HAL open science

Asynchronous Sessions with Input Races

Ilaria Castellani, Mariangiola Dezani-Ciancaglini, Paola Giannini

To cite this version:

Ilaria Castellani, Mariangiola Dezani-Ciancaglini, Paola Giannini. Asynchronous Sessions with Input Races. PLACES 2022 - 13th International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, Marco Carbone, Rumyana Neykova, Apr 2022, Munich (Allemagne), Germany. pp.12-23, 10.4204/EPTCS.356.2 . hal-03940160

HAL Id: hal-03940160

https://inria.hal.science/hal-03940160v1

Submitted on 15 Jan 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Asynchronous Sessions with Input Races

Ilaria Castellani*
INRIA, Université Côte d'Azur, France
ilaria.castellani@inria.fr

Mariangiola Dezani-Ciancaglini
Dipartimento di Informatica, Università di Torino, Italy
dezani@di.unito.it

Paola Giannini†
DiSSTE, Università del Piemonte Orientale, Italy
paola.giannini@uniupo.it

We propose a calculus for asynchronous multiparty sessions where input choices with different senders are allowed in processes. We present a type system that accepts such input races provided they do not hinder lock-freedom.

1 Introduction

The foundational work on multiparty sessions [11] introduced the notion of *global type* for specifying the overall behaviour of multiparty protocols. The criterion for a session implementation to be correct with respect to its specification was formalised via the notion of projection: each process implementing the behaviour of a session participant was required to type-check against the local type obtained by projecting the global type on that participant.

However, the work [11] imposed strong restrictions on the syntax of global types, requiring all initial communications in the branches of a choice to have the same sender and the same receiver, and every third participant[1] to have the same behaviour in all branches. Although these were useful simplifying assumptions in order to achieve multiparty session correctness, they limited the expressiveness of global types, ruling out relevant protocols. For this reason, more permissive choice constructors were investigated in subsequent work [1, 2, 9, 16, 13, 3, 5, 14, 10, 17]. A widely adopted relaxation of the choice operator, originally proposed in [1], allows third participants to behave differently in different branches, provided they are notified of which branch has been chosen. Later proposals [9, 16, 13] accommodate processes with output choices among different receivers, for instance a client choosing among different servers. On the other hand *input races*, namely input choices among different senders, continued to be considered as problematic. As a consequence, common protocols such as a server shared by different clients could not be specified by means of global types.

Recent proposals introduce more flexibility in input choices for processes [3, 5, 14, 10, 17]. The work [10] defines the property of *race-freedom* for sessions as the absence, at any stage of computation, of a branching between communications from different senders towards the same receiver leading to distinct target states. A rather permissive type system is proposed, which is shown to be both sound and complete for a range of liveness properties when restricting attention to race-free sessions. The work [14] also addresses the input race problem, referred to as the "+-problem" there. While the proposed syntax for global and local types is completely free, two well-formedness conditions are imposed on types,

*This research has been supported by the ANR17-CE25-0014-01 CISC project.
†This original research has the financial support of the Università del Piemonte Orientale.
[1]We call "third participant" any participant which is not involved in the first communication of a branch.

M. Carbone and R. Neykova (Eds.): 13th International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software
EPTCS 356, 2022, pp. 12–23, doi:10.4204/EPTCS.356.2

© I. Castellani, M. Dezani-Ciancaglini and P. Giannini
This work is licensed under the
Creative Commons Attribution License.

which are meant to prevent dangerous races. Sessions are synchronous in [10, 14] and asynchronous in [17], which is the work that is closest to ours. In that paper, input races are allowed under sophisticated conditions on projections of global types. These conditions track causalities between messages, and their soundness proof uses novel graph-theoretic techniques from the theory of message-sequence charts.

Consider for example the following session, where two participants p and q send concurrently a message to a third participant r, which is ready to receive both messages in any order:

Example 1.1 (Confluent input race) $\mathsf{p}[\![\mathsf{r}!\ell]\!] \parallel \mathsf{q}[\![\mathsf{r}!\ell']\!] \parallel \mathsf{r}[\![\mathsf{p}?\ell;\mathsf{q}?\ell' + \mathsf{q}?\ell';\mathsf{p}?\ell]\!]$

No matter whether communication is synchronous or asynchronous, this session incurs a race[2]. However, this race may be viewed as innocuous since after any branch is chosen, the input of the other branch is still available, leading to the same target state. Indeed, the race consists here of a choice between two different sequentialisations of concurrent inputs[3]. This kind of input race will be called *confluent*. The above session is well typed in [14], but not in [10, 17], where the syntax of global types forbids different senders in a choice. In [10], this session is also ruled out by the race-freedom condition.

By contrast, the following asynchronous session, where there is an apparent input race in the process of participant r, is actually race-free according to [10] because it cannot evolve to a state in which both inputs of r are simultaneously enabled. This kind of uneffective input race, which results from an agreement between the senders such that in every computation only one of them sends a message to the receiver, will be called *fake*.

Example 1.2 (Fake input race in asynchronous session)

$$\mathsf{p}[\![\mathsf{q}!a;\mathsf{q}?a;(\mathsf{q}!\ell;\mathsf{r}!b \oplus \mathsf{q}!\ell')]\!] \parallel \mathsf{q}[\![\mathsf{p}!a;\mathsf{p}?a;(\mathsf{p}?\ell + \mathsf{p}?\ell';\mathsf{r}!b)]\!] \parallel \mathsf{r}[\![\mathsf{p}?b + \mathsf{q}?b]\!]$$

This session implements the following protocol between Alice, Bob and Carol, represented by participants p, q *and* r *respectively:*

- *Alice and Bob send each other the message "I arrived" and then they read their messages;*
- *Alice sends Bob either the message "I will tell Carol", after which she sends Carol the message "We arrived", or the message "Please tell Carol";*
- *Bob reads either the message "I will tell Carol", or the message "Please tell Carol" after which he sends Carol the message "We arrived";*
- *Carol reads the message "We arrived" with sender either Alice or Bob.*

The session of Example 1.2 cannot be typed in [14, 10, 17] because the syntax of global types does not allow two participants to exchange messages by first performing both outputs and then both inputs. If we omit the initial exchange of messages between Alice and Bob, the resulting session can be typed in [10, 17] but not in [14].

Our goal is to devise a type system for asynchronous sessions that is permissive enough to accept the sessions of Example 1.1 and Example 1.2, while rejecting dangerous races that could lead to deadlock or starvation. In particular, we will not be able to type the sessions discussed in the introductions of [14, 17], since they both have a possibility of starvation.

For typing asynchronous sessions we use global types that split communications into outputs and inputs, following the approach advocated in [4, 7]. For instance, the session of Example 1.2 has the following global type: $\mathsf{p}!\mathsf{q}.a;\mathsf{q}!\mathsf{p}.a;\mathsf{p}?\mathsf{q}.a;\mathsf{q}?\mathsf{p}.a;\mathsf{p}!\{\mathsf{q}.\ell;\mathsf{G}_1\,,\,\mathsf{q}.\ell';\mathsf{G}_2\}$ where $\mathsf{G}_1 = \mathsf{p}!\mathsf{r}.b;\mathsf{q}?\mathsf{p}.\ell;\mathsf{r}?\{\mathsf{p}.b\,,\,\mathsf{q}.b\}$ and $\mathsf{G}_2 = \mathsf{q}?\mathsf{p}.\ell';\mathsf{q}!\mathsf{r}.b;\mathsf{r}?\{\mathsf{p}.b\,,\,\mathsf{q}.b\}$. Here $\mathsf{p}!\mathsf{q}.a$ denotes a send from p to q of label a, $\mathsf{p}?\mathsf{q}.a$ denotes a receive by p from q of label a, $\mathsf{p}!\{\mathsf{q}.\ell;\mathsf{G}_1\,,\,\mathsf{q}.\ell';\mathsf{G}_2\}$ is an output choice with sender p and receiver q, and $\mathsf{r}?\{\mathsf{p}.b\,,\,\mathsf{q}.b\}$ is an input choice with receiver r and senders p and q.

[2] Either initially, if communication is synchronous, or after performing both outputs, if communication is asynchronous.

[3] If we had a parallel construct | for processes, this situation would be represented as $(\mathsf{p}?\ell \mid \mathsf{q}?\ell')$.

The rest of the paper is organised as follows. In Section 2 we introduce our calculus for asynchronous sessions. In Section 3 we define the syntax and semantics of global types. In Section 4 we present our type system, illustrate it with some examples, and establish the properties of Subject Reduction, Session Fidelity and Lock-freedom. We conclude in Section 5 with a discussion on future and related work.

2 Asynchronous Sessions

We assume the following base sets: *participants*, ranged over by $\mathsf{p}, \mathsf{q}, \mathsf{r}$ and forming the set Part, and *labels*, ranged over by $\ell, \ell', \ldots$ and forming the set Lab.

Definition 2.1 (Processes) Processes *are defined by:*

$$P \quad ::=_\rho \quad \mathbf{0} \mid \bigoplus_{i \in I} \mathsf{p}_i!\ell_i;P_i \mid \Sigma_{i \in I} \mathsf{p}_i?\ell_i;P_i$$

where $I \neq \emptyset$ *and* $\mathsf{p}_h!\ell_h \neq \mathsf{p}_k!\ell_k$ *and* $\mathsf{p}_h?\ell_h \neq \mathsf{p}_k?\ell_k$ *for* $h,k \in I$ *and* $h \neq k$.

A process may be terminated, or it is an internal choice of outputs or an external choice of inputs. The symbol $::=_\rho$, in Definition 2.1 and in later definitions, indicates that the productions should be interpreted *coinductively* (they define possibly infinite processes) and that we focus on *regular* terms, namely, terms with finitely many distinct subterms. In this way, we only obtain processes which are solutions of finite sets of equations, see [6]. So, when writing processes, we shall use (mutually) recursive equations.

In the following, we will omit trailing $\mathbf{0}$'s when writing processes.

In a full-fledged calculus, processes would exchange labels of the form $\ell(v)$, where v is a value. For simplicity, we consider only pure labels here.

In our calculus, asynchronous communication is handled in the standard way, by storing sent labels in a queue together with sender and receiver names. Receivers may then fetch messages from the queue when required. We define *messages* to be triples $\langle \mathsf{p}, \ell, \mathsf{q} \rangle$, where p is the sender and q is the receiver, and *message queues* (or simply *queues*) to be possibly empty sequences of messages:

$$\mathscr{M} ::= \emptyset \mid \langle \mathsf{p}, \ell, \mathsf{q} \rangle \cdot \mathscr{M}$$

The order of messages in the queue is the order in which they will be read. Since the only reading order that matters is that between messages with the same sender and the same receiver, we consider message queues modulo the structural equivalence given by:

$$\mathscr{M} \cdot \langle \mathsf{p}, \ell, \mathsf{q} \rangle \cdot \langle \mathsf{r}, \ell', \mathsf{s} \rangle \cdot \mathscr{M}' \equiv \mathscr{M} \cdot \langle \mathsf{r}, \ell', \mathsf{s} \rangle \cdot \langle \mathsf{p}, \ell, \mathsf{q} \rangle \cdot \mathscr{M}' \ \text{ if } \ \mathsf{p} \neq \mathsf{r} \ \text{ or } \ \mathsf{q} \neq \mathsf{s}$$

Sessions are composed by a number of located processes of the form $\mathsf{p}[\![P]\!]$, each enclosed within a different participant p, and by a message queue.

Definition 2.2 (Networks and Sessions) Networks *are defined by:*

$$\mathbb{N} = \mathsf{p}_1[\![P_1]\!] \parallel \cdots \parallel \mathsf{p}_n[\![P_n]\!] \textit{ with } \mathsf{p}_h \neq \mathsf{p}_k \textit{ for any } h \neq k$$

Sessions *are defined by:*

$$\mathbb{N} \parallel \mathscr{M}$$

where $\mathbb{N}$ *is a network, and* $\mathscr{M}$ *is a message queue.*

We assume the standard structural congruence $\equiv$ on networks[4], stating that parallel composition is associative and commutative and has neutral element $\mathsf{p}[\![\mathbf{0}]\!]$ for any fresh p.

If $P \neq \mathbf{0}$ we write $\mathsf{p}[\![P]\!] \in \mathbb{N}$ as short for $\mathbb{N} \equiv \mathsf{p}[\![P]\!] \parallel \mathbb{N}'$ for some $\mathbb{N}'$. This abbreviation is justified by the associativity and commutativity of parallel composition.

To define the *operational semantics* of sessions, we use an LTS whose transitions are decorated by outputs or inputs. Therefore, we define the set of *input/output communications* (communications for

[4]By abuse of notation, we use the same symbol as for structural equivalence on queues.

$$\mathsf{p}[\![\bigoplus_{i\in I}\mathsf{q}_i!\ell_i;P_i]\!] \parallel \mathbb{N} \parallel \mathscr{M} \xrightarrow{\mathsf{p}!\mathsf{q}_k.\ell_k} \mathsf{p}[\![P_k]\!] \parallel \mathbb{N} \parallel \mathscr{M}\cdot\langle \mathsf{p},\ell_k,\mathsf{q}_k\rangle \quad \text{where } k\in I \qquad [\textsc{Send}]$$

$$\mathsf{q}[\![\Sigma_{j\in J}\mathsf{p}_j?\ell_j;Q_j]\!] \parallel \mathbb{N} \parallel \langle \mathsf{p}_k,\ell_k,\mathsf{q}\rangle\cdot\mathscr{M} \xrightarrow{\mathsf{q}?\mathsf{p}_k.\ell_k} \mathsf{q}[\![Q_k]\!] \parallel \mathbb{N} \parallel \mathscr{M} \quad \text{where } k\in J \qquad [\textsc{Rcv}]$$

Figure 1: LTS for sessions.

short), ranged over by β, β', to be $\{\mathsf{p}!\mathsf{q}.\ell, \mathsf{p}?\mathsf{q}.\ell \mid \mathsf{p},\mathsf{q}\in\mathsf{Part}, \ell\in\mathsf{Lab}\}$, where $\mathsf{p}!\mathsf{q}.\ell$ represents the output of the label ℓ from participant p to participant q, and $\mathsf{p}?\mathsf{q}.\ell$ the input by participant p of the label ℓ sent by participant q.

The LTS semantics of networks, defined modulo $\equiv$, is specified by the two Rules [SEND] and [RCV] given in Figure 1. Rule [SEND] allows a participant p with an internal choice (a sender) to send to the participant q_k the label ℓ_k by adding it to the queue. Symmetrically, Rule [RCV] allows a participant q with an external choice (a receiver) to read the label ℓ_k sent by participant p_k, provided this label is among the ℓ_j's specified in the choice. Thanks to structural equivalence, the first message from p_k to q that appears in the queue, if any, can always be moved to the top of the queue.

A key role in this paper is played by (possibly empty) sequences of communications. As usual we define them as traces.

Definition 2.3 (Traces) *(Finite) traces are defined by:*

$$\tau ::= \varepsilon \mid \beta\cdot\tau$$

We use $|\tau|$ *to denote the length of the trace* τ.

When $\tau=\beta_1\cdot\ldots\cdot\beta_n$ $(n\geq 1)$ we write $\mathbb{N}\parallel\mathscr{M}\xrightarrow{\tau}\mathbb{N}'\parallel\mathscr{M}'$ as short for

$$\mathbb{N}\parallel\mathscr{M}\xrightarrow{\beta_1}\mathbb{N}_1\parallel\mathscr{M}_1\cdots\xrightarrow{\beta_n}\mathbb{N}_n\parallel\mathscr{M}_n=\mathbb{N}'\parallel\mathscr{M}'$$

We now introduce the notion of player, which is characteristic of asynchronous communication, where only one of the involved participants is active, namely the sender for an output communication and the receiver for an input communication. The player of a communication β is the participant who is active in β. The set of players of a trace is then obtained by collecting the players of all its communications.

Definition 2.4 (Players of communications and traces) *We denote by* $\mathsf{play}(\beta)$ *the* player of a communication β *defined by*

$$\mathsf{play}(\mathsf{p}!\mathsf{q}.\ell)=\mathsf{play}(\mathsf{p}?\mathsf{q}.\ell)=\mathsf{p}$$

We denote by $\mathsf{Players}(\tau)$ *the* set of players of a trace τ *defined by*

$$\mathsf{Players}(\varepsilon)=\emptyset \qquad \mathsf{Players}(\beta\cdot\tau)=\{\mathsf{play}(\beta)\}\cup\mathsf{Players}(\tau)$$

3 Global Types

As in [4, 7], our global types can be obtained from the standard ones [11, 12] by splitting output and input communications. The novelty is that we allow multiple receivers in output choices and multiple senders in input choices.

Definition 3.1 (Global types) Global types G *are defined by the following grammar:*

$$\mathsf{G} \quad ::=_\rho \quad \mathsf{p}!\{\mathsf{q}_i.\ell_i;\mathsf{G}_i\}_{i\in I} \mid \mathsf{p}?\{\mathsf{q}_i.\ell_i;\mathsf{G}_i\}_{i\in I} \mid \mathsf{End}$$

where $I\neq\emptyset$, $\mathsf{p}\neq\mathsf{q}_i$ *for all* $i\in I$ *and* $\mathsf{q}_h.\ell_h\neq\mathsf{q}_k.\ell_k$ *for* $h,k\in I$ *and* $h\neq k$.

As for processes, $::=_\rho$ indicates that global types are coinductively defined and *regular*.

The global type $\mathsf{p}!\{\mathsf{q}_i.\ell_i;\mathsf{G}_i\}_{i\in I}$ specifies that player p sends the label ℓ_k with $k \in I$ to participant q_k and then the interaction described by the global type G_k takes place. The global type $\mathsf{p}?\{\mathsf{q}_i.\ell_i;\mathsf{G}_i\}_{i\in I}$ specifies that player p receives the label ℓ_k with $k \in I$ from participant q_k and then the interaction described by the global type G_k takes place.

We define $\mathsf{Players}(\mathsf{G})$ as the smallest set satisfying the following equations:

$\mathsf{Players}(\mathsf{End}) = \emptyset \quad \mathsf{Players}(\mathsf{p}!\{\mathsf{q_i}.\ell_i;\mathsf{G_i}\}_{\mathsf{i}\in\mathsf{I}}) = \mathsf{Players}(\mathsf{p}?\{\mathsf{q_i}.\ell_i;\mathsf{G_i}\}_{\mathsf{i}\in\mathsf{I}}) = \{\mathsf{p}\} \cup \bigcup_{\mathsf{i}\in\mathsf{I}} \mathsf{Players}(\mathsf{G_i})$

The regularity of global types ensures that the sets of players are finite. In Section 2 we used the same notation for the players of traces. In all cases, the context should make it easy to understand which function is in use.

To avoid starvation we require global types to satisfy a boundedness condition. To formalise boundedness we use ξ to denote a *path* in global type trees, i.e., a possibly infinite sequence of communications β. Note that a finite path is a trace in the sense of Definition 2.3. We extend the notation $\cdot$ to denote also the concatenation of a finite sequence with a possibly infinite sequence. The function Paths gives the set of *paths* of a global type, which is the greatest set such that:

$$\begin{array}{ll}
\mathsf{Paths}(\mathsf{End}) & = \{\varepsilon\} \\
\mathsf{Paths}(\mathsf{p}!\{\mathsf{q}_i.\ell_i;\mathsf{G}_i\}_{i\in I}) & = \bigcup_{i\in I}\{\mathsf{p}!\mathsf{q}.\ell_i \cdot \xi \mid \xi \in \mathsf{Paths}(\mathsf{G}_i)\} \\
\mathsf{Paths}(\mathsf{p}?\{\mathsf{q}_i.\ell_i;\mathsf{G}_i\}_{i\in I}) & = \bigcup_{i\in I}\{\mathsf{p}?\mathsf{q}.\ell_i \cdot \xi \mid \xi \in \mathsf{Paths}(\mathsf{G}_i)\}
\end{array}$$

If $x \in \mathbf{N} \cup \{\omega\}$ is the length of ξ, we denote by $\xi[n]$ the n-th communication in the path ξ, where $1 \leq n < x$. It is handy to define the *depth* of a player p in a global type G, $\mathsf{depth}(\mathsf{G},\mathsf{p})$.

Definition 3.2 (Depth of a player) *Let* G *be a global type. For* $\xi \in \mathsf{Paths}(\mathsf{G})$ *set*

$$\mathsf{depth}(\xi,\mathsf{p}) = \inf\{n \mid \mathsf{play}(\xi[n]) = \mathsf{p}\}$$

and define $\mathsf{depth}(\mathsf{G},\mathsf{p})$*, the* depth *of* p *in* G*, as follows:*

$$\mathsf{depth}(\mathsf{G},\mathsf{p}) = \begin{cases} \sup\{\mathsf{depth}(\xi,\mathsf{p}) \mid \xi \in \mathsf{Paths}(\mathsf{G})\} & \mathsf{p} \in \mathsf{Players}(\mathsf{G}) \\ 0 & \textit{otherwise} \end{cases}$$

Note that $\mathsf{depth}(\mathsf{G},\mathsf{p}) = 0$ iff $\mathsf{p} \notin \mathsf{Players}(\mathsf{G})$. Moreover, if $\mathsf{p} \neq \mathsf{play}(\xi[n])$ for all $n \in \mathbf{N}$, then $\mathsf{depth}(\xi,\mathsf{p}) = \inf \emptyset = \infty$. Hence, if p is a player of a global type G and there is some path in G where p does not occur as a player, then $\mathsf{depth}(\mathsf{G},\mathsf{p}) = \infty$.

Definition 3.3 (Boundedness) *A global type* G *is* bounded *if* $\mathsf{depth}(\mathsf{G}',\mathsf{p})$ *is finite for all participants* $\mathsf{p} \in \mathsf{Players}(\mathsf{G})$ *and all types* G' *which occur in* G.

Example 3.4 *The following example shows the necessity of considering all types occurring in a global type for defining boundedness. Consider* $\mathsf{G} = \mathsf{r}!\mathsf{q}.\ell;\mathsf{q}?\mathsf{r}.\ell;\mathsf{G}'$*, where*

$$\mathsf{G}' = \mathsf{p}!\{\mathsf{q}.\ell_1;\mathsf{q}?\mathsf{p}.\ell_1;\mathsf{q}!\mathsf{r}.\ell_3;\mathsf{r}?\mathsf{q}.\ell_3\,,\,\mathsf{q}.\ell_2;\mathsf{q}?\mathsf{p}.\ell_2;\mathsf{G}'\}$$

Then we have: $\mathsf{depth}(\mathsf{G},\mathsf{p}) = 3, \mathsf{depth}(\mathsf{G},\mathsf{q}) = 2, \mathsf{depth}(\mathsf{G},\mathsf{r}) = 1$*, whereas* $\mathsf{depth}(\mathsf{G}',\mathsf{p}) = 1, \mathsf{depth}(\mathsf{G}',\mathsf{q}) = 2, \mathsf{depth}(\mathsf{G}',\mathsf{r}) = \infty$.

Since global types are regular the boundedness condition is decidable.

Global types in parallel with queues, dubbed *type configurations*, are given semantics by means of the LTS in Figure 2. The first two rules allow top level outputs and inputs to be performed in the standard way. The remaining two rules allow communications to be performed inside output and input choices. These inside rules are needed to enable interleaving between independent communications despite the sequential structure of global types. For example, we want to allow $\mathsf{p}!\mathsf{q}.\ell;\mathsf{r}!\mathsf{s}.\ell' \parallel \emptyset \xrightarrow{\mathsf{r}!\mathsf{s}.\ell'} \mathsf{p}!\mathsf{q}.\ell \parallel \langle \mathsf{r},\ell',\mathsf{s}\rangle$ when $\mathsf{p} \neq \mathsf{r}$, because, intuitively, outputs performed by different players should be independent. This

$$[\textsc{Top-Out}]\ \frac{}{\mathsf{p}!\{\mathsf{q}_i.\ell_i;\mathsf{G}_i\}_{i\in I} \parallel \mathscr{M} \xrightarrow{\mathsf{p}!\mathsf{q}_h.\ell_h} \mathsf{G}_h \parallel \mathscr{M}\cdot\langle \mathsf{p},\ell_h,\mathsf{q}_h\rangle}\ h\in I$$

$$[\textsc{Top-In}]\ \frac{}{\mathsf{p}?\{\mathsf{q}_i.\ell_i;\mathsf{G}_i\}_{i\in I} \parallel \langle \mathsf{q}_h,\ell_h,\mathsf{p}\rangle\cdot\mathscr{M} \xrightarrow{\mathsf{q}_h?\mathsf{p}.\ell_h} \mathsf{G}_h \parallel \mathscr{M}}\ h\in I$$

$$[\textsc{Inside-Out}]\ \frac{\mathsf{G}_i \parallel \mathscr{M}\cdot\langle \mathsf{p},\ell_i,\mathsf{q}_i\rangle \xrightarrow{\beta} \mathsf{G}'_i \parallel \mathscr{M}'\cdot\langle \mathsf{p},\ell_i,\mathsf{q}_i\rangle \quad \forall i\in I}{\mathsf{p}!\{\mathsf{q}_i.\ell_i;\mathsf{G}_i\}_{i\in I} \parallel \mathscr{M} \xrightarrow{\beta} \mathsf{p}!\{\mathsf{q}_i.\ell_i;\mathsf{G}'_i\}_{i\in I} \parallel \mathscr{M}'}\ \mathsf{p}\neq\mathsf{play}(\beta)$$

$$[\textsc{Inside-In}]\ \frac{\mathsf{G}_j \parallel \mathscr{M} \xrightarrow{\beta} \mathsf{G}'_j \parallel \mathscr{M}' \quad \forall j\in J}{\mathsf{p}?\{\mathsf{q}_i.\ell_i;\mathsf{G}_i\}_{i\in I} \parallel \mathscr{M} \xrightarrow{\beta} \mathsf{p}?\{\mathsf{q}_i.\ell_i;\mathsf{G}'_i\}_{i\in I} \parallel \mathscr{M}'}\ \begin{array}{c} J=\mathsf{rm}(\{\langle \mathsf{q}_i,\ell_i,\mathsf{p}\rangle\}_{i\in I},\mathscr{M})\neq\emptyset \\ \mathsf{p}\neq\mathsf{play}(\beta) \quad \beta\neq \mathsf{q}_l!\mathsf{p}.\ell_l \\ \mathsf{G}'_l=\mathsf{G}_k \ \ k\in J \ \ \forall l\in I\backslash J \end{array}$$

Figure 2: LTS for type configurations.

justifies the condition $\mathsf{p}\neq\mathsf{play}(\beta)$ in Rules [INSIDE-OUT] and [INSIDE-IN]. In Rule [INSIDE-OUT] we require all branches to be able to perform the β transition. This avoids for example:

$$\begin{array}{c}\mathsf{p}!\{\mathsf{q}.\ell;\mathsf{q}?\mathsf{p}.\ell;\mathsf{r}!\mathsf{p}.\ell;\mathsf{p}?\mathsf{r}.\ell\,,\,\mathsf{q}.\ell';\mathsf{q}?\mathsf{p}.\ell';\mathsf{r}!\mathsf{p}.\ell';\mathsf{p}?\mathsf{r}.\ell'\} \parallel \emptyset \xrightarrow{\mathsf{r}!\mathsf{p}.\ell} \\ \mathsf{p}!\{\mathsf{q}.\ell;\mathsf{q}?\mathsf{p}.\ell;\mathsf{p}?\mathsf{r}.\ell\,,\,\mathsf{q}.\ell';\mathsf{q}?\mathsf{p}.\ell';\mathsf{r}!\mathsf{p}.\ell';\mathsf{p}?\mathsf{r}.\ell'\} \parallel \langle \mathsf{r},\ell,\mathsf{p}\rangle\end{array}$$

which, in case we choose the right branch, leads to the configuration $\mathsf{p}?\mathsf{r}.\ell' \parallel \langle \mathsf{r},\ell,\mathsf{p}\rangle\cdot\langle \mathsf{r},\ell',\mathsf{p}\rangle$.

The shapes of the queues appearing in the premise of Rule [INSIDE-OUT] ensure that β is not the matching input for any output in the choice. In Rule [INSIDE-IN], we consider only the branches with corresponding messages on top of the queue (called *live* branches), using the index set of ready messages $\mathsf{rm}(\{\langle \mathsf{q}_i,\ell_i,\mathsf{p}\rangle\}_{i\in I},\mathscr{M})$ defined as follows, where m ranges over messages.

Definition 3.5 *Given a set of messages $\{\mathsf{m}_i\}_{i\in I}$ and a queue $\mathscr{M}$, the index set of the "ready messages" in this set is defined by:* $\mathsf{rm}(\{\mathsf{m}_i\}_{i\in I},\mathscr{M}) = \{i\in I \mid \mathscr{M}\equiv \mathsf{m}_i\cdot\mathscr{M}_i\}$.

The mapping rm plays a crucial role also in the typing rule for input choices, as we will see in Section 4. The condition $J\neq\emptyset$ means that there is at least one live branch. The condition $\beta\neq \mathsf{q}_l!\mathsf{p}.\ell_l$ for all $l\in I\backslash J$ ensures that the occurrence of β does not generate a message that would "awaken" some dead, i.e. not live, branch of the choice. In the resulting choice, the dead branches become an arbitrary live branch (condition $\mathsf{G}'_l=\mathsf{G}_k$ for some $k\in J$ and for all $l\in I\backslash J$). In fact, such branches could also be omitted as they can never be awaken.

It is easy to check that the LTS of type configurations preserves boundedness of global types. Therefore, from now on we will assume all our global types to be bounded.

4 Type System

Global types are an abstraction of sessions. Usually, global types are projected to participants, yielding local types which are assigned to processes. The simplicity of our calculus and the flexibility of our global types allow us to formulate a type system where global types are directly derived for sessions, using judgements of the form $\mathsf{G}\vdash\mathbb{N}\parallel\mathscr{M}$. The typing rules are given in Figure 3.

Rules [OUT] and [IN] just add simultaneously outputs and inputs to global types and to the corresponding processes inside networks. The condition $\mathsf{Players}(\mathsf{G}_i)\setminus\{\mathsf{p}\}=\mathsf{Players}(\mathbb{N})$ for all $i\in I$ ensures that all players in $\mathbb{N}$ are also players in G. For example, this condition prevents the derivation of

$$[\text{END}]\ \frac{}{\mathsf{End} \vdash \mathsf{p}[\![\mathbf{0}]\!] \parallel \emptyset}$$

$$[\text{OUT}]\ \frac{\mathsf{G}_i \vdash \mathsf{p}[\![P_i]\!] \parallel \mathbb{N} \parallel \mathscr{M} \cdot \langle \mathsf{p}, \ell_i, \mathsf{q}_i \rangle \qquad \mathsf{Players}(\mathsf{G}_i) \setminus \{\mathsf{p}\} = \mathsf{Players}(\mathbb{N}) \ \ \forall i \in I}{\mathsf{p}!\{\mathsf{q}_i.\ell_i; \mathsf{G}_i\}_{i \in I} \vdash \mathsf{p}[\![\bigoplus_{i \in I} \mathsf{q}_i!\ell_i; P_i]\!] \parallel \mathbb{N} \parallel \mathscr{M}}$$

$$[\text{IN}]\ \frac{\begin{array}{c}\mathsf{G}_j \vdash \mathsf{p}[\![P_j]\!] \parallel \mathbb{N} \parallel \mathscr{M}_j \quad \boldsymbol{l}(\mathsf{G}_j, \mathbb{M}) \quad \forall j \in J \\ \mathsf{Players}(\mathsf{G}_i) \setminus \{\mathsf{p}\} = \mathsf{Players}(\mathbb{N}) \quad \forall i \in I\end{array}}{\mathsf{p}?\{\mathsf{q}_i.\ell_i; \mathsf{G}_i\}_{i \in I} \vdash \mathsf{p}[\![\Sigma_{h \in H} \mathsf{q}_h?\ell_h; P_h]\!] \parallel \mathbb{N} \parallel \mathscr{M}} \quad \begin{array}{l} J = \mathsf{rm}(\{\langle \mathsf{q}_i, \ell_i, \mathsf{p} \rangle\}_{i \in I}, \mathscr{M}) \\ \quad = \mathsf{rm}(\{\langle \mathsf{q}_h, \ell_h, \mathsf{p} \rangle\}_{h \in H}, \mathscr{M}) \neq \emptyset \\ \mathscr{M} \equiv \langle \mathsf{q}_j, \ell_j, \mathsf{p} \rangle \cdot \mathscr{M}_j \ \forall j \in J \\ \mathbb{M} = \{\langle \mathsf{q}_l, \ell_l, \mathsf{p} \rangle \mid l \in (I \cup H) \setminus J \ \&\ \mathsf{q}_l \neq \mathsf{q}_j \ \forall j \in J\} \end{array}$$

Figure 3: Typing rules for sessions.

$\mathsf{G} \vdash \mathsf{p}[\![P]\!] \parallel \mathsf{q}[\![Q]\!]$ with $\mathsf{G} = \mathsf{p}!\mathsf{q}.\ell; \mathsf{G}$ and $P = \mathsf{q}!\ell; P$ and Q arbitrary.

Rule [OUT] considers all branches of the global type, since the choice of the sent message is arbitrary. This rule requires that the session resulting from the output of a branch be typed with the corresponding branch of the global type.

Rule [IN] requires that the global type and the process read the same messages on the queue. To this end, it uses the index set of ready messages defined in Definition 3.5, collecting the indices of the live branches of the global type and of the input process[5], and asking them to be equal (condition $\mathsf{rm}(\{\langle \mathsf{q}_i, \ell_i, \mathsf{p} \rangle\}_{i \in I}, \mathscr{M}) = \mathsf{rm}(\{\langle \mathsf{q}_h, \ell_h, \mathsf{p} \rangle\}_{h \in H}, \mathscr{M})$. This set of indices must not be empty (condition $J \neq \emptyset$). Only the branches of the global type and of the input process thus selected are compared in the premises of Rule [IN]. Note that in this way we allow more freedom than in the synchronous subtyping for session types [8]. In Rule [IN], in order to ensure the condition $\beta \neq \mathsf{q}_l!\mathsf{p}.\ell_l$ for all $l \in I \setminus J$ required by the transition Rule [INSIDE-IN], we want to prevent the enqueuing of messages that would transform a dead branch of the process or of the global type into a live branch. To this end, we introduce a predicate which forbids a global type to generate such messages. Let $\mathbb{M}$ range over sets of messages.

Definition 4.1 *The type* G *is* inactive *for the set of messages* $\mathbb{M}$*, if* $\boldsymbol{l}(\mathsf{G}, \mathbb{M})$ *holds, where:*

$$\begin{array}{ll} \boldsymbol{l}(\mathsf{End}, \mathbb{M}) & \boldsymbol{l}(\mathsf{p}?\{\mathsf{q}_i.\ell_i; \mathsf{G}_i\}_{i \in I}, \mathbb{M}) \quad \textit{if } \boldsymbol{l}(\mathsf{G}_i, \mathbb{M}) \ \ \forall i \in I \\ \boldsymbol{l}(\mathsf{p}!\{\mathsf{q}_i.\ell_i; \mathsf{G}_i\}_{i \in I}, \mathbb{M}) & \textit{if } \langle \mathsf{p}, \ell_i, \mathsf{q}_i \rangle \notin \mathbb{M} \textit{ and } \boldsymbol{l}(\mathsf{G}_i, \mathbb{M}) \ \ \forall i \in I \end{array}$$

The predicate $\boldsymbol{l}(\mathsf{G}, \mathbb{M})$ looks for outputs in G which produce messages in $\mathbb{M}$. The regularity of global types guarantees the computability of this predicate. Notice that $\boldsymbol{l}(\mathsf{G}, \mathbb{M})$ also ensures that the network cannot produce messages in $\mathbb{M}$. This is due to the typing Rule [OUT] prescribing that messages put on the queue by the global type be the same as the ones of the network.

For example consider the following sequence of transitions

$$\begin{array}{lcl} \mathsf{q}[\![\mathsf{r}!\ell']\!] \parallel \mathsf{r}[\![\mathsf{p}?\ell; \mathsf{q}?\ell' + \mathsf{q}?\ell'; \mathsf{p}?\ell']\!] \parallel \langle \mathsf{p}, \ell, \mathsf{r} \rangle & \xrightarrow{\mathsf{q}!\mathsf{r}.\ell'} & \mathsf{r}[\![\mathsf{p}?\ell; \mathsf{q}?\ell' + \mathsf{q}?\ell'; \mathsf{p}?\ell']\!] \parallel \langle \mathsf{p}, \ell, \mathsf{r} \rangle \cdot \langle \mathsf{q}, \ell', \mathsf{r} \rangle \\ & \xrightarrow{\mathsf{r}?\mathsf{q}.\ell'} & \mathsf{r}[\![\mathsf{p}?\ell']\!] \parallel \langle \mathsf{p}, \ell, \mathsf{r} \rangle \end{array}$$

Since the input and the message in $\mathsf{r}[\![\mathsf{p}?\ell']\!] \parallel \langle \mathsf{p}, \ell, \mathsf{r} \rangle$ do not match, this session cannot be typed and therefore also the session $\mathsf{q}[\![\mathsf{r}!\ell']\!] \parallel \mathsf{r}[\![\mathsf{p}?\ell; \mathsf{q}?\ell' + \mathsf{q}?\ell'; \mathsf{p}?\ell']\!] \parallel \langle \mathsf{p}, \ell, \mathsf{r} \rangle$ should not be typable. Without checking the inactivity predicate we can type this session by the global type

$$(*) \quad \mathsf{r}?\{\mathsf{p}.\ell; \mathsf{q}!\mathsf{r}.\ell'; \mathsf{r}?\mathsf{q}.\ell', \mathsf{q}.\ell'; \mathsf{q}!\mathsf{r}.\ell'; \mathsf{r}?\mathsf{p}.\ell'\}$$

[5] As for global types, a branch of the input process is live if it has a corresponding message on top of the queue, and dead otherwise.

as follows:

$$\dfrac{\dfrac{\dfrac{\mathsf{End} \vdash \mathsf{r}[\![\mathbf{0}]\!] \parallel \emptyset}{\mathsf{r}?\mathsf{q}.\ell' \vdash \mathsf{r}[\![\mathsf{q}?\ell']\!] \parallel \langle \mathsf{q},\ell',\mathsf{r}\rangle}}{\mathsf{q}!\mathsf{r}.\ell';\mathsf{r}?\mathsf{q}.\ell' \vdash \mathsf{q}[\![\mathsf{r}!\ell']\!] \parallel \mathsf{r}[\![\mathsf{q}?\ell']\!] \parallel \emptyset}}{\mathsf{r}?\{\mathsf{p}.\ell;\mathsf{q}!\mathsf{r}.\ell';\mathsf{r}?\mathsf{q}.\ell',\mathsf{q}.\ell';\mathsf{q}!\mathsf{r}.\ell';\mathsf{r}?\mathsf{p}.\ell'\} \vdash \mathsf{q}[\![\mathsf{r}!\ell']\!] \parallel \mathsf{r}[\![\mathsf{p}?\ell;\mathsf{q}?\ell'+\mathsf{q}?\ell';\mathsf{p}?\ell']\!] \parallel \langle \mathsf{p},\ell,\mathsf{r}\rangle}$$

The problem here is that Rule [IN] does not check the dead branches of the global type. The role of the inactivity predicate is just to ensure that the transitions will not awake dead branches. This is done by checking the outputs in the live branches. In this example the output $\mathsf{q}!\mathsf{r}.\ell'$ is in the branch starting with the input $\mathsf{r}?\mathsf{p}.\ell$ and the queue contains $\langle \mathsf{p},\ell,\mathsf{r}\rangle$. So the typing Rule [IN] cannot be applied since $\boldsymbol{l}(\mathsf{q}!\mathsf{r}.\ell';\mathsf{r}?\mathsf{q}.\ell',\{\langle \mathsf{q},\ell',\mathsf{r}\rangle\})$ does not hold.

Notice that the session in Example 1.1 has the transition

$$\mathsf{p}[\![\mathsf{r}!\ell]\!] \parallel \mathsf{q}[\![\mathsf{r}!\ell']\!] \parallel \mathsf{r}[\![\mathsf{p}?\ell;\mathsf{q}?\ell'+\mathsf{q}?\ell';\mathsf{p}?\ell]\!] \parallel \emptyset \quad \xrightarrow{\mathsf{p}!\mathsf{r}.\ell} \quad \mathsf{q}[\![\mathsf{r}!\ell']\!] \parallel \mathsf{r}[\![\mathsf{p}?\ell;\mathsf{q}?\ell'+\mathsf{q}?\ell';\mathsf{p}?\ell]\!] \parallel \langle \mathsf{p},\ell,\mathsf{r}\rangle$$

and the resulting session differs from that of the previous example only for the label of the last input. Correspondingly, the global types $\mathsf{r}?\{\mathsf{p}.\ell;\mathsf{q}!\mathsf{r}.\ell';\mathsf{r}?\mathsf{q}.\ell',\mathsf{q}.\ell';\underline{\mathsf{r}?\mathsf{p}.\ell}\}$ and $\mathsf{r}?\{\mathsf{p}.\ell;\mathsf{q}!\mathsf{r}.\ell';\mathsf{r}?\mathsf{q}.\ell',\mathsf{q}.\ell';\underline{\mathsf{r}?\mathsf{p}.\ell'}\}$ only differ for the labels of the underlined inputs. Therefore $\mathsf{r}?\{\mathsf{p}.\ell;\mathsf{q}!\mathsf{r}.\ell';\mathsf{r}?\mathsf{q}.\ell',\mathsf{q}.\ell';\mathsf{r}?\mathsf{p}.\ell\}$ cannot be derived for the session $\mathsf{q}[\![\mathsf{r}!\ell']\!] \parallel \mathsf{r}[\![\mathsf{p}?\ell;\mathsf{q}?\ell'+\mathsf{q}?\ell';\mathsf{p}?\ell]\!] \parallel \langle \mathsf{p},\ell,\mathsf{r}\rangle$, since as we just saw the predicate $\boldsymbol{l}(\mathsf{q}!\mathsf{r}.\ell';\mathsf{r}?\mathsf{q}.\ell',\{\langle \mathsf{q},\ell',\mathsf{r}\rangle\})$ does not hold. In fact, this is expected since the session can do a transition $\xrightarrow{\mathsf{q}!\mathsf{r}.\ell'}$ that the type configuration cannot mimic. On the other hand, this session can be typed by the global type

$$\mathsf{q}!\mathsf{r}.\ell';\mathsf{r}?\{\mathsf{p}.\ell;\mathsf{r}?\mathsf{q}.\ell',\mathsf{q}.\ell';\mathsf{r}?\mathsf{p}.\ell\}$$

as follows:

$$\dfrac{\dfrac{\dfrac{\mathsf{End} \vdash \mathsf{r}[\![\mathbf{0}]\!] \parallel \emptyset}{\mathsf{r}?\mathsf{q}.\ell' \vdash \mathsf{r}[\![\mathsf{q}?\ell']\!] \parallel \langle \mathsf{q},\ell',\mathsf{r}\rangle} \qquad \dfrac{\mathsf{End} \vdash \mathsf{r}[\![\mathbf{0}]\!] \parallel \emptyset}{\mathsf{r}?\mathsf{p}.\ell \vdash \mathsf{r}[\![\mathsf{p}?\ell]\!] \parallel \langle \mathsf{p},\ell,\mathsf{r}\rangle}}{\mathsf{r}?\{\mathsf{p}.\ell;\mathsf{r}?\mathsf{q}.\ell',\mathsf{q}.\ell';\mathsf{r}?\mathsf{p}.\ell\} \vdash \mathsf{r}[\![\mathsf{p}?\ell;\mathsf{q}?\ell'+\mathsf{q}?\ell';\mathsf{p}?\ell]\!] \parallel \langle \mathsf{p},\ell,\mathsf{r}\rangle \cdot \langle \mathsf{q},\ell',\mathsf{r}\rangle}}{\mathsf{q}!\mathsf{r}.\ell';\mathsf{r}?\{\mathsf{p}.\ell;\mathsf{r}?\mathsf{q}.\ell',\mathsf{q}.\ell';\mathsf{r}?\mathsf{p}.\ell\} \vdash \mathsf{q}[\![\mathsf{r}!\ell']\!] \parallel \mathsf{r}[\![\mathsf{p}?\ell;\mathsf{q}?\ell'+\mathsf{q}?\ell';\mathsf{p}?\ell]\!] \parallel \langle \mathsf{p},\ell,\mathsf{r}\rangle}$$

In this derivation Rule [OUT] is applied first, and thus Rule [IN] is applied only when the queue contains the matching messages for both branches. Therefore Rule [IN] checks the continuations of both branches, and the inactivity predicate holds trivially for each of them.

Notice that bringing forward the output from q to r in the global type $(*)$ does not enable us to type:

$$\mathsf{q}[\![\mathsf{r}!\ell']\!] \parallel \mathsf{r}[\![\mathsf{p}?\ell;\mathsf{q}?\ell'+\mathsf{q}?\ell';\mathsf{p}?\ell']\!] \parallel \langle \mathsf{p},\ell,\mathsf{r}\rangle$$

In fact, we cannot complete the derivation:

$$\dfrac{\dfrac{\dfrac{\mathsf{End} \vdash \mathsf{r}[\![\mathbf{0}]\!] \parallel \emptyset}{\mathsf{r}?\mathsf{q}.\ell' \vdash \mathsf{r}[\![\mathsf{q}?\ell']\!] \parallel \langle \mathsf{q},\ell',\mathsf{r}\rangle} \qquad \mathsf{r}?\mathsf{p}.\ell' \vdash \mathsf{r}[\![\mathsf{p}?\ell']\!] \parallel \langle \mathsf{p},\ell,\mathsf{r}\rangle}{\mathsf{r}?\{\mathsf{p}.\ell;\mathsf{r}?\mathsf{q}.\ell',\mathsf{q}.\ell';\mathsf{r}?\mathsf{p}.\ell'\} \vdash \mathsf{r}[\![\mathsf{p}?\ell;\mathsf{q}?\ell'+\mathsf{q}?\ell';\mathsf{p}?\ell']\!] \parallel \langle \mathsf{p},\ell,\mathsf{r}\rangle \cdot \langle \mathsf{q},\ell',\mathsf{r}\rangle}}{\mathsf{q}!\mathsf{r}.\ell';\mathsf{r}?\{\mathsf{p}.\ell;\mathsf{r}?\mathsf{q}.\ell',\mathsf{q}.\ell';\mathsf{r}?\mathsf{p}.\ell'\} \vdash \mathsf{q}[\![\mathsf{r}!\ell']\!] \parallel \mathsf{r}[\![\mathsf{p}?\ell;\mathsf{q}?\ell'+\mathsf{q}?\ell';\mathsf{p}?\ell']\!] \parallel \langle \mathsf{p},\ell,\mathsf{r}\rangle}$$

Indeed, we cannot apply Rule [IN] to derive the top right judgement $\mathsf{r}?\mathsf{p}.\ell' \vdash \mathsf{r}[\![\mathsf{p}?\ell']\!] \parallel \langle \mathsf{p},\ell,\mathsf{r}\rangle$, since the only input does not match the message in the queue.

We can also type the following recursive version of Example 1.1:

$$\mathsf{p}[\![P]\!] \parallel \mathsf{q}[\![Q]\!] \parallel \mathsf{r}[\![R]\!]$$

where $P = \mathsf{r}!\ell;P$, $Q = \mathsf{r}!\ell';Q$ and $R = \mathsf{p}?\ell;\mathsf{q}?\ell';R + \mathsf{q}?\ell';\mathsf{p}?\ell;R$. A suitable global type is

$$\mathsf{G} = \mathsf{p}!\mathsf{r}.\ell;\mathsf{q}!\mathsf{r}.\ell';\mathsf{r}?\{\mathsf{p}.\ell;\mathsf{r}?\mathsf{q}.\ell';\mathsf{G},\mathsf{q}.\ell';\mathsf{r}?\mathsf{p}.\ell;\mathsf{G}\}$$

as shown by the following derivation:

$$
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\vdots \\ \hline\hline
\mathsf{G} \vdash \mathsf{p}[\![P]\!] \parallel \mathsf{q}[\![Q]\!] \parallel \mathsf{r}[\![R]\!] \parallel \emptyset
\end{array}
\\ \hline\hline
\mathsf{r}?\mathsf{q}.\ell';\mathsf{G} \vdash \mathsf{p}[\![P]\!] \parallel \mathsf{q}[\![Q]\!] \parallel \mathsf{r}[\![\mathsf{q}?\ell';R]\!] \parallel \langle \mathsf{q},\ell',\mathsf{r}\rangle
\end{array}
\qquad
\begin{array}{c}
\begin{array}{c}
\vdots \\ \hline\hline
\mathsf{G} \vdash \mathsf{p}[\![P]\!] \parallel \mathsf{q}[\![Q]\!] \parallel \mathsf{r}[\![R]\!] \parallel \emptyset
\end{array}
\\ \hline\hline
\mathsf{r}?\mathsf{p}.\ell;\mathsf{G} \vdash \mathsf{p}[\![P]\!] \parallel \mathsf{q}[\![Q]\!] \parallel \mathsf{r}[\![\mathsf{p}?\ell;R]\!] \parallel \langle \mathsf{p},\ell,\mathsf{r}\rangle
\end{array}
\\ \hline\hline\hline
\mathsf{r}?\{\mathsf{p}.\ell;\mathsf{r}?\mathsf{q}.\ell';\mathsf{G}\,,\,\mathsf{q}.\ell';\mathsf{r}?\mathsf{p}.\ell;\mathsf{G}\} \vdash \mathsf{p}[\![P]\!] \parallel \mathsf{q}[\![Q]\!] \parallel \mathsf{r}[\![R]\!] \parallel \langle \mathsf{p},\ell,\mathsf{r}\rangle \cdot \langle \mathsf{q},\ell',\mathsf{r}\rangle
\end{array}
\\ \hline\hline\hline
\mathsf{q}!\mathsf{r}.\ell';\mathsf{r}?\{\mathsf{p}.\ell;\mathsf{r}?\mathsf{q}.\ell';\mathsf{G}\,,\,\mathsf{q}.\ell';\mathsf{r}?\mathsf{p}.\ell;\mathsf{G}\} \vdash \mathsf{p}[\![P]\!] \parallel \mathsf{q}[\![Q]\!] \parallel \mathsf{r}[\![R]\!] \parallel \langle \mathsf{p},\ell,\mathsf{r}\rangle
\\ \hline\hline\hline
\mathsf{G} \vdash \mathsf{p}[\![P]\!] \parallel \mathsf{q}[\![Q]\!] \parallel \mathsf{r}[\![R]\!] \parallel \emptyset
\end{array}
$$

Our type system enjoys the properties of Session Fidelity and Subject Reduction. Moreover, it ensures the semantic property of Lock-freedom. Since every participant can freely perform outputs, to prove this property we only have to show that all inputs can be enabled. For lack of space we only give the most interesting case in the proof of Subject Reduction.

Theorem 4.2 (Session Fidelity) *If* $\mathsf{G} \vdash \mathbb{N} \parallel \mathscr{M}$ *and* $\mathsf{G} \parallel \mathscr{M} \xrightarrow{\beta} \mathsf{G}' \parallel \mathscr{M}'$*, then* $\mathbb{N} \parallel \mathscr{M} \xrightarrow{\beta} \mathbb{N}' \parallel \mathscr{M}'$ *and* $\mathsf{G}' \vdash \mathbb{N}' \parallel \mathscr{M}'$.

Theorem 4.3 (Subject Reduction) *If* $\mathsf{G} \vdash \mathbb{N} \parallel \mathscr{M}$ *and* $\mathbb{N} \parallel \mathscr{M} \xrightarrow{\beta} \mathbb{N}' \parallel \mathscr{M}'$*, then* $\mathsf{G} \parallel \mathscr{M} \xrightarrow{\beta} \mathsf{G}' \parallel \mathscr{M}'$ *and* $\mathsf{G}' \vdash \mathbb{N}' \parallel \mathscr{M}'$.

Proof. The proof is by induction on $d = \mathsf{depth}(\mathsf{G},\mathsf{p})$ where $\mathsf{p} = \mathsf{play}(\beta)$. Notice that $\mathbb{N} \parallel \mathscr{M} \xrightarrow{\beta} \mathbb{N}' \parallel \mathscr{M}'$ implies $\mathsf{p} \in \mathsf{Players}(\mathbb{N})$, which together with $\mathsf{G} \vdash \mathbb{N} \parallel \mathscr{M}$ implies $\mathsf{p} \in \mathsf{Players}(\mathsf{G})$. Then $d > 0$. Moreover d is finite since G is bounded.
Let $d > 1$ and $\mathsf{G} = \mathsf{r}?\{\mathsf{q}_i.\ell_i;\mathsf{G}_i\}_{i\in I}$ with $\mathsf{r} \neq \mathsf{p}$. Since $\mathsf{G} \vdash \mathbb{N} \parallel \mathscr{M}$ must be derived using Rule [In], we get:

$$\mathbb{N} \equiv \mathsf{r}[\![\Sigma_{h\in H}\mathsf{q}_h?\ell_h;R_h]\!] \parallel \mathbb{N}_0 \qquad \mathsf{G}_j \vdash \mathsf{r}[\![R_j]\!] \parallel \mathbb{N}_0 \parallel \mathscr{M}_j \text{ for all } j \in J \qquad \boldsymbol{\iota}(\mathsf{G}_j,\mathbb{M}) \text{ for all } j \in J$$

where $J = \mathsf{rm}(\{\langle \mathsf{q}_i,\ell_i,\mathsf{p}\rangle\}_{i\in I},\mathscr{M}) = \mathsf{rm}(\{\langle \mathsf{q}_h,\ell_h,\mathsf{p}\rangle\}_{h\in H},\mathscr{M})$ and $\mathscr{M} \equiv \langle \mathsf{q}_j,\ell_j,\mathsf{r}\rangle \cdot \mathscr{M}_j$ for all $j \in J$ and $\mathbb{M} = \{\langle \mathsf{q}_l,\ell_l,\mathsf{r}\rangle \mid l \in (I \cup H) \setminus J \ \&\ \mathsf{q}_l \neq \mathsf{q}_j \ \forall j \in J\}$. The condition $\mathsf{r} \neq \mathsf{p}$ ensures that the transition $\mathbb{N} \parallel \mathscr{M} \xrightarrow{\beta} \mathbb{N}' \parallel \mathscr{M}'$ does not modify the process of participant r and does not dequeue any message with receiver r from $\mathscr{M}$. Therefore we get $\mathbb{N}' \equiv \mathsf{r}[\![\Sigma_{h\in H}\mathsf{q}_h?\ell_h;R_h]\!] \parallel \mathbb{N}'_0$ and $\mathscr{M}' \equiv \langle \mathsf{q}_j,\ell_j,\mathsf{r}\rangle \cdot \mathscr{M}'_j$ for all $j \in J$. Moreover the transition can be done also if the process $\Sigma_{h\in H}\mathsf{q}_h?\ell_h;R_h$ is replaced by an arbitrary process and top messages with receiver r are dequeued. Therefore

$$\mathsf{r}[\![R_j]\!] \parallel \mathbb{N}_0 \parallel \mathscr{M}_j \xrightarrow{\beta} \mathsf{r}[\![R_j]\!] \parallel \mathbb{N}'_0 \parallel \mathscr{M}'_j \text{ for all } j \in J$$

It is easy to verify that $\mathsf{depth}(\mathsf{G}_j,\mathsf{p}) < \mathsf{depth}(\mathsf{G},\mathsf{p})$. Then by induction we get $\mathsf{G}_j \parallel \mathscr{M}_j \xrightarrow{\beta} \mathsf{G}'_j \parallel \mathscr{M}'_j$ and $\mathsf{G}'_j \vdash \mathsf{r}[\![R_j]\!] \parallel \mathbb{N}'_0 \parallel \mathscr{M}'_j$ for all $j \in J$. Let $\mathsf{G}' = \mathsf{r}?\{\mathsf{q}_i.\ell_i;\mathsf{G}'_i\}_{i\in I}$ where $\mathsf{G}'_l = \mathsf{G}_{j_0}$ for some $j_0 \in J$ and all $l \in I \backslash J$. From $\mathsf{G}_j \parallel \mathscr{M}_j \xrightarrow{\beta} \mathsf{G}'_j \parallel \mathscr{M}'_j$ we get $\mathsf{G}_j \parallel \mathscr{M} \xrightarrow{\beta} \mathsf{G}'_j \parallel \mathscr{M}'$ for all $j \in J$. Then we derive $\mathsf{G} \parallel \mathscr{M} \xrightarrow{\beta} \mathsf{G}' \parallel \mathscr{M}'$ by Rule [Inside-In]. The condition $\boldsymbol{\iota}(\mathsf{G}_j,\mathbb{M})$ for all $j \in J$ ensures that $\mathscr{M}'$ cannot contain a message $\langle \mathsf{q}_k,\ell_k,\mathsf{r}\rangle$ with $k \in (I \cup H) \setminus J$ and $\mathsf{q}_k \neq \mathsf{q}_j$ for all $j \in J$. The condition $\mathsf{r} \neq \mathsf{p}$ ensures that the transition $\xrightarrow{\beta}$ cannot dequeue a message with r as receiver. Hence $\mathsf{rm}(\{\langle \mathsf{q}_h,\ell_h,\mathsf{p}\rangle\}_{h\in H},\mathscr{M}') = J$. It is easy to verify that $\boldsymbol{\iota}(\mathsf{G}_j,\mathbb{M})$ implies $\boldsymbol{\iota}(\mathsf{G}'_j,\mathbb{M})$ for all $j \in J$. From $\mathsf{G}'_j \vdash \mathsf{r}[\![R_j]\!] \parallel \mathbb{N}'_0 \parallel \mathscr{M}'_j$ for all $j \in J$ we get $\mathsf{Players}(\mathsf{G}'_i) \setminus \{\mathsf{r}\} = \mathsf{Players}(\mathbb{N}'_0)$ for all $i \in I$. Then all premises of Rule [In] hold and we can derive $\mathsf{G}' \vdash \mathbb{N}' \parallel \mathscr{M}'$. □

Theorem 4.4 (Lock-freedom) *If* $\mathsf{G} \vdash \mathbb{N} \parallel \mathscr{M}$ *and* $\mathsf{p}[\![P]\!] \in \mathbb{N}$*, then* $\mathbb{N} \parallel \mathscr{M} \xrightarrow{\tau\cdot\beta}$ *with* $\mathsf{play}(\beta) = \mathsf{p}$ *for some* τ, β.

As expected, since queues in type configurations can arbitrarily grow, our type system is undecidable.

Theorem 4.5 (Undecidability) *Typing is undecidable.*

In order to recover from this undecidability result, we can define an inductive version of typing, thus obtaining a sound algorithm. This inductive definition follows the standard pattern to deal with regular structures for global types, and it requires the same queue at the beginning and at the end of each cycle.

5 Related Work and Conclusion

We proposed flexible choice operators for an asynchronous multiparty session calculus, in order to ensure the classical session correctness properties for a larger class of protocols than is usually done. Several other proposals for relaxing the constraints of the original choice operator of [11] were already mentioned in Section 1. We now discuss some of them in more detail.

In [3], which builds on [13], we pushed this flexibility even further by allowing input choices with different senders in processes, without restrictions. The same approach was followed in [5]. However, this liberal approach turned out to be incorrect, as pointed out in [10], as it allows the following (synchronous) network to be typed, while it is not deadlock-free. Indeed, this network can reach a deadlock if p chooses its second branch, leading both s and t to choose their second branch too. Then, if r chooses its first branch, it will be unable to complete it.

$$\mathsf{p}[\![\,(\mathsf{s}!a;\mathsf{t}!a;\mathsf{r}!d)\oplus(\mathsf{s}!b;\mathsf{t}!b)\,]\!] \parallel \mathsf{r}[\![\,(\mathsf{s}?c;\mathsf{t}?e;\mathsf{p}?d)+(\mathsf{t}?e;\mathsf{s}?c)\,]\!] \parallel \\ \mathsf{s}[\![\,(\mathsf{p}?a;\mathsf{r}!c)+(\mathsf{p}?b;\mathsf{r}!c)\,]\!] \parallel \mathsf{t}[\![\,(\mathsf{p}?a;\mathsf{r}!e)+(\mathsf{p}?b;\mathsf{r}!e)\,]\!]$$

In fact, this session is not race-free according to the *race-freedom* condition proposed in [10]. Note that the asynchronous session obtained by composing this network with the empty queue is not typable in our type system. Indeed, its typability would contradict Subject Reduction or Lock-freedom, since it has the derivative $\mathsf{r}[\![\,\mathsf{p}?d\,]\!] \parallel \emptyset$ which is stuck. This session cannot be typed in [14] either, since participant r does not satisfy the required well-formedness conditions. Also the type system of [17] rejects this session, the reason being that participant r can read the same message in more than one branch. More precisely, participant r can read the message c from participant s and the message e from participant t in both branches. This control is realised by annotating projections with the set of available messages. We take advantage of queues in type configurations for a similar but less refined control, which uses the predicate ensuring that a global type is inactive for a given set of messages.

As future work, we plan to investigate three different variations of our typing. The first one would be a weakening of condition $\boldsymbol{\iota}(\mathsf{G},\mathbb{M})$ in Rule [IN], taking into account the order of sent messages and the expected inputs. The second one would be a strengthening of our typing in order to forbid orphan messages. The last one would be a weakening of our typing in order to allow optional participants in the branches of choices, possibly using connecting communications as in [13, 3].

Another direction that would be worth investigating is the relationship between our approach and input races for sessions based on Classical Linear Logic, see [15, 18].

To make our type system more efficient we will design two algorithms, taking inspiration from [7], one for inferring global types for networks and the other one for checking the correctness of global types for queues, allowing also cycles in which queues increase.

Acknowledgments This paper came into being thanks to Ross Horne, who pointed out to us the example reported in the Conclusion. We are indebted to him for many interesting discussions on this subject and for suggestions on a previous version of this paper. We also thank the anonymous referees for helpful comments.

References

[1] Marco Carbone, Nobuko Yoshida & Kohei Honda (2009): *Asynchronous Session Types: Exceptions and Multiparty Interactions*. In Marco Bernardo, Luca Padovani & Gianluigi Zavattaro, editors: *Formal Methods for Web Services*, *LNCS* 5569, Springer, pp. 187–212, doi:10.1007/978-3-642-01918-0_5.

[2] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini & Luca Padovani (2012): *On Global Types and Multi-Party Sessions*. *Logical Methods in Computer Science* 8, pp. 1–45, doi:10.2168/LMCS-8(1:24)2012.

[3] Ilaria Castellani, Mariangiola Dezani-Ciancaglini & Paola Giannini (2019): *Reversible Sessions with Flexible Choices*. *Acta Informatica* 56(7), pp. 553–583, doi:10.1007/s00236-019-00332-y.

[4] Ilaria Castellani, Mariangiola Dezani-Ciancaglini & Paola Giannini (2021): *Global types and event structure semantics for asynchronous multiparty sessions*. *CoRR* abs/2102.00865, doi:10.48550/arXiv.2102.00865.

[5] Ilaria Castellani, Mariangiola Dezani-Ciancaglini, Paola Giannini & Ross Horne (2020): *Global Types with Internal Delegation*. *Theoretical Computer Science* 807, pp. 128–153, doi:10.1016/j.tcs.2019.09.027.

[6] Bruno Courcelle (1983): *Fundamental Properties of Infinite Trees*. *Theoretical Computer Science* 25, pp. 95–169, doi:10.1016/0304-3975(83)90059-2.

[7] Francesco Dagnino, Paola Giannini & Mariangiola Dezani-Ciancaglini (2021): *Deconfined Global Types for Asynchronous Sessions*. *CoRR* abs/2111.11984, doi:10.48550/arXiv.2111.11984.

[8] Romain Demangeon & Kohei Honda (2012): *Nested Protocols in Session Types*. In Maciej Koutny & Irek Ulidowski, editors: *CONCUR*, *LNCS* 7454, Springer, pp. 272–286, doi:10.1007/978-3-642-32940-1_20.

[9] Pierre-Malo Deniélou & Nobuko Yoshida (2012): *Multiparty Session Types Meet Communicating Automata*. In Helmut Seidl, editor: *ESOP*, *LNCS* 7211, Springer, pp. 194–213, doi:10.1007/978-3-642-28869-2_10.

[10] Rob van Glabbeek, Peter Höfner & Ross Horne (2021): *Assuming Just Enough Fairness to make Session Types Complete for Lock-freedom*. In Leonid Libkin, editor: *LICS*, ACM Press, pp. 1–13, doi:10.1109/LICS52264.2021.9470531.

[11] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty Asynchronous Session Types*. In George C. Necula & Philip Wadler, editors: *POPL*, ACM Press, pp. 273–284, doi:10.1145/1328438.1328472.

[12] Kohei Honda, Nobuko Yoshida & Marco Carbone (2016): *Multiparty Asynchronous Session Types*. *Journal of ACM* 63(1), pp. 9:1–9:67, doi:10.1145/2827695.

[13] Raymond Hu & Nobuko Yoshida (2017): *Explicit Connection Actions in Multiparty Session Types*. In Marieke Huisman & Julia Rubin, editors: *FASE*, *LNCS* 10202, Springer, pp. 116–133, doi:10.1007/978-3-662-54494-5_7.

[14] Sung-Shik Jongmans & Nobuko Yoshida (2020): *Exploring Type-Level Bisimilarity towards More Expressive Multiparty Session Types*. In Peter Müller, editor: *ESOP*, *LNCS* 12075, Springer, pp. 251–279, doi:10.1007/978-3-030-44914-8_10.

[15] Wen Kokke, J. Garrett Morris & Philip Wadler (2020): *Towards Races in Linear Logic*. *Logical Methods in Computer Science* 16(4), doi:10.23638/LMCS-16(4:15)2020.

[16] Julien Lange, Emilio Tuosto & Nobuko Yoshida (2015): *From Communicating Machines to Graphical Choreographies*. In Sriram K. Rajamani & David Walker, editors: *POPL*, ACM Press, pp. 221–232, doi:10.1145/2676726.2676964.

[17] Rupak Majumdar, Madhavan Mukund, Felix Stutz & Damien Zufferey (2021): *Generalising Projection in Asynchronous Multiparty Session Types*. In Serge Haddad & Daniele Varacca, editors: *CONCUR*, *LIPIcs* 203, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 35:1–35:24, doi:10.4230/LIPIcs.CONCUR.2021.35.

[18] Zesen Qian, G. A. Kavvos & Lars Birkedal (2021): *Client-server sessions in linear logic*. *Proc. ACM Program. Lang.* 5(ICFP), pp. 1–31, doi:10.1145/3473567.