Multiparty Asynchronous Session Types: A Mechanised Proof of Subject Reduction

Dawit Tirore
IT University of Copenhagen, Denmark

Jesper Bengtson
IT University of Copenhagen, Denmark

Marco Carbone
IT University of Copenhagen, Denmark

Abstract

Session types offer a type-based approach to describing the message exchange protocols between participants in communication-based systems. Initially, they were introduced in a binary setting, specifying communication patterns between two components. With the advent of multiparty session types (MPST), the typing discipline was extended to arbitrarily many components. In MPST, communication patterns are given in terms of global types, an Alice-Bob notation that gives a global view of how components interact. A central theorem of MPST is subject reduction: a well-typed system remains well-typed after reduction. The literature contains some formulations of MPST with proofs of subject reduction that have later been shown to be incorrect. In this paper, we show that the subject reduction proof of the original formulation of MPST by Honda et al. contains some flaws. Additionally, we provide a restriction to the theory and show that, for this fragment, subject reduction does indeed hold. Finally, we use subject reduction to show that well-typed processes never go wrong. All of our proofs are mechanised using the Coq proof assistant.

2012 ACM Subject Classification Theory of computation → Process calculi; Theory of computation → Type structures; Theory of computation → Program specifications

Keywords and phrases Session types, Multiparty, Mechanisation, Coq

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.31

Supplementary Material *Software (Source Code)*:
`https://github.com/Tirore96/subject_reduction/tree/ECOOP2025/theories`
archived at `swh:1:dir:00e7a131e4038ba8d267fed7d0de9715b819f9de`
Software (ECOOP 2025 Artifact Evaluation approved artifact):
`https://doi.org/10.4230/DARTS.11.2.3`

Funding This work was supported by DFF-Research Project 1 Grant no. 1026-00385A, from The Danish Council for Independent Research for the Natural Sciences (FNU).

1 Introduction

Session types provide a type-based framework for specifying how participants in a concurrent communication-based system exchange messages. These communication patterns, specified in a protocol language, are used for verifying the programs that implement them. The framework guarantees properties such as type safety (well-typed programs never go wrong), session fidelity (processes behave according to protocol descriptions), and in-session deadlock-freedom (protocols never get stuck). Originally, Honda et al. [30, 31] proposed *binary session types* where types (protocol specifications) describe only how pairs of processes exchange messages. Inspired by choreographic languages [9], Honda et al. later extended this concept to *multiparty session types* [32, 33], which generalise the framework to handle protocols with arbitrarily many participants and asynchronous communication. This seminal work has driven significant progress in both the theory of communication-based systems [6, 11, 19, 39, 49] and their implementation in mainstream programming languages [1, 2, 20, 38].

Except for some cases [13, 35, 51], the correctness results of the theory of multiparty session types rely on paper proofs, which necessitate the omission of numerous details. Known pitfalls, such as incorrect results caused by issues with binders [21, 54], or unsound projection functions [51], often arise from overlooking technical details that are nearly impossible to catch in lengthy paper proofs.

A common way to make sure that all of these details are found, and to ensure that all proofs are correct, is to use interactive proof assistants like Isabelle [44], Coq [41], or Lean [42], to have all proofs machine-checked by a computer. Proof assistants have proven immensely successful and have been used to guarantee the correctness of a wide variety of foundational results in both mathematics [25, 26, 27, 28] and computer science [5, 36]. Nevertheless, attractive as these mechanisations are, they are also very difficult to implement. The original multiparty session types paper [32], recipient of the Most Influential Paper POPL 2008 award [40], should by all means be a prime candidate for mechanisation but has, because of the difficulty of the proofs, mostly been left untouched.

This paper mechanises the subject reduction theorem for the theory of multiparty asynchronous session types, as stated in the journal version of the original paper [33]. Subject reduction states that well-typed programs remain well-typed throughout their execution, as defined by their operational semantics. Subject reduction is one of the core theorems in the meta-theory, supporting key results such as type safety, session fidelity, and in-session deadlock-freedom. Although the proofs of the original paper are quite detailed, they are also lengthy and intricate. During our mechanisation efforts, however, we discovered that subject reduction, as presented in the original work, does not hold. In this paper, we provide a counterexample to the original theorem and present a new type system for which subject reduction does hold. Our proofs are fully machine-checked using the Coq proof assistant.

Multiparty Session Types. Multiparty session types use *global types*, a protocol specification language for describing the interaction patterns among multiple communicating participants. Interactions feature a sender and a receiver, generally called *roles.* For example, the following global type specifies the communication patterns between roles p, q, and r:

$$G = \mathsf{p} \to \mathsf{q} \colon 1 \begin{Bmatrix} l_1 \colon \mathsf{q} \to \mathsf{r} : 2\langle \mathsf{bool} \rangle.\mathsf{end} \\ l_2 \colon \mathsf{p} \to \mathsf{r} : 2\langle \mathsf{bool} \rangle.\mathsf{end} \end{Bmatrix} \tag{1}$$

This global type describes a protocol where role p internally decides and informs role q over channel 1, whether to continue in branch l_1 or in branch l_2. If p picks branch l_1, role q, after receiving this label, must send a boolean to r over channel 2. Conversely, if q picks branch l_2, p must then send a boolean to r over the same channel 2. In both branches, the protocol terminates after p or q communicate with r. A channel in the protocol represents a private FIFO queue, shared between the three roles for communication. A global type describes asynchronous protocols, meaning that sends are non blocking, sent messages are immediately placed on a channel, and the sending process can continue executing while the message is in transit. In this example, role q must first receive label l_1 from p before sending a boolean on channel 2. However, if p chooses branch l_2, it can immediately send a boolean without waiting for q to read from channel 1. Global types are equipped with a *reduction semantics* that captures this asynchronous behaviour.

A key observation, relevant for the remainder of this work, is that, as in the original work, a global type dictates exactly which channels the roles communicate over, as can be seen from the explicit references to channels 1 and 2 in the example above. This practice is often referred to as *explicit channels*, and contrasts with later work where the way roles use channels is implicit and therefore more restrictive. In the implicit channel setting, each pair of roles shares private channels that no other roles can access. Implicit channels are often used in the literature because they offer a convenient abstraction when full control over channel usage is not needed, and formal proofs are significantly simplified. Explicit channels are, however, more expressive and allow fine-grained control over which channels are actually used. This is particularly useful in applications where only a limited number of channels are available and must be shared among all roles. For example, AMQP [3], the Advanced Message Queuing Protocol, is a widely used message-oriented middleware protocol where communication takes place over named queues (channels), and where efficient use of limited shared resources is critical. Encoding implicit channels using explicit ones is straightforward and just a matter of allocating unique channels to all pairs of roles. This means that our results apply to both explicit and implicit channels, and not just to the explicit setting. It is also worth noting that using implicit channels requires allocating a channel (queue) for each ordered pair of roles, resulting in a quadratic number of channels.

While global types provide a holistic view of a protocol, the behaviour of its individual roles can be described as *local types*. Local types provide a local view of the protocol for each of its roles and can be obtained from global types by an operation known as *projection*. For example, the global type from Equation (1) can be expressed as the following collection of local types:

$$\mathsf{p} : 1 \oplus \left\{ \begin{array}{l} l_1 : \mathsf{end} \\ l_2 : !2\langle\mathsf{bool}\rangle.\mathsf{end} \end{array} \right\} \qquad \mathsf{q} : 1\&\left\{ \begin{array}{l} l_1 : !2\langle\mathsf{bool}\rangle.\mathsf{end} \\ l_2 : \mathsf{end} \end{array} \right\} \qquad \mathsf{r} : ?2\langle\mathsf{bool}\rangle.\mathsf{end} \tag{2}$$

The local type for p says that label l_1 or label l_2 must be communicated over channel 1 and if label l_2 is chosen then a boolean must be sent over channel 2. Dually, role q must be ready to receive label l_1 or l_2 on channel 1, and if label l_1 is selected, a boolean must be sent over channel 2. Finally, role r is ready to receive a boolean from channel 2. Similar to global types, collections of local types such as the one seen in Equation (2), are equipped with a semantics that captures the various communications that can be performed.

Protocols described as global types are executed as *sessions*. A session is an instance of a protocol with private channels where each role is implemented as an executable process, written in the multiparty session calculus, a model of computation for concurrent system based on the π-calculus [43]. The core idea of multiparty session types is that global types can be projected into local types which are then, via a type system, used to verify that a given implementation conforms to the specification provided by the global types. For example, multiparty session types ensure that the following parallel composition of processes can participate in a session that implements the global type from Equation (1):

$$\begin{array}{l} \bar{\mathsf{a}}[\mathsf{p}]_2(s).\overbrace{s[1] \triangleleft l_2; s[2]!\langle\mathsf{true}\rangle; \mathbf{0}}^{P} \quad | \\ \mathsf{a}[\mathsf{q}](s).\underbrace{s[1] \triangleright \{l_1 : s[2]!\langle\mathsf{false}\rangle;\ \mathbf{0}, \quad l_2 : \mathbf{0}\}}_{Q} \quad | \quad \mathsf{a}[\mathsf{r}](s).\ \underbrace{s[2]?(x); \mathbf{0}}_{R} \end{array} \tag{3}$$

The term above reads as follows: three processes are about to initiate a 3-party protocol on a, where a is a *shared name* denoting the name of the protocol in the calculus. The first process, prefixed with $\bar{\mathsf{a}}[\mathsf{p}]_2(s)$, is requesting the initiation of a session featuring two queues (instantiating the channels specified in the global type), acting as p while the second and third process are performing a session accept, joining the session as roles q and r respectively. The binder s is a placeholder for a session identifier that will be created at runtime. Process P denotes a process that, playing role p, first puts a label l_2 in the queue at channel 1 and then puts the message true in queue at channel 2. Process Q awaits for a label on queue at channel 1 that determines its continuation. Finally, process R, playing role r, receives a value on the queue at channel 2. Initially, we synchronise over the prefixes $\overline{\mathsf{a}}[\mathsf{p}]_2(\mathscr{s})$, $\mathsf{a}[\mathsf{q}](\mathscr{s})$, and $\mathsf{a}[\mathsf{r}](\mathscr{s})$, yielding the term

$$(\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})\left(\begin{array}{l} \overbrace{\mathsf{s}^{\mathsf{p}}[1] \lhd l_2; \mathsf{s}^{\mathsf{p}}[2]!\langle\mathsf{true}\rangle; \mathbf{0}}^{P[\mathsf{s}^{\mathsf{p}}/\mathscr{s}]} \quad | \quad \overbrace{\mathsf{s}^{\mathsf{q}}[1] \rhd \{l_1 : \mathsf{s}^{\mathsf{q}}[2]!\langle\mathsf{false}\rangle; \mathbf{0}, l_2 : \mathbf{0}\}}^{Q[\mathsf{s}^{\mathsf{q}}/\mathscr{s}]} \quad | \\ \underbrace{\mathsf{s}^{\mathsf{r}}[2]?(x); \mathbf{0}}_{R[\mathsf{s}^{\mathsf{r}}/\mathscr{s}]} \quad | \quad \mathsf{s}[1] :: \emptyset \quad | \quad \mathsf{s}[2] :: \emptyset \end{array}\right) \tag{4}$$

The operation generates the new session identifier s (the syntax $(\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})$ is a restriction binder), which we use to replace the binder $\mathscr{s}$ with *located session identifiers* s^{p}, s^{q} and s^{r} for roles p, q, and r respectively. The two communication queues, represented by $\mathsf{s}[1]$ and $\mathsf{s}[2]$, where messages can be enqueued and dequeued, are shared between all participants and are initially empty. Role p, implemented by process $P[\mathsf{s}^{\mathsf{p}}/\mathscr{s}]$, first sends label l_2 over channel 1 and then value true over channel 2, and reduces to the following term:

$$(\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})\Big(\mathbf{0} \quad | \quad Q[\mathsf{s}^{\mathsf{q}}/\mathscr{s}] \quad | \quad R[\mathsf{s}^{\mathsf{r}}/\mathscr{s}] \quad | \quad \mathsf{s}[1] :: {l_2}^{\mathsf{p}} \cdot \emptyset \quad | \quad \mathsf{s}[2] :: \mathsf{true}^{\mathsf{p}} \cdot \emptyset\Big) \tag{5}$$

where the label ${l_2}^{\mathsf{p}}$ and the value $\mathsf{true}^{\mathsf{p}}$, both annotated with the sending role p, have been enqueued on $\mathsf{s}[1]$ and $\mathsf{s}[2]$ respectively.

A Counter-Example to Subject Reduction. Subject reduction states that any term typed by a collection of local types (referred to as the *typing environment*), derived from a global type through projection, will remain typable after the process reduces. Since local types are equipped with a reduction semantics, the subject reduction theorem also states that the term obtained after reduction can be typed either by the same typing environment or by a modified environment, reflecting a reduction of local types. The original statement of subject reduction by Honda et al. can be informally stated as follows:

> *If P is well typed in a typing environment Δ, and P reduces to P' then either P' is well-typed in the original typing environment Δ, or in another environment Δ' such that Δ can reduce to Δ'*

The reduction on the environment Δ is crucial because it establishes a connection between the reductions performed by the processes and the protocol specification defined by the local types (which are derived from global types). Proving other results, such as session fidelity (processes behave according to protocol descriptions), heavily relies on the relationship between the reductions of processes and types. Unfortunately, the original theory by Honda et al. does not satisfy subject reduction. The key issue lies in the fact that some well-typed processes can reduce to well-typed terms in a way where their typing environments are not the same nor does the initial environment reduce to the latter. We demonstrate this with our running example.

Even though the process in Equation (3) has been reduced to the one in (4) and then to (5), this new process is still typable by the same local types in Equation (2), derived by means of projection from the global type in Equation (1). This is because the semantics of environments tracks interactions, which matches an output action on a channel with an input action on the same channel. So far only outputs have been performed, and the environment is thus unchanged. However, when process $R[\mathsf{s}^{\mathsf{r}}/\mathscr{s}]$ (which is equal to $\mathsf{s}^{\mathsf{r}}[2]?(x); \mathbf{0}$) retrieves message $\mathsf{true}^{\mathsf{p}}$ from $\mathsf{s}[2]$, the term in Equation (5) reduces to

$$(\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})\Big(\mathbf{0} \quad | \quad Q[\mathsf{s}^{\mathsf{q}}/\mathscr{s}] \quad | \quad \mathbf{0} \quad | \quad \mathsf{s}[1] :: {l_2}^{\mathsf{p}} \cdot \emptyset \quad | \quad \mathsf{s}[2] :: \emptyset\Big) \tag{6}$$

which reveals a problem. Here, the process $\mathsf{s}^{\mathsf{r}}[2]?(x);\mathbf{0}$, acting as role r, has consumed the boolean $\mathsf{true}^{\mathsf{p}}$ from the $\mathsf{s}[2]$ queue while ${l_2}^{\mathsf{p}}$ is still in the $\mathsf{s}[1]$ queue. In other words, the branching interaction between roles p and q described by the global type in Equation (1) has been executed out of order, since q has not yet received l_2, but the communication of a boolean from p to r in the l_2 branch has been completely performed. The problem with the original theory of multiparty session types is that the asynchronous semantics of processes allows $Q[\mathsf{s}^{\mathsf{q}}/\mathscr{s}]$ and $R[\mathsf{s}^{\mathsf{r}}/\mathscr{s}]$ to read from queues $\mathsf{s}[1]$ and $\mathsf{s}[2]$ respectively in any order. However, the semantics of the global type from Equation (1) requires that q (corresponding to process $Q[\mathsf{s}^{\mathsf{q}}/\mathscr{s}]$) reads from channel 1 (corresponding to queue $\mathsf{s}[1]$) before r (corresponding to process $R[\mathsf{s}^{\mathsf{r}}/\mathscr{s}]$) reads from channel 2 (corresponding to queue $\mathsf{s}[2]$).

Below (left) we show a global type that describes the remaining behaviour of the session, and we show (right) the environment derived from it.

$$\mathsf{p} \to \mathsf{q} : 1 \left\{ \begin{array}{l} l_1 : \mathsf{end} \\ l_2 : \mathsf{end} \end{array} \right\} \qquad \mathsf{p} : 1 \oplus \left\{ \begin{array}{l} l_1 : \mathsf{end} \\ l_2 : \mathsf{end} \end{array} \right\}, \quad \mathsf{q} : 1 \& \left\{ \begin{array}{l} l_1 : \mathsf{end} \\ l_2 : \mathsf{end} \end{array} \right\}, \quad \mathsf{r} : \mathsf{end} \tag{7}$$

This environment types the process in Equation (6). The problem is that the two environments in Equations (2) and (7) are neither the same, nor can they be related by the semantics of local types, therefore breaking subject reduction.

A reasonable question to ask at this point is why, given how influential the work by Honda et al. is, has this not been noticed before? The answer is most likely twofold. Firstly, the proofs are very complex and have never been mechanised. Secondly, the original formulation of multiparty session types differs from most later work. In particular, they use explicit channels where all roles in a session share queues, as opposed to implicit channels, introduced by Bettini et al. [4], where each ordered pair of roles has a dedicated queue. Our counterexample hinges on the use of explicit channels since it would not have been possible to derive it with implicit channels.

This Work. We develop a theory of multiparty session types that mostly follows the original framework by Honda et al [32]. Our main results are threefold. First, as described above, we identify a counterexample to subject reduction, highlighting a flaw in the original meta-theory of multiparty asynchronous session types. Second, we fix this flaw and prove subject reduction of the amended calculus and its type system. Finally, we mechanise all of our results in the Coq proof assistant which significantly reduces the risk of errors on our part. We gradually introduce the prerequisites of our subject reduction proof throughout this paper.

Section 2 defines a process language that models communicating programs. These programs feature different types of communication operations, all based on the concept of sessions. They can start new sessions and subsequently use session channels to communicate within these sessions. The process language is formally defined as a process algebra, equipped with a standard notion of structural equivalence and reduction semantics.

Next, in Section 3, we define global and local types. Global types give a global abstraction of how a set of processes should behave within a given session. To describe how individual processes within a session should behave, global types can be transformed, by means of projection, into a set of local types (one for each process in the session), where each local type represents the behaviour of a specific process in a session. Both global and local types are equipped with a reduction semantics which specifies the behaviour enforced by the types. Using these reduction semantics, we state a projection theorem that establishes the correctness of the projection from global types to local types. However, due to the flexible definition of global types, some global types are not realisable, meaning that their semantics does not correspond to that of their projection. To address this, we introduce the linearity property, which characterises those global types that can be correctly realised, similarly to the approach in the original framework [32, 33].

The last part of this article, in Sections 4 and 5, focuses on defining the type system for which we prove subject reduction. This does not follow the original framework precisely. In particular, our typing of queues (processes communicate via FIFO queues, one for each channel) takes a different approach which, despite being equivalent, facilitates the mechanisation work. For subject reduction to hold, we need to have an extra restriction on global types, which we call unstuckness. This restriction is what avoids bad situations such as that in the example above to happen. In addition to subject reduction, we also prove communication safety (type safety), which guarantees linearity – ensuring that there is never a race on a session channel – and error freedom in communications.

All proofs have been machine-checked in the Coq proof assistant, marking this as the first mechanisation of subject reduction for a general framework of asynchronous multiparty session types with explicit channels. Definitions and results in the paper are linked to the Coq code in our online repository. These links are marked with the symbol 🐓.

2 Multiparty Asynchronous Session Processes

Our process language is an extension of the π-calculus with the notion of session which forms the foundation for inter-process communication.

Syntax. Let $\mathcal{P}$ be a finite subset of the natural numbers which denotes a set of roles (ranged over by $\mathsf{p}, \mathsf{q}, \mathsf{r}, \mathsf{s}, \mathsf{t}$), let $\mathcal{L}$ be a set of labels (ranged over by l), and let I, J denote index sets of the form $\{1, \ldots, n\}$ for any natural number n. Moreover, let indices in an index set be denoted by i, j. We identify two distinct sets of atoms, *shared names*, ranged over by $\mathsf{a}, \mathsf{b}, \mathsf{c}$, and *session identifiers*, ranged over by $\mathsf{s}, \mathsf{t}, \mathsf{r}$. Note that both shared names and session identifiers are in sans-serif font. Multiparty asynchronous session processes are defined by the following grammar (🐓):

$$
\begin{array}{llll}
a ::= x \mid \mathsf{a} & \text{(shared names)} & s ::= \mathscr{s} \mid \mathsf{s}^{\mathsf{p}} & \text{(session channels)} \\
v ::= \mathsf{a} \mid \mathsf{true} \mid \mathsf{false} & \text{(values)} & e ::= x \mid v \mid e \;\mathsf{and}\; e' & \text{(expressions)} \\
h ::= l^{\mathsf{p}} \mid v^{\mathsf{p}} \mid (\mathsf{s}^{\mathsf{q}})^{\mathsf{p}} & \text{(message)} & D ::= \{X_i(x\tilde{\mathscr{s}}) = P_i\}_{i\in I} & \text{(declarations)}
\end{array}
$$

$$
\begin{array}{rcllll}
P & ::= & \overline{a}\,[\mathsf{n}]_k(\mathscr{s}).P & \text{(session request)} & \mid\ \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q & \text{(conditional branch)} \\
 & \mid & a[\mathsf{p}](\mathscr{s}).P & \text{(session acceptance)} & \mid\ P \mid Q & \text{(parallel composition)} \\
 & \mid & s[k]!\langle e\rangle; P & \text{(value sending)} & \mid\ \mathbf{0} & \text{(inaction)} \\
 & \mid & s[k]?(x); P & \text{(value reception)} & \mid\ (\boldsymbol{\nu}^{\mathsf{a}}\mathsf{a})P & \text{(name hiding)} \\
 & \mid & s[k]!\langle\langle t\rangle\rangle; P & \text{(session delegation)} & \mid\ (\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})P & \text{(session hiding)} \\
 & \mid & s[k]?((\mathscr{s})); P & \text{(session reception)} & \mid\ X\langle e, \tilde{s}\rangle & \text{(process call)} \\
 & \mid & s[k] \lhd l; P & \text{(label selection)} & \mid\ \mathsf{s}[k] :: \tilde{h} & \text{(message queue)} \\
 & \mid & s[k] \rhd \{l_i : P_i\}_{i\in I} & \text{(label branching)} & &
\end{array}
$$

A shared name non-terminal $a, b, c, \ldots$ can be either a value variable, denoted by $x, y, z, \ldots$, or a shared name a. Located session non-terminals, denoted by $s, t, r, \ldots$, can be either a located session variable, denoted by $\mathscr{s}, \mathscr{t}, \mathscr{r}, \ldots$, or a located session, denoted by $\mathsf{s}^{\mathsf{p}}, \mathsf{t}^{\mathsf{p}}, \mathsf{r}^{\mathsf{p}}$, where the roles ($\mathsf{p}$, q, and r) denote which role is currently using the session, i.e. which location it is being used at. Figure 1 gives an overview of the various names.

A multi-cast session request $\overline{a}\,[\mathsf{n}]_k(\mathscr{s}).P$ requests to start a session on a, with roles $\{\mathsf{0}, ..., \mathsf{n}\}$ and the requester acting as n. When a session eventually is initiated on a concrete shared name a, a fresh session identifier s is created. The requester acts as n by referencing variable

a	shared name	x	value variable	a	shared name non-terminal
s^{p}	located session	$\mathscr{s}$	located session variable	s	located session non-terminal
s	session identifier	k	channel		

Figure 1 Naming of key elements.

channel $\mathscr{s}$ in P which, during the execution, is substituted for a located session s^{n}. The natural number k declares how many channels (queues) will be created when the session is initiated and each queue is addressed as $\mathsf{s}[i]$ for $i \in \{1, ..., k\}$. The term $a[\mathsf{p}](\mathscr{s}).P$ denotes the dual process that accepts a request on a, playing role p.

The construct $s[k]!\langle e\rangle; P$ denotes the sending of expression e (after its evaluation to a value) to the queue of session s addressed at $\mathsf{s}[k]$. The construct $s[k]?(x); P$ reads from $\mathsf{s}[k]$, with variable x binding the received value in the continuation P. The construct $s[k]!\langle\langle t\rangle\rangle; P$ denotes sending a session channel, an operation called delegation. The construct $s[k]?((\mathscr{s})); P$ denotes receiving a session channel, an operation called session reception. In session reception, we bind a located session variable $\mathscr{s}$ in the continuation P, similarly to the request and accept constructs. Branching communications are captured by $s[k] \triangleleft l; P$ and $s[k] \triangleright \{l_i \colon P_i\}_{i\in I}$. The former selects a branch by communicating label l, while the latter reacts to a branch that another process has chosen (external choice).

The terms $(\boldsymbol{\nu}^{\mathsf{a}}\mathsf{a})P$ and $(\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})P$ respectively bind shared names and session identifiers in P. Free shared names and free session identifiers, denoted by $\mathsf{fn}(Q)$ and $\mathsf{fs}(S)$ respectively, are defined in a standard way. The process call $X\langle e, \tilde{s}\rangle$ invokes procedures stored in the declarations D. Declarations are identified by a name X, ranging over some set of declaration names, and have the form $X(x\tilde{\mathscr{s}}) = P$, where the parameters x and $\tilde{\mathscr{s}}$ are instantiated with a value and a list of distinct located sessions, respectively. The body P may include process names, thereby supporting recursion. The terminated process is denoted by $\mathbf{0}$.

A queue addressed at $\mathsf{s}[k]$ with messages $\tilde{h}$ is denoted by $\mathsf{s}[k] :: \tilde{h}$, and a message h is either a label, a value, or a session name. A message contains an annotation of the role that has sent the message, e.g., $\mathsf{s}[k] :: \mathsf{true}^{\mathsf{p}} \cdot \tilde{h}$ contains a message from p. Syntactically, variables cannot be messages in a queue. This is sensible because the semantics introduced next enforces that all variables are substituted for concrete terms before being put on a queue.

We define two types of *substitution*: one on value variables and one on located session variables. Both are denoted by the standard syntax $P[\mathsf{a}/x]$ and $P[\mathsf{s}^{\mathsf{p}}/t]$.

Reduction Semantics. We now give a reduction semantics for processes. First, we must define the structural congruence relation.

▶ **Definition 1** (Structural Congruence). *Structural congruence, denoted by $P \equiv Q$, is the smallest equivalence relation that satisfies the following rules:*

$$P \mid \mathbf{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R)$$

$$(\boldsymbol{\nu}^{\mathsf{a}}\mathsf{a})\ \mathbf{0} \equiv \mathbf{0} \qquad (\boldsymbol{\nu}^{\mathsf{a}}\mathsf{a})P \ \mid\ Q \equiv (\boldsymbol{\nu}^{\mathsf{a}}\mathsf{a})(P \mid Q) \quad \textit{when } \mathsf{a} \notin \mathsf{fn}(Q)$$

$$(\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})\ \mathbf{0} \equiv \mathbf{0} \qquad (\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})P \ \mid\ Q \equiv (\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})(P \mid Q) \quad \textit{when } \mathsf{s} \notin \mathsf{fs}(Q)$$

$$\frac{P \equiv Q}{(\boldsymbol{\nu}^{\mathsf{a}}\mathsf{a})P \equiv (\boldsymbol{\nu}^{\mathsf{a}}\mathsf{a})Q} \qquad \frac{P \equiv Q}{(\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})P \equiv (\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})Q}$$

$$
\begin{array}{rll}
[\textsc{Link}] & \left(\prod_{\mathsf{i}=0}^{\mathsf{n}-1} \mathsf{a}[\mathsf{i}](\mathscr{s}).P_{\mathsf{i}}\right) \mid \overline{\mathsf{a}}\,[\mathsf{n}]_k(\mathscr{s}).P_{\mathsf{n}} \ \to_D\ (\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})\left(\prod_{\mathsf{i}=0}^{\mathsf{n}-1} P_{\mathsf{i}}[\mathsf{s}^{\mathsf{i}}/\mathscr{s}] \mid \prod_{j=1}^{k} \mathsf{s}\,[j] :: \epsilon\right) & \\
[\textsc{Send}] & \mathsf{s}^{\mathsf{p}}[k]!\langle e\rangle; P \mid \mathsf{s}[k] :: \tilde{h} \ \to_D\ P \mid \mathsf{s}[k] :: \tilde{h}\cdot v^{\mathsf{p}} & \boxed{e \downarrow v} \\
[\textsc{Recv}] & \mathsf{s}^{\mathsf{p}}[k]?(x); P \mid \mathsf{s}[k] :: v^{\mathsf{q}}\cdot\tilde{h} \ \to_D\ P[v/x] \mid \mathsf{s}[k] :: \tilde{h} & \\
[\textsc{Deleg}] & \mathsf{s}^{\mathsf{p}}[k]!\langle\!\langle \mathsf{t}^{\mathsf{q}}\rangle\!\rangle; P \mid \mathsf{s}[k] :: \tilde{h} \ \to_D\ P \mid \mathsf{s}[k] :: \tilde{h}\cdot(\mathsf{t}^{\mathsf{q}})^{\mathsf{p}} & \\
[\textsc{SRec}] & \mathsf{s}^{\mathsf{p}}[k]?(\!(\mathscr{s})\!); P \mid \mathsf{s}[k] :: (\mathsf{t}^{\mathsf{r}})^{\mathsf{q}}\cdot\tilde{h} \ \to_D\ P[\mathsf{t}^{\mathsf{r}}/\mathscr{s}] \mid \mathsf{s}[k] :: \tilde{h} & \\
[\textsc{Label}] & \mathsf{s}^{\mathsf{p}}[k] \lhd l; P \mid \mathsf{s}[k] :: \tilde{h} \ \to_D\ P \mid \mathsf{s}[k] :: \tilde{h}\cdot l^{\mathsf{p}} & \\
[\textsc{Branch}] & \mathsf{s}^{\mathsf{p}}[k] \rhd \{l_i : P_i\}_{i\in I} \mid \mathsf{s}[k] :: {l_j}^{\mathsf{q}}\cdot\tilde{h} \ \to_D\ P_j \mid \mathsf{s}[k] :: \tilde{h} & \boxed{j \in I} \\
[\textsc{IfT}] & \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \ \to_D\ P & \boxed{e \downarrow \mathsf{true}} \\
[\textsc{IfF}] & \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \ \to_D\ Q & \boxed{e \downarrow \mathsf{false}} \\
[\textsc{Def}] & X\langle e, \widetilde{\mathsf{s}^{\mathsf{p}}}\rangle \ \to_D\ P[v/x][\widetilde{\mathsf{s}^{\mathsf{p}}}/\tilde{\mathscr{z}}] & \boxed{e \downarrow v \text{ and } \big(X(x\tilde{\mathscr{z}}) = P\big) \in D} \\
[\textsc{Res}_{\boldsymbol{\nu}^{\mathsf{s}}}] & P \ \to_D\ P' \ \Rightarrow\ (\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})P \ \to_D\ (\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})P' & \\
[\textsc{Res}_{\boldsymbol{\nu}^{\mathsf{a}}}] & P \ \to_D\ P' \ \Rightarrow\ (\boldsymbol{\nu}^{\mathsf{a}}\mathsf{a})P \ \to_D\ (\boldsymbol{\nu}^{\mathsf{a}}\mathsf{a})P' & \\
[\textsc{Par}] & P \ \to_D\ P' \ \Rightarrow\ P \mid Q \ \to\ P' \mid Q & \\
[\textsc{Str}] & P \equiv P' \text{ and } P' \ \to_D\ Q' \text{ and } Q' \equiv Q \ \Rightarrow\ P \ \to_D\ Q &
\end{array}
$$

Figure 2 Reduction semantics for processes ().

The first line makes processes a commutative monoid over parallel composition with $\mathbf{0}$ as the unit element. The next two lines give the structural rules for name restriction, allowing the binder to be dropped when its scope is a terminated process, and allowing the scope to be extruded when this is capture-free. The rules in the last line ensure that $\equiv$ is a congruence with respect to the two restriction operators. Whilst this is not strictly necessary for the semantics, it will prove useful when we formulate communication safety in Section 5.

The reduction semantics for processes, denoted by $P \ \to_D\ P'$, is defined by the rules given in Figure 2. Rule [Link] initiates a session between $\{0, ..., \mathsf{n}\}$ roles. The request is made by n and the remaining processes accept on the same shared name a, declaring with $[\mathsf{i}]$ which role of the session they will perform as. The requester indicates by k how many queues will be initiated for the session. This information does not need to be shared with the other roles because the type system will ensure queue addresses $\mathsf{s}[k]$ are used correctly. The parallel composition of the accepting processes is defined using $\prod$ to denote parallel composition of a set of processes. We assume that the natural number n is greater than 0, i.e., there are at least two processes engaging in a session.

Rule [Send] puts a value at the end of a queue, annotating the value with the role p of the located session s^{p} it was sent on, and [Recv] reads from the front of the queue. Rules [Deleg] and [SRec] are the corresponding rules for delegation and session reception while [Label] and [Branch] are the corresponding rules for labels. Rules [IfT] and [IfF] are standard semantics for if-statements. Rule [Def] handles recursive calls by name by replacing the process variable X with its declaration in D, substituting x for the evaluation of e, and substituting $\tilde{\mathscr{z}}$ for a list of located sessions. Rules [$\text{Res}_{\boldsymbol{\nu}^{\mathsf{s}}}$], [$\text{Res}_{\boldsymbol{\nu}^{\mathsf{a}}}$] and [Par] allow reductions to occur under restriction binders and parallel composition. Rule [Str] handles structural congruence.

▶ Remark 2 (Differences with the original work). We annotate a session identifier s with a role p, yielding a located session s^{p}. Such annotations have become a common practice in the literature. They were introduced as *polarities* for binary session types [22, 54], which address unintended limitations of the semantics. As in previous work [4, 17], we annotate messages with the sending role to indicate the sender of the message. This allows us to compare the specification of p against the messages it has enqueued and its remaining actions.

3 Global Types, Local Types, and Projection

Syntax. The syntax of global and local types is given as follows ():

$$\begin{array}{lll} G & ::= & \mathsf{p} \to \mathsf{q} : k\langle U\rangle.G \mid \mathsf{p} \to \mathsf{q} : k\{l_j : G_j\}_{j\in J} \mid \mu\mathbf{t}.G \mid \mathbf{t} \mid \mathsf{end} \\ T & ::= & !k\langle U\rangle.T \mid ?k\langle U\rangle.T \mid k \oplus \{l_i : T\}_{i\in I} \mid k \,\&\, \{l_i : T_i\}_{i\in I} \mid \mu\mathbf{t}.T \mid \mathbf{t} \mid \mathsf{end} \\ U & ::= & \mathsf{bool} \mid \mathsf{int} \mid G \mid T \qquad\qquad \hat{\Delta} \ ::= \ \emptyset \mid \hat{\Delta},\ \mathsf{p} : T \end{array}$$

The global type $\mathsf{p} \to \mathsf{q} : k\langle U\rangle.G$ specifies a session with an interaction where p sends a message of type U to q via channel k, and G specifies the remaining session. The type $\mathsf{p} \to \mathsf{q} : k\{l_j : G_j\}_{j\in J}$ specifies a branching interaction where p sends a label l_i to q via k, and G_i specifies the remaining session. The constructs $\mu\mathbf{t}.G$ and $\mathbf{t}$ are used to write tail-recursive specifications such as $\mu\mathbf{t}.\mathsf{p} \to \mathsf{q} : k\langle U\rangle.\mathbf{t}$. Finally, end specifies the terminated session. A message type U is either a boolean, an integer[1], a global type or a local type. Global types are used as message types when communicating shared names and local types are used when communicating session channels.

The local type $!k\langle U\rangle.T$ (resp. $?k\langle U\rangle.T$) specifies a role that sends (resp. receives) a message of type U over channel k, with T representing the remaining actions. The type $k \oplus \{l_i : T\}_{i\in I}$ (resp. $k \,\&\, \{l_i : T_i\}_{i\in I}$) specifies sending (resp. receiving) a label l_i, with T_i being the subsequent actions. As for global types, $\mu\mathbf{t}.T$ and $\mathbf{t}$ allow tail-recursive specifications. Lastly, end is a terminated role.

We deal with recursive variables in a standard way and write capture-avoiding substitution as $G_1[G_2/\mathbf{t}]$ (resp. $T_1[T_2/\mathbf{t}]$). We assume that types are closed and contractive. Global and local types are contractive if, for any of its sub-expressions with shape $\mu\mathbf{t}_0.\mu\mathbf{t}_1...\mu\mathbf{t}_n.\mathbf{t}$, the body $\mathbf{t}$ is not $\mathbf{t}_0$ [47].

A *local type environment* $\hat{\Delta}$ is a map from roles to local types. For every p in the domain of $\hat{\Delta}$ (denoted by $\mathsf{dom}(\hat{\Delta})$), there is a unique entry $\mathsf{p} : T$ for some T.

Projection. Projection is a partial function which, given a global type and a role, attempts to generate a local type that captures the specification of this role in the global type. When the global type specifies behaviour that the role cannot implement, projection is undefined. Our mechanisation uses the projection function $\mathsf{proj}_{\mathsf{p}}(G)$ proposed by Tirore et al. [51], which is defined in Figure 3. The function $\mathsf{proj}_{\mathsf{p}}(G)$ is defined in terms of two auxiliary definitions: a translation function, denoted by $\mathsf{trans}_{\mathsf{p}}(G)$, and a projectability predicate, denoted by $\mathsf{projectable}_{\mathsf{p}}(G)$. The translation function is a total function which produces a local type representing the local behavior of the translated role. The projectability predicate checks that the global type specifies behavior that is possible to implement by the projected role.

[1] Our mechanization excludes the integer message type, as the expressiveness of the expression language is orthogonal to our focus. We include the integer type here solely to simplify examples with distinct message types.

$$\mathsf{gVar}(\mathsf{t}, G) \stackrel{\text{def}}{=} \begin{cases} \mathsf{gVar}(\mathsf{t}, G_1) \ \wedge \ \mathbf{t} \neq \mathbf{t}' & \text{if } G = \mu\mathbf{t}'.G_1 \\ \mathbf{t} \neq \mathbf{t}' & \text{if } G = \mathbf{t}' \\ \mathsf{True} & \text{otherwise} \end{cases}$$

$$\mathsf{trans}_{\mathsf{p}}(\mathsf{p}_1 \to \mathsf{p}_2 : k\langle U\rangle.G) \stackrel{\text{def}}{=} \begin{cases} !k\langle U\rangle.(\mathsf{trans}_{\mathsf{p}}(G)) & \text{if } \mathsf{p} = \mathsf{p}_1 \text{ and } \mathsf{p}_1 \neq \mathsf{p}_2 \\ ?k\langle U\rangle.(\mathsf{trans}_{\mathsf{p}}(G)) & \text{if } \mathsf{p} = \mathsf{p}_2 \text{ and } \mathsf{p}_1 \neq \mathsf{p}_2 \\ \mathsf{trans}_{\mathsf{p}}(G) & \text{if } \mathsf{p} \notin \{\mathsf{p}_1, \mathsf{p}_2\} \end{cases}$$

$$\mathsf{trans}_{\mathsf{p}}(\mathsf{p}_1 \to \mathsf{p}_2 : k\{l_j : G_j\}_{j\in J}) \stackrel{\text{def}}{=} \begin{cases} k \oplus \{l_j : \mathsf{trans}_{\mathsf{p}}(G_j)\}_{j\in J} & \text{if } \mathsf{p} = \mathsf{p}_1 \text{ and } \mathsf{p}_1 \neq \mathsf{p}_2 \\ k \ \& \ \{l_j : \mathsf{trans}_{\mathsf{p}}(G_j)\}_{j\in J} & \text{if } \mathsf{p} = \mathsf{p}_2 \text{ and } \mathsf{p}_1 \neq \mathsf{p}_2 \\ \mathsf{trans}_{\mathsf{p}}(G_1) & \text{otherwise} \end{cases}$$

$$\mathsf{trans}_{\mathsf{p}}(\mu\mathbf{t}.G) \stackrel{\text{def}}{=} \begin{cases} \mu\mathbf{t}.(\mathsf{trans}_{\mathsf{p}}(G)) & \text{if } \mathsf{gVar}(\mathsf{t}, G) \\ \mathsf{end} & \text{otherwise} \end{cases}$$

$$\mathsf{trans}_{\mathsf{p}}(\mathbf{t}) \stackrel{\text{def}}{=} \mathbf{t} \qquad\qquad \mathsf{trans}_{\mathsf{p}}(\mathsf{end}) \stackrel{\text{def}}{=} \mathsf{end}$$

$$\mathsf{proj}_{\mathsf{p}}(G) \stackrel{\text{def}}{=} \begin{cases} \mathsf{trans}_{\mathsf{p}}(G) & \text{if } \mathsf{projectable}_{\mathsf{p}}(G) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathsf{proj\_roles}(G,\ \{\mathsf{p}_1,\ \cdots,\ \mathsf{p}_n\}) \stackrel{\text{def}}{=} \begin{cases} \mathsf{p}_1 : \mathsf{proj}_{\mathsf{p}_1}(G),\ \cdots, & \text{if } \forall i \in \{1,\ \cdots,\ n\}. \\ \mathsf{p}_n : \mathsf{proj}_{\mathsf{p}_n}(G) & \mathsf{proj}_{\mathsf{p}_i}(G) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathsf{full\_proj}(G) \stackrel{\text{def}}{=} \mathsf{proj\_roles}(G,\ \mathsf{roles}(G))$$

Figure 3 Projection (🐓).

Translation is defined such that $\mathsf{trans}_{\mathsf{p}}(\mathsf{p}_1 \to \mathsf{p}_2 : k\langle U\rangle.\ G)$ produces the output $!k\langle U\rangle$ (resp. input $?k\langle U\rangle$) when the translated role is p_1 (resp. p_2), and translates the remaining global type as $\mathsf{trans}_{\mathsf{p}}(G)$. If the translated role does not occur in the interaction, translation simply ignores the interaction proceeds as $\mathsf{trans}_{\mathsf{p}}(G)$. The translation of a branching global type $\mathsf{trans}_{\mathsf{p}}(\mathsf{p}_1 \to \mathsf{p}_2 : k\{l_j : G_j\}_{j\in J})$ is similar when p occurs in the interaction. When p does not occur, translation chooses the first branch j_1 and proceeds as $\mathsf{trans}_{\mathsf{p}}(G_1)$. The auxiliary predicate $\mathsf{gVar}(\mathsf{t}, G)$ is used in the translation of $\mu\mathbf{t}.G$, and checks if $\mu\mathbf{t}$ can safely be added to $\mathsf{trans}_{\mathsf{p}}(G)$, without producing non-contractive local types. In particular, this predicate ensures that trans never produces non-contractive types like $\mu\mathbf{t}.\mathbf{t}$. Finally, $\mathsf{trans}_{\mathsf{p}}(\mathbf{t})$ (resp. $\mathsf{trans}_{\mathsf{p}}(\mathsf{end})$) returns $\mathbf{t}$ (resp. end).

On its own, the translation function is too permissive. E.g., the global type

$$\mathsf{p} \to \mathsf{q} \colon 1 \left\{ \begin{array}{l} l_1 \colon \mathsf{r} \to \mathsf{s} : 2\langle\mathsf{bool}\rangle.\mathsf{end} \\ l_2 \colon \mathsf{r} \to \mathsf{s} : 2\langle\mathsf{int}\rangle.\mathsf{end} \end{array} \right\}$$

$$\frac{}{\mathsf{p} \to \mathsf{q} : k\langle U\rangle.G \xrightarrow{\mathsf{p}\to\mathsf{q}:k\langle U\rangle} G}\;[\text{GR1}] \qquad \frac{j \in I}{\mathsf{p} \to \mathsf{q} : k\{l_i : G_i\}_{i\in I} \xrightarrow{\mathsf{p}\to\mathsf{q}:k\langle l_j\rangle} G_j}\;[\text{GR2}]$$

$$\frac{G_1 \xrightarrow{\ell} G_2 \qquad \mathsf{q} \notin \ell}{\mathsf{p} \to \mathsf{q} : k\langle U\rangle.G_1 \xrightarrow{\ell} \mathsf{p} \to \mathsf{q} : k\langle U\rangle.G_2}\;[\text{GR3}]$$

$$\frac{\forall i \in I.\ G_i \xrightarrow{\ell} G'_i \qquad \mathsf{q} \notin \ell}{\mathsf{p} \to \mathsf{q} : k\{l_i : G_i\}_{i\in I} \xrightarrow{\ell} \mathsf{p} \to \mathsf{q} : k\{l_i : G'_i\}_{i\in I}}\;[\text{GR4}] \qquad \frac{G[\mu\mathbf{t}.G/t] \xrightarrow{\ell} G'}{\mu\mathbf{t}.G \xrightarrow{\ell} G'}\;[\text{GR5}]$$

Figure 4 Semantics for global types (🐓).

specifies that p sends l_1 or l_2 to q and depending on this choice, r has to send a boolean or an integer to s. The problem is that role r cannot know which choice was made, and its specification is thus not possible to implement. The projection of role r should therefore be undefined for this global type. As a total function, translation naively ignores this and translates the l_1 branch, producing the local type $!2\langle\mathsf{bool}\rangle.\mathsf{end}$. If role p chooses l_2, role r will incorrectly behave as if l_1 was chosen.

The problem in the example above is solved by letting the projection function, which we denote with $\mathsf{proj}_\mathsf{p}(G)$, be defined only when the projectability predicate $\mathsf{projectable}_\mathsf{p}(G)$ holds. In such case, the returned local type is computed as $\mathsf{trans}_\mathsf{p}(G)$. This predicate depends on a coinductive specification of projection that is not covered in this paper. For the definition of the projectable predicate and the coinductive specification we refer the reader to the paper where they were first introduced [51].

The bottom of Figure 3 defines the function full_proj which projects a global type over all its roles, and generates a local type environment $\hat{\Delta}$.

▶ Remark 3 (Differences with the original work.). Projection is notoriously complex to define correctly. Our mechanisation uses the projection by Tirore et al. [51], which, unlike that of Honda et al., handles μ-binders correctly and has been proven sound and complete in Coq with respect to the coinductive projection specification by Ghilezan et al. [24].

Semantics of Global Types, Local Types, and Environments. We define the semantics of global types as a labeled transition system, denoted by $G \xrightarrow{\ell} G'$. A label ℓ, called *interaction*, is formally defined as:

$$\mathcal{U} ::= U \mid l \qquad\qquad \ell ::= \mathsf{p} \to \mathsf{q} : k\langle \mathcal{U}\rangle$$

The syntactic category $\mathcal{U}$ is either a message type U or a label l, and an interaction $\mathsf{p} \to \mathsf{q} : k\langle \mathcal{U}\rangle$ is either the exchange of a message type or a label.

The semantics of global types is defined by the rules given in Figure 4. Rules [GR1] and [GR2] allow a global type to reduce by its first interaction. Rules [GR3] and [GR4] allow a global type to reduce by an interaction ℓ that occurs nested in one of its continuations, which makes the semantics asynchronous by allowing p to occur in ℓ. On the process level, this corresponds to p putting a message on the queue to be read by q, followed by p completing interaction ℓ with a different role before q receives the message. The completed interaction ℓ may not involve q because inputs are blocking. Rule [GR5] unfolds a μ-binder.

$$\frac{}{!k\langle U\rangle.T \xrightarrow{!k\langle U\rangle} T}\ [\text{LR1}] \qquad \frac{}{?k\langle U\rangle.T \xrightarrow{?k\langle U\rangle} T}\ [\text{LR2}]$$

$$\frac{j \in I}{k \oplus \{l_i : T\}_{i\in I} \xrightarrow{!k\langle l_j\rangle} T_j}\ [\text{LR3}] \qquad \frac{j \in I}{k \,\&\, \{l_i : T_i\}_{i\in I} \xrightarrow{?k\langle l_j\rangle} T_j}\ [\text{LR4}]$$

$$\frac{T \xrightarrow{\zeta} T' \quad \mathsf{ch}(\zeta) \neq k}{!k\langle U\rangle.T \xrightarrow{\zeta} !k\langle U\rangle.T'}\ [\text{LR5}] \qquad \frac{\forall i \in I.\, T_i \xrightarrow{\zeta} T_i' \quad \mathsf{ch}(\zeta) \neq k}{k \oplus \{l_i : T\}_{i\in I} \xrightarrow{\zeta} k \oplus \{l_i : T_i'\}_{i\in I}}\ [\text{LR6}]$$

$$\frac{T[\mu \mathbf{t}.T/\mathbf{t}] \xrightarrow{\zeta} T'}{\mu \mathbf{t}.T \xrightarrow{\zeta} T'}\ [\text{LR7}] \qquad \frac{T_1 \xrightarrow{!k\langle \mathcal{U}\rangle} T_1' \qquad T_2 \xrightarrow{?k\langle \mathcal{U}\rangle} T_2'}{\hat{\Delta},\ \mathsf{p} : T_1,\ \mathsf{q} : T_2 \xrightarrow{\mathsf{p}\to\mathsf{q}:k\langle \mathcal{U}\rangle} \hat{\Delta},\ \mathsf{p} : T_1',\ \mathsf{q} : T_2'}\ [\text{LE}\textsc{nv}]$$

Figure 5 Semantics for local types (🏆) and local type environments (🏆).

The semantics of local types is a two-level labeled transition system: the first level gives semantics to local types T, denoted by $T \xrightarrow{\zeta} T'$, and the second level gives semantics to local type environments $\hat{\Delta}$, denoted by $\hat{\Delta} \xrightarrow{\ell} \hat{\Delta}'$. A label ζ, called *action*, is formally defined as:

$$\zeta \ ::= \ !k\langle \mathcal{U}\rangle \ \mid \ ?k\langle \mathcal{U}\rangle$$

An action is the sending or receiving of a message type or a label. We write $\mathsf{ch}(\zeta)$ (resp. $\mathsf{ch}(\ell)$) to extract the channel of a local label ζ (resp. global label ℓ).

Formally, we define the semantics of local types and typing environments by the rules given in Figure 5. Rules [LR1], [LR2], [LR3] and [LR4] allow a local type to reduce by its first action. Similar to the global type rules [GR3] and [GR4], rules [LR5] and [LR6] allow a local type to reduce by an action ζ that occurs nested in one of its continuations. These rules make the local type semantics asynchronous. A condition for reducing by a nested action ζ, is that ζ uses a different channel than the first action, i.e., $\mathsf{ch}(\zeta) \neq k$. The intuition is that the first action, which is an output, already has occurred and it has resulted in a value being put on channel k; but before this value has been received by another participant, another interaction has in the meantime completed on another channel k'. Interactions between local types is captured by the semantics of local type environments $\hat{\Delta} \xrightarrow{\ell} \hat{\Delta}'$, given by the single rule [LE<span style="font-variant:small-caps">nv</span>]. An environment may reduce by $\mathsf{p} \to \mathsf{q} : k\langle \mathcal{U}\rangle$ when it maps p and q to local types that can reduce by $!k\langle \mathcal{U}\rangle$ and $?k\langle \mathcal{U}\rangle$.

▶ Remark 4 (Differences with the original work). We define the semantics of local types differently from Honda et al. Unlike their semantics, ours includes the asynchronous rules [LR5] and [LR6]. Instead, Honda et al. achieve asynchrony by defining a relation that relates local types with permuted outputs; for example, $!k\langle U\rangle.!k'\langle U'\rangle.T$ is related to $!k'\langle U'\rangle.!k\langle U\rangle.T$. Similar to our asynchronous rules, this permutation is only allowed if $k \neq k'$. However, that by Honda et al. is a more restrictive semantics, as it misses cases where an output is preceded by a nested input on a different channel, thereby permitting the input to be placed in front of the output. For instance, with the types $!k\langle U\rangle; ?k'\langle U'\rangle; T$ and $?k'\langle U'\rangle; !k\langle U\rangle; T$, for $k \neq k'$, our rule [LR5] with $\zeta = ?k'\langle U\rangle$ allows the output to be preceded by the input. Preceding an output with an input is essential for proving the equivalence between the semantics of global types and that of local type environments, a property known as the *Projection Theorem.* Consequently, their restrictive semantics means that their projection theorem (Lemma 5.11 [33]) does not hold. At the end of this section, we present a projection theorem for our semantics.

Linearity. The Honda et al. formulation of multiparty session types began an entire line of research, but to the best of our knowledge, all future work has adopted the *implicit* channels introduced by Bettini et al. [4]. Although they are more expressive, explicit channels are more complicated, allowing global types to contain races. For example, the global type:

$$\mathsf{p} \to \mathsf{q} : k\langle \mathsf{int}\rangle.\ \mathsf{r} \to \mathsf{q} : k\langle \mathsf{bool}\rangle.\mathsf{end} \tag{8}$$

is problematic because the two interactions $\mathsf{p} \to \mathsf{q} : k\langle \mathsf{int}\rangle$ and $\mathsf{r} \to \mathsf{q} : k\langle \mathsf{bool}\rangle$ share the same explicit channel k which introduces a race between p and r. The problem is that roles p, q, and r are freely using channel k without coordination. Hence, if r writes on channel k before p, it wins the race and a type error occurs when q receives bool while expecting int. Linearity of global types (not to be confused with standard linear types) is a property that rules out global types with interactions that introduce races. The main intuition behind linearity is that those interactions of a global type that share the same channel must have some causal dependency that avoids races. In the example, there is no causal dependency between the output of p and the output of q. However, the global type

$$\mathsf{p} \to \mathsf{q} : k\langle \mathsf{int}\rangle.\ \mathsf{p} \to \mathsf{r} : k'\langle \mathsf{string}\rangle.\ \mathsf{r} \to \mathsf{q} : k\langle \mathsf{bool}\rangle.\mathsf{end} \tag{9}$$

enforces a dependency between the output of p and the output of r: p first sends int on channel k and then sends string on k' while r has to receive string on channel k' before sending bool on channel k. Linearity checks that all execution paths of a global type enforce dependencies between interactions on the same channel, ensuring that no race can occur.

We first introduce the notion of a trace as:

▶ **Definition 5** (Trace 🐓). *A trace ρ of a global type G is a (possibly empty) sequence of interactions such that*

$$\frac{\mathsf{trace}\ \rho\ G}{\mathsf{trace}\ (\mathsf{p} \to \mathsf{q} : k\langle U\rangle \cdot \rho)\ (\mathsf{p} \to \mathsf{q} : k\langle U\rangle.G)} \qquad \frac{\mathsf{trace}\ \rho\ (G[\mu \mathbf{t}.G\ \mathsf{t}])}{\mathsf{trace}\ \rho\ (\mu \mathbf{t}.G)}$$

$$\frac{\mathsf{trace}\ \rho\ G_i}{\mathsf{trace}\ (\mathsf{p} \to \mathsf{q} : k\langle l_i\rangle \cdot \rho)\ (\mathsf{p} \to \mathsf{q} : k\{l_i : G_i\}_{i\in I})} \qquad \frac{}{\mathsf{trace}\ \epsilon\ G}$$

Intuitively, a trace records the interactions of a finite walk through the global type, starting from the first interaction and unfolding μ-binders when they occur. For example, the global type G in Equation (1) has traces $\mathsf{p} \to \mathsf{q} : 1\langle l_1\rangle \cdot \mathsf{q} \to \mathsf{r} : 2\langle \mathsf{bool}\rangle \cdot \epsilon$ and $\mathsf{p} \to \mathsf{q} : 1\langle l_2\rangle \cdot \mathsf{p} \to \mathsf{r} : 2\langle \mathsf{bool}\rangle \cdot \epsilon$. However, since a trace is a finite walk, also $\mathsf{p} \to \mathsf{q} : 1\langle l_1\rangle \cdot \epsilon$ and $\mathsf{p} \to \mathsf{q} : 1\langle l_2\rangle \cdot \epsilon$ are traces of G. We say $\mathsf{p} \to \mathsf{q} : k$ happens before $\mathsf{r} \to \mathsf{s} : k'$ in G if, for some ρ and ρ', we can derive:

$$\mathsf{trace}\ (\rho \cdot \mathsf{p} \to \mathsf{q} : k\langle \mathcal{U}\rangle \cdot \rho' \cdot \mathsf{r} \to \mathsf{s} : k'\langle \mathcal{U}'\rangle)\ G \tag{10}$$

We omit $\mathcal{U}$ and $\mathcal{U}'$ from the interactions as they provide no causal information. Given two interactions ℓ_1 and ℓ_2, we say that ℓ_1 and ℓ_2 are ordered if ℓ_1 happens before ℓ_2 or vice versa. Linearity asserts that ordered interactions that share the same channel must have dependencies between their inputs and outputs, which then guarantees the absence of races. Formally, we first define dependencies as:

▶ **Definition 6** (Input and Output Dependencies). *The interaction dependencies* $\mathsf{IO}, \mathsf{II}, \mathsf{OO}, \mathsf{IOOO} \subseteq \ell \times \ell$ (🐓) *and judgments* $\mathsf{input\ dep}\ \rho$ (🐓) *and* $\mathsf{output\ dep}\ \rho$ (🐓) *are defined as:*

$$\mathsf{p} \to \mathsf{q} : k\ \mathsf{IO}\ \mathsf{s} \to \mathsf{r} : k' \ \stackrel{\mathsf{def}}{=}\ (\mathsf{q} = \mathsf{s}) \qquad \mathsf{p} \to \mathsf{q} : k\ \mathsf{II}\ \mathsf{r} \to \mathsf{s} : k' \ \stackrel{\mathsf{def}}{=}\ (\mathsf{q} = \mathsf{s})$$

$$\mathsf{p} \to \mathsf{q} : k\ \mathsf{OO}\ \mathsf{s} \to \mathsf{r} : k' \ \stackrel{\mathsf{def}}{=}\ (\mathsf{p} = \mathsf{s}) \wedge (k = k') \qquad \ell_1\ \mathsf{IOOO}\ \ell_2 \ \stackrel{\mathsf{def}}{=}\ \ell_1\ \mathsf{IO}\ \ell_2\ \vee\ \ell_1\ \mathsf{OO}\ \ell_2$$

$$\text{[I-BASE]}\ \frac{\ell_1\ \mathsf{II}\ \ell_2}{\mathsf{input\ dep}\ \ell_1 \cdot \ell_2} \qquad \text{[I-SKIP]}\ \frac{\mathsf{input\ dep}\ \ell_1 \cdot \rho \cdot \ell_3}{\mathsf{input\ dep}\ \ell_1 \cdot \ell_2 \cdot \rho \cdot \ell_3} \qquad \text{[I-TAIL]}\ \frac{\mathsf{input\ dep}\ \ell_2 \cdot \rho \cdot \ell_3 \qquad \ell_1\ \mathsf{IO}\ \ell_2}{\mathsf{input\ dep}\ \ell_1 \cdot \ell_2 \cdot \rho \cdot \ell_3}$$

$$\text{[O-BASE]}\ \frac{\ell_1\ \mathsf{IOOO}\ \ell_2}{\mathsf{output\ dep}\ \ell_1 \cdot \ell_2} \qquad \text{[O-SKIP]}\ \frac{\mathsf{output\ dep}\ \ell_1 \cdot \rho \cdot \ell_3}{\mathsf{output\ dep}\ \ell_1 \cdot \ell_2 \cdot \rho \cdot \ell_3} \qquad \text{[O-TAIL]}\ \frac{\mathsf{output\ dep}\ \ell_2 \cdot \rho \cdot \ell_3 \qquad \ell_1\ \mathsf{IOOO}\ \ell_2}{\mathsf{output\ dep}\ \ell_1 \cdot \ell_2 \cdot \rho \cdot \ell_3}$$

Interaction dependencies relate interactions by asserting different ways the same role can occur in both interactions, such as IO (input-output) meaning the input role of the first interaction is also the output role of the second interaction (cf. [32]). Likewise we have OO (output-output) and II (input-input) dependencies. Input and output dependencies are obtained by composing interaction dependencies. An input dependency is a sequence of IO dependencies that ends in II while an output dependency is a sequence of IOOO dependencies. We use these chains to show causal orderings between the roles in the first and last interactions of the chain. For the sake of presentation of rules [O-BASE] and [O-TAIL], we have defined the interaction dependency IOOO which simply states that there can be either an IO or an OO dependency.

We begin by examining each rule of input dependency to understand why they assert and preserve the ordering of inputs. The shortest input dependency is established by [I-BASE], where the two inputs originate from the same role and are therefore ordered. Rule [I-SKIP] reflects that chains do not need to include all interactions in the trace. Finally rule [I-TAIL] extends an existing input dependency between ℓ_2 and ℓ_3 by adding ℓ_1 in front of ℓ_2 when it satisfies ℓ_1 IO ℓ_2. We can assume that the inputs of ℓ_2 and ℓ_3 are ordered and this is also the case between ℓ_1 and ℓ_3 if it is the case for ℓ_1 and ℓ_2. If the channels of ℓ_1 and ℓ_2 are different then the inputs are ordered because the input of ℓ_2 cannot intercept the output of ℓ_1. If the channels are the same, then there must exist input and output dependencies between ℓ_1 and ℓ_2, ensuring in particular that their inputs are ordered. More generally, input and output dependencies show that a specific pair of ordered interactions, with the same channel, have ordered inputs and outputs, by relying on this property to hold for all other ordered interactions with the same channel.

Next, we consider why the rules of output dependency preserve ordering of outputs. We start by the shortest output dependency built by [O-BASE], relating the interactions by either OO or IO. The former ensures the ordering by having the same role in both outputs. IO preserves output dependency by blocking the output of the second interaction. Rule [O-SKIP] is the corresponding skip. Finally, rule [O-TAIL] allows output dependencies to be extended either by OO or IO, extending the ordering of the outputs between ℓ_2 and ℓ_3 by adding ℓ_1 in front, as long as the outputs of ℓ_1 and ℓ_2 are ordered as well. The argument for why output ordering is preserved is similar to that of rule [O-BASE].

Linearity asserts dependencies between any ordered interactions with the same channel. Unlike Honda et al., we define linearity in terms of a weaker property, $\mathsf{linearHead}(G)$, which simplifies the decision procedure. This formulation only asserts the existence of dependencies when the first of the two interactions is also the first interaction in the trace.

We define linearity of a global type, $\mathsf{linear}(G)$, in terms of $\mathsf{linearHead}(G)$ in Figure 7. Intuitively, $\mathsf{linearHead}(G)$ checks if the first interaction is free of races and $\mathsf{next}(G)$ proceeds to the continuation of the global type. Due to global types having only finitely many distinct subterms, a consequence of μ-types being regular [47], only finitely many applications of the coinductive rule of linear is necessary before a previously encountered global type inevitably reappears, at which point the proof can circularly be closed.

$$|G| \overset{\text{def}}{=} \begin{cases} 1 + |G_1| & \text{if } G = \mu\mathbf{t}.G_1 \\ 0 & \text{otherwise} \end{cases} \qquad \mathsf{unfold\_once}(G) \overset{\text{def}}{=} \begin{cases} G_1[\mu\mathbf{t}.G_1/t] & \text{if } G = \mu\mathbf{t}.G_1 \\ G & \text{otherwise} \end{cases}$$

$$\mathsf{unf}(G) \overset{\text{def}}{=} \mathsf{unfold\_once}^{|G|}(G)$$

Figure 6 Calculating μ-height ( ) and unfolding global types ( ). Corresponding operations exist on local types (respectively and).

$$\begin{aligned} \mathsf{linearHead}(G) \overset{\text{def}}{=} \forall \ell_1\ \rho\ \ell_2.\ \ \mathsf{trace}\ (\ell_1 \cdot \rho \cdot \ell_2)\ G\ \rightarrow\ \mathsf{ch}(\ell_1) = \mathsf{ch}(\ell_2)\ \rightarrow \\ \mathsf{input\ dep}\ \ell_1 \cdot \rho \cdot \ell_2 \wedge \mathsf{output\ dep}\ \ell_1 \cdot \rho \cdot \ell_2 \end{aligned}$$

$$\mathsf{next}(G) \overset{\text{def}}{=} \begin{cases} \{G_1\} & \text{if } \mathsf{unf}(G) = \mathsf{p}_1 \to \mathsf{p}_2 : k\langle U\rangle.G_1 \\ \{G_i \mid i \in I\} & \text{if } \mathsf{unf}(G) = \mathsf{p}_1 \to \mathsf{p}_2 : k\{l_i : G_i\}_{i\in I} \\ \emptyset & \text{otherwise} \end{cases}$$

$$\frac{\mathsf{linearHead}(G) \quad \forall G' \in \mathsf{next}(G).\ \mathsf{linear}(G')}{\mathsf{linear}(G)}$$

Figure 7 Linearity of global types ( ).

The definition of the function $\mathsf{next}(G)$ uses the unfold operation $\mathsf{unf}(G)$ which we define in Figure 6. The definition uses μ-height, denoted by $|G|$, which is the number of top-level μ-binders occurring before another construct appears. For example, $|\mu\mathbf{t}.\mathsf{end}| = 1$ and $|\mathsf{p} \to \mathsf{q} : k\langle U\rangle.G| = 0$. We create an auxiliary function $\mathsf{unfold\_once}(\cdot)$ which unfolds a single binder by turning $\mu\mathbf{t}.G$ into $G[\mu\mathbf{t}.G/\mathbf{t}]$ and define the unfolding operation for global types, denoted by $\mathsf{unf}(G)$, as the repeated application of $\mathsf{unfold\_once}(G)$ μ-height times. The operation $\mathsf{unf}(T)$ can be similarly defined on local types and we elide its definition.

▶ **Theorem 7** (Decidability of Linearity). *The predicate* $\mathsf{linear}(G)$ *is decidable.*

Decidability of linearity can be reduced to a reachability problem on a graph induced by a global type with $\mathsf{next}(\cdot)$ as an adjacency list.

Projection Theorem. We have seen that interactions in a protocol can be described by $G \xrightarrow{\ell} G'$ and $\hat{\Delta} \xrightarrow{\ell} \hat{\Delta}'$. Naturally, we want to establish a correspondence between these two semantics with respect to the projection operation. Intuitively, if $\hat{\Delta}$ and $\hat{\Delta}'$ are derived from G and G', respectively, then $G \xrightarrow{\ell} G'$ holds if and only if $\hat{\Delta} \xrightarrow{\ell} \hat{\Delta}'$. However, this result is too strong due to recursion, which may unfold some terms during reduction, making projection slightly different. The issue lies in syntactic equality on local types being too restrictive to prove the projection theorem. Local types, however, can be more loosely related by a coinductive equality on their continuously unfolded terms:

▶ **Definition 8** (Coinductive equality). *Coinductive equality is a relation on local types defined by the following rules:*

$$\frac{\mathit{unf}(T) = \mathsf{end} \quad \mathit{unf}(T') = \mathsf{end}}{T \approx T'} \; [\textsc{Eq-End}]$$

$$\frac{\mathit{unf}(T) = !k\langle U\rangle.T'' \quad \mathit{unf}(T') = !k\langle U\rangle.T''' \quad T'' \approx T'''}{T \approx T'} \; [\textsc{Eq-!}]$$

$$\frac{\mathit{unf}(T) = ?k\langle U\rangle.T'' \quad \mathit{unf}(T') = ?k\langle U\rangle.T''' \quad T'' \approx T'''}{T \approx T'} \; [\textsc{Eq-?}]$$

$$\frac{\mathit{unf}(T) = k \oplus \{l_j : T_j\}_{j\in J} \quad \mathit{unf}(T') = k \oplus \{l_j : T'_j\}_{j\in J} \quad \forall j \in J.\ T_j \approx T'_j}{T \approx T'} \; [\textsc{Eq-}\oplus]$$

$$\frac{\mathit{unf}(T) = k \;\&\; \{l_j : T_j\}_{j\in J} \quad \mathit{unf}(T') = k \;\&\; \{l_j : T'_j\}_{j\in J} \quad \forall j \in J.\ T_j \approx T'_j}{T \approx T'} \; [\textsc{Eq-}\&]$$

This relation checks the shape of the unfolded local types, and checks whether the same actions are specified, before proceeding to the continuations. Because of unfolding, it might require a circular proof to derive the equality. With an abuse of notation, $\hat{\Delta}_1 \approx \hat{\Delta}_2$ is the point-wise extension of $\approx$ to local type environments. Coinductive equality on continuously unfolded μ-types is decidable [47], and this has been mechanised [18]. Our mechanisation also includes a proof of the decidability of the relation $\approx$, which, to the best of our knowledge, is the first mechanisation for session types. Using this relation, we can state the following:

▶ **Theorem 9** (Projection). *Let G be a global type such that* $\mathsf{linear}(G)$ *holds and* $\mathsf{full\_proj}(G)$ *is defined, then*

1. (🐓) *if $G \xrightarrow{l} G'$ then $\hat{\Delta}$ exists such that* $\mathsf{full\_proj}(G) \xrightarrow{l} \hat{\Delta}$ *and* $\mathsf{proj\_roles}(G', \mathsf{roles}(G)) \approx \hat{\Delta}$
2. (🐓) *if* $\mathsf{full\_proj}(G) \xrightarrow{l} \hat{\Delta}$ *then G' exists such that $G \xrightarrow{l} G'$ and* $\mathsf{proj\_roles}(G', \mathsf{roles}(G)) \approx \hat{\Delta}$.

▶ Remark 10 (Differences with the original work). The projection theorem of Honda et al. does not take into account that $\mathsf{roles}(G')$ may be a strict subset of $\mathsf{roles}(G)$. The set of roles in a global type can decrease after reduction if the reduced interaction involves a role that only occurs in that interaction. This results in an invalid environment reduction because the domain shrinks, exemplifying an innocent error that is hard to catch on paper.

4 Queue Types, Environment Decomposition, and Coherence

Queue types. Honda et al. handle the typing of queues using type contexts $\mathcal{T}$, which are local types with a hole $[\cdot]$. They decompose a local type T into a context $\mathcal{T}$ and another local type T', such that the applied context $\mathcal{T}[T']$ is related to T by a subtyping relation (see Definition 5.1 [33]). The type checking of a process against an environment $\hat{\Delta}$, may involve multiple of these decompositions. Mechanising this approach is non-trivial, and for convenience, we take a different approach: we type processes with an environment that contains local types that are decomposed *a priori*. We refer to this as a decomposed environment, denoted by $\hat{\Delta}; \hat{\mathfrak{Q}}$. We formally define queue environments $\hat{\mathfrak{Q}}$, and the queue types Q it contains, as:

▶ **Definition 11** (Queue type and Queue Environment). Queue types (🐓)*, denoted by* $\mathtt{Q}$*, and* queue environments (🐓)*, denoted by* $\hat{\mathfrak{Q}}$ *, are defined as:*

$$\mathtt{Q} \;::=\; \emptyset \mid (k, \mathcal{U}) \cdot \mathtt{Q} \qquad\qquad \hat{\mathfrak{Q}} \;::=\; \emptyset \mid \hat{\mathfrak{Q}},\ \mathtt{p} : \mathtt{Q}$$

A queue environment $\hat{\mathcal{Q}}$ is a map from roles to queue types, and a queue type is a sequence of pairs. In each pair, the left component is always a channel, and the right component is either a message type U or a label l. A pair $(k, \mathcal{U})$ in the queue type models the presence of a message $\mathcal{U}$ in the queue at channel k, and the order of these pairs indicates the order in which these messages were sent. Queue types are given with respect to roles in the environment $\hat{\mathcal{Q}}$, and this environment describes the content of all queues in the session. Queue types are derived from local types using the following notion of path:

▶ **Definition 12** (Path). *A path, denoted by σ, is a (possibly empty) sequence of elements from the set $\mathcal{L} \cup \{\bot\}$, where $\mathcal{L}$ is the set of labels defined in Section 2.*

A path is similar to a trace, it represents a walk through the μ-type. Unlike traces, we use paths to decompose local types.

▶ **Definition 13** (Decomposition of Local Types). *The decomposition of local types, denoted by* $\mathsf{decomp} : \sigma \to T \to T \times Q$, *is defined as:*

$$\mathsf{decomp}_\sigma\ T \overset{\text{def}}{=} \begin{cases} (T', (k, U) \cdot \mathtt{Q}) & \mathsf{unf}\ T = !k\langle U\rangle.T_1 \ \wedge\ \sigma = \bot \cdot \sigma' \ \wedge\ \mathsf{decomp}_{\sigma}\ T_1 = (T', \mathtt{Q}) \\ (T', (k, l_j) \cdot \mathtt{Q}) & \mathsf{unf}\ T = k \oplus \{l_i : T\}_{i \in I} \ \wedge\ \sigma = l_j \cdot \sigma' \ \wedge \\ & \mathsf{decomp}_{\sigma}\ T_j = (T', \mathtt{Q}) \ \wedge\ j \in I \\ (\mathsf{unf}\ T, \emptyset) & \sigma = \emptyset \\ \text{undefined} & \text{otherwise} \end{cases}$$

With respect to a path, decomposition splits a local type into a pair consisting of a residual local type and a queue type. We overload notation, denoting the point-wise extension of decomposition onto environments as $\mathsf{decomp}_f\ \hat{\Delta}$, which splits the environment $\hat{\Delta}$, with respect to a path function $f : \mathcal{P} \to \sigma$, into a residual environment $\hat{\Delta}'$ and a queue environment $\hat{\mathcal{Q}}$.

▶ **Definition 14** (Decomposition of Environments). *Let the path function f be such that* $\forall i \in \{1, \ldots, n\}.\ \mathsf{decomp}_{f(\mathsf{p_i})}\ T_i = (T_i', \mathtt{Q}_i)$. *Environment decomposition, denoted by* $\mathsf{decomp} : (\mathcal{P} \to \sigma) \to \hat{\Delta} \to \hat{\Delta} \times \hat{\mathcal{Q}}$, *is defined as:*

$$\mathsf{decomp}_f\ (\mathsf{p}_1 : T_1, ..., \mathsf{p}_n : T_n) \quad \overset{\text{def}}{=} \quad \mathsf{p}_1 : T_1', ..., \mathsf{p}_n : T_n';\ \ \mathsf{p}_1 : \mathtt{Q}_1, ..., \mathsf{p}_n : \mathtt{Q}_n$$

▶ **Example 15.** The path function tracks, within a session, the messages that are in transit. Recall that the initial state of our running example from Section 1, Equation (4), has two empty queues. Because the queues are empty, the decomposition of the local types that type the three roles, should all produce the empty queue type $\emptyset$. This is achieved with the path function: $f(\mathsf{p}) = \emptyset$. After p has put a message on each of the two queues in Equation (5), the local type of p must be decomposed in a way that reflects the two outputs have been performed, and that the remaining roles have performed no outputs. We achieve this with the path function: $f(\mathsf{p}') = \mathsf{if}\ \mathsf{p}' = \mathsf{p}\ \mathsf{then}\ l_1 \cdot \bot\ \mathsf{else}\ \emptyset$. The decomposition of $G\!\restriction_{\mathsf{p}}$ with respect to $f(\mathsf{p})$, where G is the global type in our running example, yields the residual local type end and queue type $(1, l_2) \cdot (2, \mathsf{bool})$.

Coherence. Honda et al. use a well-formedness condition on global types called coherence, which they define as a global type being linear and projectable, i.e., its projection is defined for all roles. Coherence is key to the soundness of Honda et al.'s work and other approaches based on global types, as it ensures that local behaviours are compatible [11, 49, 12] and correctly follow the global specification. However, coherence is not sufficient to rule out our counterexample. For this reason, we introduce an additional property to our definition of coherence, called unstuckness and denoted by $\mathsf{unstuck}\ (G)$.

$$\frac{}{\mathsf{p} \to \mathsf{q} : k\langle U\rangle.G \downarrow^1 \mathsf{p} \to \mathsf{q} : k\langle U\rangle}\text{[GB-1]} \qquad \frac{j \in I}{\mathsf{p} \to \mathsf{q} : k\{l_i : G_i\}_{i\in I} \downarrow^1 \mathsf{p} \to \mathsf{q} : k\langle l_j\rangle}\text{[GB-2]}$$

$$\frac{\mathsf{q} \notin \ell \quad G \downarrow^1 \ell}{\mathsf{p} \to \mathsf{q} : k\langle U\rangle.G \downarrow^1 \ell}\text{[GB-3]} \qquad \frac{\mathsf{q} \notin \ell \quad j \in I \quad G_j \downarrow^1 \ell}{\mathsf{p} \to \mathsf{q} : k\{l_i : G_i\}_{i\in I} \downarrow^1 \ell}\text{[GB-4]} \qquad \frac{G[\mu\mathbf{t}.G/t] \downarrow^1 \ell}{\mu\mathbf{t}.G \downarrow^1 \ell}\text{[GB-5]}$$

Figure 8 Barb relation for global types ().

The idea behind unstuckness is the following. The global type which specifies the counterexample is stuck because the only relevant rule, [GR4], is not applicable due to its restrictive premise, which requires universal quantification $\forall i \in I.\ G_i \xrightarrow{\ell} G_i'$. If this premise were relaxed to existential quantification $\exists i \in I.\ G_i \xrightarrow{\ell} G_i'$, then the global type would not be stuck. It is however not an option to relax the rule because it would break the Projection theorem. Instead, we define $\mathsf{unstuck}\ (G)$, which captures global types where the universally quantified premise implies the existentially quantified premise. The global type in the counterexample does not satisfy this property and is thus ruled out.

In order to define the $\mathsf{unstuck}$ predicate, we use the notion of barb, denoted by $G \downarrow^1 \ell$, formally defined by the rules in Figure 8. Intuitively, $G \downarrow^1 \ell$ holds whenever it is possible to derive that G can reduce by ℓ, in the relaxed setting where the existentially quantified premise is used in [GR4]. This lets us quantify over the labels for which reduction derivations must exist. This property must also be preserved by reduction, so we additionally assert that the reduced global type G' satisfies the property as well. For recursive global types, a derivation will not be finite, and the predicate is therefore coinductively defined:

▶ **Definition 16** (Unstuck). *We say a global type is unstuck if it satisfies the predicate* $\mathsf{unstuck}\ (G)$*, coinductively defined as:*

$$\begin{array}{c} \forall \ell.\ G \downarrow^1 \ell \implies \exists G'.G \xrightarrow{\ell} G' \wedge \mathsf{unstuck}\ (G') \\ \hline\hline \mathsf{unstuck}\ (G) \end{array}$$

We can finally introduce our stricter notion of coherence:

▶ **Definition 17** (Coherence). *The coherence of global types, local types and decomposed environments, denoted respectively by* $\mathsf{coherent}(G)$, $\mathsf{coherent}(L)$ *and* $\mathsf{coherent}(\hat{\Delta}; \hat{\mathcal{Q}})$*, is defined as:*

- () $\mathsf{coherent}(G)$ *holds iff G is projectable, linear and unstuck*
- () $\mathsf{coherent}(\hat{\Delta})$ *holds iff there exists a coherent G whose full projection is $\hat{\Delta}$*
- () $\mathsf{coherent}(\hat{\Delta}; \hat{\mathcal{Q}})$ *holds iff there exists a coherent $\hat{\Delta}'$ and a function f such that* $\mathsf{decomp}_f\ \hat{\Delta}' = \hat{\Delta}; \hat{\mathcal{Q}}$.

5 Typing System and Subject Reduction

This section introduces the type system for processes, connecting the concepts we have covered so far. This is formalised by the following typing judgement ():

$$\Gamma \vdash_{\mathcal{D}} P \triangleright_{\mathcal{C}} \mathcal{Q};\ \Delta$$

The intuitive meaning of the judgement is that process P is well-typed with values typed by the unrestricted environment Γ, including shared names, which Γ maps to global types. Ongoing sessions are typed by the two linear environments Δ and $\mathcal{Q}$. Process variables are typed by $\mathcal{D}$, and the linear environment $\mathcal{C}$ ensures that each queue address is unique. Each entry $\mathsf{s}[k]$ in $\mathcal{C}$ acts as a unique token associated to a queue. We use $\mathcal{C}$ to ensure that queue addresses are unique across a process, e.g., $\mathsf{s}[k] :: \emptyset \mid \mathsf{s}[k] :: \emptyset$. Formally, we define these environments as

$$\text{(Unrestricted)} \qquad \Gamma \ ::= \ \emptyset \ \mid \ \Gamma, a : G \ \mid \ \Gamma, x : U \qquad \mathcal{D} \ ::= \ \emptyset \ \mid \ \mathcal{D}, X : (U, \vec{T})$$

$$\text{(Linear)} \qquad \mathcal{C} \ ::= \ \emptyset \ \mid \ \mathcal{C}, \mathsf{s}[k] \qquad \Delta \ ::= \ \emptyset \ \mid \ \Delta, s : T \qquad \mathcal{Q} \ ::= \ \emptyset \ \mid \ \mathcal{Q}, \mathsf{s}^{\mathsf{p}} : Q$$

The environment $\mathcal{Q}$ associates a queue type with each located session, recording what actions have been performed, while Δ associates a local type with each session channel, indicating what actions remain. $\mathcal{C}$ ensures that all queue addresses are disjoint. Note that the environments $\mathcal{Q}$ and Δ may specify multiple sessions, while $\hat{\mathcal{Q}}$ and $\hat{\Delta}$ specifies a single session.

▶ **Definition 18** (Session Filtering and Lifting)**.** *The filtering of Δ by session identifier s, denoted by $\Delta(\mathsf{s})$, is the environment $\hat{\Delta}$ containing the local types of session s in Δ. Similarly, the filtering of $\mathcal{Q}$ by session identifier s, denoted by $\mathcal{Q}(\mathsf{s})$, is the environment $\hat{\mathcal{Q}}$ containing the local types of session s in $\mathcal{Q}$. These operations are always defined and return the empty environment when s is not present.*
Conversely, we define a lifting operation on $\hat{\Delta}$ and $\hat{\mathcal{Q}}$, denoted respectively by $[\hat{\Delta}]_{\mathsf{s}}$ and $[\hat{\mathcal{Q}}]_{\mathsf{s}}$, which yields environments Δ and $\mathcal{Q}$, corresponding to the initial environments where roles p have been replaced with located sessions s^{p}.

Using session filtering, we can extend the definition of coherence to $\Delta; \mathcal{Q}$.

▶ **Definition 19** (Coherence of $\Delta; \mathcal{Q}$)**.** *The coherence of the environment $\Delta; \mathcal{Q}$, denoted by $\mathsf{coherent}(\Delta; \mathcal{Q})$, holds iff for any session identifier s, $\Delta(\mathsf{s}); \mathcal{Q}(\mathsf{s})$ is coherent.*
When we, in stating the coherence of $\Delta; \mathcal{Q}$, want to refer to the underlying coherent environment Δ', whose decomposition yields $\Delta; \mathcal{Q}$, we write $\mathsf{coherent}(\Delta; \mathcal{Q})$ as Δ'.

The environments Δ, $\mathcal{Q}$ and $\mathcal{C}$ are linear, and this means different things for each. For Δ and $\mathcal{C}$, linearity carries the standard meaning: each entry in the environment must be used exactly once. In other words, entries cannot be duplicated, such that a local type in Δ is used to type two different processes. The linearity of $\mathcal{Q}$ is based on the partitioning of a queue type, which we explain next:

▶ **Example 20** (Partitioning a queue)**.** Consider two queues with messages from the same p

$$\mathsf{s}[k] :: \mathsf{true}^{\mathsf{p}} \mid \mathsf{s}[k'] :: \mathsf{true}^{\mathsf{p}}$$

The queue type of p is $(k, \mathsf{bool}) \cdot (k', \mathsf{bool})$. We partition the queue type into two queue types, one for each queue, namely (k, bool) and (k', bool).

In the following definition, let $\mathsf{filter}\ f\ Q$ denote the queue type obtained by filtering out all those entries in the channels of Q that do not satisfy the predicate $f : k \to \{\mathsf{True}, \mathsf{False}\}$. Partitioning is then defined as:

▶ **Definition 21** (Partitioning)**.** *Partitioning of a queue type by a predicate $f : k \to \mathsf{bool}$, denoted by $\mathsf{partition} : (k \to \{\mathsf{True}, \mathsf{False}\}) \to Q \to Q \times Q$, is defined as:*

$$\mathsf{partition}^f(Q) = (\mathsf{filter}\ f\ Q, \mathsf{filter}\ (\lambda k.\ \neg f(k))\ Q)$$

We write $(Q_0, Q_1) \hookrightarrow Q$ when there exists a predicate f such that $\mathsf{partition}^f(Q) = (Q_0, Q_1)$ and we point-wise extend this to environments, abusing notation by writing $(\mathcal{Q}_0, \mathcal{Q}_1) \hookrightarrow \mathcal{Q}$.

$$\frac{\Gamma(x) = S}{\Gamma \vdash x : S} \qquad \Gamma \vdash b : \mathsf{bool} \qquad \frac{\Gamma \vdash e : \mathsf{bool} \qquad \Gamma \vdash e' : \mathsf{bool}}{\Gamma \vdash e \;\mathsf{and}\; e' : \mathsf{bool}}$$

Figure 9 Typing rules for expressions (🐓).

The terminated process **0** and the empty queue $\mathsf{s}[k] :: \emptyset$ are typed using the following definitions for terminated environments:

▶ **Definition 22** (Terminated Environments 🐓). *A local type environment is terminated, denoted by* $\mathsf{ends}(\Delta)$ *iff* $\forall(\mathsf{p} : T) \in \Delta.\ \mathit{unf}(T) = \mathsf{end}$. *A queue environment is terminated, denoted by* $\mathsf{ends}(\mathcal{Q})$ *iff* $\forall(\mathsf{s}^{\mathsf{p}} : Q) \in \mathcal{Q}.\ Q = \epsilon$

Typing Rules. The rules defining our type system can be found in Figures 9, 10, and 11. Figure 9 contains the typing rules for expressions. The typing rules for processes are split into two parts: the first part is presented in Figure 10, where all rules include the premise $\mathsf{ends}(\mathcal{Q})$, which is omitted in the figure to conserve space. [T-MCast] and [T-MAcc] type session request and accept, respectively. Roles are totally ordered, and the largest role of G is n which is assigned to the requesting process. The unfolding of its projection is inserted into Δ. [T-Send] and [T-Rcv] are the rules for sending and receiving. The session channel s is mapped to a local type in Δ and checked to correspond with the action of the process. In the typing of the subterm P, the s entry in Δ is updated to the unfolded continuation $\mathsf{unf}\ T$. [T-Deleg] and [T-SRec] type session delegation and reception. Similarly to the rules for send and receive, they perform the delegation of session channel s', which requires the type T_2 for s' in the environment to be coinductively equal to the carried type T_1 in the type of s. Here, coinductive equality is necessary to ensure the type system is closed under $\approx$. The rules [T-Sel] and [T-Branch] are similar to [T-Send] and [T-Rcv]. [T-IF] is the rule for if statements, which has no effect on the type environments. [T-Inact] and [T-QNil] are the rules for inaction and the empty queue, using the predicates $\mathsf{ends}(\Delta)$ and $\mathsf{ends}(\mathcal{Q})$ to check that the respective environments are terminated.

The remaining rules are in Figure 11, where the premises of all rules are explicitly stated. [T-Conc] types parallel composition and implements the split of the environments as discussed above. [T-NRes] and [T-CRes] are the rules for name and channel restriction. [T-NRes] introduces a coherent global type into Γ. [T-CRes] adds the decomposed environment $\hat{\Delta}; \hat{\mathcal{Q}}$ into $\Delta; \mathcal{Q}$ and extends $\mathcal{C}$ by a disjoint set of unique tokens $\{\mathsf{s}[k_i]\}_{i \in I}$. The tokens in $\mathcal{C}$ are used to ensure the uniqueness of queue addresses such that if $\mathsf{s}_k \in \mathsf{dom}\ \mathcal{C}$, then exactly one queue of shape $\mathsf{s}[k] :: \tilde{h}$ exists in the typed process. [T-QVal], [T-QSess], and [T-QSel] are type queues. A message is annotated by the role p that sent the message, which is used to identify the queue type in $\mathcal{Q}$ associated with the located session s^{p}, checking that k and $\mathcal{U}$ in the queue type correspond to the channel of the queue and the message it contains. [T-Var] is the rule for procedure call, calling procedure X with an expression and a list of distinct session channels. Here, we write $\tilde{s} : \tilde{T}$ to mean then environment with entries $(\tilde{s}_i : \tilde{T}_i)$ for $i \in \{1, .., |\tilde{s}|\}$, where $|\tilde{s}| = |\tilde{T}|$.

Our semantics uses a fixed set of definitions D in $\rightarrow_D$. We now show how to type them.

▶ **Definition 23** (Typing of declarations 🐓). *Declarations D are typed by the environment $\mathcal{D}$ whenever* $\mathsf{dom}\ D = \mathsf{dom}\ \mathcal{D}$ *and*

$$\forall X(x\tilde{\mathfrak{z}}) = P \in D. \quad X : (U, \tilde{T}) \in \mathcal{D}\ \textit{implies}\ x : U \vdash_{\mathcal{D}} P \triangleright_{\emptyset} \tilde{\mathfrak{z}} : \tilde{T};\ \emptyset$$

$$\frac{\Gamma \vdash a : G \qquad \begin{matrix}\mathsf{roles}(G) = \{0, \dots, \mathsf{n}\} \\ \mathsf{channels}(G) = \{1, \dots, k\}\end{matrix} \qquad \Gamma \vdash_{\mathscr{D}} P \triangleright_{\emptyset} \Delta, \mathscr{s} : \mathsf{unf}(G \lfloor_{\mathsf{n}});\ \mathscr{Q}}{\Gamma \vdash_{\mathscr{D}} \overline{a}\,[\mathsf{n}]_k(\mathscr{s}).P \triangleright_{\emptyset} \Delta;\ \mathscr{Q}}\ [\text{T-MCast}]$$

$$\frac{\Gamma \vdash a : G \quad \mathsf{p} \in \mathsf{roles}(G) \quad \Gamma \vdash_{\mathscr{D}} P \triangleright_{\emptyset} \Delta, \mathscr{s} : \mathsf{unf}(G \lfloor_{\mathsf{p}});\ \mathscr{Q}}{\Gamma \vdash_{\mathscr{D}} a[\mathsf{p}](\mathscr{s}).P \triangleright_{\emptyset} \Delta;\ \mathscr{Q}}\ [\text{T-MAcc}]$$

$$\frac{\Gamma \vdash e : U \quad \Gamma \vdash_{\mathscr{D}} P \triangleright_{\emptyset} \Delta, s : \mathsf{unf}(T);\ \mathscr{Q}}{\Gamma \vdash_{\mathscr{D}} s[k]!\langle e \rangle; P \triangleright_{\emptyset} \Delta, s :!k\langle U \rangle.T;\ \mathscr{Q}}\ [\text{T-Send}] \qquad \frac{\Gamma, x : U \vdash_{\mathscr{D}} P \triangleright_{\emptyset} \Delta, s : \mathsf{unf}(T);\ \mathscr{Q}}{\Gamma \vdash_{\mathscr{D}} s[k]?(x); P \triangleright_{\emptyset} \Delta, s :?k\langle U \rangle.T;\ \mathscr{Q}}\ [\text{T-Rcv}]$$

$$\frac{T_1 \approx T_2 \quad s \neq s' \quad \Gamma \vdash_{\mathscr{D}} P \triangleright_{\emptyset} \Delta, s : \mathsf{unf}(T);\ \mathscr{Q}}{\Gamma \vdash_{\mathscr{D}} s[k]!\langle\!\langle s' \rangle\!\rangle; P \triangleright_{\emptyset} \Delta, s :!k\langle T_1 \rangle.T, s' : T_2;\ \mathscr{Q}}\ [\text{T-Deleg}]$$

$$\frac{\Gamma \vdash_{\mathscr{D}} P \triangleright_{\emptyset} \Delta, s : \mathsf{unf}(T), t : \mathsf{unf}(T');\ \mathscr{Q}}{\Gamma \vdash_{\mathscr{D}} s[k]?((t)); P \triangleright_{\emptyset} \Delta, s :?k\langle T' \rangle.T;\ \mathscr{Q}}\ [\text{T-SRec}]$$

$$\frac{i \in I \quad \Gamma \vdash_{\mathscr{D}} P \triangleright_{\emptyset} \Delta, s : \mathsf{unf}(T_i);\ \mathscr{Q}}{\Gamma \vdash_{\mathscr{D}} s[k] \triangleleft l_i; P \triangleright_{\emptyset} \Delta, s : k \oplus \{l_j : T_j\}_{j \in I};\ \mathscr{Q}}\ [\text{T-Sel}]$$

$$\frac{\forall i \in I \quad \Gamma \vdash_{\mathscr{D}} P_i \triangleright_{\emptyset} \Delta, s : \mathsf{unf}(T_i);\ \mathscr{Q}}{\Gamma \vdash_{\mathscr{D}} s[k] \triangleright \{l_j : P_j\}_{j \in I} \triangleright_{\emptyset} \Delta, s : k\ \&\ \{l_j : T_j\}_{j \in I};\ \mathscr{Q}}\ [\text{T-Branch}]$$

$$\frac{\Gamma \vdash_{\mathscr{D}} P \triangleright_{\emptyset} \Delta;\ \mathscr{Q} \quad \Gamma \vdash_{\mathscr{D}} Q \triangleright_{\emptyset} \Delta;\ \mathscr{Q} \quad \Gamma \vdash e : \mathsf{bool}}{\Gamma \vdash_{\mathscr{D}} \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \triangleright_{\emptyset} \Delta;\ \mathscr{Q}}\ [\text{T-If}]$$

$$\frac{\mathsf{ends}(\Delta_0) \quad \Gamma \vdash e : U \quad X : (U, \tilde{T}) \in \mathscr{D} \quad \Delta_1 \approx \tilde{s} : \tilde{T}}{\Gamma \vdash_{\mathscr{D}} X\langle e, \tilde{s} \rangle \triangleright_{\mathcal{C}} \Delta_0, \Delta_1;\ \mathscr{Q}}\ [\text{T-Var}] \qquad \frac{\mathsf{ends}(\Delta)}{\Gamma \vdash_{\mathscr{D}} \mathbf{0} \triangleright_{\emptyset} \Delta;\ \mathscr{Q}}\ [\text{T-Inact}]$$

Figure 10 Typing rules ( ): all rules implicitly include the premise $\mathsf{ends}(\mathscr{Q})$.

▶ **Example 24** (Typing process definitions). Let D contain the single declaration $X(x\mathscr{s}) = \mathscr{s}[k]!\langle x \rangle; X\langle x, \mathscr{s} \rangle$, which is typed by the environment $X : \mathsf{bool} \times \mathsf{unf}(T)$, where $T = \mu\mathbf{t}.!k\langle \mathsf{bool} \rangle.\mathbf{t}$. We name this environment $\mathscr{D}$ and to assert that D is typed by $\mathscr{D}$, we must show:

$$x : \mathsf{bool} \vdash_{\mathscr{D}} \mathscr{s}[k]!\langle x \rangle; X\langle x, \mathscr{s} \rangle \triangleright_{\emptyset} \mathscr{s} : \mathsf{unf}(T);\ \emptyset$$

Which is derivable by [T-Send] and and [T-Var].

The example above highlights an important aspect of the type system. All local types in Δ are unfolded. Note for example that we put the unfolded local type in $\mathscr{D}$. This is necessary because all rules assume local types are unfolded, and we preserve this property by only introducing unfolded local types into Δ.

Subject Reduction. We now state the properties of the mechanised type system. We start with the substitution lemma, which is required by the subject reduction theorem to derive typings for value substituted processes $P[v/x]$ and located session substituted processes $P[\mathsf{s}^{\mathsf{p}}/t]$.

▶ **Lemma 25** (Substitution).
1. *If* $\Gamma, x : U \vdash_{\mathscr{D}} P \triangleright_{\mathcal{C}} \Delta;\ \mathscr{Q}$ *and* $\Gamma \vdash v : U$ *then* $\Gamma \vdash_{\mathscr{D}} P[v/x] \triangleright_{\mathcal{C}} \Delta;\ \mathscr{Q}$.
2. *If* $\Gamma \vdash_{\mathscr{D}} P \triangleright_{\mathcal{C}} \Delta, \mathscr{s} : T;\ \mathscr{Q}$ *then* $\Gamma \vdash_{\mathscr{D}} P[\mathsf{s}^{\mathsf{p}}/\mathscr{s}] \triangleright_{\mathcal{C}} \Delta, \mathsf{s}^{\mathsf{p}} : T;\ \mathscr{Q}$.

$$\frac{\Gamma \vdash_{\mathscr{D}} P \triangleright_{\mathcal{C}_0} \Delta_0;\ \mathcal{Q}_0 \quad \Gamma \vdash_{\mathscr{D}} Q \triangleright_{\mathcal{C}_1} \Delta_1;\ \mathcal{Q}_1 \quad (\mathcal{Q}_0, \mathcal{Q}_1) \hookrightarrow \mathcal{Q}}{\Gamma \vdash_{\mathscr{D}} P \mid Q \triangleright_{\mathcal{C}_0,\mathcal{C}_1} \Delta_0, \Delta_1;\ \mathcal{Q}}\ \text{[T-Conc]}$$

$$\frac{\mathsf{ends}(\mathcal{Q}) \quad \mathsf{ends}(\Delta)}{\Gamma \vdash_{\mathscr{D}} \mathsf{s}[k] :: \emptyset \triangleright_{\mathsf{s}[k]} \Delta;\ \mathcal{Q}}\ \text{[T-QNil]} \qquad \frac{\mathsf{coherent}(G) \quad \Gamma, \mathsf{a} : G \vdash_{\mathscr{D}} P \triangleright_{C} \Delta;\ \mathcal{Q}}{\Gamma \vdash_{\mathscr{D}} (\boldsymbol{\nu}^{\mathsf{a}}\mathsf{a})P \triangleright_{\emptyset} \Delta;\ \mathcal{Q}}\ \text{[T-NRes]}$$

$$\frac{\mathsf{coherent}(\hat{\Delta}; \hat{\mathcal{Q}}) \quad \Gamma \vdash_{\mathscr{D}} P \triangleright_{(\mathcal{C}, \{\mathsf{s}[k_i]\}_{i \in I})} \Delta, [\hat{\Delta}]_{\mathsf{s}};\ \mathcal{Q}, [\hat{\mathcal{Q}}]_{\mathsf{s}}}{\Gamma \vdash_{\mathscr{D}} (\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})P \triangleright_{\mathcal{C}} \Delta;\ \mathcal{Q}}\ \text{[T-CRes]}$$

$$\frac{\Gamma \vdash v : U \quad \Gamma \vdash_{\mathscr{D}} \mathsf{s}[k] :: \tilde{h} \triangleright_{\mathcal{C}} \Delta;\ \mathcal{Q}, \mathsf{s}^{\mathsf{p}} : Q}{\Gamma \vdash_{\mathscr{D}} \mathsf{s}[k] :: v^{\mathsf{p}} \cdot \tilde{h} \triangleright_{\mathcal{C}} \Delta;\ \mathcal{Q}, \mathsf{s}^{\mathsf{p}} : (k, U) \cdot Q}\ \text{[T-QVal]}$$

$$\frac{\Gamma \vdash_{\mathscr{D}} \mathsf{s}[k] :: \tilde{h} \triangleright_{\mathcal{C}} \Delta;\ \mathcal{Q}, \mathsf{s}^{\mathsf{p}} : Q}{\Gamma \vdash_{\mathscr{D}} \mathsf{s}[k] :: l^{\mathsf{p}} \cdot \tilde{h} \triangleright_{\mathcal{C}} \Delta;\ \mathcal{Q}, \mathsf{s}^{\mathsf{p}} : (k, l) \cdot Q}\ \text{[T-QSel]}$$

$$\frac{T_0 \approx T_1 \quad \Gamma \vdash_{\mathscr{D}} \mathsf{s}[k] :: \tilde{h} \triangleright_{\mathcal{C}} \Delta;\ \mathcal{Q}, \mathsf{s}^{\mathsf{p}} : Q}{\Gamma \vdash_{\mathscr{D}} \mathsf{s}[k] :: (\mathsf{t}^{\mathsf{q}})^{\mathsf{p}} \cdot \tilde{h} \triangleright_{\mathcal{C}} \Delta, \mathsf{t}^{\mathsf{q}} : T_0;\ \mathcal{Q}, \mathsf{s}^{\mathsf{p}} : (k, T_1) \cdot Q}\ \text{[T-QSess]}$$

Figure 11 Typing rules (🐓): there are no implicit premises for these rules.

The lemma states that processes remain well-typed after variables are substituted for values and located session variables are substituted for located session identifiers. This lemma is necessary for proving the cases of subject reduction dealing with value and session reception.

We are now ready to state subject congruence and subject reduction:

▶ **Theorem 26** (Subject congruence and reduction). *Let P, P', and Q be processes, and Γ, Δ, Δ_0, and $\mathcal{Q}$ environments. Then,*

1. (🐓) *if $P \equiv Q$ and $\Gamma \vdash_{\mathscr{D}} P \triangleright_{\mathcal{C}} \Delta;\ \mathcal{Q}$ then $\Gamma \vdash_{\mathscr{D}} Q \triangleright_{\mathcal{C}} \Delta;\ \mathcal{Q}$*
2. (🐓) *if $\Gamma \vdash_{\mathscr{D}} P \triangleright_{\mathcal{C}} \Delta;\ \mathcal{Q}$, $\mathsf{coherent}(\Delta; \mathcal{Q})$ as Δ_0, and $P \ \to_D\ P'$ then there exists $\Delta_1, \Delta', \mathcal{Q}'$ s.t. $\mathsf{coherent}(\Delta'; \mathcal{Q}')$ as Δ_1 and $\Gamma \vdash_{\mathscr{D}} P' \triangleright_{\mathcal{C}} \Delta';\ \mathcal{Q}'$ and either $\Delta_0 = \Delta_1$ or there exists ℓ s.t. $\Delta_0 \xrightarrow{\ell} \Delta_1$*
3. (🐓) *if $\Gamma \vdash_{\mathscr{D}} P \triangleright_{\emptyset} \emptyset;\ \emptyset$ and $P \ \to_D\ P'$ then $\Gamma \vdash_{\mathscr{D}} P' \triangleright_{\emptyset} \emptyset;\ \emptyset$*

The theorem contains three statements: (1) typing is preserved by congruence; (2) typing is preserved by process reductions, relating environments either by equality or reduction; and, (3) typing is preserved by closed processes with no queues , which is a special case of (2). The proof of (1) proceeds by structural induction on the congruence relation, while the proof of (2), the core subject reduction result, is by induction on process reductions.

▶ Remark 27 (Differences with the original work). The type system enforces linear resource use by defining smaller resources as splits of larger ones, contrasting with Honda et al., who define larger resources as compositions of smaller ones. Our focus on splitting rather than composing is evident in several aspects of the type system. Firstly, the environment $\Delta; \mathcal{Q}$ provides an intuitive temporal split: queue types in $\mathcal{Q}$ indicate completed actions, while residual local types in Δ indicate remaining actions. Keeping these concepts distinct simplifies the definition of environment splitting, as Δ and $\mathcal{Q}$ split orthogonally. Secondly, queue types and residual local types are obtained by decomposition. Our definition allows us to avoid using the subtyping relation, unlike Honda et al. This is significant because introducing subtyping through a non-structural subsumption rule complicates reasoning about type derivations, particularly in proof assistants.

$$
\begin{array}{ll}
\mathbf{error}(\mathsf{s}^{\mathsf{p}}[k]?(x); P \mid \mathsf{s}^{\mathsf{q}}[k]?(x); Q) & \mathbf{error}(\mathsf{s}^{\mathsf{p}}[k]!\langle e\rangle; P \mid \mathsf{s}^{\mathsf{q}}[k]!\langle e'\rangle; Q) \\
\mathbf{error}(\mathsf{s}^{\mathsf{p}}[k]?(x); P \mid \mathsf{s}^{\mathsf{q}}[k] \rhd \{l_j : Q_j\}_{i\in J}) & \mathbf{error}(\mathsf{s}^{\mathsf{p}}[k]!\langle e\rangle; P \mid \mathsf{s}^{\mathsf{q}}[k] \lhd l; Q) \\
\mathbf{error}(\mathsf{s}^{\mathsf{p}}[k]?(\!(\mathscr{s})\!); P \mid \mathsf{s}^{\mathsf{q}}[k]?(\!(\mathscr{s})\!); Q) & \mathbf{error}(\mathsf{s}^{\mathsf{p}}[k]!\langle e\rangle; P \mid \mathsf{s}^{\mathsf{q}}[k]!\langle\!\langle t\rangle\!\rangle; Q) \\
\mathbf{error}(\mathsf{s}^{\mathsf{p}}[k] \rhd \{l_i : P_i\}_{i\in I} \mid \mathsf{s}^{\mathsf{q}}[k] \rhd \{l_j : Q_j\}_{j\in J}) & \mathbf{error}(\mathsf{s}^{\mathsf{p}}[k] \lhd l;_1 P \mid \mathsf{s}^{\mathsf{q}}[k] \lhd l;_2 Q) \\
\mathbf{error}(\mathsf{s}^{\mathsf{p}}[k] \rhd \{l_i : P_i\}_{i\in I} \mid \mathsf{s}^{\mathsf{q}}[k]?(\!(\mathscr{s})\!); Q) & \mathbf{error}(\mathsf{s}^{\mathsf{p}}[k] \lhd l; P \mid \mathsf{s}^{\mathsf{q}}[k]!\langle\!\langle t\rangle\!\rangle; Q) \\
\mathbf{error}(\mathsf{s}^{\mathsf{p}}[k]?(\!(\mathscr{s})\!); P \mid \mathsf{s}^{\mathsf{q}}[k]?(\!(\mathscr{s})\!); Q) & \mathbf{error}(\mathsf{s}^{\mathsf{p}}[k]!\langle\!\langle t\rangle\!\rangle; P \mid \mathsf{s}^{\mathsf{q}}[k]!\langle\!\langle u\rangle\!\rangle; Q)
\end{array}
$$

Figure 12 The **error** relation (🐓): violation of linearity (race on the same channel).

$$
\begin{array}{ll}
\mathbf{error}(\mathsf{s}^{\mathsf{p}}[k]?(x); P \mid \mathsf{s}[k] :: {l_j}^{\mathsf{q}} \cdot \tilde{h}) & \\
\mathbf{error}(\mathsf{s}^{\mathsf{p}}[k]?(x); P \mid \mathsf{s}[k] :: (\mathsf{t}^{\mathsf{r}})^{\mathsf{q}} \cdot \tilde{h}) & \dfrac{\mathbf{error}(P)}{\mathbf{error}((\boldsymbol{\nu}^{\mathsf{a}}\mathsf{a})P)} \\
\mathbf{error}(\mathsf{s}^{\mathsf{p}}[k] \rhd \{l_i : P_i\}_{i\in I} \mid \mathsf{s}[k] :: v^{\mathsf{q}} \cdot \tilde{h}) & \\
\mathbf{error}(\mathsf{s}^{\mathsf{p}}[k] \rhd \{l_i : P_i\}_{i\in I} \mid \mathsf{s}[k] :: (\mathsf{t}^{\mathsf{r}})^{\mathsf{q}} \cdot \tilde{h}) & \dfrac{\mathbf{error}(P)}{\mathbf{error}((\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})P)} \\
\mathbf{error}(\mathsf{s}^{\mathsf{p}}[k]?(\!(\mathscr{s})\!); P \mid \mathsf{s}[k] :: v^{\mathsf{q}} \cdot \tilde{h}) & \\
\mathbf{error}(\mathsf{s}^{\mathsf{p}}[k]?(\!(\mathscr{s})\!); P \mid \mathsf{s}[k] :: {l_j}^{\mathsf{q}} \cdot \tilde{h}) & \dfrac{\mathbf{error}(P)}{\mathbf{error}(P \mid Q)}
\end{array}
$$

Figure 13 The **error** relation (🐓): communication error and structural rules.

Communication Safety. We conclude this section by presenting communication safety, the property that states that a well-typed process never goes wrong. We first define a notion of error, in order to capture processes behaviour that we consider erroneous:

▶ **Definition 28** ($\mathbf{error}_{\equiv}$ 🐓)**.** *The relation* **error** *on processes is the minimal relation satisfying the rules in Figure 12 and Figure 13. Then,* $\mathbf{error}_{\equiv}(P)$ *is defined as* $\exists Q.\ P \equiv Q \wedge \mathbf{error}(Q)$*.*

The **error** relation captures situations in which processes cause a run-time error. Following the original work, we have categorised errors into two groups. The rules in Figure 12 address situations where linearity is violated. In session types, linearity ensures that there is never a race condition on a session channel. Such an error situation can arise in different ways, notably when two inputs or outputs competing on the same channel are ready to be executed (recall the notion of linearity of global types). The second type of error is a communication error, where a process reads from a queue a message of the wrong kind, e.g., it reads a label when a delegated session is expected. These cases are presented in Figure 13, along with three structural rules that permit lifting the relation under restriction or parallel composition.

The predicate $\mathbf{error}_{\equiv}(P)$ lifts the error relation to structural congruence in order to capture different structurally congruent processes. This design choice (as opposed to including a structural congruence rule in the definition of **error**) was made to simplify our mechanised proof of communication safety. Note that we can always rewrite a process into the form $(\nu x_1 \dots x_n)(P \mid Q \mid R)$, where P and Q are the components to be checked for errors. Restriction rules are then applied to eliminate binders, and parallel rules are used to strip away the R component. We formally state communication safety as:

▶ **Theorem 29** (Communication Safety 🐓)**.** *If* $\Gamma \vdash_{\mathcal{D}} P \triangleright_{\mathcal{C}} \Delta;\ \mathcal{Q}$ *and* $\mathsf{coherent}(\Delta; \mathcal{Q})$ *as* Δ_0 *then it is never the case that* $\mathbf{error}_{\equiv}(P)$*.*

From the previous theorem and subject reduction, we obtain the following:

▶ **Corollary 30** (Safety Preservation). *If* $\Gamma \vdash_{\mathcal{D}} P \triangleright_{C} \Delta;\ \mathcal{Q}$*,* $\mathsf{coherent}(\Delta;\mathcal{Q})$ *as* Δ_0*, and* $P \rightarrow_{D} P'$ *then it is never the case that* $\mathbf{error}_{\equiv}(P')$*.*

▶ Remark 31 (Differences with the original work). Our approach to defining communication safety differs from that of the original work by Honda et al. While we structurally define what constitutes an erroneous process and negate this property to derive safety, they define safety directly in terms of available redexes in the processes. In their approach to safety, when a redex is present in a session, session channels must occur linearly, and the messages read from queues must conform to the expectations of the receiving process. We departed from their non-structural approach in favour of an error predicate, as this proved more convenient for mechanisation, allowing us to perform induction on the predicate.

Given that the original formulation is not entirely formal, we have not mechanised the original definition nor demonstrated whether the two approaches are equivalent. However, it is worth noting that the use of an error predicate for defining safety is fairly standard and, for example, was employed in the original work on binary session types by Honda et al. [31].

6 Related Work and Discussion

Related Work. We discuss the most relevant related work, with a particular focus on mechanisations of session types.

Mechanisation of multiparty session types. Castro-Perez et al. [13] mechanise in Coq a trace equivalence between processes, coinductive local types and coinductive global types. Their processes, equipped with a trace semantics and no structural congruence, use implicit channels and are single-session without delegation; therefore, there is no multi-cast session request nor session acceptance and a process is always typed by a single global type. As a consequence, subject reduction is replaced by showing that processes and types have the same traces. Their process language is a domain specific language embedded in Coq. This allows a user to write a correct-by-construction process in Coq for each role of a session, and use Coq's extraction feature to extract executable OCaml code. The generated code has a transport API that allows interaction with external code. While their setting is more applied than ours, they also propose an elegant approach to representing session types as coinductive types that are coinductively related to unfolded μ-types. Castro-Perez et al. prove their definition of projection sound with respect to a coinductive specification of projection by Ghilezan et a. [24]. For a similar setting but with explicit channels, Tirore et al. [51] propose a projection for which both soundness and completeness hold: this is the projection that we adopt in our mechanisation.

Jacobs et al. [35] mechanise multiparty GV in Coq using the Iris framework [37]. Multiparty GV is an extension of the linear lambda calculus with multiparty asynchronous session types with implicit channels. Their language, equipped with a thread pool semantics and no structural congruence, features an n-ary fork operator that combines session initiation and process spawning, ensuring an acyclic topology, therefore ensuring progress [8]. This is unlike our setting where already existing processes can initiate sessions on shared names. Their methodology is significantly different from ours, and is based on separation logic [45, 46] and connectivity graphs [34]. They define an invariant on runtime configurations, ensuring well-typedness and buffer consistency. They then prove preservation of this invariant under operational semantics and show that configurations satisfying the invariant cannot deadlock.

Mechanisation of binary session types. Using the Iris framework, Hinrichsen et al. [29] introduce semantic typing for a variant of binary session types using logical relations. Gay et al. [23] compare duality definitions in an Agda mechanisation of binary session types.

Duality is the binary notion of compatibility used for binary session types, that is a special case of the multiparty notion of coherence. Gay et al. demonstrate that some definitions are unsound when type variables can appear in messages. Like them, our work does not take the equi-recursive view and we explicitly state when syntactic equality of μ-types is used and thus when to trigger unfoldings and when we use coinductive equality modulo unfolding. Castro-Perez et al. [14] mechanise in Coq a subject reduction theorem for the binary session type meta-theory given by Yoshida and Vasconcelos [54]. They use the locally nameless representation for variables by Charguéraud [15]. Like us, Castro-Perez et al. distinguish between variable and value session channels. Unlike our setting, theirs does not include recursion nor process call and definitions. Instead of process call, they use replication $!P$. Their definition of structural congruence lacks commutativity and transitivity which, although it makes some simple reductions impossible, simplifies substantially the subject reduction proof. Recently, Sano et al. [48] mechanised a version of binary session types that are in a Curry-Howard correspondence with linear logic [52]. They use linearity predicates, which enable the separate checking of linearity properties in the type system, allowing the typing judgement to focus solely on non-linear contexts.

Session types on paper. There is a vast literature on multiparty session types that builds on the original paper. An introduction was given by Yoshida and Gheri [53] and a comprehensive overview was given by Scalas and Yoshida [49] where they argue the inconvenience of using global types to define compatibility. It should be noted that, to the best of our knowledge, all later work uses implicit channels and, therefore, the counter example does not invalidate these results. However, as argued in the introduction, explicit channels are more expressive and particularly useful in applications where only a limited number of channels are available, e.g., AMQP [3]. A binary version of session types has also been applied to type a choreographic language, i.e., a synchronous process-level language with global interactions that features explicit channels [9, 10]. Like for global types, choreographies must undergo a well-formedness check, which resembles a more restrictive linearity analysis for global types.

Discussion. We discuss our contributions and potential further developments.

On the Mechanisation in Coq. Our mechanisation in Coq consists of over 20K lines of code and features several design choices to address the complexity of the theory of multiparty session types. For handling binders, we employed the *Autosubst2* library, developed in Stark's PhD thesis [50]. From a user-defined signature in higher-order abstract syntax (HOAS), Autosubst2 generates Coq code with an inductive definition that uses a de Bruijn representation, along with substitution operations on this datatype. We reuse the definitions in the work by Tirore et al [51] for representing global types, local types and the projection function. We reused their code off-the-shelf, significantly streamlining our development.

For representing typing environments, we adopted a straightforward approach using lists of pairs $(\text{name}, \text{type})$. Other common forms of representation include lists with indices or functions from names to types. We found our approach more convenient to use for parallel substitutions, which is the kind of substitution operation that Autosubst2 is based on. In particular, while low-level, it simplified context splitting for parallel composition rules, requiring only standard checks for linearity of name usage, as dictated by linear type systems. We believe this representation offers an accessible starting point for researchers aiming to mechanise session types. Finally, our work heavily relies on a combination of *inductive* and *coinductive* definitions, which we implemented using the Paco library [16], which proved invaluable for developing our mechanisation.

On Communication Safety, Session Fidelity, and Progress. Subject reduction serves as a critical internal lemma that can be used to prove many key results, including communication safety (also known as type safety), session fidelity, and progress. The primary goal of this paper is to prove subject reduction and uncover the issues in the original theorem by Honda et al., demonstrating its incorrectness; establishing communication safety, which ensures that well-typed processes do not exhibit wrong behaviour, was an additional achievement. On the other hand, other results, though highly interesting, lie beyond the intended scope of this work. Session fidelity expresses that communications within a session progress according to the global type. While we have not mechanised session fidelity as a formal theorem, addressing this remains a promising avenue for future research. Proving session fidelity would require augmenting the reduction semantics with labels to specify the communication performed and the channels involved, enabling a finer-grained analysis of interaction behaviours.

Another key result from Honda et al. is progress, which asserts that a well-typed single-session process never deadlocks. In our work, linearity of global types plays a crucial role in ensuring that interleavings of send and receive operations adhere to the global type. However, our counterexample raises questions about the implications of explicit channels and causality on progress. Investigating these implications and formally proving progress for our system would further strengthen our meta-theoretical framework.

Finally, our results offer a foundation for extending our work to mechanise more advanced properties of session-typed systems, particularly in multi-session scenarios. Explicit channels, as demonstrated by our counterexample, introduce non-trivial distinctions in causality that deserve further exploration.

Alternative Semantics for Global and Local Types. The counterexample highlighted the restrictiveness of the global type semantics. Later work on multiparty session types [49, 13] introduces a more flexible global type semantics that, in the context of implicit channels, is unrelated to the counterexample. This work suggests semantic definitions that could allow the unstuck predicate to be removed from the meta-theory.

Other Observations. The mechanisation revealed some inconveniences in the meta-theory. For example, the representation of a finished session channel is ambiguous; it can be represented either as not being contained in the environment or as being contained but mapped to `end`. We conjecture that it would simplify proofs to resolve this ambiguity by choosing only the non-presence representation in conjunction with a closure operation on sessions, as done by Caires et al. [7] and Wadler [52].

Another inconvenience is the use of global types as the basis for multiparty compatibility. Scalas and Yoshida [49] pointed out that more general notions of compatibility exist beyond global types. This broader perspective may not only provide a more encompassing definition of compatibility but also simplify proofs by eliminating the need for two specification languages in the meta-theory.

References

1 Scribble: Describing Multy Party Protocols. URL: `http://www.scribble.org`.

2 Session Types in Programming Languages: A Collection of Implementations. URL: `http://groups.inf.ed.ac.uk/abcd/session-implementations.html`.

3 Advanced Message Queuing Protocol. `http://www.iona.com/opensource/amqp/`, 2015.

4 Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically interleaved multiparty sessions. In Franck van Breugel and Marsha Chechik, editors, *CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings*, volume 5201 of *Lecture Notes in Computer Science*, pages 418–433. Springer, 2008. `doi:10.1007/978-3-540-85361-9_33`.

5 Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. A formally-verified C compiler supporting floating-point arithmetic. In Alberto Nannarelli, Peter-Michael Seidel, and Ping Tak Peter Tang, editors, *21st IEEE Symposium on Computer Arithmetic*, pages 107–115. IEEE Computer Society, 2013. doi:10.1109/ARITH.2013.30.

6 Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. Undecidability of asynchronous session subtyping. *Inf. Comput.*, 256:300–320, 2017. doi:10.1016/J.IC.2017.07.010.

7 Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Math. Struct. Comput. Sci.*, 26(3):367–423, 2016. doi:10.1017/S0960129514000218.

8 Marco Carbone and Søren Debois. A graphical approach to progress for structured communication in web services. In Simon Bliudze, Roberto Bruni, Davide Grohmann, and Alexandra Silva, editors, *Proceedings Third Interaction and Concurrency Experience: Guaranteed Interaction, ICE 2010, Amsterdam, The Netherlands, 10th of June 2010*, volume 38 of *EPTCS*, pages 13–27, 2010. doi:10.4204/EPTCS.38.4.

9 Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In Rocco De Nicola, editor, *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4421 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2007. doi:10.1007/978-3-540-71316-6_2.

10 Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8:1–8:78, 2012. doi:10.1145/2220365.2220367.

11 Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. Coherence generalises duality: A logical explanation of multiparty session types. In Josée Desharnais and Radha Jagadeesan, editors, *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*, volume 59 of *LIPIcs*, pages 33:1–33:15, Germany, 2016. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CONCUR.2016.33.

12 Marco Carbone, Sonia Marin, and Carsten Schürmann. A logical interpretation of asynchronous multiparty compatibility. In Robert Glück and Bishoksan Kafle, editors, *Logic-Based Program Synthesis and Transformation - 33rd International Symposium, LOPSTR 2023, Cascais, Portugal, October 23-24, 2023, Proceedings*, volume 14330 of *Lecture Notes in Computer Science*, pages 99–117. Springer, 2023. doi:10.1007/978-3-031-45784-5_7.

13 David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, and Nobuko Yoshida. Zooid: a DSL for certified multiparty computation: from mechanised metatheory to certified multiparty processes. In *Proceedings of PLDI*, pages 237–251. ACM, 2021. doi:10.1145/3453483.3454041.

14 David Castro-Perez, Francisco Ferreira, and Nobuko Yoshida. EMTST: engineering the meta-theory of session types. In Armin Biere and David Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II*, volume 12079 of *Lecture Notes in Computer Science*, pages 278–285. Springer, 2020. doi:10.1007/978-3-030-45237-7_17.

15 Arthur Charguéraud. The locally nameless representation. *J. Autom. Reason.*, 49(3):363–408, 2012. doi:10.1007/S10817-011-9225-2.

16 Chung-Kil, Georg Neis Georg, Derek Dreyer, and Victor Vafeiadis. The power of parameterization in coinductive proof. *SIGPLAN Notices*, 48(1):193–206, 2013. doi:10.1145/2480359.2429093.

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

18 Nils Anders Danielsson and Thorsten Altenkirch. Subtyping, declaratively. In Claude Bolduc, Josée Desharnais, and Béchir Ktari, editors, *Mathematics of Program Construction, 10th International Conference, MPC 2010, Québec City, Canada, June 21-23, 2010. Proceedings*, volume 6120 of *Lecture Notes in Computer Science*, pages 100–118. Springer, 2010. doi:10.1007/978-3-642-13321-3_8.

19 Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata. In Helmut Seidl, editor, *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7211 of *Lecture Notes in Computer Science*, pages 194–213. Springer, 2012. doi:10.1007/978-3-642-28869-2_10.

20 Simon Fowler. An erlang implementation of multiparty session actors. In Massimo Bartoletti, Ludovic Henrio, Sophia Knight, and Hugo Torres Vieira, editors, *Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, 8-9 June 2016*, volume 223 of *EPTCS*, pages 36–50, 2016. doi:10.4204/EPTCS.223.3.

21 Simon J. Gay and Malcolm Hole. Types and subtypes for client-server interactions. In S. Doaitse Swierstra, editor, *Programming Languages and Systems, 8th European Symposium on Programming, ESOP'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, 22-28 March, 1999, Proceedings*, volume 1576 of *Lecture Notes in Computer Science*, pages 74–90. Springer, 1999. doi:10.1007/3-540-49099-X_6.

22 Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3):191–225, 2005. doi:10.1007/S00236-005-0177-Z.

23 Simon J. Gay, Peter Thiemann, and Vasco T. Vasconcelos. Duality of session types: The final cut. In Stephanie Balzer and Luca Padovani, editors, *Proceedings of the 12th International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2020, Dublin, Ireland, 26th April 2020*, volume 314 of *EPTCS*, pages 23–33, 2020. doi:10.4204/EPTCS.314.3.

24 Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic, Alceste Scalas, and Nobuko Yoshida. Precise subtyping for synchronous multiparty sessions. *Journal of Logical and Algebraic Methods in Programming*, 104:127–173, 2019. doi:10.1016/j.jlamp.2018.12.002.

25 Georges Gonthier. The four colour theorem: Engineering of a formal proof. In Deepak Kapur, editor, *Computer Mathematics, 8th Asian Symposium, ASCM*, volume 5081 of *Lecture Notes in Computer Science*, page 333. Springer, 2007. doi:10.1007/978-3-540-87827-8_28.

26 Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving - 4th International Conference, ITP*, volume 7998 of *LNCS*, pages 163–179. Springer, 2013. doi:10.1007/978-3-642-39634-2_14.

27 Thomas C. Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason M. Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller. A formal proof of the kepler conjecture. *CoRR*, abs/1501.02155, 2015. arXiv:1501.02155.

28 Jesse Michael Han and Floris van Doorn. A formal proof of the independence of the continuum hypothesis. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 353–366. ACM, 2020. doi:10.1145/3372885.3373826.

29 Jonas Kastberg Hinrichsen, Daniël Louwrink, Robbert Krebbers, and Jesper Bengtson. Machine-checked semantic session typing. In Catalin Hritcu and Andrei Popescu, editors, *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, pages 178–198. ACM, 2021. doi:10.1145/3437992.3439914.

30 Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993. doi:10.1007/3-540-57208-2_35.

31 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *Proceedings of ESOP*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998. doi:10.1007/BFb0053567.

32 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of POPL*, pages 273–284. ACM, 2008. doi:10.1145/1328438.1328472.

33 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *Journal of the ACM*, 63(1):9:1–9:67, 2016. doi:10.1145/2827695.

34 Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Connectivity graphs: a method for proving deadlock freedom based on separation logic. *Proc. ACM Program. Lang.*, 6(POPL):1–33, 2022. doi:10.1145/3498662.

35 Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Multiparty GV: functional multiparty session types with certified deadlock freedom. *Proceedings of the ACM on Programming Languages*, 6(ICFP):466–495, 2022. doi:10.1145/3547638.

36 Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David A. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an os kernel. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 207–220. ACM, 2009. doi:10.1145/1629575.1629596.

37 Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 205–217. ACM, 2017. doi:10.1145/3009837.3009855.

38 Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. A static verification framework for message passing in go using behavioural types. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 1137–1148. ACM, 2018. doi:10.1145/3180155.3180157.

39 Julien Lange and Nobuko Yoshida. On the undecidability of asynchronous session subtyping. In Javier Esparza and Andrzej S. Murawski, editors, *Foundations of Software Science and Computation Structures - 20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10203 of *Lecture Notes in Computer Science*, pages 441–457, 2017. doi:10.1007/978-3-662-54458-7_26.

40 SIPLAN selection committee. Most Influential POPL Paper Award (2018). https://www.sigplan.org/Awards. Accessed: July 2024.

41 The Coq development team. The Coq Proof Assistant. https://coq.inria.fr. Accessed: July 2024.

42 The Lean development team. The Lean Proof Assistant. https://lean-lang.org. Accessed: July 2024.

43 Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40,41–77, September 1992.

44 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. doi:10.1007/3-540-45949-9.

45 Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bull. Symb. Log.*, 5(2):215–244, 1999. doi:10.2307/421090.

46 Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg, editor, *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001. doi:10.1007/3-540-44802-0_1.

47 Benjamin C. Pierce. *Types and programming languages.* MIT Press, 2002.

48 Chuta Sano, Ryan Kavanagh, and Brigitte Pientka. Mechanizing session-types using a structural view: Enforcing linearity without linearity. *Proc. ACM Program. Lang.*, 7(OOPSLA2):374–399, 2023. doi:10.1145/3622810.

49 Alceste Scalas and Nobuko Yoshida. Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.*, 3(POPL):30:1–30:29, 2019. doi:10.1145/3290343.

50 Kathrin Stark. *Mechanising Syntax with Binders in Coq.* Phd thesis, Universität des Saarlandes, Fakultät für Mathematik und Informatik, Saarbrücken, Germany, 2019.

51 Dawit Legesse Tirore, Jesper Bengtson, and Marco Carbone. A sound and complete projection for global types. In Adam Naumowicz and René Thiemann, editors, *14th International Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023, Białystok, Poland*, volume 268 of *LIPIcs*, pages 28:1–28:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.ITP.2023.28.

52 Philip Wadler. Propositions as sessions. In *Proceedings of ICFP*, pages 273–286. ACM, 2012. doi:10.1145/2364527.2364568.

53 Nobuko Yoshida and Lorenzo Gheri. A very gentle introduction to multiparty session types. In Dang Van Hung and Meenakshi D'Souza, editors, *Distributed Computing and Internet Technology - 16th International Conference, ICDCIT 2020, Bhubaneswar, India, January 9-12, 2020, Proceedings*, volume 11969 of *Lecture Notes in Computer Science*, pages 73–93. Springer, 2020. doi:10.1007/978-3-030-36987-3_5.

54 Nobuko Yoshida and Vasco Thudichum Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. In Maribel Fernández and Claude Kirchner, editors, *Proceedings of the First International Workshop on Security and Rewriting Techniques, SecReT@ICALP 2006, Venice, Italy, July 15, 2006*, volume 171(4) of *Electronic Notes in Theoretical Computer Science*, pages 73–93. Elsevier, 2006. doi:10.1016/J.ENTCS.2007.02.056.