve along the terminated trace $\sigma_e : \mathtt{p} \mapsto \mathtt{q}!m_1.\mathtt{r}!m_3, \mathtt{q} \mapsto \mathtt{p}?m_1.\mathtt{r}!m_2, \mathtt{r} \mapsto \mathtt{q}?m_2.\mathtt{p}!m_3$ even if non-terminated traces are different; for instance $\sigma_t : \mathtt{p} \mapsto \mathtt{q}!m_1, \mathtt{q} \mapsto \emptyset, \mathtt{r} \mapsto \emptyset$ is a valid trace from G_e by $(M,1)$ and not by $(0,0)$.

3.2 Expressiveness via denotational semantics

We can extract from Definition 2 *a denotation* of a global type G, w.r.t a particular semantics, as the set of terminated traces of G. We can compare, for a given type G, the terminated traces of G for two different semantics. As sessions ensure the local interaction follows an expected behaviour, local traces are strongly constrained by the semantics. This observation is still useful for two reasons: (i) it establishes a distinction between safe semantics which prevents deadlocks from arising and unsafe semantics, and (ii) further operations (§ 4, 5.1 and 5.2) on types will remove this constraint. Secondly, we can associate, to a type G containing n participants, the languages $\{L_i\}_{i \leq n}$ corresponding to the local traces in the set of terminated traces for G. We can then consider the *expressive power* of ϕ as the set of all languages obtainable for all possible G with ϕ. We define ϕ_1 has greater expressive power than ϕ_2 if all languages in the expressive power of ϕ_2 are in the expressive power of ϕ_1.

▶ **Definition 4** (Denotation of a type under a semantics). We define the *denotation* of global type G under semantics ϕ, noted $\mathbf{D}(G, \phi)$, as the set of all terminated traces from G w.r.t. ϕ.

▶ **Definition 5** (Progress). We say that ϕ ensures progress if for all G, $\mathbf{D}(G, \phi)$ contains only completed traces.

If role $\mathtt{r}$ appears in G, $\mathbf{D}(G, \phi, \mathtt{r})$ is the set of all local traces for $\mathtt{r}$ obtained from terminated traces of $\mathbf{D}(G, \phi)$, that is $\mathbf{D}(G, \phi, \mathtt{r}) = \{\sigma(\mathtt{r}) | \sigma \in \mathbf{D}(G, \phi)\}$.

▶ **Definition 6** (Expressive power of a semantics). We define the *expressive power* of semantics ϕ as follows: $\{\mathbf{D}(G, \phi, \mathtt{r}) \mid \mathtt{r} \in G \text{ and } G \text{ well-formed}\}$, that is, the collection of all languages of local traces corresponding to terminated traces from well-formed global types.

3.3 Expressiveness results (without subtyping)

The first theorem (Theorem 10) states that all safe semantics give the same denotations: adding non-single-input queues has no influence on denotations. We first prove confluence of semantics by stating that we can always complete a trace of any semantics by using synchronous semantics.

▶ **Lemma 7** (Confluence of trace semantics). *Let ϕ_1 be a safe semantics and G a well-formed global type. If $\delta(G) \rightsquigarrow^{\sigma}_{\phi_1} \Delta$, there exists Δ', σ' s.t. $\delta(G) \rightsquigarrow^{\sigma.\sigma'}_{\phi_1} \Delta'$ and $\delta(G) \rightsquigarrow^{\sigma.\sigma'}_{(0,0)} \Delta'$*

Proof. We first prove that if σ is a valid trace for ϕ but not for $(0, 0)$, it has outputs with unmatched inputs. We then prove that (as the semantics is safe), messages in transit can be moved and consumed at their destination. We thus reach a configuration with only empty queues with a trace corresponding to a synchronous one. ◀

▶ **Definition 8** (Prefix). Let T be a type, σ a trace and $\mathtt{r}$ a participant. The prefix relation $\sigma(\mathtt{r}) <_{\mathtt{p}} T$ is defined as: (1) $\epsilon <_{\mathtt{p}} T$; (2) $\mathtt{p}!m_j.\sigma <_{\mathtt{p}} \mathtt{q}!\{T_i\}_{i \in I}$ and $\sigma <_{\mathtt{p}} T_j$; and (3) $\mathtt{p}?m_j.\sigma <_{\mathtt{p}} \mathtt{q}?\{T_i\}_{i \in I}$ and $\sigma <_{\mathtt{p}} T_j$.

▶ **Lemma 9** (Session fidelity). *Let ϕ be a safe semantics and G a well-formed global type. If $\delta(G) \rightsquigarrow^{\sigma}_{\phi} \Delta$, $\mathtt{r} \in G$, then $\sigma(\phi) <_{\mathtt{p}} G\lceil(\mathtt{r})$.*

Proof. Easy by induction over the length of σ, noticing that the possible actions of $\mathtt{r}$ are constrained by $G\lceil(\mathtt{r})$. ◀

▶ **Theorem 10** (Expressiveness of safe semantics). *For any G, the sets $\mathbf{D}(G, \phi)$ for all safe ϕ are the same.*

Proof. Done by proving that all safe semantics are equivalent to the synchronous one: local traces stay the same, as they are constrained by the initial global types. Suppose σ_1 is a completed trace for ϕ_1. By using Lemma 7 there exists σ'_1 s.t. $\sigma_1.\sigma'_1$ is a trace for ϕ_1 and $(0, 0)$. By Lemma 9, σ'_1 is ϵ. It follows that σ_1 is a completed trace for $(0, 0)$, thus a completed trace for ϕ_2. ◀

However, unsafe semantics are not comparable with the others, as they lead to deadlocks.

▶ Proposition 11 (Single input deadlock). *Unsafe semantics do not ensure progress.*

Proof. Consider a global type $G_e = \mathtt{p} \to \mathtt{q} : m_1.\mathtt{q} \to \mathtt{r} : m_2.\mathtt{p} \to \mathtt{r} : m_3$ with the $(1, 0)$ semantics (same reasoning applies to $(1, 0)$ and $(1, M)$). After the sequence $\mathtt{p}!\mathtt{q} : m_1.\mathtt{p}!\mathtt{r} : m_3.\mathtt{p}?\mathtt{q} : m_1.\mathtt{q}!\mathtt{r} : m_2$, $\mathtt{r}$ type is $\mathtt{q}?m_2.\mathtt{p}?m_3.\mathtt{end}$ and its queue is $(\mathtt{r} \blacktriangleleft \bullet : \langle \mathtt{p}, \mathtt{r}, m_3\rangle\langle \mathtt{q}, \mathtt{r}, m_2\rangle)$ meaning the system can no longer proceed: $\mathtt{r}$ expects m_2 then m_3 but the queue offers m_3 then m_2. ◀

▶ **Proposition 12 (Regularity).** The languages in expressive power of safe semantics are regular.

Proof. Suppose G is a session type containing participant $\mathtt{p}$. We prove that the traces accepted by local $T = G\lceil(\mathtt{r})$ by induction on T.

- If $T = \mathtt{end}$, the set of possible completed traces is $\emptyset$ and we conclude.
- If $T = \mathtt{q}?\{m_i.T_i\}_{i\in I}$, then the possible completed traces $\mathtt{q}?l_j.t$ all start with one $\mathtt{q}?l_j$ and follows by an accepted trace t_j of T_j. By induction, the language L_j of all t_j is regular. Thus the accepted language is the sum, for all j of the traces $\mathtt{q}?l_j.L_j$, and is regular. Same reasoning holds for $T = \mathtt{q}!\{m_i.T_i\}_{i\in I}$.
- If $T = \mu\mathtt{t}.T'(\mathtt{t})$ we check the number of occurrences of $\mathtt{end}$ in $T'(\mathtt{t})$. If $\mathtt{end}$ does not appear, the accepted language of T is $\emptyset$. Otherwise, by masking about the recursion token $\mathtt{t}$, we can see the accepted traces of T' as a sum of language ΣL_k, each language L_k corresponding of one branch of T'. There is at most one branch ending with $\mathtt{t}$, suppose it is the branch corresponding to L_1, it means the accepted traces of T are $(L_1) * .(\Sigma_{k\neq 1} L_k)$, which is regular. ◀

▶ **Remark (Asymmetry of expressiveness).** Progress requires input queues to be multiple (or no input queues at all), but is independent from output queues. Output queues do not affect expressiveness as they cannot block endpoints: if there is an input queue, the former can always unload its content in the latter, if there is none, as the order of messages in the output queue matches the order of the session, session type soundness ensures progress. Without input queues, *session fidelity* [11] is required, for instance consider the configuration $\mathtt{r} : \mathtt{p}!m_1.\mathtt{q}!m_2.\mathtt{end}, \mathtt{p} : \mathtt{q}!m_3.\mathtt{r}?m_1.\mathtt{end}, \mathtt{q} : \mathtt{r}?m_2.\mathtt{p}?m_1.\mathtt{end}$. Terminated traces are different for $(0, 0)$ and $(0, 1)$ (in this case, the system is blocked as it would require m_2 to be sent before m_1) and for $(0, M)$ (the system can proceed to a completed state); however, the initial configuration does not correspond to a global type.

4 Expressiveness of subtyping

As described in the previous section, the mechanisms of session maintain an order over local components traces: actions performed by the participants happen in the exact order expected by types. As a consequence, if transport structures change the order of arrival of messages from different sources – a realistic assumption, this condition can lead to deadlocks.

Implementations of session types [22, 18] sometimes enforce *flexibility* for the endpoint application: for instance, by allowing two outputs expected to be sent sequentially to be performed in any order. This flexibility is represented formally by the use of *subtyping*, as in [16, 6, 17], describing transformations on types by switching pairs of consecutive actions. We study input-input and output-output flexibility which make it possible to exchange two consecutive actions of the same nature. Input-output (resp. output-input) flexibility allows one output (resp. input) to be performed before previously expected inputs *and* outputs.

4.1 Subtyping rules

As explained above, input-input flexibility offers the possibility for the second input in a sequence of two consecutive inputs to be executed first. For instance, input flexibility allows type

$\mathtt{p}?m_1.\mathtt{q}?m_2.\mathtt{p}!m_3.\mathtt{end}$ to be converted into $\mathtt{q}?m_2.\mathtt{p}?m_1.\mathtt{p}!m_3.\mathtt{end}$. Consider the following types:

$$T = \mathtt{p}?\begin{cases} m_{11}.\mathtt{q}?m_2.\mathtt{p}!m_3.\mathtt{end} \\ m_{12}.\mathtt{q}?m_2.\mathtt{p}!m_4.\mathtt{end} \end{cases} \qquad T' = \mathtt{q}?m_2.\mathtt{p}?\begin{cases} m_{11}.\mathtt{p}!m_3.\mathtt{end} \\ m_{12}.\mathtt{p}!m_4.\mathtt{end} \end{cases}$$

In T, the first input is branching and models a program reacting to two different messages from $\mathtt{p}$, either m_{11} or m_{12}. In both branches, a message m_2 from $\mathtt{q}$ is expected. One would expect an input-input flexible program to be able to accept message m_2 if it arrives first, which would mean converting T to T', which means T' is a subtyping of T.

For a subtyping definition, we introduce the input and output contexts. They are parametered with the name of the participant of the action being permuted, as we do not exchange two actions with the same participant. An input (resp. output) context is a term formed only of branched input (resp. output) actions, seen as tree where each branch finishes with a hole. Input-output and output-input contexts contain both type of actions, they only differ in the name verification.

▶ **Definition 13** (Input/Output contexts). Input and input/output type contexts are defined by:

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

Flexibility is defined by the subtyping rules, which realise the possible exchanges in branched types.

▶ **Definition 14** (Subtyping). Subtyping $\leq$ is coinductively defined by the following rules:

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

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

Subtyping is extended on configurations. To describe semantics with subtyping, we define a *subtyping policy* $\mathcal{P}$ as a subset of $\{\mathsf{OO}, \mathsf{II}, \mathsf{OI}, \mathsf{IO}\}$ abiding to $\mathsf{OO} \in \mathcal{P} \Rightarrow \mathsf{OI} \in \mathcal{P}$ and $\mathsf{OI} \in \mathcal{P} \Rightarrow \mathsf{OO} \in \mathcal{P}$. We use the notation $T_1 \leq_{\mathcal{P}} T_2$ to state that $T_1 \leq T_2$ is derivable using only rules in $\mathcal{P}$. We consider semantics ϕ associated to a subtyping policy $\mathcal{P}$, which are obtained by adding the following rule:

$$(\mathsf{Sub}:\mathcal{P}) \qquad \Delta_1 \xrightarrow{l} \Delta_2 \quad {\Delta'}_1 \leq_{\mathcal{P}} \Delta_1 \quad \Longrightarrow \quad {\Delta'}_1 \xrightarrow{l} \Delta_2$$

Subtyping makes it possible to first perform an action that is present in the branches of all possible behaviours. (II) performs inputs from different senders in any order; (OO) performs an output before other outputs to different receivers, (IO) allows an output to be performed before other actions with different participants, and (OI) allows an input to be performed before other actions with different participants.

4.2 Progress and expressiveness of flexibility

Flexibility introduces deadlock under the synchronous semantics. For instance, consider the global type $\mathtt{r} \to \mathtt{q} : m.\mathtt{q} \to \mathtt{p} : m_1.\mathtt{r} \to \mathtt{p} : m_2.\mathtt{end}$. In one application of (IO) on the initial configuration, we reach configuration $(\mathtt{r} : \mathtt{q}!m.\mathtt{p}!m_2, \mathtt{q} : \mathtt{p}!m_1.\mathtt{r}?m_2, \mathtt{p} : \mathtt{q}?m_1.\mathtt{r}!m_2)$ which is locked for synchronous semantics, and we have no means to retrieve initial configuration. Proposition 16 and Table 2 describe which association of a semantics ϕ and a set of subtyping rules $\mathcal{P}$ avoid deadlocks. A subtyping policy $\mathcal{P}$ is *safe* w.r.t. a semantics ϕ if the system $\mathcal{P}$ ensure progress under ϕ.

▶ **Lemma 15.**
- $\emptyset$, $\{\mathsf{OO}\}$ *and* $\{\mathsf{II}\}$ *and* $\{\mathsf{OI}, \mathsf{IO}\}$ *are safe w.r.t. all safe semantics.*
- $\{\mathsf{IO}\}$ *is safe w.r.t. all non-synchronous semantics.*
- $\{\mathsf{OI}\}$ *is unsafe w.r.t. all semantics.*

	$(0,0)$	$(0,1)$	$(0,D)$	$(1,0)$	$(1,1)$	$(1,D)$	$(D,0)$	$(D,1)$	(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	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$

Table 2 Safe subtyping with respect to semantics

▶ Proposition 16 (Safe subtyping). *Safety for subtyping policy and semantics is given by Table 2: "$\surd$" represents a safe semantics, and "$\times$" an unsafe one.*

For a given semantics, one can use subtyping to complete traces that are not possible without it. As a simple example consider $\mathtt{p} \to \mathtt{q} : m_1.\mathtt{p} \to \mathtt{r} : m_2$, trace $\mathtt{r}!m_2.\mathtt{q}!m_1$ is accepted with OO-subtyping but cannot be accepted otherwise. We use $\mathbf{D}(G, \mathcal{P}, \phi)$ to represent the denotation of type G under semantics ϕ and subtyping rules $\mathcal{P}$. Below Proposition 17 confirms that subtyping actually changes the denotations of types. Theorem 18 states that subtyping accept traces beyond the regular languages.

▶ Proposition 17 (Denotations in presence of subtyping). If ϕ is safe and $\mathcal{P}_1 \subsetneq \mathcal{P}_2$,
1. for all projectable global type G, $\mathbf{D}(G, \mathcal{P}_1, \phi) \subseteq \mathbf{D}(G, \mathcal{P}_1, \phi)$
2. there exists a projectable global type G, $\mathbf{D}(G, \mathcal{P}_1, \phi) \subsetneq \mathbf{D}(G, \mathcal{P}_1, \phi)$

Proof. 1. Direct, as all completed traces for $\mathcal{P}_1, \phi$ are completed traces for $\mathcal{P}_2, \phi$.
2. For instance if $\mathcal{P}_1$ is $\emptyset$ and $\mathcal{P}_2$ is II, the denotation $\mathbf{D}(\mathtt{r} \to \mathtt{p} : m_1, \mathtt{q} \to \mathtt{p} : m_2, \mathcal{P}_2, \phi)$ contains a trace σ s.t. $\sigma(\mathtt{p}) = \mathtt{q}?m_2.\mathtt{p}?m_1$, whereas all traces σ in $\mathbf{D}(\mathtt{r} \to \mathtt{p} : m_1, \mathtt{q} \to \mathtt{p} : m_2, \mathcal{P}_1, \phi)$ are s.t. $\sigma(\mathtt{p}) = \mathtt{p}?m_1.\mathtt{q}?m_2$. ◀

▶ **Theorem 18** (Expressive power of subtyping). *The expressive power of subtyped sessions is strictly greater than the expressive power of standard sessions.*

Proof. We prove that the expressive power of subptyped sessions contained non-regular languages. Consider the type $G = \mu\mathtt{t}.\mathtt{p} \to \mathtt{q}.\mathtt{p} \to \mathtt{r}.\mathtt{t}$. Its projection on $\mathtt{p}$ is $T = \mu\mathtt{t}.(\mathtt{q}!.\mathtt{r}!.\mathtt{t} + \mathtt{q}.\mathtt{end})$. Although it is of no importance for the following, we can establish the possible completed traces from T by the safe semantics $(\emptyset, \emptyset)$ is $(\mathtt{q}!.\mathtt{r}!)^*.\mathtt{q}!$. (1) With the semantics $(\emptyset, \{\mathsf{OO}\})$, it is easy to see the language L of possible completed traces are all the words obtain by shuffling $\mathtt{q}!^{n+1}$ and $\mathtt{r}!^n$: all completed traces that contains n $!\mathtt{q}$ contains exactly $(n-1)$ $\mathtt{r}!$ and type permutations allow us to move any $\mathtt{r}!$ leftwards in any trace and stay inside the completed trace set – thanks to the OO rule. (2) The shuffling of $\mathtt{q}!^{n+1}$ and $\mathtt{r}!^n$ is not regular. It follows from Proposition 12 that there does not exist a type G s.t. the language of possible traces for $\mathtt{p}$ in $G{\upharpoonright}(\mathtt{p})$ is L. ◀

5 Expressiveness of parallel and interruptible sessions

We study here the influence on expressive power of two different updates of the language syntax: the addition of parallel composition and interruptible scopes.

5.1 Influence of parallel composition

In the previous section, the reduction rules are updated with subtyping, giving flexibility to the local endpoint. Similar behaviour can also be reached by adding in the syntax a parallel operator explicitly stating that two actions can be performed in any order. We consider in the following, *parallel sessions* which are sessions with the additional constructs for parallel composition $G_1 \mid G_2$ and $T_1 \mid T_2$. Related projection and semantics rules are standard and can be found in [11].

As expected, adding parallel composition of actions, leads to irregularity of the safe semantics.

▶ Proposition 19 (Parallel). *Expressive power of parallel sessions contains irregular languages.*

Proof. The language of completed traces accepted at $\mathtt{p}$ for the type $\mu\mathtt{t}.(\mathtt{p} \to \mathtt{q}.\mathtt{t}{+}\mathtt{p} \to \mathtt{q}.\mathtt{end} \mid \mathtt{p} \to \mathtt{r}.\mathtt{end})$ is not regular – and is a particular shuffling of $\mathtt{q}!^n$ and $\mathtt{r}!^n$. ◀

Both subtyping rule and parallel syntax can be used to get rid of potential deadlocks. Parallel composition allows one session designer to precisely describe which actions are unordered whereas subtyping is a global policy for permutation. As a result, parallel sessions are more expressive than subtyping sessions, giving a finer control over which actions can be exchanged.

▶ Proposition 20 (Parallel and subtyping). Parallel sessions have a strictly greater expressive power than subtyping sessions.

Proof. For every G without parallel and every safe combination of ϕ and $\mathcal{P}$, we can find G' which has exactly the same trace set. G' is obtained from G by putting in parallel consecutive events from G w.r.t. $\mathcal{P}$. ◀

5.2 Expressiveness of Interruptible Session Types

Interruptible sessions introduce a mechanism for participants to exceptionally exit blocks of interactions; we study here the resulting gain in expressiveness. Interruptible session types are presented in [8, 12]: in the global type syntax, particular ranges of actions (called *scopes*) can be interrupted at any time by a participant; a special message is broadcasted to all participants of the scope. As soon as one of them is notified of the interruption, it gives up any action related to this scope. Interruptions have been included in protocol language *Scribble* [22] because it was needed to represent usecases in [1], see [12]. We prove here that this inclusion is necessary, and that such protocols cannot be described with standard sessions.

As a simple example of the interrupt, consider:

$$G = \{\!|\mathtt{r} \to \mathtt{p} : m.(\mu\mathtt{t}.\mathtt{p} \to \mathtt{q} : m_1.\mathtt{q} \to \mathtt{p} : m_2.\mathtt{t})|\!\}^{\mathtt{c}}\langle i \text{ by } \mathtt{r}\rangle; \mathtt{q} \to \mathtt{r} : a.\mathtt{end}$$

G is a type consisting of one interruptible scope $\mathtt{c}$. It starts with message m from $\mathtt{r}$ to $\mathtt{p}$, which initiates a loop of messages m_1 and m_2 between $\mathtt{p}$ and $\mathtt{q}$. At any time during this loop, $\mathtt{r}$ can decide to stop it by raising an interruption: message i is sent to both $\mathtt{p}$ and $\mathtt{q}$ which are expected to stop interacting with each other as soon as they receive it. After interruption, the session then resumes by a message a from $\mathtt{q}$ to $\mathtt{r}$.

Interruptible session types are standard session types with the addition of scope constructions. $\{\!|G|\!\}^{\mathtt{c}}\langle l \text{ by } \mathtt{r}\rangle; G'$ is the global type composed of one scope name $\mathtt{c}$ encompassing the type G. At any time, progress inside G can be interrupted by participant $\mathtt{r}$ with a special interrupt message carrying label l and $\mathtt{Eend}$ stands for a exceptionally ended scope. After G is finished - either normally or exceptionally, the protocol continues as G'. Each scope $\mathtt{c}$ is associated to a set of participants involved through the mapping Γ. Such information allows the semantics to notify each participant when an exceptional behaviour arises. Additional projection rules needed for interruptible scopes, projection remembers whether it projects on the name which can interrupt the scope or not, resulting in two different constructs for interruptible local types:

$$\begin{array}{rcll}
\{\!|G|\!\}^{\mathtt{c}}\langle l \text{ by } \mathtt{r}\rangle; G'\lceil(\mathtt{r}) & = & \{\!|G\lceil(\mathtt{r})|\!\}^{\mathtt{c}} \rhd \langle \mathtt{r}?l\rangle; G'\lceil(\mathtt{r}) & \\
\{\!|G|\!\}^{\mathtt{c}}\langle l \text{ by } \mathtt{r}'\rangle; G'\lceil(\mathtt{r}) & = & \{\!|G\lceil(\mathtt{r})|\!\}^{\mathtt{c}} \lhd \langle \mathtt{r}'!l\rangle; G'\lceil(\mathtt{r}) & \text{when } \mathtt{r} \in G \qquad \text{otherwise} \quad G'\lceil(\mathtt{r})
\end{array}$$

Excerpt of configuration semantics for interruptible sessions (details are in [8, 12]) is given below through the use of evaluation contexts $\mathbb{C}^{\mathtt{c}}$ which has a hole in $\{\!|\_|\!\}$ and after the sequential composition (formally defined in Appendix).

$$\begin{array}{ll}
(\mathsf{EOut}) & \mathtt{r} : \mathbb{C}^{\mathsf{c}_0}[\{\!|T|\!\}^{\mathsf{c}} \rhd \langle \mathtt{r}?l\rangle; T'], \mathtt{r}_1 : h, \ldots, \mathtt{r}_n : h \\
 & \quad \to \mathtt{r} : \mathbb{C}^{\mathsf{c}_0}[\{\!|\mathtt{Eend}|\!\}^{\mathsf{c}} \rhd \langle \mathtt{r}?l\rangle; T'], \mathtt{r}_1 : \langle \mathsf{c}^{\mathtt{I}}, \mathtt{r}, \mathtt{r}_1, l\rangle.h, \ldots, \mathtt{r}_n : \langle \mathsf{c}^{\mathtt{I}}, \mathtt{r}, \mathtt{r}_n, l\rangle.h \\
(\mathsf{EIn}) & \mathtt{r} : \mathbb{C}^{\mathsf{c}_0}[\{\!|T|\!\}^{\mathsf{c}} \rhd \langle \mathtt{q}?l\rangle; T']; \mathtt{r} : h.\langle \mathsf{c}^{\mathtt{I}}, \mathtt{q}, \mathtt{r}, l\rangle.h \to \mathtt{r} : \mathbb{C}^{\mathsf{c}_0}[\{\!|\mathtt{Eend}|\!\}^{\mathsf{c}} \rhd \langle \mathtt{q}?l\rangle; T']; \mathtt{r} : h \\
(\mathsf{Disc}) & \mathtt{r} : \mathbb{C}^{\mathsf{c}_0}[\{\!|\mathtt{Eend}|\!\}^{\mathsf{c}} \rhd \langle \mathtt{q}?l\rangle; T']; \mathtt{r} : \langle \mathsf{c}_1, \mathtt{q}, \mathtt{r}, l\rangle.h \to \mathtt{r} : \mathbb{C}^{\mathsf{c}_0}[\{\!|\mathtt{Eend}|\!\}^{\mathsf{c}} \rhd \langle \mathtt{q}?l\rangle; T']; \mathtt{r} : h \\
(\mathsf{EDisc}) & \mathtt{r} : \mathbb{C}^{\mathsf{c}_0}[\{\!|\mathtt{Eend}|\!\}^{\mathsf{c}} \rhd \langle \mathtt{q}?l\rangle; T']; \mathtt{r} : \langle \mathsf{c}_1^{\mathtt{I}}, \mathtt{q}, \mathtt{r}, l\rangle.h \to \mathtt{r} : \mathbb{C}^{\mathsf{c}_0}[\{\!|\mathtt{Eend}|\!\}^{\mathsf{c}} \rhd \langle \mathtt{q}?l\rangle; T']; \mathtt{r} : h
\end{array}$$

In this framework, in-transit messages contains scope information c, it allows interrupt messages to exit the right scope. In (EOut), participant $\mathtt{r}$ decides to raise an interruption of scope c and continues as T' but remembers that scope c was exited exceptionally; interruption messages are broadcasted to all participants present in scope c. In (EIn) a participant $\mathtt{r}$ executing actions in scope c receives an interrupt messages from $\mathcal{Q}$, and immediately exits c. Rule (Disc) (resp. rule (EDisc)) is used to discard incoming standard (resp. interruption) messages to already-exited scopes.

Theorem 22 claims that session languages with interruptions have greater expressiveness. Denotations resulting from interruptible session types cannot be obtained by use of parallel, choice and flexibility subtyping. We use the following lemma whose detailed proofs can be found in Appendix.

▶ **Lemma 21.** *Expressive powers of standard, parallel and subtyped sessions do not contain languages of the form $a^n.b^k$ with $k \leq n$.*

▶ **Theorem 22** (Expressiveness of interruptible sessions). ***1.*** *Interruptible sessions have a strictly greater expressive power than sessions.*
2. *Interruptible sessions have a different expressive power than parallel sessions.*
3. *Interruptible sessions have a different expressive power than subptyped sessions.*

Proof. Standard session behaviours are included into interruptible sessions.

Separation proofs are based on, stating that only interruptible sessions allows trace languages of the form $a^n.b^k$ with $k \leq n$. Lemma 21 is proved by stating that without interruptions, a denotation containing traces where actions a all appear before actions b is such that the number of a and b are independent (coming from the unfoldings of two different recursions).

Interruptible type $\mu\mathtt{t}.\{\!|\mathtt{p} \to \mathtt{q} : m_1.\mathtt{t}|\!\}^{\mathsf{c}}\langle i \text{ by } \mathtt{q}\rangle; \mathtt{q} \to \mathtt{p} : m_2.\mathtt{end}$, is a loop of nested scopes. Messages m_1 are continuously received by $\mathtt{q}$ (unfolding the recursion once at each reception) until an interruption is raised, using rule (EOut). Afterwards, $\mathtt{q}$ discards further incoming m_1 messages with rule (Disc) and the only possible actions is sending of m_2 messages. The number of m_2 messages to be sent corresponds to the number of unfoldings done while receiving m_1, and thus is lower than the number of messages m_1 received. As a consequence, the expressive power of interruptible sessions contains the languages $a^n.b^k$ with $k \leq n$, which does not appear in the expressive power of standard, parallel and subtyped sessions.

Interruptible sessions are not strictly more expressive than parallel and subtyped sessions. The languages corresponding to types $\mu\mathtt{t}.(\mathtt{p} \to \mathtt{q}.\mathtt{t} + \mathtt{p} \to \mathtt{q}.\mathtt{end} \mid \mathtt{p} \to \mathtt{r}.\mathtt{end})$ from Proposition 19 and $\mu\mathtt{t}.\mathtt{p} \to \mathtt{q}.\mathtt{p} \to \mathtt{r}.\mathtt{t}$ from Theorem 18 are not in the expressive power of interruptible sessions, which does not contain shufflings. ◀

6 Related Works

There is a vast literature on expressiveness studies for process calculi; we refer to [20] for a survey (see also [21, § 2.3]). Our work is original as (i) we study expressiveness of *types*, based on the language theory; (ii) we compare the design choices of the network (queue) topology (§ 3); the local permutations (§ 4); and the type constructs (parallel in § 5.1 and and interrupt in § 5.2); and (iii) our notion of expressiveness is based on denotational and operational: we compare completed traces of local actions induced by a global type. As far as we have known, this is the first work to define and investigate expressiveness based on denotations and languages made by traces of concurrecy types.

Our concurrent model stems from a previous work [2] in which networks are modeled as configurations, i.e. collections of types and queues. The model used in [2] is *multisession* and uses routing information updated at runtime to maintain network topology. This feature has been removed for the sake of clarity, as it has little impact on expressiveness.

The first part of our work, focusing on expressiveness on different queue configurations, is inspired by [14] where they studied the typed bisimulation theories of binary sessions with located queues.

Existing works about expressiveness in process algebra is based on encodings; for example, the early work [19] compares expressiveness of synchronous and asynchronous CCS through the impossibility of an encoding. Our work focus on the semantics of types based on the language acceptance of local traces induced by types without encodings. Another paper [9] compares the expressiveness of several process algebras (asynchronous π, distributed π, ambients) through the use of encodings in order to state possibility and impossibility results. Our approach is different, as we use both operation expressiveness and language comparisons.

The syntax and semantics for interruptible sessions have been defined in [12] and are driven by implementation. The gap in expressiveness created by the addition of operator for exceptional behaviours has been studied in [3]. However, the setting is different and the comparisons are based on Turing-(in)completeness of the different calculi defined, whereas, in § 5.2 we use a comparison based on language inclusion. Our interrupt [12, 8] differs from exceptions in sessions studied in [5, 13, 4] as we provide distributed mechanisms for exceptional behaviours. None of [12, 5, 13, 4, 8] studies the expressiveness.

References

1 Ocean Observatories Initiative (OOI). `http://www.oceanobservatories.org/`.

2 Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. Monitoring networks through multiparty session types. In *FMOODS/FORTE 2013*, pages 50–65, 2013.

3 Mario Bravetti and Gianluigi Zavattaro. On the expressive power of process interruption and compensation. *Mathematical Structures in Computer Science*, 19(3):565–599, 2009.

4 Sara Capecchi, Elena Giachino, and Nobuko Yoshida. Global escape in multiparty sessions. *MSCS*, 29:1–50, 2015.

5 Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured interactional exceptions in session types. In *CONCUR*, volume 5201 of *LNCS*, pages 402–417. Springer, 2008.

6 Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. On the preciseness of subtyping in session types. In *PPDP 2014*, pages 146–135. ACM Press, 2014.

7 Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global progress for dynamically interleaved multiparty sessions. *MSCS*, 760:1–65, 2015.

8 Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Practical interruptible conversations: Distributed dynamic verification with multiparty session types and python. *FMSD*, pages 1–29, 2015.

9 Daniele Gorla. On the relative expressive power of asynchronous communication primitives. In *FOSSACS 2006*, pages 47–62, 2006.

10 Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138, 1998.

11 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.

12 Raymond Hu, Rumyana Neykova, Nobuko Yoshida, Romain Demangeon, and Kohei Honda. Practical interruptible conversations - distributed dynamic verification with session types and python. In *RV 2013*, pages 130–148, 2013.

13 Svetlana Jaksic and Luca Padovani. Exception handling for copyless messaging. *Sci. Comput. Program.*, 84:22–51, 2014.

14 Dimitrios Kouzapas, Nobuko Yoshida, Raymond Hu, and Kohei Honda. On asynchronous eventful session semantics. *MSCS*, 2015.

15 Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From communicating machines to graphical choreographies. In *POPL*, pages 221–232, 2015.

16 Dimitris Mostrous and Nobuko Yoshida. Session typing and asynchronous subtyping for the higher-order π-calculus. *Inf. Comput.*, 241:227–263, 2015.

17 Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global principal typing in partially commutative asynchronous sessions. In *ESOP'09*, number 5502 in LNCS. Springer, 2009.

18 Nicholas Ng, Nobuko Yoshida, and Kohei Honda. Multiparty Session C: Safe parallel programming with message optimisation. In *TOOLS*, volume 7304 of *LNCS*, pages 202–218. Springer, 2012.

19 Catuscia Palamidessi. Comparing the expressive power of the synchronous and asynchronous pi-calculi. *Mathematical Structures in Computer Science*, 13(5):685–719, 2003.

20 Joachim Parrow. Expressiveness of process algebras. *Electr. Notes Theor. Comput. Sci.*, 209:173–186, 2008.

21 Jorge A. Pérez. *Higher-Order Concurrency: Expressiveness and Decidability Results*. PhD thesis, University of Bologna, 2010.

22 Scribble Project homepage. `www.scribble.org`.

A Appendix: Interruptible contexts

$$\begin{array}{rllll}
\mathbb{C}^{\epsilon} = & [] & & \\
\mathbb{C}^{\mathsf{c}} = & \{\!|\mathbb{C}^{\mathsf{c}}|\!\}^{\mathsf{c}'\neq\mathsf{c}} \triangleright \langle \mathtt{r}?l\rangle; T' & \mid \{\!|\mathbb{C}^{\mathsf{c}}|\!\}^{\mathsf{c}'\neq\mathsf{c}} \triangleleft \langle \mathtt{r}!l\rangle; T' & \mid \{\!|\mathbb{C}^{\epsilon}|\!\}^{\mathsf{c}} \triangleright \langle \mathtt{r}?l\rangle; T' \\
 & \mid \{\!|\mathbb{C}^{\epsilon}|\!\}^{\mathsf{c}} \triangleleft \langle \mathtt{r}!l\rangle; T' & \mid \{\!|\mathtt{Eend}|\!\}^{\mathsf{c}'\neq\mathsf{c}} \triangleright \langle \mathtt{r}?l\rangle; \mathbb{C}^{\mathsf{c}} & \mid \{\!|\mathtt{Eend}|\!\}^{\mathsf{c}'\neq\mathsf{c}} \triangleleft \langle \mathtt{r}!l\rangle; \mathbb{C}^{\mathsf{c}} \\
 & \mid \{\!|\mathtt{end}|\!\}^{\mathsf{c}'\neq\mathsf{c}} \triangleright \langle \mathtt{r}?l\rangle; \mathbb{C}^{\mathsf{c}} & \mid \{\!|\mathtt{end}|\!\}^{\mathsf{c}'\neq\mathsf{c}} \triangleleft \langle \mathtt{r}!l\rangle; \mathbb{C}^{\mathsf{c}} &
\end{array}$$

B Proofs

In this section, we provide details for proofs of Section 5.2. Lemma 21 formally states the impossibility for non-interruptible sessions to express languages containing two sequences consecutive of actions whose sizes are not independent.

Proof of Lemma 21. We prove that for any global type G, L the language of completed traces for a non-interruptible semantics of $G\lceil(\mathtt{p})$, and a and b two actions s.t. there exist traces in L with an arbitrary number of a and b, if in all traces of L, all occurrences of a happen before all occurrences of b, then there exists traces in L s.t. the number of occurrences of b are strictly greater than the number of occurrences of a. This prevent L to be of the form $a^n.b^k$ with $k \leq n$.

We prove that if in all traces of L, all occurrences of a happen before all occurrences of b, then there exists traces in L s.t. the number of occurrences of b are strictly greater than the number of occurrences of a. This prevent L to be of the form $a^n.b^k$ with $k \leq n$.

If, in all traces, all occurrences of a follows all occurrences of b it means that the occurrences of a and the occurrences of b in T are in two separate recursions – and that a and b are not allowed to permute by the semantics. If they are in separate recursions, it follows that for a given trace t, we can unfold the recursion containing b and obtain a possible trace. ◀

Proof of Theorem 22. It is easy to see, as interrupts are an enrichment of the syntax, that session types with interrupts are at least as expressive as session types.

- We place ourselves in the case where the semantics is not fully asynchronous, for instance, $\mathtt{oo}$ is not in the type permutation scheme, it is easy to prove the following by replacing $\mathtt{oo}$ by any other type permutation.
  We propose the following type and prove that its trace semantics does not correspond to a valid semantics of the standard session types:

$$G = \mu\mathtt{t}.\{\!|\mathtt{p} \to \mathtt{q}.\mathtt{t}|\!\}^{\mathsf{c}}\langle i \text{ by } \mathtt{q}\rangle; \mathtt{q} \to \mathtt{p}.\mathtt{end}$$

  Its projection on $\mathtt{p}$ is $\mu\mathtt{t}.\{\!|\mathtt{q}!.\mathtt{t}|\!\}^{\mathsf{c}} \triangleright \langle \mathtt{q}?i\rangle; \mathtt{q}?.\mathtt{end}$. This type is composed of infinitely many embedded scopes. For instance it is equal to $\{\!|\mathtt{q}!.\{\!|\mathtt{q}!.\mu\mathtt{t}.\{\!|\mathtt{q}!.\mathtt{t}|\!\}^{\mathsf{c}_3} \triangleright \langle \mathtt{q}.\mathtt{end}?i\rangle; \mathtt{q}?|\!\}^{\mathsf{c}_2} \triangleright \langle \mathtt{q}?i\rangle; \mathtt{q}?.\mathtt{end}|\!\}^{\mathsf{c}_1} \triangleright \langle \mathtt{q}?i\rangle; \mathtt{q}?.\mathtt{end}$ after three unfoldings.
  A completed trace can be described as the following: first $\mathtt{p}$ continuously sends messages to $\mathtt{q}$, the ith such message belongs to the scope c_i (and to all scopes $k \leq i$). At some point, let us say after n messages, $\mathtt{q}$ decides to interrupt a scope. It can interrupt any scope $\leq n$ with a message i. Suppose it interrupts scope $m \leq n$, then $\mathtt{p}$ receives the interruption $\mathtt{q}?i$ and is then bound to receive $n - m$ messages from q, in a descending scope order (first a message from the continuation of scope m, then a message from continuation of scope $m - 1, \ldots$). However, q can further interrupt any remaining scope at any time: in this case, the number of remaining receptions $\mathtt{q}?$ expected by $\mathtt{p}$ decreases. Note that after the first interruption, no message *to* $\mathtt{q}$ can be sent and that the total number of receptions is strictly smaller than n. The number of initial

outputs to q is either equal to n or to $n-1$, depending whether the first interruption was received before or after the sending of the n-th message.
As oo is not in the permutation scheme, q! and q? cannot commute.
It follows that L the language of possible completed traces for T is $\mathtt{q}!^n.\Pi(\mathtt{q}?i.\mathtt{q}?^{k_i})$ with $(\Sigma k_i \leq n+1)$, which is not regular.
We note that occurrences of q! and q? are not bound in L and that all occurrences of q? follow all occurrences of q!. From Lemma 12, we conclude.

- Suppose semantics is fully asynchronous $\mathcal{P}_{\{\mathtt{ii},\mathtt{oi},\mathtt{io},\mathtt{oo}\}}$. Without interruptible scopes, completed traces are full shuffling of actions. Yet, the semantics of the type described above is still $\mathtt{q}!^n.\Pi(\mathtt{q}?i.\mathtt{q}?^{k_i})$, as type permutations cannot be used to receive something from the continuation of an interruptible scope before the scope is terminated – either exceptionally or normally.

◀