A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming*

Alceste Scalas[1], Ornela Dardha[2], Raymond Hu[3], and Nobuko Yoshida[4]

1 Imperial College London, UK
alceste.scalas@imperial.ac.uk

2 University of Glasgow, UK
ornela.dardha@glasgow.ac.uk

3 Imperial College London, UK
raymond.hu@imperial.ac.uk

4 Imperial College London, UK
n.yoshida@imperial.ac.uk

Abstract

Multiparty Session Types (MPST) is a typing discipline for message-passing distributed processes that can ensure properties such as absence of communication errors and deadlocks, and protocol conformance. Can MPST provide a theoretical foundation for concurrent and distributed programming in "mainstream" languages? We address this problem by *(1)* developing the first encoding of a *full-fledged* multiparty session π-calculus into linear π-calculus, and*(2)* using the encoding as the foundation of a practical toolchain for safe multiparty programming in Scala.

Our encoding is type-preserving and operationally sound and complete. Crucially, it keeps the distributed *choreographic* nature of MPST, illuminating that the safety properties of multiparty sessions can be precisely represented with a decomposition into *binary linear channels*. Previous works have only studied the relation between (limited) multiparty and binary sessions via centralised *orchestration* means. We exploit these results to implement an automated generation of Scala APIs for multiparty sessions, abstracting existing libraries for binary communication channels. This allows multiparty systems to be safely implemented over binary message transports, as commonly found in practice. Our implementation is the first to support *distributed multiparty delegation*: our encoding yields it for free, via existing mechanisms for binary delegation.

1998 ACM Subject Classification D.1.3 Concurrent Programming; D.3.1 Formal Definitions and Theory; F.3.3 Studies of Program Constructs — Type structure

Keywords and phrases process calculi, session types, concurrent programming, Scala

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.24

Supplementary Material ECOOP Artifact Evaluation approved artifact available at http://dx.doi.org/10.4230/DARTS.3.2.3

1 Introduction

Correct design and implementation of concurrent and distributed applications is notoriously difficult. Programmers must confront challenges involving *protocol conformance* (are messages

[*] Partially supported by EPSRC (grants EP/K034413/1, EP/K011715/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1) and EU (FP7 612985 "Upscale"). Dardha was awarded a SICSA PECE bursary for visiting Imperial College London in January–March 2016.

31st European Conference on Object-Oriented Programming (ECOOP 2017).
Editor: Peter Müller; Article No. 24; pp. 24:1–24:31
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

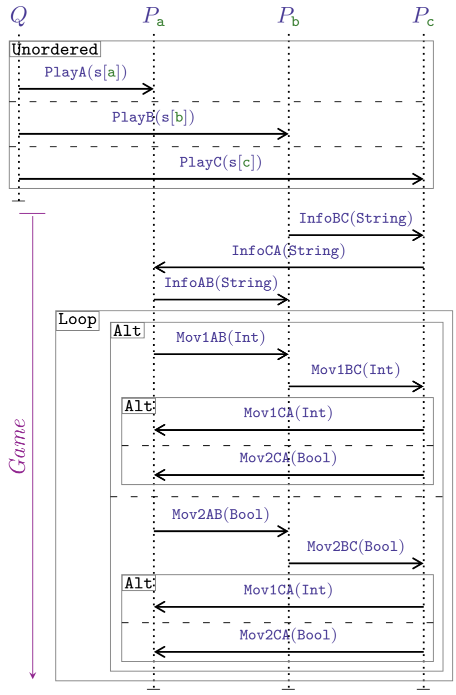

Figure 1 Game server with 3 clients.

sent/received according to a specification?) and *communication mechanics* (how are the interactions actually performed?). These difficulties are worsened by the potential complexity of interactions among *multiple* participants, and if the *communication topology* is not fixed.

For example, consider a common scenario for a peer-to-peer multiplayer game: the clients, initially unknown to each other, connect to a "matchmaking" server, whose task is to group players and setup a game session in which they can interact directly. Figure 1 depicts this scenario: Q is the server, connected to three clients P_a, P_b and P_c. To set up a game, Q sends to each client some networking information (denoted by $s[a]/s[b]/s[c]$, payloads of the PlayA/B/C messages) to "introduce" the clients to each other and allow them to communicate. Then, the clients follow the game protocol (marked as "*Game*"), consisting in some initial message exchanges (Info), and a game loop: P_a chooses a message to send to P_b (Mov1AB or Mov2AB) followed by a message from P_b to P_c, who chooses which message send back to P_a.

Figure 1 features structured protocols with inter-role message dependencies, and a dynamic communication topology (starting client-to-server, becoming client-to-client). Implementing them is not easy: programmers would benefit from tools to *statically* detect protocol violations in source code, and realise the communication topology changes.

Multiparty Session Types (MPST) [27] are a theoretical framework for channel-based communication, capable of modelling our example. In MPST, participants are modelled as *roles* (e.g., game players a, b, c) and programs are *session π-calculus processes*; the "networking information payloads" $s[a]/s[b]/s[c]$ can be modelled as *multiparty channels*, for interpreting roles a/b/c on the game *session* s. Notably, channels can *themselves* be sent/received: this allows to *delegate* a multiparty interaction to another process, thus changing the communicating topology. In Figure 1, the server Q sends (i.e., delegates) the channel $s[b]$ to P_b; the latter can then use $s[b]$ to interact with the processes owning channels $s[a]$ and $s[c]$ (i.e., P_a and P_c, after two more delegations).

The MPST framework formalises protocols as *session types*: structured sequences of inputs/outputs and choices. The MPST typing system assigns such types to channels, and checks the processes using them. In our example, channel $s[\mathtt{b}]$ could have type:

$$\begin{aligned} S_{\mathtt{b}} \quad = \quad & \mathtt{c}!\mathtt{InfoBC}(\mathtt{String}) \,.\, \mathtt{a}?\mathtt{InfoAB}(\mathtt{String}) \,. \\ & \qquad \mu\mathbf{t}.\big(\mathtt{a} \,\&\, \{\, ?\mathtt{Mov1AB}(\mathtt{Int}).\mathtt{c}!\mathtt{Mov1BC}(\mathtt{Int}).\mathbf{t} \;,\; ?\mathtt{Mov2AB}(\mathtt{Bool}).\mathtt{c}!\mathtt{Mov2BC}(\mathtt{Bool}).\mathbf{t} \,\}\big) \end{aligned}$$

$S_{\mathtt{b}}$ says that $s[\mathtt{b}]$ must be used to realise the *Game* interactions of $P_{\mathtt{b}}$ in Figure 1: first to send $\mathtt{InfoBC}(\mathtt{String})$ to $\mathtt{c}$, then receive $\mathtt{InfoAB}$ from $\mathtt{a}$, then enter the recursive game "loop" $\mu\mathbf{t}.(\ldots)$. Inside the recursion, $\mathtt{a} \,\&\, \{\ldots\}$ is a *branching from* $\mathtt{a}$: depending on $\mathtt{a}$'s choice, the channel will deliver either $\mathtt{Mov1AB}(\mathtt{Int})$ (in which case, it must be used to send $\mathtt{Mov1BC}(\mathtt{Int})$ to $\mathtt{c}$, and loop), or $\mathtt{Mov2AB}$ (then, it must be used to send $\mathtt{Mov2BC}$ to $\mathtt{c}$, and loop). Analogous types can be assigned to $s[\mathtt{a}]$ and $s[\mathtt{c}]$. *Delegation* is represented by types like $\mathtt{q}?\mathtt{PlayB}(S_{\mathtt{b}}).\mathbf{end}$, meaning: from role $\mathtt{q}$, receive a message $\mathtt{PlayB}$ carrying a channel that must be used according to $S_{\mathtt{b}}$ above; then, $\mathbf{end}$ the session. Session type checking ensures that, e.g., process $P_{\mathtt{b}}$ uses its channels abiding by the types above — thus safely implementing the expected channel dynamics and fulfilling role $\mathtt{b}$ in the game. Finally, MPST can formalise the whole *Game* protocol in Figure 1 as a *global type*, and validate that it is *deadlock-free*; then, via typing, ensure that a set of processes interacts according to the global type (and is, thus, deadlock-free).

MPST in practice: challenges. MPST could offer a promising formal foundation for *safe distributed programming*, helping to develop type-safe and deadlock-free concurrent programs. However, bridging the gap between theory and implementation raises several challenges:

C1 Multiparty sessions can have 2, 3 *or more* interacting roles; but in practice, communication occurs over *binary* channels (e.g., TCP sockets). Can multiparty channels be implemented as compositions of binary channels, preserving their type safety properties?

C2 MPST are far from the types of "mainstream" programming languages, as shown by $S_{\mathtt{b}}$ above. Can they be rendered, e.g., as objects? If so, what are their API and internals?

C3 How should *multiparty delegation* be realised, especially in *distributed* settings?

The current state-of-the-art has not addressed these challenges. On one hand, existing theoretical works on encoding multiparty sessions into binary sessions [8, 9] introduce centralised *medium* (or *arbiter*) processes to *orchestrate* the interactions between the multiparty session roles: hence, they depart from the choreographic (i.e., decentralised) nature of the MPST framework [27], and preclude examples like our peer-to-peer game in Figure 1. On the other hand, there are *no* existing implementations of full-fledged MPST; e.g., [57, 32, 33, 42, 52, 61, 55] only support *binary* sessions, while none of [29, 64, 17, 20] support session delegation.

Our approach. In this work, we tackle the three challenges above with a two-step strategy:

S1 we give the first *choreographic* encoding of a *"full" MPST calculus* into *linear π-calculus*;

S2 we implement a *multiparty session API generation* for Scala, based on our encoding.

By step **S1**, we formally address challenge **C1**. Linear π-calculus provides a theoretical framework with typed channels that cater only for *binary* communication, and may only be used *once* for input/output. These "limitations" are key to the practicality of our approach. In fact, they force us to figure out whether *multiparty channels can be represented by a decomposition into binary channels* — and whether *multiparty session types can be represented by a decomposition into linear types.* To solve these issues, we need study how to "decompose" the intricate MPST theory in (much simpler) π-calculus terms. This endeavour was not tackled before, and its feasibility was unclear. Its practical payoff is that linear π-calculus channels/types are amenable for an (almost) direct object-based representation (shown in [61]): this tackles challenge **C2**. Further, using π-calculus we can *prove* whether such a decomposition is "correct", i.e., whether MPST processes can be encoded to only interact on *binary* channels, *preserving their type-safety and behaviour* and "inheriting" deadlock-freedom.

In step **S2**, we generate high-level typed APIs for multiparty session programming, ensuring their "correctness" by reflecting the types and process behaviours formalised in step **S1**. Following the binary decomposition in step **S1**, we can implement such APIs as a layer over *existing* libraries for binary sessions (available for Java [30], Haskell [57, 32, 42], Links [44], Rust [33], Scala [61], ML [55]), in a way that solves challenge **C3** "for free".

Contributions. We present the *first* encoding (Section 5) of a full multiparty session π-calculus (Section 2) into standard π-calculus with linear, labelled tuple and variant types (Section 3).

- We present a novel, streamlined MPST formulation, sharply separating global/local typing. Using this formulation, we "close the gaps" between the intricacies of the MPST theory and the (much simpler) π-calculus, and spot a longstanding issue with *type merging* [18] (Definition 2.9, Section 2.1 "On Consistency"). We fix it, with a *revised subject reduction* (Theorem 2.16).
- At the heart of our encoding there is the discovery that the *type safety* property of MPST is *precisely* characterised as a *decomposition* into linear π-calculus types (Theorem 6.3). Our encoding of *types* preserves *duality* and *subtyping* (Theorem 6.1);our encoding of *processes* is *type-preserving* and *operationally sound and complete* (Theorem 6.2 and Theorem 6.5).
- We subsume the encodings of *binary* sessions into π-calculus [14, 15], and support *recursion* (Section 4), which was not properly handled in [13]. Further, we show that multiparty sessions can be encoded into binary sessions *choreographically*, i.e., while *preserving process distribution* (homomorphically w.r.t. parallel composition), in contrast to [8, 9].

In Section 7, we use our encoding as formal basis for the *first implementation of multiparty sessions* supporting *distributed multiparty delegation*, over existing Scala libraries (**paper's artifact**[1]).

Conventions. Derivations use *single*/*double* lines for *inductive*/*coinductive* rules. Recursive types $\mu\mathbf{t}.T$ are always *closed*, and *guarded*: e.g., $\mu\mathbf{t}_1\ldots\mu\mathbf{t}_n.\mathbf{t}_1$ is not a type. We define $\mathrm{unf}(\mu\mathbf{t}.T) = \mathrm{unf}(T\{^{\mu\mathbf{t}.T}/_{\mathbf{t}}\})$, and $\mathrm{unf}(T) = T$ if $T \neq \mu\mathbf{t}.T'$. Type equality is *syntactic*: $\mu\mathbf{t}.T$ is not equal to $\mathrm{unf}(\mu\mathbf{t}.T)$. We write $P \to P'$ for process reductions, $\to^*$ for the reflexive+transitive closure of $\to$, and $P \not\to$ iff $\nexists P'$ such that $P \to P'$. We assume a *basic subtyping* $\leqslant_{\mathsf{B}}$ capturing e.g. $\mathtt{Int} \leqslant_{\mathsf{B}} \mathtt{Real}$. For readability, we use **blue**/**red** for **multiparty**/**standard** π-calculus.

2 Multiparty Session π-Calculus

In this section we illustrate a multiparty session π-calculus [27] (Definition 2.1), and its typing system — including recursion, subtyping [19] and type merging [67, 18] (Section 2.1). The calculus models processes that interact via *multiparty channels* connecting two or more participants: this is a departure from many "classic" and simpler process calculi, like the

[1] `http://dx.doi.org/10.4230/DARTS.3.2.3`

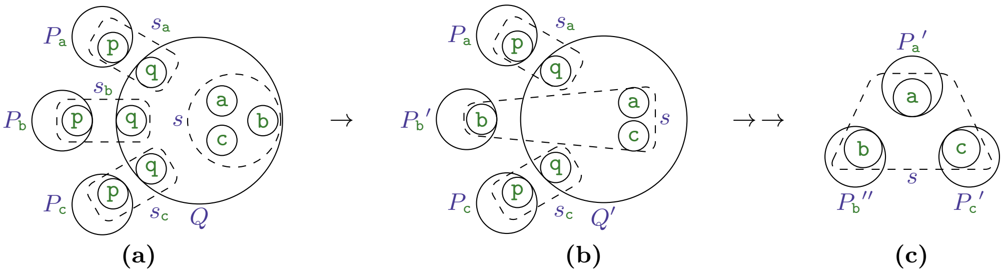

Figure 2 Multiparty peer-to-peer game. Dashed lines represent session scopes, and circled roles represent channels with roles. *(a)* initial configuration; *(b)* delegation of channel with role $s[\mathtt{b}]$ (and end of session $s_{\mathtt{b}}$); *(c)* clients directly interacting on session s, after "complete" delegation.

linear π-calculus (Section 3), that model *binary* channels. We provide various examples based on the scenario in Section 1.

▶ **Definition 2.1.** The *syntax* of multiparty session π-calculus *processes* and *values* is:

$$\begin{array}{rrcll}
\text{Processes} & P, Q & ::= & \mathbf{0} \mid P \,|\, Q \mid (\boldsymbol{\nu} s)P & \text{(inaction, composition, restriction)} \\
 & & & c[\mathtt{p}] \oplus \langle l(v) \rangle . P & \text{(selection towards role } \mathtt{p}\text{)} \\
 & & & c[\mathtt{p}] \,\&_{i \in I} \{ l_i(x_i) . P_i \} & \text{(branching from role } \mathtt{p} \text{ — with } I \neq \emptyset\text{)} \\
 & & & \mathbf{def}\ D\ \mathbf{in}\ Q \mid X\langle \tilde{x} \rangle & \text{(process definition, process call)} \\
\text{Declarations} & D & ::= & X(\tilde{x}) = P & \text{(process declaration)} \\
\text{Channels} & c & ::= & x \mid s[\mathtt{p}] & \text{(variable, channel with role } \mathtt{p}\text{)} \\
\text{Values} & v & ::= & c \mid \mathtt{false} \mid \mathtt{true} \mid 42 \mid \ldots & \text{(channel, base value)}
\end{array}$$

fc(P) is the set of *free channels with roles* in P, and fv(P) is the set of *free variables* in P.

A **channel** c can be either a variable or a **channel with role** $s[\mathtt{p}]$, i.e., a multiparty communication endpoint whose user impersonates role $\mathtt{p}$ in the session s. **Values** v can be variables, or channels with roles, or base values. The **inaction** $\mathbf{0}$ represents a terminated process. The **parallel composition** $P \,|\, Q$ represents two processes that can execute concurrently, and potentially communicate. The **session restriction** $(\boldsymbol{\nu} s)P$ declares a new session s with scope limited to process P. Process $c[\mathtt{p}] \oplus \langle l(v) \rangle . P$ performs a **selection (internal choice)** towards role $\mathtt{p}$, using the channel c: the labelled value $l(v)$ is sent, and the execution continues as process P. Dually, process $c[\mathtt{p}] \,\&_{i \in I} \{ l_i(x_i) . P_i \}$ uses channels c to wait for a **branching (external choice)** from role $\mathtt{p}$: if the labelled value $l_k(v)$ is received (for some $k \in I$), then the execution continues as P_k (with x_k holding value v). Note that for all $i \in I$, variable x_i is bound with scope P_i. In both branching and selection, the labels l_i $(i \in I)$ are all different and their order is irrelevant. **Process definition** $\mathbf{def}\ D\ \mathbf{in}\ Q$ and **process call** $X\langle \tilde{x} \rangle$ model recursion, with D being a **process declaration** $X(\tilde{x}) = P$: the call invokes X by expanding it into P, and replacing its formal parameters with the actual ones. We postulate that process declarations are *closed*, i.e., in $X(\tilde{x}) = P$, we have fv(P) $\subseteq \tilde{x}$ and fc(P) $= \emptyset$. Note that our syntax is simplified in the style of [19]: it does not have dedicated input/output prefixes, but they can be easily encoded using $\&$ (with *one* branch) and $\oplus$.

▶ **Example 2.2.** The following MPST π-calculus process implements the scenario in Figure 1:

$$\begin{array}{l}
\mathbf{def}\ \mathit{Loop}_{\mathtt{b}}(x) = x[\mathtt{a}] \,\& \left\{ \mathtt{Mov1AB}(y) . x[\mathtt{c}] \oplus \langle \mathtt{Mov1BC}(y) \rangle . \mathit{Loop}_{\mathtt{b}}\langle x \rangle \ ,\ \mathtt{Mov2AB}(z) . x[\mathtt{c}] \oplus \langle \mathtt{Mov2BC}(z) \rangle . \mathit{Loop}_{\mathtt{b}}\langle x \rangle \right\}\ \mathbf{in} \\
\quad \mathbf{def}\ \mathit{Client}_{\mathtt{b}}(y) = y[\mathtt{q}] \,\&\, \mathtt{PlayB}(z) \,.\, z[\mathtt{c}] \oplus \langle \mathtt{InfoBC}(\text{“…”}) \rangle \,.\, z[\mathtt{a}] \,\&\, \mathtt{InfoBA}(y) \,.\, \mathit{Loop}_{\mathtt{b}}\langle z \rangle\ \mathbf{in} \\
\qquad (\boldsymbol{\nu} s_{\mathtt{a}}, s_{\mathtt{b}}, s_{\mathtt{c}}) \big(Q \mid P_{\mathtt{a}} \mid P_{\mathtt{b}} \mid P_{\mathtt{c}} \big)
\end{array}$$

where: $\quad P_{\mathtt{b}} \; = \; \mathit{Client}_{\mathtt{b}}\langle s_{\mathtt{b}}[\mathtt{p}] \rangle \quad$ (for brevity, we omit the definitions of $P_{\mathtt{a}}$ and $P_{\mathtt{c}}$)

$$Q \; = \; (\boldsymbol{\nu} s) \Big(s_{\mathtt{a}}[\mathtt{q}][\mathtt{p}] \oplus \langle \mathtt{PlayA}(s[\mathtt{a}]) \rangle \;\mid\; s_{\mathtt{b}}[\mathtt{q}][\mathtt{p}] \oplus \langle \mathtt{PlayB}(s[\mathtt{b}]) \rangle \;\mid\; s_{\mathtt{c}}[\mathtt{q}][\mathtt{p}] \oplus \langle \mathtt{PlayC}(s[\mathtt{c}]) \rangle \Big)$$

In the 3rd line, $s_{\mathtt{a}}, s_{\mathtt{b}}, s_{\mathtt{c}}$ are the sessions between the server process Q and the clients $P_{\mathtt{a}}, P_{\mathtt{b}}, P_{\mathtt{c}}$, which are composed in parallel with $|$. Each sessions has 2 roles: $\mathtt{q}$ (server) and $\mathtt{p}$ (client); e.g., $s_{\mathtt{b}}$ is accessed by the server (through the channel with role $s_{\mathtt{b}}[\mathtt{q}]$) and by the client $P_{\mathtt{b}}$ (through $s_{\mathtt{b}}[\mathtt{p}]$); similarly, $s_{\mathtt{a}}$ (resp. $s_{\mathtt{c}}$) is accessed by $P_{\mathtt{a}}$ (resp. $P_{\mathtt{c}}$) through $s_{\mathtt{a}}[\mathtt{p}]$ (resp. $s_{\mathtt{c}}[\mathtt{p}]$), while the server owns $s_{\mathtt{a}}[\mathtt{q}]$ (resp. $s_{\mathtt{c}}[\mathtt{q}]$). The body of the server process Q defines a session s (with 3 roles $\mathtt{a}, \mathtt{b}, \mathtt{c}$) for playing the game. Note that the scope of s does *not* include $P_{\mathtt{a}}, P_{\mathtt{b}}, P_{\mathtt{c}}$: see Figure 2(a) for a schema of processes and sessions.

The server Q uses the channel with role $s_{\mathtt{b}}[\mathtt{q}]$ (resp. $s_{\mathtt{a}}[\mathtt{q}], s_{\mathtt{c}}[\mathtt{q}]$) to send the message $\mathtt{PlayB}$ (resp. $\mathtt{PlayA}, \mathtt{PlayC}$) carrying the channel with role $s[\mathtt{b}]$ (resp. $s[\mathtt{a}], s[\mathtt{c}]$) to $\mathtt{p}$. The result is a *delegation* of the channel to the client process $P_{\mathtt{b}}$ (resp. $P_{\mathtt{a}}, P_{\mathtt{c}}$). This way, each client obtains a channel endpoint to interact in the game session s, interpreting a role among $\mathtt{a}$, $\mathtt{b}$ and $\mathtt{c}$.

The client $P_{\mathtt{b}}$ is implemented by invoking $\mathit{Client}_{\mathtt{b}}\langle s_{\mathtt{b}}[\mathtt{p}]\rangle$ (defined in the 2nd line). Here, $y[\mathtt{q}] \,\&\, \mathtt{PlayB}(z)$ means that y (that becomes $s_{\mathtt{b}}[\mathtt{p}]$ after the invocation) is used to receive $\mathtt{PlayB}(z)$ from $\mathtt{q}$, while $z[\mathtt{c}] \oplus \langle \mathtt{InfoBC}(\text{“...”})\rangle$ means that z (that becomes $s[\mathtt{b}]$ after the delegation is received) is used to send $\mathtt{InfoBC}(\text{“...”})$ to $\mathtt{c}$. The game loop is implemented with the recursive process call $\mathit{Loop}_{\mathtt{b}}\langle z\rangle$ (defined in the 1st line) — which becomes $\mathit{Loop}_{\mathtt{b}}\langle s[\mathtt{b}]\rangle$ after delegation.

▶ **Definition 2.3.** The *operational semantics* of multiparty session processes is:

$$\begin{array}{rl}
(\textsc{R-Comm}) & s[\mathtt{p}][\mathtt{q}] \,\&_{i\in I}\, \{l_i(x_i).P_i\} \mid s[\mathtt{q}][\mathtt{p}] \oplus \langle l_j(v)\rangle.Q \;\;\to\;\; P_j\{{}^{v}\!/\!{}_{x_j}\} \mid Q \quad (\text{if } j \in I \text{ and } \mathrm{fv}(v) = \emptyset)\\
(\textsc{R-Call}) & \mathbf{def}\, X(\widetilde{x}) = P \,\mathbf{in}\, (X\langle\widetilde{v}\rangle \mid Q) \;\to\; \mathbf{def}\, X(\widetilde{x}) = P \,\mathbf{in}\, (P\{\widetilde{v}/\widetilde{x}\} \mid Q)\\
 & \qquad\qquad\qquad\qquad (\text{if } \widetilde{x} = x_1,\dots,x_n,\ \widetilde{v} = v_1,\dots,v_n,\ \mathrm{fv}(\widetilde{v}) = \emptyset)\\
(\textsc{R-Par}) & P \to Q \;\text{ implies }\; P \mid R \to Q \mid R \qquad (\textsc{R-Res}) \;\; P \to Q \;\text{ implies }\; (\boldsymbol{\nu} s)P \to (\boldsymbol{\nu} s)Q\\
(\textsc{R-Def}) & P \to Q \;\text{ implies }\; \mathbf{def}\, D \,\mathbf{in}\, P \to \mathbf{def}\, D \,\mathbf{in}\, Q\\
(\textsc{R-Struct}) & P \equiv P' \;\text{ and }\; P \to Q \;\text{ and }\; Q' \equiv Q \;\text{ implies }\; P' \to Q' \quad (\text{with } \equiv \text{ standard — see [60]})
\end{array}$$

Rule ($\textsc{R-Comm}$) models communication: it says that the parallel composition of a branching and a selection process, both operating on the same session s respectively as roles $\mathtt{p}$ and $\mathtt{q}$ (i.e., via $s[\mathtt{p}]$ and $s[\mathtt{q}]$) and targeting each other (i.e., $s[\mathtt{p}]$ is used to branch from $\mathtt{q}$, and $s[\mathtt{q}]$ is used to select towards $\mathtt{p}$) reduces to the corresponding continuations, with a value substitution on the receiver side. ($\textsc{R-Call}$) says that a process call $X\langle\widetilde{v}\rangle$ in the scope of $\mathbf{def}\, X(\widetilde{x}) = P \,\mathbf{in} \dots$ reduces by expanding $X\langle\widetilde{v}\rangle$ into P, and replacing the formal parameters $(\widetilde{x})$ with the actual ones $(\widetilde{v})$. The remaining rules are standard: reduction can happen under parallel composition, restriction and process definition. By ($\textsc{R-Struct}$), reduction is closed under a structural congruence [60] stating, e.g., that $|$ is commutative and associative, and has $\mathbf{0}$ as neutral element (i.e., $P \mid Q \equiv Q \mid P$, $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$ and $P \mid \mathbf{0} \equiv P$).

▶ **Example 2.4.** The process in Example 2.2 reduces as (see also Figure 2(b), noting the scope of s):

$$\begin{array}{l}
(\boldsymbol{\nu} s_{\mathtt{a}}, s_{\mathtt{b}}, s_{\mathtt{c}})\big(Q \mid P_{\mathtt{a}} \mid P_{\mathtt{b}} \mid P_{\mathtt{c}}\big) \;\to \qquad (\text{by } (\textsc{R-Comm}) \text{ between } Q \text{ and } P_{\mathtt{b}}, (\textsc{R-Par}), (\textsc{R-Struct}), (\textsc{R-Res}))\\
\quad (\boldsymbol{\nu} s_{\mathtt{a}}, s_{\mathtt{c}})\Big((\boldsymbol{\nu} s)\Big(\big(s_{\mathtt{a}}[\mathtt{q}][\mathtt{p}] \oplus \langle \mathtt{PlayA}(s[\mathtt{a}])\rangle \mid s_{\mathtt{c}}[\mathtt{q}][\mathtt{p}] \oplus \langle \mathtt{PlayC}(s[\mathtt{c}])\rangle\big) \mid s[\mathtt{b}][\mathtt{c}] \oplus \langle \mathtt{InfoBC}(\text{“...”})\rangle \dots\Big) \mid P_{\mathtt{a}} \mid P_{\mathtt{c}}\Big)
\end{array}$$

2.1 Multiparty Session Typing

We now illustrate the typing system for the MPST π-calculus, and its properties. We adopt standard definitions from literature — except for some crucial (and duly noted) adaptations.

The goal of the MPST typing system is to ensure that processes interact on their channels according to given specifications, represented as *session types.* MPST foster a *top-down* approach: a *global type* G describes a protocol involving various *roles* — e.g., the game with roles $\mathtt{a}, \mathtt{b}, \mathtt{c}$ in Section 1; G is *projected* into a set of *(local) session types* $S_{\mathtt{a}}$, $S_{\mathtt{b}}$, $S_{\mathtt{c}}, \dots$ (one per role) that specify how each role is expected to use its channel endpoint; finally, session types are assigned to channels, and the processes using them are type-checked. Typing ensures that processes *(1) never go wrong* (i.e., use their channels type-safely), and *(2)* interact according to G, by respecting its projections — thus realising a *multiparty, deadlock-free session.*

In the following, we provide a revised and streamlined presentation that clearly outlines the *interplay between the global/local typing levels.* For this reason, unlike most papers, we discuss *local types first*, and *global types later*, at the end of the section.

Session Types: Local and Partial. Session types describe the expected usage of a channel, as a communication protocol involving two or more *roles.* They allow to declare structured sequences of input/output actions, specifying who is the source/target role of interaction.

▶ **Definition 2.5** (Types and roles)**.** The syntax of *(local) session types* is:

$$\begin{array}{rcll} S & ::= & \mathsf{p}\,\&_{i\in I}\, ?l_i(U_i).S_i & \text{(branching from role } \mathsf{p} \text{ — with } I\neq\emptyset) \\ & & \mathsf{p}\oplus_{i\in I}\, !l_i(U_i).S_i & \text{(selection towards role } \mathsf{p} \text{ — with } I\neq\emptyset) \\ & & \mu\mathbf{t}.S \;\mid\; \mathbf{t} \;\mid\; \mathbf{end} & \text{(recursive type, type variable, termination)} \end{array}$$

$$B \;::=\; \mathtt{Bool} \mid \mathtt{Int} \mid \ldots \quad \text{(base type)} \qquad U \;::=\; B \mid S\,{\scriptstyle\text{(closed)}} \quad \text{(payload type)}$$

We omit $\&/\oplus$ when I is a singleton: $\mathsf{p}!l_1(\mathtt{Int}).S_1$ stands for $\mathsf{p}\oplus_{i\in\{1\}} !l_i(\mathtt{Int}).S_i$. The set of *roles in* S, denoted as $\mathrm{roles}(S)$, is defined as follows:

$$\mathrm{roles}(\mathsf{p}\oplus_{i\in I} !l_i(U_i).S_i) \;\triangleq\; \mathrm{roles}(\mathsf{p}\,\&_{i\in I}\, ?l_i(U_i).S_i) \;\triangleq\; \{\mathsf{p}\}\cup\bigcup_{i\in I}\mathrm{roles}(S_i)$$
$$\mathrm{roles}(\mathbf{end}) \;\triangleq\; \emptyset \qquad \mathrm{roles}(\mathbf{t}) \;\triangleq\; \emptyset \qquad \mathrm{roles}(\mu\mathbf{t}.S) \;\triangleq\; \mathrm{roles}(S)$$

We will write $\mathsf{p}\in S$ for $\mathsf{p}\in\mathrm{roles}(S)$, and $\mathsf{p}\in S\backslash\mathsf{q}$ for $\mathsf{p}\in\mathrm{roles}(S)\setminus\{\mathsf{q}\}$.

The **branching type** $\mathsf{p}\,\&_{i\in I}\, ?l_i(U_i).S_i$ describes a channel that can receive a label l_i from role p (for some $i\in I$, chosen by p), together with a *payload* of type U_i; then, the channel must be used as S_i. The **selection** $\mathsf{p}\oplus_{i\in I} !l_i(U_i).S_i$, describes a channel that can choose a label l_i (for any $i\in I$), and send it to p together with a payload of type U_i; then, the channel must be used as S_i. The labels of branch/select types are all distinct and their order is irrelevant. The **recursive type** $\mu\mathbf{t}.S$ and **type variable** $\mathbf{t}$ model infinite behaviours. $\mathbf{end}$ is the type of a **terminated channel** (often omitted). **Base types** $B, B', \ldots$ can be types like $\mathtt{Bool}$, $\mathtt{Int}$, *etc.* **Payload types** $U, U', \ldots$ are either base types, or *closed* session types.

▶ **Example 2.6.** See the definition and description of session type S_{b} in Section 1 (p. 3).

To define session typing contexts later on, we also need *partial* session types.

▶ **Definition 2.7.** *Partial session types*, denoted by H, are:

$$\begin{array}{rcll} H & ::= & \&_{i\in I}\, ?l_i(U_i).H_i \;\mid\; \oplus_{i\in I}\, !l_i(U_i).H_i & \text{(branching, selection)}\ {\scriptstyle(\text{with } I\neq\emptyset,\ U_i \text{ closed})} \\ & & \mu\mathbf{t}.H \;\mid\; \mathbf{t} \;\mid\; \mathbf{end} & \text{(recursive type, type variable, termination)} \end{array}$$

A partial session type H is either a branching, a selection, a recursion, a type variable, or a terminated channel type. Unlike Definition 2.5, partial types have *no role annotations*: they are similar to *binary* session types (but the payloads U_i can be *multiparty*) — and similarly, they endow a notion of *duality*: the outputs of a type match the inputs of its dual, and *vice versa.*

▶ **Definition 2.8.** $\overline{H}$ is the *dual of* H, defined as:

$$\overline{\oplus_{i\in I}\, !l_i(U_i).H_i} \;\triangleq\; \&_{i\in I}\, ?l_i(U_i).\overline{H_i} \qquad \overline{\&_{i\in I}\, ?l_i(U_i).H_i} \;\triangleq\; \oplus_{i\in I}\, !l_i(U_i).\overline{H_i}$$
$$\overline{\mathbf{end}} \;\triangleq\; \mathbf{end} \qquad\qquad \overline{\mathbf{t}} \;\triangleq\; \mathbf{t} \qquad\qquad \overline{\mu\mathbf{t}.H} \;\triangleq\; \mu\mathbf{t}.\overline{H}$$

The dual of a selection type is a branching with dualised continuations, and *vice versa*; the payloads U_i are the same. Duality is the identity on **end** and $\mathbf{t}$, and homomorphic on $\mu\mathbf{t}.H$.

Multiparty session types can be *projected* onto a role q (Definition 2.9 below): this yields a partial type that only describes the communications where q is involved. This is technically necessary for typing rules, as we will see in Definition 2.11 later on.

▶ **Definition 2.9.** $S \restriction \mathsf{q}$ is the *partial projection of S onto* q:

$$\mathbf{end} \restriction \mathsf{q} \triangleq \mathbf{end} \qquad \mathbf{t} \restriction \mathsf{q} \triangleq \mathbf{t} \qquad (\mu\mathbf{t}.S) \restriction \mathsf{q} \triangleq \begin{cases} \mu\mathbf{t}.(S \restriction \mathsf{q}) & \text{if } S \restriction \mathsf{q} \neq \mathbf{t}' \ (\forall \mathbf{t}') \\ \mathbf{end} & \text{otherwise} \end{cases}$$

$$(\mathsf{p} \oplus_{i \in I} !l_i(U_i).S_i) \restriction \mathsf{q} \triangleq \begin{cases} \oplus_{i \in I} !l_i(U_i).(S_i \restriction \mathsf{q}) & \text{if } \mathsf{q} = \mathsf{p}, \\ \bigsqcap_{i \in I} (S_i \restriction \mathsf{q}) & \text{if } \mathsf{p} \neq \mathsf{q} \end{cases}$$

$$(\mathsf{p} \,\&_{i \in I}\, ?l_i(U_i).S_i) \restriction \mathsf{q} \triangleq \begin{cases} \&_{i \in I}\, ?l_i(U_i).S_i \restriction \mathsf{q} & \text{if } \mathsf{q} = \mathsf{p}, \\ \bigsqcap_{i \in I} (S_i \restriction \mathsf{q}) & \text{if } \mathsf{p} \neq \mathsf{q} \end{cases}$$

where $\bigsqcap$ is the *merge operator for partial session types*:

$$\mathbf{end} \sqcap \mathbf{end} \triangleq \mathbf{end} \qquad \mathbf{t} \sqcap \mathbf{t} \triangleq \mathbf{t} \qquad \mu\mathbf{t}.H \sqcap \mu\mathbf{t}.H' \triangleq \mu\mathbf{t}.(H \sqcap H')$$

$$\&_{i \in I}\, ?l_i(U_i).H_i \sqcap \&_{i \in I}\, ?l_i(U_i).H'_i \triangleq \&_{i \in I}\, ?l_i(U_i).(H_i \sqcap H'_i)$$

$$\oplus_{i \in I} !l_i(U_i).H_i \sqcap \oplus_{j \in J} !l_j(U_j).H'_j \triangleq$$
$$\big(\oplus_{k \in I \cap J} !l_k(U_k).(H_k \sqcap H'_k)\big) \oplus \big(\oplus_{i \in I \setminus J} !l_i(U_i).H_i\big) \oplus \big(\oplus_{j \in J \setminus I} !l_j(U_j).H'_j\big)$$

The projection of **end** or a type variable $\mathbf{t}$ onto any role is the identity. Projecting a recursive type $\mu\mathbf{t}.S$ onto q, means projecting S onto q, if $S \restriction \mathsf{q}$ is *not* some $\mathbf{t}'$, for all possible recursive variables $\mathbf{t}'$; otherwise, the projection is **end**. The projection of a selection $\mathsf{p} \oplus_{i \in I} !l_i(U_i).S_i$ (resp. branching $\mathsf{p} \,\&_{i \in I}\, ?l_i(U_i).S_i$) on role p, produces a partial selection type $\oplus_{i \in I} !l_i(U_i).(S_i \restriction \mathsf{p})$ (resp. branching $\&_{i \in I}\, ?l_i(U_i).S_i \restriction \mathsf{p}$) with the continuations projected on p. Otherwise, if projecting on $\mathsf{q} \neq \mathsf{p}$, the select/branch is "skipped", and the projection is the *merging of the continuations*, i.e., $\bigsqcap_{i \in I} (S_i \restriction \mathsf{q})$. The $\sqcap$ operator (introduced in [67, 18]) expands the set of session types whose partial projections are defined, which allows to type more processes (as we will see in Definition 2.11 and Example 2.14 later on). Crucially, $\sqcap$ can compose different *internal* choices, but *not* external choices (because this could break type safety).

Subtyping. The *subtyping relation* (Definition 2.10) says that a session type S is "smaller" than S' when S is "less demanding" than S' — i.e., when S permits more internal choices, and imposes less external choices, than S'. When typing processes (Definition 2.12), a channel with a smaller type can be used whenever a channel with a larger type is required, according to Liskov's Substitution Principle [45]. Subtyping is defined on both local and partial types.

▶ **Definition 2.10** (Subtyping)**.** The *subtyping $\leqslant_{\mathsf{S}}$ on multiparty session types* is the largest relation such that

(i) if $S \leqslant_{\mathsf{S}} S'$, then $\forall \mathsf{p} \in (\text{roles}(S) \cup \text{roles}(S'))\ S \restriction \mathsf{p} \leqslant_{\mathsf{P}} S' \restriction \mathsf{p}$, and

(ii) is closed backwards under coinductive rules at the top of Figure 3.

The *subtyping $\leqslant_{\mathsf{P}}$ on partial session types* is coinductively defined by the rules at the bottom of Figure 3.

Definition 2.10 uses coinduction to support recursive types [56, Section 20 and Section 21]. Clause *(i)* links local and partial subtyping, and ensures that if two types are related, then their partial projections exist: this will be necessary later, for typing contexts (Definition 2.11). The gist of Definition 2.10 lies in clause *(ii)*. Rules (S-Brch)/(S-Sel) define subtyping on

$$\frac{\forall i \in I \quad U_i \leqslant_{\mathsf{S}} U_i' \quad S_i \leqslant_{\mathsf{S}} S_i'}{\mathtt{p} \mathbin{\&}_{i \in I} ?l_i(U_i).S_i \;\leqslant_{\mathsf{S}}\; \mathtt{p} \mathbin{\&}_{i \in I \cup J} ?l_i(U_i').S_i'}\;(\text{S-Brch}) \qquad \frac{\forall i \in I \quad U_i' \leqslant_{\mathsf{S}} U_i \quad S_i \leqslant_{\mathsf{S}} S_i'}{\mathtt{p} \oplus_{i \in I \cup J} !l_i(U_i).S_i \leqslant_{\mathsf{S}} \mathtt{p} \oplus_{i \in I} !l_i(U_i').S_i'}\;(\text{S-Sel})$$

$$\frac{B \leqslant_{\mathsf{B}} B'}{B \leqslant_{\mathsf{S}} B'}\;(\text{S-B}) \qquad \frac{}{\mathbf{end} \leqslant_{\mathsf{S}} \mathbf{end}}\;(\text{S-End}) \qquad \frac{S\{^{\mu\mathbf{t}.S}/_{\mathbf{t}}\} \leqslant_{\mathsf{S}} S'}{\mu\mathbf{t}.S \leqslant_{\mathsf{S}} S'}\;(\text{S-}\mu\text{L}) \qquad \frac{S \leqslant_{\mathsf{S}} S'\left\{^{\mu\mathbf{t}.S'}/_{\mathbf{t}}\right\}}{S \leqslant_{\mathsf{S}} \mu\mathbf{t}.S'}\;(\text{S-}\mu\text{R})$$

$$\frac{\forall i \in I \quad U_i \leqslant_{\mathsf{S}} U_i' \quad H_i \leqslant_{\mathsf{P}} {H_i}'}{\mathbin{\&}_{i \in I} ?l_i(U_i).H_i \leqslant_{\mathsf{P}} \mathbin{\&}_{i \in I \cup J} ?l_i(U_i').{H_i}'}\;(\text{S-ParBrch}) \qquad \frac{\forall i \in I \quad U_i' \leqslant_{\mathsf{S}} U_i \quad H_i \leqslant_{\mathsf{P}} {H_i}'}{\oplus_{i \in I \cup J} !l_i(U_i).H_i \leqslant_{\mathsf{P}} \oplus_{i \in I} !l_i(U_i').{H_i}'}\;(\text{S-ParSel})$$

$$\frac{}{\mathbf{end} \leqslant_{\mathsf{P}} \mathbf{end}}\;(\text{S-ParEnd}) \qquad \frac{H\{^{\mu\mathbf{t}.H}/_{\mathbf{t}}\} \leqslant_{\mathsf{P}} H'}{\mu\mathbf{t}.H \leqslant_{\mathsf{P}} H'}\;(\text{S-Par}\mu\text{L}) \qquad \frac{H \leqslant_{\mathsf{P}} H'\left\{^{\mu\mathbf{t}.H'}/_{\mathbf{t}}\right\}}{H \leqslant_{\mathsf{P}} \mu\mathbf{t}.H'}\;(\text{S-Par}\mu\text{R})$$

Figure 3 Subtyping for session types (top) and partial session types (bottom).

branch/select types. Both rules are covariant in the continuation types, i.e., they require $S_i \leqslant_{\mathsf{S}} S_i'$. (S-Brch) is covariant also in the number of branches offered, whereas (S-Sel) is contravariant. (S-B) relates base types, if they are related by $\leqslant_{\mathsf{B}}$. (S-End) relates terminated channel types. (S-μL) and (S-μR) are standard under coinduction: they say that a recursive session type $\mu\mathbf{t}.S$ is related to S', iff its unfolding is related, too. The subtyping $\leqslant_{\mathsf{P}}$ for partial types is similar, except for the lack of role annotations (thus resembling the *binary* session subtyping [22]).

Multiparty Session Typing System. Before delving into the session typing rules (Definition 2.12), we need to formalise the notions of *typing context* and *typing judgement*, defined below.

▶ **Definition 2.11.** A *session typing context* Γ is a partial mapping defined as:

$$\Gamma \;::=\; \varnothing \;\mid\; \Gamma, x\!:\!U \;\mid\; \Gamma, s[\mathtt{p}]\!:\!S \;\; (\text{with } \mathtt{p} \notin S)$$

We say that Γ *is consistent* iff for all $s[\mathtt{p}]\!:\!S_{\mathtt{p}}, s[\mathtt{q}]\!:\!S_{\mathtt{q}} \in \Gamma$ with $\mathtt{p} \neq \mathtt{q}$, we have $\overline{S_{\mathtt{p}} \upharpoonright \mathtt{q}} \leqslant_{\mathsf{P}} S_{\mathtt{q}} \upharpoonright \mathtt{p}$. We say that Γ *is complete* iff for all $s[\mathtt{p}]\!:\!S_{\mathtt{p}} \in \Gamma$, $\mathtt{q} \in S_{\mathtt{p}}$ implies $s[\mathtt{q}] \in \operatorname{dom}(\Gamma)$. We say that Γ *is unrestricted*, $\operatorname{un}(\Gamma)$, iff for all $c \in \operatorname{dom}(\Gamma)$, $\Gamma(c)$ is either a base type or $\mathbf{end}$. The *typing contexts composition* $\circ$ is the commutative operator with $\varnothing$ as neutral element:

$$\Gamma_1, c\!:\!U \;\circ\; \Gamma_2, c'\!:\!U' \;\triangleq\; (\Gamma_1 \circ \Gamma_2), c\!:\!U, c'\!:\!U' \quad (\text{if } \operatorname{dom}(\Gamma_2) \not\ni c \neq c' \notin \operatorname{dom}(\Gamma_1))$$
$$\Gamma_1, x\!:\!B \;\circ\; \Gamma_2, x\!:\!B \;\triangleq\; (\Gamma_1 \circ \Gamma_2), x\!:\!B$$

A typing context can map a channel with role $s[\mathtt{p}]$ to a session type S (that cannot refer to $\mathtt{p}$ itself, ruling out "self-interactions"), but *not* to a base type. Variables can be mapped to either session or base types. The clause "$\forall c\!:\!S \in \Gamma : S \upharpoonright \mathtt{p}$ is defined" is discussed below.

On Consistency. In Definition 2.11, and in the rest of this work, we emphasise the importance of *consistency* of the context Γ for session typing: this condition is, in fact, *necessary to prove subject reduction*, and will be central for our encoding (Section 5 and Section 6). As an example of *non*-consistent typing context, consider $s[\mathtt{p}]\!:\!\mathbf{end}, s[\mathtt{q}]\!:\!\mathtt{p}?l(U).S$: we have $\overline{\mathbf{end} \upharpoonright \mathtt{q}} = \mathbf{end} \not\leqslant_{\mathsf{P}} ?l(U).S = (\mathtt{p}?l(U).S) \upharpoonright \mathtt{p}$.

Note that our consistency in Definition 2.11 is *weaker* than the one in previous papers (where it is sometimes called *coherency*): we use $\leqslant_{\mathsf{P}}$, instead of (syntactic) type equality $=$, to relate dual partial projections. The reason being: if we use $=$, *and* adopt partial projections with type merging (Definition 2.9), subject reduction does *not* hold. Hence, by

$$\text{(T-Name)}\ \frac{\mathrm{un}(\Gamma)}{\Gamma, c{:}S \vdash c{:}S} \qquad \text{(T-Basic)}\ \frac{\mathrm{un}(\Gamma) \qquad v \in B}{\Gamma \vdash v{:}B} \qquad \text{(T-DefCtx)}\ \frac{}{\Theta, X{:}\widetilde{U} \vdash X{:}\widetilde{U}} \qquad \text{(T-Sub)}\ \frac{\Theta \cdot \Gamma, c{:}U \vdash P \qquad U' \leqslant_{\mathsf{S}} U}{\Theta \cdot \Gamma, c{:}U' \vdash P}$$

$$\text{(T-Nil)}\ \frac{\mathrm{un}(\Gamma)}{\Theta \cdot \Gamma \vdash \mathbf{0}} \qquad \text{(T-Par)}\ \frac{\Theta \cdot \Gamma_1 \vdash P \qquad \Theta \cdot \Gamma_2 \vdash Q}{\Theta \cdot \Gamma_1 \circ \Gamma_2 \vdash P \mid Q} \qquad \text{(T-Res)}\ \frac{\Theta \cdot \Gamma, \Gamma' \vdash P \qquad \Gamma' = \{s[\mathtt{p}]{:}S_{\mathtt{p}}\}_{\mathtt{p} \in I} \text{ complete}}{\Theta \cdot \Gamma \vdash (\boldsymbol{\nu} s{:}\Gamma')P}$$

$$\text{(T-Brch)}\ \frac{\forall i \in I \qquad \Theta \cdot \Gamma, x_i{:}U_i, c{:}S_i \vdash P_i}{\Theta \cdot \Gamma, c{:}\mathtt{p} \,\&_{i \in I}\, ?l_i(U_i).S_i \vdash c[\mathtt{p}] \,\&_{i \in I}\, \{l_i(x_i).P_i\}} \qquad \text{(T-Sel)}\ \frac{\Gamma_1 \vdash v{:}U \qquad \Theta \cdot \Gamma_2, c{:}S \vdash P}{\Theta \cdot \Gamma_1 \circ \Gamma_2, c{:}\mathtt{p} \oplus !l(U).S \vdash c[\mathtt{p}] \oplus \langle l(v) \rangle.P}$$

$$\text{(T-Def)}\ \frac{\Theta, X{:}\widetilde{U} \cdot \widetilde{x}{:}\widetilde{U} \vdash P \qquad \Theta, X{:}\widetilde{U} \cdot \Gamma \vdash Q}{\Theta \cdot \Gamma \vdash \mathbf{def}\ X(\widetilde{x}{:}\widetilde{U}) = P\ \mathbf{in}\, Q} \qquad \text{(T-Call)}\ \frac{\forall i \in \{1..n\} \qquad \Gamma_i \vdash v_i{:}U_i \qquad \mathrm{un}(\Gamma)}{\Theta, X{:}U_1, \ldots, U_n \ \cdot\ \Gamma_1 \circ \ldots \circ \Gamma_n \circ \Gamma \vdash X\langle v_1, \ldots, v_n \rangle}$$

Figure 4 Typing rules for the multiparty session π-calculus.

relaxing our definition, and proving Theorem 2.16 later on, we fix a longstanding mistake appearing e.g., in [67, 18].

▶ **Definition 2.12** (Session typing judgements)**.** The *process declaration typing context* Θ maps process variables X to n-tuples of types $\widetilde{U}$ (one per argument of X), and is defined as:

$$\Theta \ ::= \ \varnothing \mid \Theta, X{:}\widetilde{U}$$

Typing judgements are inductively defined by the rules in Figure 4, and have the forms:

for processes: $\Theta \cdot \Gamma \vdash P$ (with Γ consistent, and $\forall c{:}S \in \Gamma$, $S \upharpoonright \mathtt{p}$ is defined $\forall \mathtt{p} \in S$)
for values: $\Gamma \vdash v{:}U$ for process variables: $\Theta \vdash X{:}\widetilde{U}$

The judgement $\Theta \cdot \Gamma \vdash P$ reads: "process P is well-typed in Θ and Γ". Θ and Γ, in turn, type respectively process variables (judgement $\Theta \vdash X{:}\widetilde{U}$) and values, including channels (judgement $\Gamma \vdash v{:}U$). Rule (T-Name) says that a channel has the type assumed in the session typing context. (T-Basic) relates base values to their type. By (T-DefCtx), a process name has the type assumed in the process declaration typing context. (T-Sub) is the standard subsumption rule, using $\leqslant_{\mathsf{S}}$ (Definition 2.10). By (T-Nil), the terminated process is well typed in any unrestricted typing context. By (T-Par), the parallel composition of P and Q is well typed under the composition of the corresponding typing contexts, as per Definition 2.11. By (T-Res), $(\boldsymbol{\nu} s)P$ is well typed in Γ, if s occurs in a *complete* set of typed channels with roles (denoted with Γ'), and the open process P is well typed in the "full" context Γ, Γ'. For convenience, we annotate the restricted s with Γ' in the process, giving $(\boldsymbol{\nu} s{:}\Gamma')P$. (T-Brch) (resp. (T-Sel)) state that branching (resp. selection) process on $c[\mathtt{p}]$ is well typed if $c[\mathtt{p}]$ is of compatible branching (resp. selection) type, and the continuations P_i, for all $i \in I$, are well typed with the continuation session types. By (T-Def), a process definition $\mathbf{def}\ X(\widetilde{x}) = P\ \mathbf{in}\, Q$ is well typed if both P and Q are well typed in their typing contexts enriched with $\widetilde{x}{:}\widetilde{U}$. For convenience, we annotate $\widetilde{x}$ with types $\widetilde{U}$. By (T-Call), process call $X\langle v_1, \ldots, v_n \rangle$ is well typed if the actual parameters $v_1, \ldots, v_n$ have compatible types w.r.t. X.

As mentioned above, we emphasise consistency by restricting typing judgements to *consistent* typing contexts — i.e., those allowing to prove subject reduction. The clause "$\forall c{:}S \in \Gamma : S \upharpoonright \mathtt{p}$ is defined" is unusual in MPST works, but arises naturally: by requiring the existence of partial projections, it rejects processes containing

(a) a channel with role $s[\mathtt{p}]{:}S$ that, for some $\mathtt{q} \in S$, cannot be (consistently) paired with $s[\mathtt{q}]$, or

(b) a variable $x{:}S$ that, in a consistent and complete Γ, cannot be substituted by any $s[\mathtt{p}]{:}S$.

Rejected processes cannot join any complete session (case *(a)*), or are never-executed "dead code" (case *(b)*).

▶ Remark 2.13. Unlike most MPST papers (e.g., [19, 11]), our rule (T-Res) does *not* directly map a session s to a global type: this is explained in the next section, "Global Types".

▶ **Example 2.14.** Consider the session type S_{b} in Section 1 (p. 3), and the client process $P_{\mathsf{b}} = \mathit{Client}_{\mathsf{b}}\langle s_{\mathsf{b}}[\mathsf{p}]\rangle$ from Example 2.2. By Definition 2.12, the following typing judgement holds:

$$\mathit{Client}_{\mathsf{b}} : \mathsf{q}?\mathtt{PlayB}(S_{\mathsf{b}}),\ \mathit{Loop}_{\mathsf{b}} : \mu\mathbf{t}.\,\mathsf{a}\,\&\left\{\begin{array}{l} ?\mathtt{Mov1AB}(\mathtt{Int}).\mathsf{c}!\mathtt{Mov1BC}(\mathtt{Int}).\mathbf{t}\,, \\ ?\mathtt{Mov2AB}(\mathtt{Bool}).\mathsf{c}!\mathtt{Mov2BC}(\mathtt{Bool}).\mathbf{t} \end{array}\right\} \cdot s_{\mathsf{b}}[\mathsf{p}] : \mathsf{q}?\mathtt{PlayB}(S_{\mathsf{b}}) \ \vdash\ \mathit{Client}_{\mathsf{b}}\langle s_{\mathsf{b}}[\mathsf{p}]\rangle$$

It says that the channel with role $s_{\mathsf{b}}[\mathsf{p}]$ is used following type $\mathsf{q}?\mathtt{PlayB}(S_{\mathsf{b}}).\mathbf{end}$ (with a delegation of a S_{b}-typed channel); the argument of $\mathit{Client}_{\mathsf{b}}$ has the same type; the argument of $\mathit{Loop}_{\mathsf{b}}$ is used following the game loop. This example *cannot be typed* without merging $\sqcap$ (Definition 2.9): its derivation requires to compute
$S_{\mathsf{b}}\!\upharpoonright\!\mathsf{c} \ = !\mathtt{InfoBC}(\mathtt{String}).\mu\mathbf{t}.(!\mathtt{Mov1BC}(\mathtt{Int}).\mathbf{t} \sqcap !\mathtt{Mov2BC}(\mathtt{Bool}).\mathbf{t}) \ = !\mathtt{InfoBC}(\mathtt{String}).\mu\mathbf{t}.(!\mathtt{Mov1BC}(\mathtt{Int}).\mathbf{t} \oplus !\mathtt{Mov2BC}(\mathtt{Bool}).\mathbf{t})$, which is undefined without merging.

The typing rules in Figure 4 satisfy a subject reduction property (Theorem 2.16) based on *typing context reductions*. Reduction relations for typing contexts are common in typed process calculi, and reflect the communications required by the types in Γ.

▶ **Definition 2.15** (Typing context reduction)**.** The *reduction* $\Gamma \to \Gamma'$ is:

$$\begin{array}{rcll} s[\mathsf{p}]:S_{\mathsf{p}},\ s[\mathsf{q}]:S_{\mathsf{q}} & \to & s[\mathsf{p}]:S_k,\ s[\mathsf{q}]:S'_k & \text{if} \left\{\begin{array}{ll} \mathrm{unf}(S_{\mathsf{p}}) = \mathsf{q} \oplus_{i\in I} !l_i(U_i).S_i & k \in I \\ \mathrm{unf}(S_{\mathsf{q}}) = \mathsf{p}\ \&_{i\in I\cup J}\ ?l_i(U'_i).S'_i & U_k \leqslant_{\mathsf{S}} U'_k \end{array}\right. \\ \Gamma,\ c{:}U & \to & \Gamma',\ c{:}U' & \text{if } \Gamma \to \Gamma' \text{ and } U \leqslant_{\mathsf{S}} U' \end{array}$$

Our Definition 2.15 is a bit less straightforward than the ones in literature: it accommodates subtyping (hence, uses $\leqslant_{\mathsf{S}}$) and our iso-recursive type equality (hence, unfolds types explicitly).

▶ **Theorem 2.16** (Subject reduction)**.** *If* $\Theta\cdot\Gamma\vdash P$ *and* $P{\to}P'$*, then* $\exists\Gamma'$*:* $\Gamma{\to}^{*}\Gamma'$ *and* $\Theta\cdot\Gamma'\vdash P'$*.*

Global Types. We conclude this section with *global types*, mentioned in Section 2.1 and Remark 2.13.

▶ **Definition 2.17.** The syntax of global types, ranged over by G, is:

$$\begin{array}{rcll} G & ::= & \mathsf{p}\to\mathsf{q}{:}\{l_i(U_i).G_i\}_{i\in I} & \text{(interaction — with } U_i \text{ closed)} \\ & & \mu\mathbf{t}.G \ \mid\ \mathbf{t} \ \mid\ \mathbf{end} & \text{(recursive type, type variable, termination)} \end{array}$$

Type $\mathsf{p}\to\mathsf{q}{:}\{l_i(U_i).G_i\}_{i\in I}$ states that role p sends to role q one of the (pairwise distinct) labels l_i for $i \in I$, together with a payload U_i (Definition 2.5). If the chosen label is l_j, then the interaction proceeds as G_j. Type $\mu\mathbf{t}.G$ and type variable $\mathbf{t}$ model recursion. Type $\mathbf{end}$ states the termination of a protocol. We omit the braces $\{...\}$ from interactions when I is a singleton: e.g., $\mathsf{a}{\to}\mathsf{b}{:}l_1(U_1).G_1$ stands for $\mathsf{a}{\to}\mathsf{b}{:}\{l_i(U_i).G_i\}_{i\in\{1\}}$.

▶ **Example 2.18.** The following global type formalises the *Game* described in Section 1 and Figure 1:

$$\begin{array}{rcl} G_{\mathit{Game}} & = & \mathsf{b}{\to}\mathsf{c}{:}\mathtt{InfoBC}(\mathtt{String})\,.\,\mathsf{c}{\to}\mathsf{a}{:}\mathtt{InfoCA}(\mathtt{String})\,.\,\mathsf{a}{\to}\mathsf{b}{:}\mathtt{InfoAB}(\mathtt{String})\,. \\ & & \mu\mathbf{t}.\mathsf{a}{\to}\mathsf{b}{:}\left\{\begin{array}{l} \mathtt{Mov1AB}(\mathtt{Int}).\mathsf{b}{\to}\mathsf{c}{:}\mathtt{Mov1BC}(\mathtt{Int}).\mathsf{c}{\to}\mathsf{a}{:}\left\{\begin{array}{l}\mathtt{Mov1CA}(\mathtt{Int}).\mathbf{t}\,, \\ \mathtt{Mov2CA}(\mathtt{Bool}).\mathbf{t}\end{array}\right\}, \\ \mathtt{Mov2AB}(\mathtt{Bool}).\mathsf{b}{\to}\mathsf{c}{:}\mathtt{Mov2BC}(\mathtt{Bool}).\mathsf{c}{\to}\mathsf{a}{:}\left\{\begin{array}{l}\mathtt{Mov1CA}(\mathtt{Int}).\mathbf{t}\,, \\ \mathtt{Mov2CA}(\mathtt{Bool}).\mathbf{t}\end{array}\right\} \end{array}\right\} \end{array}$$

In MPST theory, a global type G with roles $\mathtt{p}_i$ $(i \in I)$ is used to *project*[2] a set of session types S_i (one per role). E.g., projecting G_{Game} in Example 2.18 onto $\mathtt{b}$ yields the session type $S_{\mathtt{b}}$ (p. 3). When *all* such projections S_i are defined, *and* all partial projections of each S_i are defined (as per Definition 2.9), then we can define the *projected typing context of* G:

$$\Gamma_G \;=\; \{s[\mathtt{p}_i] : S_i\}_{i\in I} \qquad \text{where } \forall i \in I:\; S_i \text{ is the projection of } G \text{ onto } \mathtt{p}_i$$

and Γ_G can be shown to be:

(a) *consistent and complete*, i.e., can be used to type the session s by rule (T-Res) (Figure 4), and

(b) *deadlock-free*, i.e.: $\Gamma_G \rightarrow^* \Gamma'_G \not\rightarrow$ implies $\forall i \in I : \Gamma'_G(s[\mathtt{p}_i]) = \mathbf{end}$.

Similarly, it can be shown that Γ_G reduces as prescribed by G.

Now, from observation *(a)* above, we can easily define a "strict" version of rule (T-Res) (Figure 4) in the style of [19, 11], where

1. the clause "Γ' *complete*" is replaced with "Γ' *is the projected typing context of some* G", and
2. in the conclusion, the annotation $(\boldsymbol{\nu} s{:}\Gamma')$ is replaced with $(\boldsymbol{\nu} s{:}G)$.

Further, observation *(b)* allows to prove Theorem 2.19 below, as shown e.g. in [5]: a typed ensemble of processes interacting on a single G-typed session is deadlock-free (note: with our rules in Figure 4, the annotation $(\boldsymbol{\nu} s{:}G)$ would be $(\boldsymbol{\nu} s{:}\Gamma_G)$).

▶ **Theorem 2.19** (Deadlock freedom). *Let* $\varnothing \cdot \varnothing \vdash P$*, where* $P \equiv (\boldsymbol{\nu} s{:}G) \big|_{i\in I} P_i$ *and each* P_i *only interacts on* $s[\mathtt{p_i}]$*. Then,* P *is* deadlock-free*: i.e.,* $P \rightarrow^* P' \not\rightarrow$ *implies* $P' \equiv \mathbf{0}$.

Note that the properties above emerge by placing suitable session types S_i in the premises of (T-Res) — but our streamlined typing rules in Figure 4 do *not* require it, *nor* mention G. The main property of such rules is ensuring *type safety* (Theorem 2.16). We will exploit this insight (obtained by our separation of global/local typing) in our encoding (Section 5), preserving semantics and types (and thus, Theorem 2.19) *without* explicit references to global types.

3 Linear π-Calculus

The π-calculus is the canonical model for communication and concurrency based on message-passing and *channel mobility.* It was developed in the late 1980's, with the first publication in 1992 [47], followed by various proposals for types and type systems. In this section we summarise the theory of the π-calculus with linear types [37], adopting a standard formulation and well-known results from [59]. We will present new π-calculus-related results in Section 4.

▶ **Definition 3.1.** The *syntax* of π-calculus *processes* and *values* is:

$$\begin{array}{rcll}
P, Q & ::= & \mathbf{0} \;\mid\; P \mid Q \;\mid\; (\boldsymbol{\nu} x)P & \text{(inaction, parallel composition, restriction)}\\
 & & *P \;\mid\; \overline{x}\langle v\rangle.P \;\mid\; x(y).P & \text{(process replication, output, input)}\\
 & & \mathbf{case}\, v\, \mathbf{of}\, \{l_i(x_i) \triangleright P_i\}_{i\in I} & \text{(variant destruct)}\\
 & & \mathbf{with}\, [l_i : x_i]_{i\in I} = v\, \mathbf{do}\, P & \text{(labelled tuple destruct)}\\
u, v & ::= & x, y, w, z \;\mid\; l(v) \;\mid\; [l_i : v_i]_{i\in I} & \text{(name, variant value, labelled tuple value)}\\
 & & \mathtt{false} \;\mid\; \mathtt{true} \;\mid\; 42 \;\mid\; \ldots & \text{(base value)}
\end{array}$$

In π-calculus, *names* $x, y, \ldots$ can be intutively seen as variables (i.e., they can be substituted with values), and as communication channels (i.e., they can be used for input/output). Values can be names, base values like $\mathtt{false}$ or 42, variant values $l(v)$ and labelled tuples

[2] We use a *standard* projection with merging [67, 18]: for its definition (not crucial here), see [60].

$[l_i\,{:}\,v_i]_{i \in I}$. The **inaction** $\mathbf{0}$ and the **parallel composition** $P \mid Q$ are similar to Definition 2.1. The **restriction** $(\boldsymbol{\nu} x)P$ creates a new name x and binds it with scope P. The **replicated process** $*P$ represents infinite replicas of P, composed in parallel. The **output** $\overline{x}\langle v \rangle.P$ uses the name x to send a value v, and proceeds as P; the **input** $x(y).P$ uses x to receive a value that will substitute y in the continuation P. Process $\textbf{case}\, v \,\textbf{of}\, \{l_i(x_i) \triangleright P_i\}_{i \in I}$ **pattern matches** a variant value v, and if it has label l_i, substitutes x_i and continues as P_i. Process $\textbf{with}\, [l_i\,{:}\,x_i]_{i \in I} = v \,\textbf{do}\, P$ **destructs a labelled tuple** v, substituting each x_i in P. For brevity, we will often write “record” instead of “labelled tuple”.

▶ **Definition 3.2.** The π*-calculus operational semantics* is the relation $\to$ defined as:

$$\begin{array}{rl}
(\text{R}\pi\text{-Com}) & \overline{x}\langle v \rangle.P \mid x(y).Q \;\to\; P \mid Q\{^{v}\!/_{y}\} \\
(\text{R}\pi\text{-Case}) & \textbf{case}\, l_j(v) \,\textbf{of}\, \{l_i(x_i) \triangleright P_i\}_{i \in I} \;\to\; P_j\{^{v}\!/_{x_j}\} \quad (j \in I) \\
(\text{R}\pi\text{-With}) & \textbf{with}\, [l_i\,{:}\,x_i]_{i \in I} = [l_i\,{:}\,v_i]_{i \in I} \,\textbf{do}\, P \;\to\; P\{^{v_i}\!/_{x_i}\}_{i \in I} \\
(\text{R}\pi\text{-Res}) & P \to Q \;\;\text{implies}\;\; (\boldsymbol{\nu} x)P \;\to\; (\boldsymbol{\nu} x)Q \\
(\text{R}\pi\text{-Par}) & P \to Q \;\;\text{implies}\;\; P \mid R \;\to\; Q \mid R \\
(\text{R}\pi\text{-Struct}) & P \equiv P' \;\wedge\; P \to Q \;\wedge\; Q' \equiv Q \;\;\text{implies}\;\; P' \to Q'
\end{array}$$

Rule (Rπ-Com) models communication between output and input on a name x: it reduces to the corresponding continuations, with a value substitution on the receiver process. (Rπ-Case) says that **case** applied on a variant value $l_j(v)$ reduces to P_j, with v in place of x_j — provided that l_j is one of the supported cases (i.e., $l_j = l_i$ for some $i \in I$). Rule (Rπ-With) deconstructs a labelled tuple $[l_i\,{:}\,v_i]_{i \in I}$: it says that **with** reduces to its continuation P with v_i in place of each x_i, for all $i \in I$. By (Rπ-Res) and (Rπ-Par), reductions can happen under restriction and parallel composition, respectively. By (Rπ-Struct), reduction is closed under the structural congruence $\equiv$, whose definition is standard (see [59, Table 1.1] and [60]).

π-Calculus Typing. We now summarise the π-calculus types, subtyping, and typing rules.

▶ **Definition 3.3** (π-types)**.** The syntax of a π*-calculus type* T is given by:

$$\begin{array}{rll}
T \;::=\; & \mathsf{Li}(T) \mid \mathsf{Lo}(T) \mid \mathsf{L}\sharp(T) & \text{(linear input, linear output, linear connection)} \\
& \sharp(T) \mid \bullet & \text{(unrestricted connection, no capability)} \\
& \langle l_i\_T_i \rangle_{i \in I} \mid [l_i\,{:}\,T_i]_{i \in I} & \text{(variant, labelled tuple a.k.a. “record”)} \\
& \mu\mathbf{t}.T \mid \mathbf{t} \mid \mathtt{Bool} \mid \mathtt{Int} \mid \ldots & \text{(recursive type, type variable, base type)}
\end{array}$$

Linear types $\mathsf{Li}(T)$, $\mathsf{Lo}(T)$ denote, respectively, names used *exactly once* to input/output a value of type T. $\mathsf{L}\sharp(T)$ denotes a name used once for sending, and once for receiving, a message of type T. $\sharp(T)$ denotes an *unrestricted connection*, i.e., a name that can be used both for input/output any number of times. $\bullet$ is assigned to names that cannot be used for input/output. $\langle l_i\_T_i \rangle_{i \in I}$ is a labelled disjoint union of types, while $[l_i\,{:}\,T_i]_{i \in I}$ (that we will often call “record”) is a labelled product type; for both, labels l_i are all distinct, and their order is irrelevant. As syntactic sugar, we write $(T_i)_{i \in 1..n}$ for a record with integer labels $[i\,{:}\,T_i]_{i \in \{1,..,n\}}$. Recursive types and variables, and base types like $\mathtt{Bool}$, are standard.

The predicate $\mathrm{lin}(T)$ (Definition 3.4 below) holds iff T has some linear input/output component.

▶ **Definition 3.4** (Linear/unrestricted types)**.** The predicate lin is inductively defined as:

$$\mathrm{lin}(\mathsf{Li}(T)) \qquad \mathrm{lin}(\mathsf{Lo}(T)) \qquad \frac{\exists j \in I : \mathrm{lin}(T_j)}{\mathrm{lin}(\langle l_i\_T_i \rangle_{i \in I})} \qquad \frac{\exists j \in I : \mathrm{lin}(T_j)}{\mathrm{lin}([l_i\,{:}\,T_i]_{i \in I})} \qquad \frac{\mathrm{lin}\,(T)}{\mathrm{lin}\,(\mu\mathbf{t}.T)}$$

We write $\mathrm{un}(T)$ iff $\neg\,\mathrm{lin}(T)$ (i.e., T is unrestricted iff is not linear).

$$\text{(T}\pi\text{-Name)}\ \frac{\mathrm{un}(\Gamma)}{\Gamma, x\!:\!T \vdash x\!:\!T} \qquad \text{(T}\pi\text{-Basic)}\ \frac{\mathrm{un}(\Gamma) \qquad v \in B}{\Gamma \vdash v\!:\!B} \qquad \text{(T}\pi\text{-LVal)}\ \frac{\Gamma \vdash v\!:\!T}{\Gamma \vdash l(v)\!:\!\langle l\_T \rangle}$$

$$\text{(T}\pi\text{-LTup)}\ \frac{\mathrm{un}(\Gamma) \qquad \forall i \in I \qquad \Gamma_i \vdash v_i\!:\!T_i}{\left(\biguplus_{i\in I} \Gamma_i\right) \uplus \Gamma \vdash [l_i\!:\!v_i]_{i\in I}\!:\![l_i\!:\!T_i]_{i\in I}} \qquad \text{(T}\pi\text{-Sub)}\ \frac{\Gamma \vdash x\!:\!T \qquad T \leqslant_\pi T'}{\Gamma \vdash x\!:\!T'} \qquad \text{(T}\pi\text{-Nil)}\ \frac{\mathrm{un}(\Gamma)}{\Gamma \vdash \mathbf{0}}$$

$$\text{(T}\pi\text{-Par)}\ \frac{\Gamma_1 \vdash P \qquad \Gamma_2 \vdash Q}{\Gamma_1 \uplus \Gamma_2 \vdash P \mid Q} \qquad \text{(T}\pi\text{-Res1)}\ \frac{\Gamma, x\!:\!\dagger(T) \vdash P \qquad \dagger \in \{\mathsf{L}\sharp, \sharp\}}{\Gamma \vdash (\boldsymbol{\nu} x)P} \qquad \text{(T}\pi\text{-Res2)}\ \frac{\Gamma, x\!:\!\bullet \vdash P}{\Gamma \vdash (\boldsymbol{\nu} x)P}$$

$$\text{(T}\pi\text{-Inp)}\ \frac{\begin{array}{c}\Gamma_1 \vdash x\!:\!\dagger(T) \qquad \dagger \in \{\mathsf{Li}, \sharp\}\\ \Gamma_2, y\!:\!T \vdash P\end{array}}{\Gamma_1 \uplus \Gamma_2 \vdash x(y).P} \qquad \text{(T}\pi\text{-Out)}\ \frac{\begin{array}{c}\Gamma_1 \vdash x\!:\!\dagger(T) \qquad \dagger \in \{\mathsf{Lo}, \sharp\}\\ \Gamma_2 \vdash v\!:\!T \qquad \Gamma_3 \vdash P\end{array}}{\Gamma_1 \uplus \Gamma_2 \uplus \Gamma_3 \vdash \overline{x}\langle v \rangle.P} \qquad \text{(T}\pi\text{-Repl)}\ \frac{\Gamma \vdash P \qquad \mathrm{un}(\Gamma)}{\Gamma \vdash *P}$$

$$\text{(T}\pi\text{-Case)}\ \frac{\Gamma_1 \vdash v\!:\!\langle l_i\_T_i \rangle_{i\in I} \quad \forall i \in I \quad \Gamma_2, x_i\!:\!T_i \vdash P_i}{\Gamma_1 \uplus \Gamma_2 \vdash \mathbf{case}\, v\, \mathbf{of}\, \{l_i(x_i) \triangleright P_i\}_{i\in I}} \qquad \text{(T}\pi\text{-With)}\ \frac{\Gamma_1 \vdash v\!:\![l_i\!:\!T_i]_{i\in I} \quad \Gamma_2, \{x_i\!:\!T_i\}_{i\in I} \vdash P}{\Gamma_1 \uplus \Gamma_2 \vdash \mathbf{with}\, [l_i\!:\!x_i]_{i\in I} = v\, \mathbf{do}\, P}$$

Figure 5 Typing rules for the linear π-calculus.

▶ **Definition 3.5.** *Subtyping* $\leqslant_\pi$ for π-types is coinductively defined as:

$$\frac{B \leqslant_{\mathsf{B}} B'}{B \leqslant_\pi B'}\ \text{(S-LB)} \qquad \frac{}{\bullet \leqslant_\pi \bullet}\ \text{(S-LEnd)} \qquad \frac{T \leqslant_\pi T'}{\mathsf{Li}(T) \leqslant_\pi \mathsf{Li}(T')}\ \text{(S-Li)} \qquad \frac{T' \leqslant_\pi T}{\mathsf{Lo}(T) \leqslant_\pi \mathsf{Lo}(T')}\ \text{(S-Lo)}$$

$$\frac{\forall i \in I \quad T_i \leqslant_\pi T'_i}{\langle l_i\_T_i \rangle_{i\in I} \leqslant_\pi \langle l_i\_T'_i \rangle_{i\in I\cup J}}\ \text{(S-Variant)} \qquad \frac{\forall i \in I \quad T_i \leqslant_\pi T'_i}{[l_i\!:\!T_i]_{i\in I} \leqslant_\pi [l_i\!:\!T'_i]_{i\in I}}\ \text{(S-LTuple)} \qquad \frac{T\{^{\mu\mathbf{t}.T}/_{\mathbf{t}}\} \leqslant_\pi T'}{\mu\mathbf{t}.T \leqslant_\pi T'}\ \text{(S-L}\mu\text{L)}$$

By rule (S-LB), $\leqslant_\pi$ includes basic subtyping $\leqslant_{\mathsf{B}}$. (S-LEnd) relates types without I/O capabilities. By (S-Li) (resp. (S-Lo)), linear input (resp. output) subtyping is *covariant* (resp. *contravariant*) in the carried type. By (S-Variant), subtyping for variant types is *covariant* in *both* carried types *and* number of components. By (S-LTuple), subtyping for labelled tuples, a.k.a records, is *covariant* in the carried types. (Note: "full" record subtyping allows to add/remove entries [59, §7.3]; but here, "record" just means "labelled tuple".) Rule (S-LμL) (and its symmetric, omitted) relates a recursive type $\mu\mathbf{t}.T$ to T' iff its unfolding is related to T'.

▶ **Definition 3.6** (Typing context, type combination)**.** The *linear π-calculus typing context* Γ is a partial mapping defined as: $\Gamma \ ::= \ \emptyset \mid \Gamma, x\!:\!T$

We write $\mathrm{lin}(\Gamma)$ iff $\exists x\!:\!T \in \Gamma : \mathrm{lin}(T)$, and $\mathrm{un}(\Gamma)$ iff $\neg\,\mathrm{lin}(\Gamma)$. The *type combinator* $\uplus$ is defined as follows (and undefined in other cases), and is extended to typing contexts as expected.

$$\mathsf{Li}(T) \uplus \mathsf{Lo}(T) \ \triangleq\ \mathsf{L}\sharp(T) \qquad \mathsf{Lo}(T) \uplus \mathsf{Li}(T) \ \triangleq\ \mathsf{L}\sharp(T) \qquad T \uplus T \ \triangleq\ T \quad \text{if } \mathrm{un}(T)$$

$$(\Gamma_1 \uplus \Gamma_2)(x) \ \triangleq\ \begin{cases} \Gamma_1(x) \uplus \Gamma_2(x) & \text{if } x \in \mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) \\ \Gamma_i(x) & \text{if } x \in \mathrm{dom}(\Gamma_i) \setminus \mathrm{dom}(\Gamma_j) \end{cases}$$

Figure 5 shows the **typing system for the linear π-calculus**. Typing judgements have two forms: $\Gamma \vdash v\!:\!T$ and $\Gamma \vdash P$. (Tπ-Name) says that a name has the type assumed in the typing context; (Tπ-Basic) relates base values to their types; both rules require unrestricted typing contexts. By (Tπ-LVal), a variant value $l(v)$ is of type $\langle l\_T \rangle$ if value v is of type T. By (Tπ-LTup), a record value $[l_i\!:\!v_i]_{i\in I}$ is of type $[l_i\!:\!T_i]_{i\in I}$ if for all $i \in I$, v_i is of type T_i. (Tπ-Sub) is the *subsumption rule*: if x has type T in Γ, then it also has any *super*type of T. By (Tπ-Nil), $\mathbf{0}$ is well typed in every unrestricted typing context. By (Tπ-Par), the parallel composition of two processes is typed by combining the respective typing contexts. By (Tπ-Res1), the restriction process $(\boldsymbol{\nu} x)P$ is well typed if P is typed by augmenting the context with $x\!:\!\mathsf{L}\sharp(T)$. or $x\!:\!\sharp T$. In the first case, by applying Definition 3.6 ($\uplus$), we have $x\!:\!\mathsf{L}\sharp(T) = x\!:\!\mathsf{Li}(T) \uplus \mathsf{Lo}(T)$: this implies that P owns *both* capabilities of linear input/output

$$\mathbf{let}\,x\!=\!v\,\mathbf{in}\,P \quad \triangleq \quad (\boldsymbol{\nu} z)\,(\overline{z}\langle v\rangle.\mathbf{0} \mid z(x).P) \quad (\text{where } z \notin \{x\} \cup \text{fn}(v) \cup \text{fn}(P))$$

$$(\text{R}\pi\text{-Let}) \;\; \mathbf{let}\,x\!=\!v\,\mathbf{in}\,P \;\to\; P\{^{v}/_{x}\} \qquad (\text{T}\pi\text{-Let}) \; \frac{\Gamma_1 \vdash v : T \qquad \Gamma_2, x : T \vdash P}{\Gamma_1 \uplus \Gamma_2 \vdash \mathbf{let}\,x\!=\!v\,\mathbf{in}\,P}$$

$$(\text{T}\pi\text{-Narrow}) \; \frac{\Gamma, x : T \vdash P \quad T' \leqslant_\pi T}{\Gamma, x : T' \vdash P} \qquad (\text{T}\pi\text{-MSubst}) \; \frac{\forall i \in I \quad \Gamma_i \vdash v_i : T_i \quad \Gamma, \{x_i : T_i\}_{i \in I} \vdash P}{\left(\biguplus_{i\in I} \Gamma_i\right) \uplus \Gamma \vdash P\{^{v_i}/_{x_i}\}_{i\in I}}$$

Figure 6 “Let” binder (definition, reduction, typing), and narrowing / substitution rules.

of x. By (Tπ-Res2), the restriction $(\boldsymbol{\nu} x)P$ is typed if P is typed and x has no capabilities. By (Tπ-Inp) (resp. (Tπ-Out)), the input and output processes are typed if x is a (possibly linear) name used in input (resp. output), and the carried types are compatible with the type of y (resp. value v). The typing context used to type the input and output process is obtained by applying $\uplus$ on the premises. By (Tπ-Repl), a replicated process $*P$ is typed in the same unrestricted context that types P. By (Tπ-Case), $\mathbf{case}\,v\,\mathbf{of}\,\{l_i(x_i) \triangleright P_i\}_{i\in I}$ is typed if the guard value v has variant type, and every P_i is typed assuming $x_i : T_i$, for all $i \in I$. By (Tπ-With), process $\mathbf{with}\,[l_i : x_i]_{i\in I}\!=\!v\,\mathbf{do}\,P$ is typed if v is of record type and for all $i \in I$, each v_i has the same type as x_i, i.e., T_i.

4 Some Typed π-Calculus Extensions and Results

We introduce some definitions and results on typed π-calculus: we will need them in Section 5 and Section 6, to state our encoding and its properties. As we target *standard* typed π-calculus (Section 3), all our extensions are *conservative*, so to preserve standard results (e.g., subject reduction).

“Let” binder, narrowing, substitution. Figure 6 shows several auxiliary definitions and typing rules. $\mathbf{let}\,x\!=\!v\,\mathbf{in}\,P$ binds x in P, and reduces by replacing x with v in P. It is a macro on other π-calculus contructs: hence, rules (Rπ-Let)/(Tπ-Let) are based on the reduction/typing of its expansion (details in [60]). Rule (Tπ-Narrow) derives from the narrowing lemma [59, 7.2.5]. (Tπ-MSubst) represents zero or more applications of the substitution lemma [59, 8.1.4].

Duality and Recursive π-Types. The *duality* for linear π-types relates opposite but compatible input/output capabilities. Intuitively, the dual of a $\mathsf{Li}(T)$ is $\mathsf{Lo}(T)$ (and *vice versa*) [15]. Note that the carried type T *is the same*: i.e., dual types can be combined with $\uplus$ (Definition 3.6), yielding $\mathsf{L}\sharp(T)$. However, defining duality for *recursive* π-types is not straightforward: what is the dual of $T = \mu\mathbf{t}.\mathsf{Lo}(\mathbf{t})$? Is it maybe $T' = \mu\mathbf{t}.\mathsf{Li}(\mathbf{t})$? Since $\uplus$ is *not* defined for μ-types, we can check whether it is defined for the *unfoldings* of our hypothetical duals T and T'. Unfortunately, we have $\text{unf}(T) = \mathsf{Lo}(\mu\mathbf{t}.\mathsf{Lo}(\mathbf{t}))$ and $\text{unf}(T') = \mathsf{Li}(\mu\mathbf{t}.\mathsf{Li}(\mathbf{t}))$: i.e., $\uplus$ is again undefined, so T,T' cannot be considered duals. Solving this issue is crucial: in Section 5, we will need to encode recursive partial types, preserving their duality (Definition 2.8) in linear π-types.

What we want is a notion of duality that *commutes with unfolding*, so that if two recursive types are dual, and we unfold them, we get a dual pair $\mathsf{Lo}(T)/\mathsf{Li}(T)$ that can be combined with $\uplus$ (since they carry the same T). We address this issue by extending the π-calculus type variables (Definition 3.3) with their *dualised* counterpart, denoted with $\overline{\mathbf{t}}$. We allow recursive types such as $\mu\mathbf{t}.\mathsf{Li}(\overline{\mathbf{t}})$ (but *not* $\mu\overline{\mathbf{t}}\ldots$), and postulate that when unfolding, $\overline{\mathbf{t}}$ is substituted by a “dual” type $\mu\mathbf{t}.\mathsf{Lo}(\mathbf{t})$, as formalised in Definition 4.1 below. Quite interestingly, our approach reminds of the “logical duality” for session types [43], but we study it in the context of π-calculus (we will further discuss this topic in Section 8).

▶ **Definition 4.1.** $\overline{T}$ is the *dual of* T, and is defined as follows:

$$\overline{\mathsf{Li}(T)} \triangleq \mathsf{Lo}(T) \qquad \overline{\mathsf{Lo}(T)} \triangleq \mathsf{Li}(T) \qquad \overline{\bullet} \triangleq \bullet \qquad \overline{(\mathbf{t})} \triangleq \overline{\mathbf{t}} \qquad \overline{(\overline{\mathbf{t}})} \triangleq \mathbf{t} \qquad \overline{\mu\mathbf{t}.T} \triangleq \mu\mathbf{t}.\overline{T}\{\overline{\mathbf{t}}/\mathbf{t}\}$$

The *substitution* of T for a type variable $\mathbf{t}$ or $\overline{\mathbf{t}}$ is: $\mathbf{t}\{T/\mathbf{t}\} \triangleq T \quad \overline{\mathbf{t}}\{T/\mathbf{t}\} \triangleq \overline{T}$

The dual of a linear input type $\mathsf{Li}(T)$ is a linear output type $\mathsf{Lo}(T)$, and *vice versa*, with the payload type T unchanged, as expected. The dual of a terminated channel type $\bullet$ is itself. The dual of a type variable $\mathbf{t}$ is $\overline{\mathbf{t}}$, and the dual of a dualised type variable $\overline{\mathbf{t}}$ is $\mathbf{t}$, implying that duality on linear π-types is convolutive. The dual of $\mu\mathbf{t}.T$ is $\mu\mathbf{t}.\overline{T}\{\overline{\mathbf{t}}/\mathbf{t}\}$, where type T is dualised to $\overline{T}$, and every occurrence of $\mathbf{t}$ is replaced by its dual $\overline{\mathbf{t}}$ by Definition 4.1. Now, the desired commutativity between duality and unfolding holds, as per Lemma 4.2 below.

▶ **Lemma 4.2.** $\mathrm{unf}(\overline{T}) = \overline{\mathrm{unf}(T)}$.

▶ **Example 4.3.** Let $T = \mu\mathbf{t}.\mathsf{Li}((\mathbf{t},\overline{\mathbf{t}}))$. Then:

$\mathrm{unf}(T) = \mathsf{Li}\Big(\big(\mu\mathbf{t}.\mathsf{Li}((\mathbf{t},\overline{\mathbf{t}})), \overline{\mu\mathbf{t}.\mathsf{Li}((\mathbf{t},\overline{\mathbf{t}}))}\big)\Big) = \mathsf{Li}((\mu\mathbf{t}.\mathsf{Li}((\mathbf{t},\overline{\mathbf{t}})), \mu\mathbf{t}.\mathsf{Lo}((\overline{\mathbf{t}},\mathbf{t}))))$; and

$\mathrm{unf}(\overline{T}) = \mathrm{unf}(\mu\mathbf{t}.\mathsf{Lo}((\overline{\mathbf{t}},\mathbf{t}))) = \mathsf{Lo}((\mu\mathbf{t}.\mathsf{Li}((\mathbf{t},\overline{\mathbf{t}})), \mu\mathbf{t}.\mathsf{Lo}((\overline{\mathbf{t}},\mathbf{t})))) = \overline{\mathrm{unf}(T)}$

By adding dualised type variables in Definition 3.3, we naturally extend the definition of $\mathrm{fv}(T)$ (with $\mu\mathbf{t}\ldots$ binding both $\mathbf{t}$ and $\overline{\mathbf{t}}$), the subtyping relation $\leqslant_\pi$ in Definition 3.5 (by letting rules (S-LμL) and (S-LμR) use the substitution in Definition 4.1) and ultimately the typing system in Definition 3.6. Using these extensions, we will obtain a rather simple encoding of recursive session types (Definition 5.1), and solve a subtle issue involving duality, recursion and continuations (Example 5.3).

The reader might be puzzled about the impact of dualised variables in the π-calculus theory. We show that dualised variables *do not increase the expressiveness of linear π-types*, and *do not unsafely enlarge subtyping* $\leqslant_\pi$: this is proved in Lemma 4.4, that allows to erase dualised variables from recursive π-types. It uses

1. a substitution that *only* replaces dualised variables, i.e.: $\overline{\mathbf{t}}\{\mathbf{t}'/\overline{\mathbf{t}}\} = \mathbf{t}'$; and
2. the equivalence $=_\pi$ defined as: $\leqslant_\pi \cap \leqslant_\pi^{-1}$.

▶ **Lemma 4.4** (Erasure of $\overline{\mathbf{t}}$)**.** $\mu\mathbf{t}.T =_\pi \mu\mathbf{t}.T\{\mu\mathbf{t}'.\overline{T}\{\mathbf{t}'/\overline{\mathbf{t}}\}/\overline{\mathbf{t}}\}$, *for all* $\mathbf{t}' \not\in \mathrm{fv}(T)$.

▶ **Example 4.5** (Application of erasure)**.** Take T from Example 4.3. By Lemma 4.4, we have:

$T =_\pi \mu\mathbf{t}.\mathsf{Li}\Big(\big(\mathbf{t}, \mu\mathbf{t}'.\overline{\mathsf{Li}((\mathbf{t},\overline{\mathbf{t}}))}\{\mathbf{t}'/\overline{\mathbf{t}}\}\big)\Big) = \mu\mathbf{t}.\mathsf{Li}((\mathbf{t}, \mu\mathbf{t}'.\mathsf{Lo}((\mathbf{t},\mathbf{t}')))).$

Since $T =_\pi T'$ implies $T \leqslant_\pi T'$ and $T' \leqslant_\pi T$, Lemma 4.4 says that any $\mu\mathbf{t}.T$ is equivalent to a μ-type *without occurrences of* $\overline{\mathbf{t}}$: i.e., any typing relation with instances of $\overline{\mathbf{t}}$ corresponds to a $\overline{\mathbf{t}}$-free one. As a consequence, any typing derivation using $\overline{\mathbf{t}}$ can be turned into a $\overline{\mathbf{t}}$-free one. Summing up: adding dualised variables preserves the standard results of typed π-calculus.

Type Combinator $\cap\!\!\!\!+$. Definition 4.6 introduces a type combinator that is a “relaxed” version of $\uplus$ (Definition 3.6) extended with subtyping. We will use it to encode MPST typing contexts (Definition 5.6).

▶ **Definition 4.6.** The *π-calculus type combinator* $\cap\!\!\!\!+$ is defined on π-types as follows (and undefined in other cases), and naturally extended to typing contexts:

$$\left.\begin{array}{lcl} \mathsf{Lo}(T) \cap\!\!\!\!+ \mathsf{Li}(T') & \triangleq & \mathsf{Li}(T) \uplus \mathsf{Lo}(T) \\ \mathsf{Li}(T') \cap\!\!\!\!+ \mathsf{Lo}(T) & \triangleq & \mathsf{Li}(T) \uplus \mathsf{Lo}(T) \end{array}\right\}\text{if } T \leqslant_\pi T' \qquad T \cap\!\!\!\!+ T \triangleq T \quad \text{if } \mathrm{un}(T)$$

$$(\Gamma_1 \cap\!\!\!\!+ \Gamma_2)(x) \triangleq \begin{cases} \Gamma_1(x) \cap\!\!\!\!+ \Gamma_2(x) & \text{if } x \in \mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) \\ \Gamma_i(x) & \text{if } x \in \mathrm{dom}(\Gamma_i) \setminus \mathrm{dom}(\Gamma_j) \end{cases}$$

The difference between $\uplus$ and $\Cap$ is that the former combines linear inputs/outputs with *the same carried type*, while $\Cap$ is more relaxed: it allows a carried type to be subtype of the other — more exactly, the type carried by the output side can be smaller than the type carried by the input side. This is shown in Lemma 4.7 and Example 4.8 below.

▶ **Lemma 4.7.** *If* $T = T_1 \Cap T_2$*, and* $T'_1 \uplus T'_2 = T$*, then either*
(a) $T'_1 \leqslant_\pi T_1$ *and* $T'_2 \leqslant_\pi T_2$*, or*
(b) $T'_1 \leqslant_\pi T_2$ *and* $T'_2 \leqslant_\pi T_1$*.*

Lemma 4.7 says that $T_1 \Cap T_2$ (when defined) is a type that, when split using $\uplus$, yields linear I/O types that are subtypes of the originating T_1, T_2. Intuitively, it means that $\Cap$ can be soundly used to simplify typing derivations: if used to type some name x, it will yield (when defined) a type that can also be obtained by suitably using $\uplus$ and (Tπ-Sub) (Figure 5).

▶ **Example 4.8.** Let $T_1 = \mathsf{Li}(\mathtt{Real})$, $T_2 = \mathsf{Lo}(\mathtt{Int})$, and $T = T_1 \Cap T_2$. We have $T = \mathsf{L}\sharp(\mathtt{Int})$; if we let $T'_1 \uplus T'_2 = T$, then we get either *(a)* $T'_1 = \mathsf{Li}(\mathtt{Int}) \leqslant_\pi T_1$ and $T'_2 = \mathsf{Lo}(\mathtt{Int}) \leqslant_\pi T_2$, or *(b)* $T'_1 = \mathsf{Lo}(\mathtt{Int}) \leqslant_\pi T_2$ and $T'_2 = \mathsf{Li}(\mathtt{Int}) \leqslant_\pi T_1$.

5 Encoding Multiparty Session-π into Linear π-Calculus

We now present our encoding of MPST π-calculus into linear π-calculus. It consists of an *encoding of types* and an *encoding of processes*: combined, they preserve the safety properties of MPST communications, both w.r.t. typing and process behaviour.

Encoding of Types. Our goal is to decompose MPST channel endpoints into point-to-point π-calculus channels. This leads to the main intuition behind our approach: *encode MPST channel endpoints as labelled tuples*, whose labels are roles, and whose values are names (for communication). The idea is that if a multiparty channel of type S allows to talk with role $\mathtt{p}$, then the corresponding π-calculus record should have a label $\mathtt{p}$, mapping to a name that can send/receive messages to/from the process that plays the role $\mathtt{p}$. This suggests the type of an encoded MPST channel endpoint: it should be a π-calculus record — and since each name appearing in such record is used to communicate, it should have an input/output type.

▶ **Definition 5.1.** The *encoding of session type S into linear π-types* is: $[\![S]\!] \triangleq [\mathtt{p}{:}\,[\![S\restriction\mathtt{p}]\!]]_{\mathtt{p}\in S}$ where the encoding of the partial projections $[\![S\restriction\mathtt{p}]\!]$ is:

$$\begin{array}{llll}
[\![\oplus_{i\in I}\,!l_i(U_i).H_i]\!] \triangleq \mathsf{Lo}\big(\langle l_i\_([\![U_i]\!],\overline{[\![H_i]\!]})\rangle_{i\in I}\big) & [\![B]\!] \triangleq B & [\![\mathbf{end}]\!] \triangleq \bullet \\
[\![\&_{i\in I}\,?l_i(U_i).H_i]\!] \triangleq \mathsf{Li}\big(\langle l_i\_([\![U_i]\!],[\![H_i]\!])\rangle_{i\in I}\big) & [\![\mathbf{t}]\!] \triangleq \mathbf{t} & [\![\mu\mathbf{t}.H]\!] \triangleq \mu\mathbf{t}.[\![H]\!]
\end{array}$$

The encoding of a session type S, namely $[\![S]\!]$, is a record that maps each role $\mathtt{p}\in S$ to the encoding of the *partial projection* $[\![S\restriction\mathtt{p}]\!]$. The latter adopts the basic idea of the encoding of *binary, non-recursive* session types [36, 15]: it is the identity on a base type B, while a terminated channel type $\mathbf{end}$ becomes $\bullet$, with no capabilities. Selection $\oplus_{i\in I}\,!l_i(U_i).H_i$ and branching $\&_{i\in I}\,?l_i(U_i).H_i$ are encoded as linear output and input types, respectively, adopting a *continuation-passing style (CPS)*. In both cases, the carried types are variants: $\langle l_i\_([\![U_i]\!],\overline{[\![H_i]\!]})\rangle_{i\in I}$ for select and $\langle l_i\_([\![U_i]\!],[\![H_i]\!])\rangle_{i\in I}$ for branch, with the same labels as the originating partial projections. Such variants carry tuples $([\![U_i]\!],\overline{[\![H_i]\!]})$ and $([\![U_i]\!],[\![H_i]\!])$: the first element is the encoded payload type, and the second (i.e., the encoding of H_i) is the type of a *continuation name*: it is sent together with the encoded payload, and will be used to send/receive the *next* message (unless H_i is $\mathbf{end}$). Note that selection sends the *dual* of $[\![H_i]\!]$: this is because the *sender* must keep interacting according to $[\![H_i]\!]$, while the *recipient* must operate *dually* (cf. Definition 4.1). E.g., if $[\![H_i]\!]$ requires to send a message, the recipient of $\overline{[\![H_i]\!]}$ must receive it. The encodings of type variables and recursive types are homomorphic.

Note that by encoding session types as labelled tuples, we untangle the order of the interactions among different roles. We will recover this order later, when encoding processes.

▶ **Example 5.2.** Consider the session type $S \triangleq \mathtt{p}!l_1(\mathtt{Int}).\mathtt{q}?l_2(S').\mathbf{end}$, where $S' \triangleq \mathtt{r}!l_3(\mathtt{Bool}).\mathtt{q}?l_4(\mathtt{String}).\mathbf{end}$. By Definition 5.1, the encoding of S is:

$$\begin{aligned}[\![S]\!] \quad &= \quad [\mathtt{p}\colon [\![S \upharpoonright \mathtt{p}]\!], \mathtt{q}\colon [\![S \upharpoonright \mathtt{q}]\!]] \;=\; [\mathtt{p}\colon [\![!l_1(\mathtt{Int})]\!], \mathtt{q}\colon [\![?l_2(S')]\!]] \\ &= \quad [\mathtt{p}\colon \mathsf{Lo}(\langle l_1\_(\mathtt{Int}, \bullet)\rangle), \mathtt{q}\colon \mathsf{Li}(\langle l_2\_([\mathtt{r}\colon \mathsf{Lo}(\langle l_3\_(\mathtt{Bool}, \bullet)\rangle), \mathtt{q}\colon \mathsf{Li}(\langle l_4\_(\mathtt{String}, \bullet)\rangle)], \bullet)\rangle)]\end{aligned}$$

Recursion, Continuations and Duality. We now point out a subtle (but crucial) difference between Definition 5.1 and the encoding of *binary, non-recursive* session types in [15]. When encoding partial selections, our continuation type is the *dual of the encoding of* H_i, i.e., $\overline{[\![H_i]\!]}$; in [15], instead, it is the *encoding of the dual of* H_i, i.e., $[\![\overline{H_i}]\!]$. This difference is irrelevant for *non-recursive* types (Example 5.2); but for *recursive* types, using $[\![\overline{H_i}]\!]$ would yield the wrong continuations. Using $\overline{[\![H_i]\!]}$, instead, gives the expected result, by generating *dualised recursion variables* (cf. Definition 4.1). We explain it in Example 5.3 below.

▶ **Example 5.3.** Let $H = \mu\mathbf{t}.!l(\mathtt{Bool}).\mathbf{t}$. By Definition 5.1, we have:

$$[\![H]\!] \quad = \quad [\![\mu\mathbf{t}.!l(\mathtt{Bool}).\mathbf{t}]\!] = \mu\mathbf{t}.\mathsf{Lo}\Big(\langle l\_([\![\mathtt{Bool}]\!], \overline{[\![\mathbf{t}]\!]})\rangle\Big) = \mu\mathbf{t}.\mathsf{Lo}\big(\langle l\_(\mathtt{Bool}, \overline{\mathbf{t}})\rangle\big)$$

Let us now unfold the encoding of H. By Definition 4.1, we have:

$$\operatorname{unf}([\![H]\!]) \;=\; \operatorname{unf}\big(\mu\mathbf{t}.\mathsf{Lo}(\langle l\_(\mathtt{Bool}, \overline{\mathbf{t}})\rangle)\big) \;=\; \mathsf{Lo}\big(\langle l\_\big(\mathtt{Bool},\, \mu\mathbf{t}.\mathsf{Li}(\langle l\_(\mathtt{Bool}, \mathbf{t})\rangle)\big)\rangle\big)$$

This is what we want: since H requires a recursive output of $\mathtt{Bool}$eans, its encoding should output a $\mathtt{Bool}$ean, together with a *recursive input name* as continuation. Hence, the recipient will receive the first $\mathtt{Bool}$ean together with a continuation name, whose type mandates to recursively input more $\mathtt{Bool}$s. If encoding continuations as in [15], instead, we would have:

$$\begin{aligned}[\![H]\!] \quad &= \quad \mu\mathbf{t}.\mathsf{Lo}\big(\langle l\_([\![\mathtt{Bool}]\!], [\![\overline{\mathbf{t}}]\!])\rangle\big) = \mu\mathbf{t}.\mathsf{Lo}\big(\langle l\_(\mathtt{Bool}, \mathbf{t})\rangle\big) \;\; (\mathbf{t} \text{ is } \textit{not} \text{ dualised}) \\ \operatorname{unf}([\![H]\!]) \quad &= \quad \mathsf{Lo}\big(\langle l\_(\mathtt{Bool}, \mu\mathbf{t}.\mathsf{Lo}(\langle l\_(\mathtt{Bool}, \mathbf{t})\rangle))\rangle\big)\end{aligned}$$

which is wrong: the recipient is required to recursively *output* $\mathtt{Bool}$s. This wrong encoding would also prevent us from obtaining Theorem 6.1 later on.

Encoding of Typing Contexts. In order to preserve type safety, we want to *encode a session judgement (Figure 4) into a π-calculus typing judgement (Figure 5)*. For this reason, we now use the encoding of session types (Definition 5.1) to formalise the encoding of session typing contexts.

▶ **Definition 5.4.** The *encoding of a session typing context* is:

$$\begin{array}{cccc} [\![\varnothing]\!] \triangleq \varnothing & [\![\Theta \cdot \Gamma]\!] \triangleq [\![\Theta]\!], [\![\Gamma]\!] & [\![c\!:\!U]\!] \;\triangleq\; [\![c]\!]\!:\![\![U]\!] & [\![s[\mathtt{p}]]\!] \;\triangleq\; z_{s[\mathtt{p}]} \\ \Big[\!\!\Big[\Theta, X\!:\!\widetilde{U}\Big]\!\!\Big] \triangleq [\![\Theta]\!], \Big[\!\!\Big[X\!:\!\widetilde{U}\Big]\!\!\Big] & [\![\Gamma, c\!:\!U]\!] \triangleq [\![\Gamma]\!], [\![c\!:\!U]\!] & [\![x]\!] \;\triangleq\; x & [\![X]\!] \;\triangleq\; z_X \\ [\![\Gamma_1 \circ \Gamma_2]\!] \triangleq [\![\Gamma_1]\!] \uplus [\![\Gamma_2]\!] & \multicolumn{3}{c}{[\![X\!:\!U_1, \ldots, U_n]\!] \;\triangleq\; [\![X]\!]\!:\!\sharp\big(([\![U_i]\!])_{i \in 1..n}\big)} \end{array}$$

When encoding typing contexts, variables (x) keep their name, while process variables (X) and channels with roles ($s[\mathtt{p}]$) are turned into distinguished names with a subscript: e.g., X becomes z_X. The composition $\Gamma_1 \circ \Gamma_2$ (Definition 2.11) is encoded using $\uplus$ (Definition 3.6): such an operation is always defined, since the domains of $[\![\Gamma_1]\!]$,$[\![\Gamma_2]\!]$ can only overlap on basic types.

Note that encoded process variables have an *unrestricted* connection type, carrying an n-tuple of encoded argument types; encoded sessions, instead, are linearly-typed, similarly to [15]: this will allow to exploit the (partial) confluence properties of linear π-calculus [37] to prove Theorem 6.5 later. Moreover, this will lead to the implementation discussed in Section 7.

Encoding Typing Judgements: Overview. With these definitions at hand, we can now have a first look at the encoding of session typing judgements in Figure 7 (but we postpone the formal statement to Definition 5.7 later on, as it requires some more technical developments).

Terminated processes are encoded homomorphically. **Parallel composition** is also encoded homomorphically — i.e., our encoding preserves the choreographic distribution of the originating processes. Note that $\llbracket P \rrbracket_{\Theta\cdot\Gamma_1}$ and $\llbracket Q \rrbracket_{\Theta\cdot\Gamma_2}$ are the encoded processes yielded respectively by $\llbracket \Theta\cdot\Gamma_1 \vdash P \rrbracket$ and $\llbracket \Theta\cdot\Gamma_2 \vdash Q \rrbracket$: they exist because such typing judgements hold, by inversion of (T-Par) (Figure 4). Similar uses of sub-processes encoded w.r.t. their typing occur in the other cases. **Process declaration** $\mathbf{def}\ X(\widetilde{x}\,{:}\,U) = P\ \mathbf{in}\, Q$ is encoded as a replicated π-calculus process that inputs a value z on a name $\llbracket X \rrbracket = z_X$ (matching Definition 5.4), deconstructs it into $x_1, \ldots, x_n$ (using **with**, and hence assuming that z is an n-tuple), and then continues as the encoding of the body P; meanwhile, the encoding of Q runs in parallel, enclosed by a delimitation on z_X (that matches the scope of the original declaration). Correspondingly, a **process call** $X\langle\widetilde{v}\rangle$ is encoded as a process that sends the encoded values $\llbracket \widetilde{v} \rrbracket$ on z_X and ends (in MPST π-calculus, process calls are in tail position).

Selection on $c[\mathtt{p}]$ is encoded using information from the session typing context: the fact that c has type $S = \mathtt{p} \oplus !l(U).S'$ — i.e., $\llbracket S \rrbracket$ is a record type with one entry $\mathtt{q}\,{:}\,z_{\mathtt{q}}$ for each $\mathtt{q} \in S$. Therefore, the encoding first deconstructs $\llbracket c \rrbracket$ (using **with**), an then uses the (linear) name in its $\mathtt{p}$-entry to output on $z_{\mathtt{p}}$. Before performing the output, however, a new name z is created: it is the *continuation* of the interaction with $\mathtt{p}$. Then, one endpoint of z is sent through $z_{\mathtt{p}}$ as part of $l(\llbracket v \rrbracket, z)$, which is a variant value carrying a tuple. The other endpoint of z is kept, and used to rebind $\llbracket c \rrbracket$ (using **let**) with a "new" record, consisting in *all* the entries of the "original" $\llbracket c \rrbracket$, *except* $z_{\mathtt{p}}$ (which has been used for output). More in detail, the "new" $\llbracket c \rrbracket$ has an entry for $\mathtt{p}$ (mapping $\mathtt{p}$ to z) iff S' still involves $\mathtt{p}$ (otherwise, if $\mathtt{p} \notin S'$, then z is discarded, since it has type $\llbracket S' {\upharpoonright} \mathtt{p} \rrbracket = \llbracket \mathbf{end} \rrbracket = \bullet$). After **let**, the encoding continues as $\llbracket P \rrbracket$.

Symmetrically, **branching** on $c[\mathtt{p}]$ is also encoded using information from the typing context, i.e., that c has type $S = \mathtt{p}\, \&_{i\in I}\, ?l_i(U_i).S'_i$ — and therefore, $\llbracket S \rrbracket$ is a record type with one entry $\mathtt{q}\,{:}\,z_{\mathtt{q}}$ for each $\mathtt{q} \in S$. As above, the encoded process deconstructs $\llbracket c \rrbracket$ (using **with**), an then uses the (linear) name in its $\mathtt{p}$-entry to perform an input $z_{\mathtt{p}}(y)$; y is assumed to be a variant, and is pattern matched to determine the continuation. If y matches l_i (for some $i \in I$), *and* it carries a tuple $z_i = (x_i, z)$ (where z is a continuation name), then $\llbracket c \rrbracket$ is rebound (using **let**) and the process continues as $\llbracket P_i \rrbracket$. The rebinding of $\llbracket c \rrbracket$ depends on l_i and the continuation type S'_i: the "new" $\llbracket c \rrbracket$ is a record with *all* the linear names of the "original" $\llbracket c \rrbracket$, *except* $z_{\mathtt{p}}$ (which has been used for input); as above, an entry for $\mathtt{p}$ will exist (and map $\mathtt{p}$ to z) iff S'_i still involves $\mathtt{p}$ (otherwise, if $\mathtt{p} \notin S'_i$, then z has type $\bullet$ and is discarded).

We will explain the encoding of **session restriction** $(\boldsymbol{\nu} s)P$ later, after Definition 5.7, as it requires some technicalities: namely, the substitution $\boldsymbol{\sigma}(\Gamma')$. We can, however, have an intuition about the role of $\boldsymbol{\sigma}(\Gamma')$ by considering an obvious discrepancy. Consider the following session π-calculus process, that reduces by communication (cf. Definition 2.3):

$$\Gamma, s[\mathtt{p}] : S, s[\mathtt{q}] : S' \vdash s[\mathtt{p}][\mathtt{q}]\&\{l(x).P\} \mid s[\mathtt{q}][\mathtt{p}] \oplus \langle l(v)\rangle.Q \quad\to\quad P\{{}^{v}\!/\!{}_{x}\} \mid Q \tag{1}$$

We would like its encoding to reduce and communicate, too — but it is *not* the case:

$$\mathbf{with}\,[\mathtt{r} : z_{\mathtt{r}}]_{\mathtt{r}\in S} = \llbracket s[\mathtt{p}] \rrbracket\ \mathbf{do} \ldots \mid \mathbf{with}\,[\mathtt{r} : z_{\mathtt{r}}]_{\mathtt{r}\in S'} = \llbracket s[\mathtt{q}] \rrbracket\ \mathbf{do} \ldots \quad \not\to \tag{2}$$

$$\llbracket \Gamma \vdash \mathbf{0} \rrbracket \triangleq \llbracket \Gamma \rrbracket \vdash \mathbf{0} \qquad \llbracket \Theta \cdot \Gamma_1 \circ \Gamma_2 \vdash P \mid Q \rrbracket \triangleq \llbracket \Theta \cdot \Gamma_1 \circ \Gamma_2 \rrbracket \vdash \llbracket P \rrbracket_{\Theta\cdot\Gamma_1} \mid \llbracket Q \rrbracket_{\Theta\cdot\Gamma_2}$$

$$\underbrace{\left\llbracket \Theta \cdot \Gamma \vdash \mathbf{def}\ X(\widetilde{x}:\widetilde{U}) = P\ \mathbf{in}\ Q \right\rrbracket}_{\text{where } \widetilde{U} = U_1,\dots,U_n \text{ and } \widetilde{x} = x_1,\dots,x_n \text{ and } \widetilde{v} = v_1,\dots,v_n} \triangleq \begin{array}{l} \llbracket \Theta \cdot \Gamma \rrbracket \vdash \\ (\boldsymbol{\nu} \llbracket X \rrbracket) \Big(* \Big(\llbracket X \rrbracket(z).\mathbf{with}\,(x_i)_{i\in\{1..n\}} = z\ \mathbf{do}\ \llbracket P \rrbracket_{\Theta,X:\widetilde{U}\cdot\widetilde{x}:\widetilde{U}} \Big) \mid \llbracket Q \rrbracket_{\Theta,X:\widetilde{U}\cdot\Gamma} \Big) \end{array}$$

$$\overbrace{\left\llbracket \Theta, X:\widetilde{U} \cdot \Gamma_1 \circ \ldots \circ \Gamma_n \circ \Gamma \vdash X\langle \widetilde{v} \rangle \right\rrbracket} \triangleq \left\llbracket \Theta, X:\widetilde{U} \cdot \Gamma_1 \circ \ldots \circ \Gamma_n \circ \Gamma \right\rrbracket \vdash \overline{\llbracket X \rrbracket}\langle (\llbracket v_i \rrbracket)_{i\in\{1..n\}} \rangle.\mathbf{0}$$

$$\underbrace{\llbracket \Theta \cdot c:S, \Gamma_1 \circ \Gamma_2 \vdash c[\mathsf{p}] \oplus \langle l(v) \rangle.P \rrbracket}_{\text{where } S = \mathsf{p} \oplus !l(U).S'} \triangleq \underbrace{\begin{array}{l} \llbracket \Theta, c:S, \Gamma_1 \circ \Gamma_2 \rrbracket \vdash \\ \mathbf{with}\ \big[\mathsf{q}:z_{\mathsf{q}}\big]_{\mathsf{q}\in S} = \llbracket c \rrbracket\ \mathbf{do}\ (\boldsymbol{\nu} z)\overline{z_{\mathsf{p}}}\langle l(\llbracket v \rrbracket, z) \rangle.\mathbf{let}\ \llbracket c \rrbracket = \maltese\ \mathbf{in}\ \llbracket P \rrbracket_{\Theta\cdot\Gamma_2,c:S'} \end{array}}_{\text{where } \maltese = \begin{cases} [\mathsf{p}:z,\ \mathsf{q}:z_{\mathsf{q}}]_{\mathsf{q}\in S'\setminus\mathsf{p}} & \text{if } \mathsf{p} \in S' \\ \big[\mathsf{q}:z_{\mathsf{q}}\big]_{\mathsf{q}\in S'} & \text{otherwise} \end{cases}}$$

$$\underbrace{\llbracket \Theta \cdot c:S, \Gamma \vdash c[\mathsf{p}]\ \&_{i\in I}\ \{l_i(x_i).P_i\} \rrbracket}_{\text{where } S = \mathsf{p}\ \&_{i\in I}\ ?l_i(U_i).S'_i} \triangleq \underbrace{\begin{array}{l} \llbracket \Theta, c:S, \Gamma \rrbracket \vdash \mathbf{with}\ \big[\mathsf{q}:z_{\mathsf{q}}\big]_{\mathsf{q}\in S} = \llbracket c \rrbracket\ \mathbf{do}\ z_{\mathsf{p}}(y).\mathbf{case}\ y\ \mathbf{of}\ \Big\{ \\ \qquad l_i(z_i) \triangleright \mathbf{with}\,(x_i, z) = z_i\ \mathbf{do\ let}\ \llbracket c \rrbracket = \maltese_i\ \mathbf{in}\ \llbracket P_i \rrbracket_{\Theta\cdot\Gamma'} \Big\}_{i\in I} \end{array}}$$

$$\text{where } \Gamma' = \Gamma, x_i:U_i, c:S'_i \text{ and } \maltese_i = \begin{cases} [\mathsf{p}:z,\ \mathsf{q}:z_{\mathsf{q}}]_{\mathsf{q}\in S'_i\setminus\mathsf{p}} & \text{if } \mathsf{p} \in S'_i \\ \big[\mathsf{q}:z_{\mathsf{q}}\big]_{\mathsf{q}\in S'_i} & \text{otherwise} \end{cases}$$

$$\underbrace{\llbracket \Theta \cdot \Gamma \vdash (\boldsymbol{\nu} s:\Gamma')P \rrbracket}_{\text{where } \mathrm{conn}(s,\Gamma') = \{\{\mathsf{p}_1,\mathsf{q}_1\},\dots,\{\mathsf{p}_n,\mathsf{q}_n\}\}} \triangleq \llbracket \Theta \cdot \Gamma \rrbracket \vdash \underbrace{\llbracket (\boldsymbol{\nu} s) \rrbracket \llbracket P \rrbracket_{\Theta\cdot\Gamma,\Gamma'} \boldsymbol{\sigma}(\Gamma')}_{\text{where } \llbracket (\boldsymbol{\nu} s) \rrbracket = (\boldsymbol{\nu} z_{\{s,\mathsf{p}_i,\mathsf{q}_i\}})_{i\in\{1..n\}}}$$

Figure 7 Encoding of typing judgements. Here, $\llbracket P \rrbracket_{\Theta\cdot\Gamma} = Q$ iff $\llbracket \Theta\cdot\Gamma \vdash P \rrbracket = \llbracket \Theta\cdot\Gamma \rrbracket \vdash Q$ (Definition 5.7).

and the reason is that $\llbracket s[\mathsf{p}] \rrbracket, \llbracket s[\mathsf{q}] \rrbracket$ are "just" record-typed *names* (respectively $z_{s[\mathsf{p}]}, z_{s[\mathsf{q}]}$, as per Definition 5.4), whereas **with**-prefixes only reduce when applied to *record values* (cf. Definition 3.2). Hence, to let our encoded terms reduce, we must first substitute $\llbracket s[\mathsf{p}] \rrbracket, \llbracket s[\mathsf{q}] \rrbracket$ with two records; moreover, to let the two encoded processes synchronise and exchange $\llbracket v \rrbracket$, such records must be suitably defined: we must ensure that the entries for q (in one record) and p (in the other) map to *the same (linear) name.* In the following, we show how $\boldsymbol{\sigma}(\Gamma')$ handles this issue.

Reification of Multiparty Sessions. By simply translating a channel with role $s[\mathsf{p}]$ into a π-calculus name $z_{s[\mathsf{p}]}$, we have not yet captured the insight behind our approach, i.e., the idea that a multiparty session can be decomposed into a labelled tuple of linear channels (i.e., π-calculus names), connecting *pairs of roles.* We can formalise "connections" as follows.

▶ **Definition 5.5.** The *connections of* s *in* Γ are: $\mathrm{conn}(s,\Gamma) \triangleq \big\{ \{\mathsf{p},\mathsf{q}\} \bigm| s[\mathsf{p}]:S_{\mathsf{p}} \in \Gamma \wedge \mathsf{q} \in S_{\mathsf{p}} \big\}$

Intuitively, two roles p, q are connected by s in Γ if p occurs in the type $\Gamma(s[\mathsf{q}])$ (but q *might not* occur in $\Gamma(s[\mathsf{p}])$; note, however, that q will always occur if Γ is consistent).
Now, as anticipated above, we want to substitute each $\llbracket s[\mathsf{p}] \rrbracket$ with a suitably defined record, containing π-calculus names; moreover, such names must be typed in the typing context. But what are exactly such names, and their types? This is answered by Definition 5.6.

▶ **Definition 5.6** (Reification and decomposition of MPST contexts). The *reification of a session typing context* Γ_{S} is the substitution:

$$\boldsymbol{\sigma}(\Gamma_{\mathsf{S}}) \;=\; \Big\{ \big[\mathsf{q}:z_{\{s,\mathsf{p},\mathsf{q}\}}\big]_{\mathsf{q}\in S_{\mathsf{p}}} / \llbracket s[\mathsf{p}] \rrbracket \Big\}_{s[\mathsf{p}]:S_{\mathsf{p}}\in\Gamma_{\mathsf{S}}}$$

The *linear decomposition of* Γ_{S} is the π-calculus typing context $\boldsymbol{\delta}(\Gamma_{\mathsf{S}})$, defined as:

$$\boldsymbol{\delta}(\Gamma_{\mathsf{S}}) \;=\; \mathop{\cap\!\!\!\!+}_{s[\mathsf{p}]:S_{\mathsf{p}}\in\Gamma_{\mathsf{S}}} \Big\{ z_{\{s,\mathsf{p},\mathsf{q}\}} : \big\llbracket \mathrm{unf}\big(S_{\mathsf{p}} \upharpoonright \mathsf{q}\big) \big\rrbracket \Big\}_{\{\mathsf{p},\mathsf{q}\}\in\mathrm{conn}(s,\Gamma_{\mathsf{S}})}$$

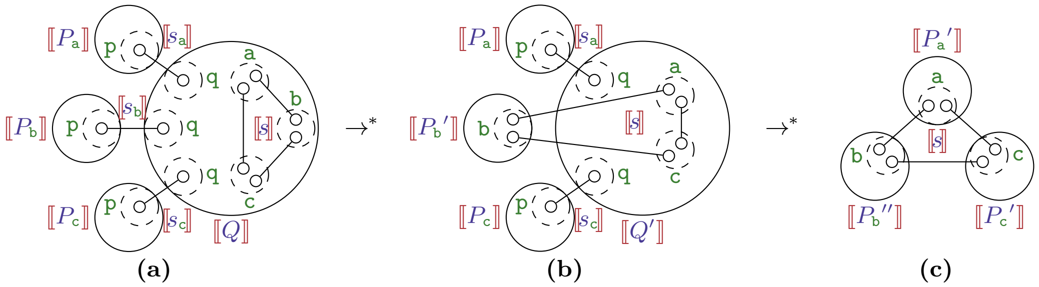
$\llbracket P_{a} \rrbracket$
$\llbracket s_{a} \rrbracket$
$\llbracket P_{b} \rrbracket$
$\llbracket s_{b} \rrbracket$
$\llbracket P_{c} \rrbracket$
$\llbracket s_{c} \rrbracket$
$\llbracket s \rrbracket$
$\llbracket Q \rrbracket$
p
q
a
b
c
$\to^*$
$\llbracket P_{b}' \rrbracket$
$\llbracket Q' \rrbracket$
$\llbracket P_{a}' \rrbracket$
$\llbracket P_{b}'' \rrbracket$
$\llbracket P_{c}' \rrbracket$
(a)
(b)
(c)

Figure 8 Multiparty peer-to-peer game: encoded version of Figure 2. Lines are binary channels.

The π-calculus *reification typing rule* is (note that Γ_{S}, Γ'_{S} are *MPST* typing contexts):

$$\frac{\llbracket \Theta \cdot \Gamma_{S} \rrbracket, \llbracket \Gamma'_{S} \rrbracket \vdash P}{\llbracket \Theta \cdot \Gamma_{S} \rrbracket, \boldsymbol{\delta}(\Gamma'_{S}) \vdash P\boldsymbol{\sigma}(\Gamma'_{S})} \; (\text{T}\pi\text{-Reify})$$

The simplest part of Definition 5.6 is $\boldsymbol{\sigma}(\Gamma_{S})$: it is a substitution that, for each $s[\mathtt{p}]{:}S_{\mathtt{p}} \in \Gamma_{S}$, replaces $\llbracket s[\mathtt{p}] \rrbracket$ with a record containing one entry $\mathtt{q}{:}z_{\{s,\mathtt{p},\mathtt{q}\}}$ for each $\mathtt{q} \in S_{\mathtt{p}}$. Note that if there is also some $s[\mathtt{q}]{:}S_{\mathtt{q}} \in \Gamma_{S}$ with $\mathtt{p} \in S_{\mathtt{q}}$, then the corresponding record (replacing $\llbracket s[\mathtt{q}] \rrbracket$) has an entry $\mathtt{p}{:}z_{\{s,\mathtt{q},\mathtt{p}\}} = z_{\{s,\mathtt{p},\mathtt{q}\}}$; i.e., $\mathtt{p}$ (in one record) and $\mathtt{q}$ (in the other) *map to the same name*. This realises the intuition of "multiparty sessions as records of interconnected binary channels".

The definition of $\boldsymbol{\sigma}(\Gamma_{S})$ was the last ingredient needed to formalise our encoding, presented in Definition 5.7 below. The rest of Definition 5.6 will be used later on, to prove its correctness (Theorem 6.2): hence, we postpone its explanation to page 22.

▶ **Definition 5.7** (Encoding). The *encoding of session typing judgements* is given in Figure 7. We define $\llbracket P \rrbracket_{\Theta \cdot \Gamma} = Q$ iff $\llbracket \Theta \cdot \Gamma \vdash P \rrbracket = \llbracket \Theta \cdot \Gamma \rrbracket \vdash Q$. Sometimes, we write $\llbracket P \rrbracket$ for $\llbracket P \rrbracket_{\Theta \cdot \Gamma}$ when Θ, Γ are empty, or clear from the context.

We conclude by explaining the last case in Figure 7, which was not addressed on p.19. The process $(\boldsymbol{\nu} s{:}\Gamma')P$ is encoded by generating one delimitation for each $z_{\{s,\mathtt{p}_i,\mathtt{q}_i\}}$ whenever $\{\mathtt{p}_i, \mathtt{q}_i\}$ is a connection of s in Γ' (Definition 5.5). Then, P is encoded, and the substitution $\boldsymbol{\sigma}(\Gamma')$ is applied: it replaces each $\llbracket s[\mathtt{p}_i] \rrbracket$, $\llbracket s[\mathtt{q}_i] \rrbracket$ in $\llbracket P \rrbracket$ with records based on the delimited $z_{\{s,\mathtt{p}_i,\mathtt{q}_i\}}$.

▶ **Example 5.8.** Consider (1). If we delimit s and encode the resulting process, we obtain a π-calculus process based on (2), enclosed by the delimitations yielded by $\llbracket (\boldsymbol{\nu} s) \rrbracket$, and the substitution $\boldsymbol{\sigma}(s[\mathtt{p}]{:}S, s[\mathtt{q}]{:}S', \ldots)$. Since the latter replaces $\llbracket s[\mathtt{p}] \rrbracket$, $\llbracket s[\mathtt{q}] \rrbracket$ with records whose entries reflect roles(S) and roles(S'), the encoding can now reduce, firing the two **with**s.

▶ **Example 5.9.** Consider the main server/clients parallel composition in Example 2.2:

$$(\boldsymbol{\nu} s_{a}, s_{b}, s_{c})\big(Q \mid P_{a} \mid P_{b} \mid P_{c}\big) \quad \textit{where}$$
$$Q \;=\; (\boldsymbol{\nu} s)\Big(s_{a}[\mathtt{q}][\mathtt{p}] \oplus \langle \mathtt{PlayA}(s[\mathtt{a}]) \rangle \mid s_{b}[\mathtt{q}][\mathtt{p}] \oplus \langle \mathtt{PlayB}(s[\mathtt{b}]) \rangle \mid s_{c}[\mathtt{q}][\mathtt{p}] \oplus \langle \mathtt{PlayC}(s[\mathtt{c}]) \rangle\Big)$$

Its encoding is the following process, with s decomposed into 3 linear channels (see Figure 8):

$$(\boldsymbol{\nu} z_{\{s_{a},\mathtt{p},\mathtt{q}\}}, z_{\{s_{b},\mathtt{p},\mathtt{q}\}}, z_{\{s_{c},\mathtt{p},\mathtt{q}\}})\big(\llbracket Q \rrbracket \mid \llbracket P_{a} \rrbracket \mid \llbracket P_{b} \rrbracket \mid \llbracket P_{c} \rrbracket\big) \quad \textit{where}$$
$$\llbracket Q \rrbracket \;=\; (\boldsymbol{\nu} z_{\{s,\mathtt{a},\mathtt{b}\}}, z_{\{s,\mathtt{b},\mathtt{c}\}}, z_{\{s,\mathtt{a},\mathtt{c}\}})\Big(\llbracket s_{a}[\mathtt{q}][\mathtt{p}] \oplus \langle \mathtt{PlayA}(s[\mathtt{a}]) \rangle \rrbracket \mid \llbracket s_{b}[\mathtt{q}][\mathtt{p}] \oplus \langle \mathtt{PlayB}(s[\mathtt{b}]) \rangle \rrbracket \mid \llbracket s_{c}[\mathtt{q}][\mathtt{p}] \oplus \langle \mathtt{PlayC}(s[\mathtt{c}]) \rangle \rrbracket\Big)$$

6 Properties of the Encoding

In this section we present some crucial properties ensuring the correctness of our encoding.

Encoding of Types. Theorem 6.1 below says that our encoding

1. commutes the duality between partial session types (Definition 2.8) and π-types (Definition 4.1), and
2. also preserves subtyping.

▶ **Theorem 6.1** (Duality/subtyping preservation). $[\![\overline{H}]\!] = \overline{[\![H]\!]}$*; if* $U \leqslant_{\mathsf{S}} U'$*, then* $[\![U]\!] \leqslant_{\pi} [\![U']\!]$.

Encoding of Typing Judgements. Theorem 6.2 shows that the encoding of session typing judgements into π-calculus typing judgements is valid. As a consequence, a well-typed MPST process also enjoys the type safety guarantees that can be expressed in standard π-calculus.

▶ **Theorem 6.2** (Correctness of encoding). $\Gamma \vdash v : U$ *implies* $[\![\Gamma]\!] \vdash [\![v]\!] : [\![U]\!]$*,* $\Theta \vdash X : \widetilde{U}$ *implies* $[\![\Theta]\!] \vdash [\![X]\!] : \widetilde{[\![U]\!]}$*, and* $\Theta \cdot \Gamma \vdash P$ *implies* $[\![\Theta \cdot \Gamma \vdash P]\!]$.

The proof is by induction on the MPST typing derivation, and yields a corresponding π-calculus typing derivation. One simple case is the following, that relates subtyping:

$$\text{(T-Sub)}\ \frac{\Theta \cdot \Gamma, c{:}U \vdash P \quad U' \leqslant_{\mathsf{S}} U}{\Theta \cdot \Gamma, c{:}U' \vdash P} \qquad \text{implies} \qquad \frac{[\![\Theta \cdot \Gamma, c{:}U \vdash P]\!] \quad [\![U']\!] \leqslant_{\pi} [\![U]\!]}{[\![\Theta \cdot \Gamma, c{:}U']\!] \vdash [\![P]\!]_{\Theta \cdot \Gamma, c:U}}\ \begin{array}{c}\text{(T}\pi\text{-Narrow)} \\ \text{(Figure 6)}\end{array}$$

and holds by the induction hypothesis and Theorem 6.1. The most delicate case is the encoding of session restriction $\Theta \cdot \Gamma \vdash (\boldsymbol{\nu} s{:}\Gamma')P$ (Figure 7): its encoding turns $(\boldsymbol{\nu} s)$ into a set of delimited names, used in the substitution $\boldsymbol{\sigma}(\Gamma')$ applied to $[\![P]\!]_{\Theta \cdot \Gamma, \Gamma'}$. Hence, to prove Theorem 6.2 in this case, we need to type such names, i.e., produce a context that types $[\![P]\!]_{\Theta \cdot \Gamma, \Gamma'} \boldsymbol{\sigma}(\Gamma')$. This is where $\boldsymbol{\delta}(\Gamma')$ and (Tπ-Reify) (Definition 5.6) come into play, as we now explain.

More on reification and decomposition. By Definition 5.6, the typing context $\boldsymbol{\delta}(\Gamma_{\mathsf{S}})$, *when defined*, $\boldsymbol{\delta}(\Gamma_{\mathsf{S}})$ has an entry for each role of each channel in Γ_{S}; more precisely, an entry $z_{\{s,\mathsf{p},\mathsf{q}\}}$ for each $s[\mathsf{p}] : S_{\mathsf{p}} \in \Gamma_{\mathsf{S}}$ and $\mathsf{q} \in S_{\mathsf{p}}$. Such entries are used to type the records yielded by $\boldsymbol{\sigma}(\Gamma_{\mathsf{S}})$. The type of $z_{\{s,\mathsf{p},\mathsf{q}\}}$ is based on the encoding of the unfolded partial projection $S_{\mathsf{p}} \restriction \mathsf{q}$, that can be either $\bullet$, or $\mathsf{Li}(T)/\mathsf{Lo}(T)$ (for some T). Note that if there is also some $s[\mathsf{q}] : S_{\mathsf{q}} \in \Gamma_{\mathsf{S}}$ with $\mathsf{p} \in S_{\mathsf{q}}$, the type of $z_{\{s,\mathsf{q},\mathsf{p}\}} = z_{\{s,\mathsf{p},\mathsf{q}\}}$ *(when defined)* is $[\![\mathrm{unf}(S_{\mathsf{p}} \restriction \mathsf{q})]\!] \mathbin{⩀} [\![\mathrm{unf}(S_{\mathsf{q}} \restriction \mathsf{p})]\!]$. This creates a deep correspondence between the consistency of Γ_{S} and the existence of $\boldsymbol{\delta}(\Gamma_{\mathsf{S}})$, shown in Theorem 6.3: *the precondition for MPST type safety (i.e., consistency of* Γ_{S}*) is precisely characterised in* π*-calculus by the linear decomposition at the roots of our encoding.*

▶ **Theorem 6.3** (Precise decomposition). Γ_{S} *is consistent* if and only if $\boldsymbol{\delta}(\Gamma_{\mathsf{S}})$ *is defined.*

The final part of Definition 5.6 is the π-calculus typing rule (Tπ-Reify), that uses $\boldsymbol{\delta}(\Gamma'_{\mathsf{S}})$ to type a process on which $\boldsymbol{\sigma}(\Gamma'_{\mathsf{S}})$ has been applied. Intuitively, $\boldsymbol{\delta}(\Gamma'_{\mathsf{S}})$ provides a typing context that types each record yielded by $\boldsymbol{\sigma}(\Gamma'_{\mathsf{S}})$. We now explain how the rule works and why it is sound (with a slight simplification). Let $\Gamma'_{\mathsf{S}} = \{s[\mathsf{p}] : S_{\mathsf{p}}\}_{\mathsf{p} \in I}$, for some I. Then, by Definition 5.6:

$$\boldsymbol{\delta}(\Gamma'_{\mathsf{S}}) \;=\; ⩀_{\mathsf{p} \in I} \left\{ z_{\{s,\mathsf{p},\mathsf{q}\}} : [\![\mathrm{unf}\big(S_{\mathsf{p}} \restriction \mathsf{q}\big)]\!] \right\}_{\{\mathsf{p},\mathsf{q}\} \in \mathrm{conn}(s, \Gamma_{\mathsf{S}})} \qquad \boldsymbol{\sigma}(\Gamma'_{\mathsf{S}}) \;=\; \left\{ {}^{[\mathsf{q} : z_{\{s,\mathsf{p},\mathsf{q}\}}]_{\mathsf{q} \in S_{\mathsf{p}}}} / {}_{[\![s[\mathsf{p}]]\!]} \right\}_{\mathsf{p} \in I}$$

(Note: $\boldsymbol{\delta}(\Gamma'_{\mathsf{S}})$ is defined *iff* Γ'_{S} is consistent, by Theorem 6.3). Take the I/O types yielded by $\boldsymbol{\delta}(\Gamma'_{\mathsf{S}})$, i.e., $\{T_{(s,\mathsf{p},\mathsf{q})}\}_{\{\mathsf{p},\mathsf{q}\} \in \mathrm{conn}(s, \Gamma_{\mathsf{S}})}$ such that $\boldsymbol{\delta}(\Gamma'_{\mathsf{S}}) = \biguplus_{\mathsf{p} \in I} \{z_{\{s,\mathsf{p},\mathsf{q}\}} : T_{(s,\mathsf{p},\mathsf{q})}\}_{\{\mathsf{p},\mathsf{q}\} \in \mathrm{conn}(s, \Gamma_{\mathsf{S}})}$ (note $T_{(s,\mathsf{p},\mathsf{q})}$, $T_{(s,\mathsf{q},\mathsf{p})}$ are distinct). If we assume $[\![\Theta \cdot \Gamma_{\mathsf{S}}]\!], [\![\Gamma'_{\mathsf{S}}]\!] \vdash P$, this derivation holds:

$$\frac{\left\{ \forall \mathsf{q} \in S_{\mathsf{p}} \quad \dfrac{\dfrac{\text{(T}\pi\text{-Name)}\ \dfrac{}{z_{\{s,\mathsf{p},\mathsf{q}\}} : T_{(s,\mathsf{p},\mathsf{q})} \vdash z_{\{s,\mathsf{p},\mathsf{q}\}} : T_{(s,\mathsf{p},\mathsf{q})}} \qquad T_{(s,\mathsf{p},\mathsf{q})} \leqslant_{\pi} [\![S_{\mathsf{p}} \restriction \mathsf{q}]\!]}{z_{\{s,\mathsf{p},\mathsf{q}\}} : T_{(s,\mathsf{p},\mathsf{q})} \vdash z_{\{s,\mathsf{p},\mathsf{q}\}} : [\![S_{\mathsf{p}} \restriction \mathsf{q}]\!]}\ \text{(T}\pi\text{-Sub)}}{\{z_{\{s,\mathsf{p},\mathsf{q}\}} : [\![S_{\mathsf{p}} \restriction \mathsf{q}]\!]\}_{\mathsf{q} \in S_{\mathsf{p}}} \vdash [\mathsf{q} : z_{\{s,\mathsf{p},\mathsf{q}\}}]_{\mathsf{q} \in S_{\mathsf{p}}}}\ \text{(T}\pi\text{-Rec)} \right\}_{\mathsf{p} \in I} \quad [\![\Theta \cdot \Gamma_{\mathsf{S}}]\!], [\![\Gamma'_{\mathsf{S}}]\!] \vdash P}{[\![\Theta \cdot \Gamma_{\mathsf{S}}]\!], \boldsymbol{\delta}(\Gamma'_{\mathsf{S}}) = [\![\Theta \cdot \Gamma]\!] \uplus \boldsymbol{\delta}(\Gamma'_{\mathsf{S}}) \vdash P\boldsymbol{\sigma}(\Gamma'_{\mathsf{S}})}\ \text{(T}\pi\text{-MSubst - Figure 6)}$$

In particular, the assumptions $T_{(s,\mathrm{p},\mathrm{q})} \leqslant_{\pi} [\![S_{\mathrm{p}} \upharpoonright \mathrm{q}]\!]$ hold by Lemma 4.7, since each $T_{(s,\mathrm{p},\mathrm{q})}$ is obtained by splitting $\boldsymbol{\delta}(\Gamma'_{\mathsf{S}})$ (that combines types with $\boxplus$) using $\uplus$. The equivalence in the conclusion holds since $\mathrm{dom}([\![\Theta\cdot\Gamma_{\mathsf{S}}]\!])\cap\mathrm{dom}(\boldsymbol{\delta}(\Gamma'_{\mathsf{S}}))=\emptyset$. Hence: if the (Tπ-Reify) premise ($[\![\Theta \cdot \Gamma_{\mathsf{S}}]\!], [\![\Gamma'_{\mathsf{S}}]\!] \vdash P$) holds, the above derivation holds, proving the conclusion of (Tπ-Reify).

Now, we can finish the proof of Theorem 6.2 for the case $\Theta\cdot\Gamma\vdash(\boldsymbol{\nu} s:\Gamma')P$. Assuming that the judgement holds, we also have $\Theta \cdot \Gamma, \Gamma' \vdash P$ and Γ' complete (by the premise of (T-Res), Figure 4): hence, Γ' is consistent, and $\boldsymbol{\delta}(\Gamma')$ is defined (by Theorem 6.3). Assuming that $[\![\Theta\cdot\Gamma, \Gamma' \vdash P]\!]$ holds (by the induction hypothesis), we obtain:

$$\dfrac{[\![\Theta \cdot \Gamma]\!], [\![\Gamma']\!] \vdash [\![P]\!]_{\Theta\cdot\Gamma,\Gamma'}}{[\![\Theta \cdot \Gamma]\!], \boldsymbol{\delta}(\Gamma') \vdash [\![P]\!]_{\Theta\cdot\Gamma,\Gamma'}\boldsymbol{\sigma}(\Gamma')}\ (\text{T}\pi\text{-Reify})$$

where $\boldsymbol{\delta}(\Gamma')$ types all the names $z_{\{s,\mathrm{p},\mathrm{q}\}}$ in $\boldsymbol{\sigma}(\Gamma')$, that are also delimited by $[\![(\boldsymbol{\nu} s)]\!]$. We can conclude by applying (Tπ-Res1) to type such delimitations (cf. Figure 5 — this is allowed by the completeness of Γ'): we get $[\![\Theta \cdot \Gamma]\!] \vdash [\![(\boldsymbol{\nu} s)]\!][\![P]\!]_{\Theta\cdot\Gamma,\Gamma'}\boldsymbol{\sigma}(\Gamma')$, i.e., we match Figure 7.

Finally, notice (from Figure 7) that our encoding of processes uses some typing information. In principle, a process could be typed by applying the rules in multiple ways (especially (T-Sub) in Figure 4), and one might wonder whether an MPST process could have multiple encodings. Proposition 6.4 says that this is *not* the case: the reason is that the only typing information being used is the set of roles in each session type, which does not depend on the typing rule — and is constant w.r.t. subtyping (i.e., $S\leqslant_{\mathsf{S}} S'$ implies $\mathrm{roles}(S)=\mathrm{roles}(S')$).

▶ **Proposition 6.4** (Uniqueness). *If* $\Theta \cdot \Gamma \vdash P$ *and* $\Theta' \cdot \Gamma' \vdash P$, *then* $[\![P]\!]_{\Theta\cdot\Gamma} = [\![P]\!]_{\Theta'\cdot\Gamma'}$.

Encoding and Reduction. One usual way to assess that an encoding is "behaviourally correct" (i.e., a process and its encoding behave "in the same way") consists in proving *operational correspondence*. Roughly, it says that the encoding is:

1. *complete*, i.e., any reduction of the original process is simulated by its encoding; and
2. *sound*, i.e., any reduction of the encoded process matches some reduction of the original process.

This is formalised in Theorem 6.5, where $\xrightarrow{\mathbf{with}}$ denotes a reduction induced by (Rπ-With) (Definition 3.2).

▶ **Theorem 6.5** (Operational correspondence). *If* $\varnothing \cdot \varnothing \vdash P$, *then:*

1. (Completeness) $P \to^* P'$ *implies* $\exists \widetilde{x}, P''$ *such that* $[\![P]\!] \to^* (\boldsymbol{\nu}\widetilde{x})P''$ *and* $P'' = [\![P']\!]$;
2. (Soundness) $[\![P]\!] \to^* P_*$ *implies* $\exists\widetilde{x}, P'', P'$ *s.t.* $P_* \to^* (\boldsymbol{\nu}\widetilde{x})P''$, $P \to^* P'$ *and* $[\![P']\!] \xrightarrow{\mathbf{with}}{}^* P''$.

The statement of Theorem 6.5 is standard [23, §5.1.3]. Item 1 says that if P reduces to P', then the encoding of the former can reduce to the encoding of the latter. Item 2 says (roughly) that no matter how the encoding of P reduces, it can always further reduce to the encoding of some P', such that P reduces to P'. Note that when we write $[\![P']\!]$, we mean $[\![P']\!]_{\varnothing\cdot\varnothing}$, which implies $\varnothing \cdot \varnothing \vdash P'$ (cf. Definition 5.7). The restricted variables $\widetilde{x}$ in items 1-2 are generated by the encoding of selection (Figure 7): it creates a (delimited) linear name to continue the session. To see why item 2 uses $\xrightarrow{\mathbf{with}}{}^*$, consider the following MPST process:

$$\varnothing \cdot \Gamma, s[\mathrm{p}] : S \vdash s[\mathrm{p}][\mathrm{q}]\&\{l(x).P\} \quad \not\to \qquad \text{(the process is stuck)}$$

If we encode it (and apply $\boldsymbol{\sigma}(\Gamma, s[\mathrm{p}] : S)$ as per Example 5.8), we get a π-calculus process that gets stuck, too — but *only after firing one internal* **with***-reduction*:

$$\mathbf{with}\,[\mathtt{r} : z_{\mathtt{r}}]_{\mathtt{r}\in S} = [\mathtt{r} : z_{\{s,\mathrm{p},\mathrm{r}\}}]_{\mathtt{r}\in S}\ \mathbf{do}\ z_{\mathrm{q}}(y).\ldots \ \xrightarrow{\mathbf{with}}\ z_{\{s,\mathrm{p},\mathrm{q}\}}(y).\ldots \ \not\to$$

This happens whenever a process is deadlocked, because in Figure 7, the "atomic" MPST branch/select actions are encoded with multiple π-calculus steps: first **with** to deconstruct the channels tuple, and then input/output. In general, if an MPST process is stuck, its encoding fires *one* **with** for each branch/select, then blocks on an input/output.

Theorem 6.5 yields a corollary on deadlock freedom (Corollary 6.6), that in turn allows to transfer deadlock freedom (Theorem 2.19) from MPST to π-calculus processes (Corollary 6.7 below).

▶ **Corollary 6.6.** *P is deadlock-free* if and only if *$[\![P]\!]$ is deadlock-free, i.e.: $[\![P]\!] \to^* P' \not\to$ implies $\exists Q \equiv \mathbf{0}$ such that $P' = [\![Q]\!]$.*

▶ **Corollary 6.7.** *Let $\varnothing \cdot \varnothing \vdash P$, where $P \equiv (\boldsymbol{\nu} s\!:\!G)\big|_{i \in I} P_i$ and each P_i* only interacts on *$s[\mathtt{p_i}]$. Then, $[\![P]\!]$ is* deadlock-free*.*

7 From Theory to Implementation

We can now show how our encoding directly guides the implementation of a toolchain for generating safe multiparty session APIs in Scala, supporting *distributed delegation*. We continue our Game example from Section 1, focusing on player b: we sketch the API generation and an implementation of a client, following the results in Section 6. Our approach is to:

1. exploit *type safety and distribution* provided by an existing library for *binary* session channels, and then
2. treat the *ordering* of communications *across separate channels* in the API generation.

Scala and `lchannels`. Our Scala toolchain is built upon the `lchannels` library [61, 62]. `lchannels` provides two key classes, `Out[T]` and `In[T]`, whose instances must be used *linearly* (i.e., *once*) to send/receive (by method calls) a `T`-typed message: i.e., they represent channel endpoints with π-calculus types $\mathsf{Lo}(T)$ and $\mathsf{Li}(T)$ (Definition 3.3). This approach enforces the typing of I/O actions via *static* Scala typing; the *linear usage of channels*, instead, goes beyond the capabilities of the Scala typing system, and is therefore enforced with *run-time* checks.

`lchannels` delivers messages by abstracting over various transports: local memory, TCP sockets, Akka actors [41]. Notably, `lchannels` promotes session type-safety through a *continuation-passing-style* encoding of *binary* session types [61] that is close to our encoding of partial projections (formalised in Definition 5.1). Further, `lchannels` allows to send/receive `In[T]`/`Out[T]` instances for *binary session delegation* [61, Example 4.3]; on *distributed* message transports, instances of `In[T]`/`Out[T]` can be sent remotely (e.g., via the Akka-based transport).

Type-safe, distributed multiparty delegation. By Theorem 6.2, Definition 5.1 and Theorem 6.3, we know that the game player session type $S_{\mathtt{b}}$ in our example (see Section 1, page 3) provides the type safety guarantees of a tuple of (linear) channels, whose types are given by the encoded partial projections of $S_{\mathtt{b}}$ onto a and c (Definition 2.9). This suggests that, using `lchannels`, the delegation of an $S_{\mathtt{b}}$-typed channel (as seen in Section 1) could be rendered in Scala as:

```
In[PlayB]   with definitions:   case class PlayB(payload: Sb)
                                case class Sb(a: In[InfoAB],c: Out[InfoBC])
```

i.e., as a linear input channel carrying a message of type `PlayB`, whose `payload` has type $\mathtt{S}_{\mathtt{b}}$; $\mathtt{S}_{\mathtt{b}}$, in turn, is a Scala `case class`, which can be seen as a labelled tuple, that maps `a`,`c` to I/O channels — whose types derive from $[\![S_{\mathtt{b}}\!\restriction\!\mathtt{a}]\!]$ and $[\![S_{\mathtt{b}}\!\restriction\!\mathtt{c}]\!]$ (in fact, they carry messages of type `InfoAB`/`InfoBC`). In this view, $\mathtt{S}_{\mathtt{b}}$ is our Scala rendering of the encoded session type $[\![S_{\mathtt{b}}]\!]$. As said above, `lchannels` allows to send channels remotely — hence, also allows to remotely send *tuples* of channels (e.g., instances of $\mathtt{S}_\mathtt{b}$); thus, with this simple approach, we obtain *type-safe distributed multiparty delegation* of an $[\![S_\mathtt{b}]\!]$-typed channel tuple "for free".

Multiparty API generation. Corresponding to the π-calculus labelled tuple type yielded by the *type* encoding $[\![S_\mathtt{b}]\!]$, the $\mathtt{S}_\mathtt{b}$ class outlined above can ensure communication safety, i.e., no unexpected message will be sent or received on any of its binary channels. Like $[\![S_\mathtt{b}]\!]$, however, $\mathtt{S}_\mathtt{b}$ does not convey any *ordering* to communications *across* channels: i.e., $\mathtt{S}_\mathtt{b}$ does not suggest the order in which its fields a,c should be used. (Indeed, $[\![S_\mathtt{b}]\!]$ may type π-processes that use its separate channels in *any* order, while preserving type safety.) To recover the "desired" ordering of communications, and implement it *correctly*, we can refine our classes so that:

1. each multiparty channel class (e.g., $\mathtt{S}_\mathtt{b}$) exposes a send() or receive() method, according to the I/O action expected by the multiparty session type (e.g., $S_\mathtt{b}$);
2. the implementation of such method uses the binary channels as per our *process encoding*.

E.g., consider again $S_\mathtt{b}$ and $\mathtt{S}_\mathtt{b}$. $S_\mathtt{b}$ requires to *send* towards c, so $\mathtt{S}_\mathtt{b}$ could provide the API:

```
case class Sb(a: In[InfoAB], c: Out[InfoBC]) {
  def send(v: String) = {   // v is the payload of InfoBC message
    val c′ = c !! InfoBC(v)_ // lchannels method: send v, and return continuation
    S′b(a, c′) } }           // return a "continuation object"
```

Now, $\mathtt{S}_\mathtt{b}$.send() behaves *exactly* as our process encoding in Figure 7 (case for selection $\oplus$): it picks the correct channel from the tuple (in this case, c), creates a new tuple $\mathtt{S}'_\mathtt{b}$ where c maps to a continuation channel, and returns it — so that the caller can use it to continue the multiparty session interaction. The class $\mathtt{S}'_\mathtt{b}$ should be similar, with a receive() method that uses a for input (by following the encoding of $\&$). This way, a programmer is correctly led to write, e.g., val x = s.send(...).receive() (using method call chaining) — whereas attempting, e.g., s.receive() is rejected by the Scala compiler (method undefined). These send()/receive() APIs are mechanical, and can be automatically generated: we did it by extending Scribble.

Scribble-Scala Toolchain. Scribble is a practical MPST-based language and tool for describing global protocols [63, 68]. To implement our results, we have extended Scribble (both the language and the tool) to support the full MPST theory in Section 2, including, e.g., projection, type merging and delegation (not previously supported). Our extension supports protocols with the syntax in Figure 9 (left), by augmenting Scribble with a *projection operator* @; then, it computes the projections/encodings explained in Section 5, and automates the Scala API generation as outlined above (producing, e.g., the $\mathtt{S}_\mathtt{b}$, $\mathtt{S'}_\mathtt{b}$,... classes and their send/receive methods). This approach reminds the Java API generation in [29] — but we follow a formal foundation and target the type-safe binary channels provided by lchannels (that, as shown above, takes care of most irksome aspects — e.g., delegation). As a result, the $P_\mathtt{b}$ client in Figure 1 can be written as in Figure 9 (right); and although conceptually programmed as Figure 2, the networking mechanisms of the game will concretely follow Figure 8.

8 Conclusion and Related Works

We presented the *first* encoding of a full-fledged multiparty session π-calculus into standard π-calculus (Section 5), and used it as the foundation of the *first* implementation of multiparty sessions (based on Scala API generation) supporting *distributed multiparty delegation*, on top of existing libraries (Section 7). We proved that the type safety property of MPST is precisely characterised by our decomposition into linear π-calculus (Theorem 6.3). We encode types by preserving duality and subtyping (Theorem 6.1); our encoding of processes is type-preserving,

```
global protocol ClientA(role p, role q) {
  PlayA(Game@a) from q to p; } // Delegation payload
global protocol ClientB(role p, role q) {
  PlayB(Game@b) from q to p; }
global protocol ClientC(role p, role q) {
  PlayC(Game@c) from q to p; }


global protocol Game(role a, role b, role c) {
  InfoBC(String) from b to c;
  InfoCA(String) from c to a;
  InfoAB(String) from a to b;
  rec t { choice at a {
    Mov1AB(Int) from a to b;
    Mov1BC(Int) from b to c;
    choice at c { Mov1CA(Int) from c to a; continue t; }
             or { Mov2CA(Bool) from c to a; continue t; }
   } or {
    Mov2AB(Bool) from a to b;
    Mov2BC(Bool) from b to c;
    choice at c { Mov1CA(Int) from c to a; continue t; }
             or { Mov2CA(Bool) from c to a; continue t; }
} } }
```

```
def P_b(c_bin: In[binary.PlayB]) = { // Cf. Ex.2.2
  // Wrap binary chan in generated multiparty API
  Client_b(MPPlayB(c_bin))
}

def Client_b(y: MPPlayB): Unit = {
  // Receive Game chan (wraps binary chans to a/c)
  val z = y.receive().p // p is the payload field
  // Send info to c; wait info from a; enter loop
  Loop_b(z.send(InfoBC("...")).receive())
}

def Loop_b(x: MPMov1ABOrMov2AB): Unit = {
  x.receive() match { // Check a's move
    case Mov1AB(p, cont) => {
      // cont only allows to send Mov1BC
      Loop_b(cont.send(Mov1BC(p)))
    }
    case Mov2AB(p, cont) => {
      // cont only allows to send Mov2BC
      Loop_b(cont.send(Mov2BC(p)))
}}} // If e.g. case Mov2AB missing: compiler warn
```

Figure 9 Game example (Section 1). Left: Scribble protocols for client/server setup sessions, and main *Game* (Example 2.18). Right: Scala code for player b, using Scribble-generated APIs to mimick Example 2.2.

and operationally sound and complete (Theorem 6.2 and Theorem 6.5); hence, our encoding preserves the type-safety and deadlock-freedom properties of MPST (Corollary 6.7). These results ensure the correctness of our (encoding-based) Scala implementation. Moreover, our encoding *preserves process distribution* (i.e., is homomorphic w.r.t. parallel composition); correspondingly, our implementation of multiparty sessions is decentralised and *choreographic*.

Session Types for "Mainstream" Languages. We mentioned *binary* session implementations for various languages in Section 1. Notably, [57, 32, 33, 42, 52, 61, 55] seek to integrate session types in the *native* host language, without language extensions, to avoid hindering their use in practice. To do so, one approach (e.g. in [61, 55]) is combining *static* typing of I/O actions with *run-time* checking of linear channel usage. Our implementation adopts this idea (Section 7). Haskell-based works exploit its richer typing system to statically enforce linearity — with various expressiveness/usability trade-offs based on their session types embedding strategy.

Implementations of *multiparty* sessions are few and limited, due to the intricacies of the theory (e.g., the interplay between *projections*, *mergability* and *consistency*), and practical issues (e.g., realising multiparty abstractions over binary transports, including distributed delegation), as discussed in Section 1. [64] was the first implementation of MPST, based on extending Java with session primitives. [29] proposes MPST-based API generation for Java, based on CFSMs [7], but has no formalisation — unlike our implementation, that follows our encoding. [17, 20] develop MPST-influenced networking APIs in Python and Erlang; [50] implements recovery strategies in Erlang. [17, 20, 50] focus on *purely dynamic* MPST verification via run-time monitoring. [51, 48] extends [17] with actors and timed specifications. [46] uses a dependent MPST theory to verify MPI programs. Crucially, *none* of these implementations supports delegation (nor projection merging, needed by our Game example, cf. Example 2.14).

Encodings of Session Types and Processes. [16] encodes binary session π-calculus into an augmented π-calculus with branch/select constructs. [15], following [36], and [21] encode *non-recursive, binary* session π-calculus, respectively into linear π-calculus and the Generic Type System for π-calculus [31], proving correctness w.r.t. typing and reduction. All the above works investigate *binary* and (except [16]) *non-recursive* session types, while in this paper we study the encoding of *multiparty* session types, subsuming binary ones; and unlike [16], we target *standard* π-calculus. We encode branching/selection using variants as in [15, 13], but our treatment of recursion, and the rest of the MPST theory, is novel.

Encodings of multiparty into binary sessions are studied in [9, 8]. Both use *orchestration* approaches that add centralised *medium/arbiter* processes, and target session calculi (*not* π-calculus). [53] uses a limited class of global types to extract "characteristic" deadlock-free π-calculus processes — without addressing session calculi, nor proving operational properties.

Recursion and Duality. The interplay between recursion and duality has been a thorny issue in session types literature, requiring our careful treatment in Section 4. [6, 1] noticed that the *standard duality* in [26] does *not* commute with the unfolding of recursion when type variables occur as payload, e.g., $\mu \mathbf{t}.!\mathbf{t}.\mathbf{end}$. To solve this issue, [6, 1] define a new notion of duality, called *complement* [1], then used in [13] to encode *recursive binary* session types into linear π-types. Unfortunately, [2] later noticed that even complement does *not* commute, e.g., when unfolding $\mu \mathbf{t}.\mu \mathbf{t}'.!\mathbf{t}.\mathbf{t}'$. As observed in Section 4, to encode *recursive* session types we encounter similar issues in π-types. The reason seems natural: π-types do not distinguish "payloads" and "continuations", and in recursive linear inputs/outputs, type variables always occur as "payload", e.g., $\mu \mathtt{t}.\mathsf{Lo}(\mathtt{t})$. Since, in the light of [2], we could not adopt the approach of [13], we propose a solution similar to [43]: introduce *dualised type variables* $\overline{\mathtt{t}}$. [43] also sketches a property similar to our Lemma 4.4. The main difference is that we add dualised variables to π-types (while [43] adds $\overline{\mathbf{t}}$ to session types). An alternative idea is given in [61]: encoding recursive session types as *non-recursive* linear I/O types with *recursive payloads*. This avoids dualised variables (e.g., $\mathsf{Lo}(\mu \mathtt{t}.\mathsf{Li}(\mathtt{t}))$ instead of $\mu \mathtt{t}.\mathsf{Lo}(\overline{\mathtt{t}})$), but if adopted, would complicate Definition 5.1. Moreover, [61] studies the encoding of recursive types, but not processes.

Future work. On the practical side, we plan to study whether Scala language extensions could provide stronger *static* channel usage checks. E.g., [25, 24] (capabilities) could allow to ensure that a channel endpoint is not used after being sent; [58, 65] (effects) could allow to ensure that a channel endpoint is actually used in a given method. We also plan to extend our multiparty API generation approach beyond Scala and `lchannels`, targeting other languages and implementations of binary sessions/channels [57, 32, 33, 42, 52, 55].

On the theoretical side, our encoding provides a basis for reusing theoretical results and tools between MPST π-calculus and standard π-calculus. E.g., we could now exploit Corollary 6.6, to verify deadlock-freedom of processes with interleaved multiparty sessions (studied in [3, 10, 12]) by applying π-calculus deadlock detection methods to their encodings [38, 35, 66]. Moreover, we can prove that our encoding is *barb-preserving*: hence, we plan to study its *full abstraction* properties w.r.t. *barbed congruence* in session π-calculus [40, 39] and π-calculus.

Thanks to the reviewers for their remarks, and to B. Toninho for fruitful discussions. Thanks to S.S. Jongmans, R. Neykova, N. Ng, B. Toninho for testing the companion artifact.

References

1 Giovanni Bernardi and Matthew Hennessy. Using higher-order contracts to model session types (extended abstract). In *CONCUR*, 2014. doi:10.1007/978-3-662-44584-6_27.

2 Giovanni Bernardi and Matthew Hennessy. Using higher-order contracts to model session types. *Logical Methods in Computer Science*, 12(2), 2016. doi:10.2168/LMCS-12(2:10)2016.

3 Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, 2008. doi:10.1007/978-3-540-85361-9_33.

4 Laura Bocchi, Julien Lange, and Nobuko Yoshida. Meeting Deadlines Together. In *CONCUR*, 2015. doi:http://dx.doi.org/10.4230/LIPIcs.CONCUR.2015.283.

5 Laura Bocchi, Julien Lange, and Nobuko Yoshida. Meeting Deadlines Together (long version). Technical report, 2015. Long version of [4]. URL: http://mrg.doc.ic.ac.uk/publications/meeting-deadlines-together/long.pdf.

6 Viviana Bono and Luca Padovani. Typing copyless message passing. *Logical Methods in Computer Science*, 8(1), 2012. doi:10.2168/LMCS-8(1:17)2012.

7 Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2), April 1983. doi:10.1145/322374.322380.

8 Luís Caires and Jorge A. Pérez. Multiparty session types within a canonical binary theory, and beyond. In *FORTE*, 2016. doi:10.1007/978-3-319-39570-8_6.

9 Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. Coherence generalises duality: A logical explanation of multiparty session types. In *CONCUR*, 2016. doi:10.4230/LIPIcs.CONCUR.2016.33.

10 Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. Inference of global progress properties for dynamically interleaved multiparty sessions. In *COORDINATION*, 2013. doi:10.1007/978-3-642-38493-6_4.

11 Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. A gentle introduction to multiparty asynchronous session types. In *Formal Methods for Multicore Programming*, 2015. doi:10.1007/978-3-319-18941-3_4.

12 Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global Progress for Dynamically Interleaved Multiparty Sessions. *Mathematical Structures in Computer Science*, 760, 2015. doi:10.1017/S0960129514000188.

13 Ornela Dardha. Recursive session types revisited. In *BEAT*, 2014. doi:10.4204/EPTCS.162.4.

14 Ornela Dardha. *Type Systems for Distributed Programs: Components and Sessions*, volume 7 of *Atlantis Studies in Computing*. Atlantis Press, July 2016. doi:10.2991/978-94-6239-204-5.

15 Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *PPDP*, 2012. doi:10.1145/2370776.2370794.

16 Romain Demangeon and Kohei Honda. Full abstraction in a subtyped pi-calculus with linear types. In *CONCUR*, 2011. doi:10.1007/978-3-642-23217-6_19.

17 Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Practical interruptible conversations: Distributed dynamic verification with multiparty session types and Python. *Formal Methods in System Design*, 2015. doi:10.1007/s10703-014-0218-8.

18 Pierre-Malo Deniélou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. Parameterised multiparty session types. *Logical Methods in Computer Science*, 8(4), 2012. doi:10.2168/LMCS-8(4:6)2012.

19 Mariangiola Dezani-Ciancaglini, Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic, and Nobuko Yoshida. Precise subtyping for synchronous multiparty sessions. In *PLACES*, pages 29–43, 2015. doi:10.4204/EPTCS.203.3.

20 Simon Fowler. An Erlang implementation of multiparty session actors. In *ICE*, 2016. doi:10.4204/EPTCS.223.3.

21 Simon J. Gay, Nils Gesbert, and António Ravara. Session types as generic process types. In *EXPRESS/SOS*, 2014. doi:10.4204/EPTCS.160.9.

22 Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3), 2005. doi:10.1007/s00236-005-0177-z.

23 Daniele Gorla. Towards a unified approach to encodability and separation results for process calculi. *Inf. Comput.*, 208(9), 2010. doi:10.1016/j.ic.2010.05.002.

24 Philipp Haller and Alexander Loiko. LaCasa: lightweight affinity and object capabilities in Scala. In *OOPSLA*, 2016. doi:10.1145/2983990.2984042.

25 Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In *ECOOP*, 2010. doi:10.1007/978-3-642-14107-2_17.

26 Kohei Honda, Vasco Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP*, 1998. doi:10.1007/BFb0053567.

27 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL*, 2008. Full version in [28]. doi:10.1145/1328438.1328472.

28 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1), March 2016. doi:10.1145/2827695.

29 Raymond Hu and Nobuko Yoshida. Hybrid session verification through endpoint API generation. In *FASE*, 2016. doi:10.1007/978-3-662-49665-7_24.

30 Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in java. In *ECOOP*, 2008. doi:10.1007/978-3-540-70592-5_22.

31 Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. *Theo. Comput. Sci.*, 311(1-3), 2004. doi:10.1016/S0304-3975(03)00325-6.

32 Keigo Imai, Shoji Yuen, and Kiyoshi Agusa. Session type inference in Haskell. In *PLACES*, 2010. doi:10.4204/EPTCS.69.6.

33 Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. Session types for Rust. In *WGP@ICFP*, 2015. doi:10.1145/2808098.2808100.

34 Naoki Kobayashi. Type systems for concurrent programs. In *10th Anniversary Colloquium of UNU/IIST*, 2002. doi:10.1007/978-3-540-40007-3_26.

35 Naoki Kobayashi. A new type system for deadlock-free processes. In *CONCUR*, 2006. doi:10.1007/11817949_16.

36 Naoki Kobayashi. Type systems for concurrent programs. Extended version of [34], Tohoku University, 2007. URL: http://www.kb.ecei.tohoku.ac.jp/~koba/papers/tutorial-type-extended.pdf.

37 Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.*, 21(5), September 1999. doi:10.1145/330249.330251.

38 Naoki Kobayashi and Davide Sangiorgi. A hybrid type system for lock-freedom of mobile processes. *ACM Trans. Program. Lang. Syst.*, 32(5), 2010.

39 Dimitrios Kouzapas and Nobuko Yoshida. Globally governed session semantics. In *CONCUR*, 2013. doi:10.1007/978-3-642-40184-8_28.

40 Dimitrios Kouzapas and Nobuko Yoshida. Globally governed session semantics. *Logical Methods in Computer Science*, 10(4), 2014. doi:10.2168/LMCS-10(4:20)2014.

41 Lightbend, Inc. The Akka framework, 2017. URL: http://akka.io/.

42 Sam Lindley and J. Garrett Morris. Embedding session types in Haskell. In *Haskell*, 2016. doi:10.1145/2976002.2976018.

43 Sam Lindley and J. Garrett Morris. Talking bananas: Structural recursion for session types. In *ICFP*, 2016. doi:10.1145/2951913.2951921.

44 Links homepage. http://links-lang.org/. S. Fowler and D. Hillerström and S. Lindley and G. Morris and P. Wadler.

45 Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6), November 1994. doi:10.1145/197320.197383.

46 Hugo A. Lopez, Eduardo R. B. Marques, Francisco Martins, Nicholas Ng, Casar Santos, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. Protocol-based verification of message-passing parallel programs. In *OOPSLA*, 2015. doi:10.1145/2814270.2814302.

47 Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Inf. Comput.*, 100(1), 1992. doi:10.1016/0890-5401(92)90008-4.

48 Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. Timed Runtime Monitoring for Multiparty Conversations. In *BEAT*, volume 162. EPTCS, 2014. Full version in [49]. doi:10.4204/EPTCS.162.3.

49 Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. Timed runtime monitoring for multiparty conversations. *Formal Aspects of Computing*, 2017. doi:10.1007/s00165-017-0420-8.

50 Rumyana Neykova and Nobuko Yoshida. Let It Recover: Multiparty Protocol-Induced Recovery. In *CC*, 2017. doi:10.1145/3033019.3033031.

51 Rumyana Neykova and Nobuko Yoshida. Multiparty Session Actors. *Logical Methods in Computer Science*, 13(1), March 2017. doi:10.23638/LMCS-13(1:17)2017.

52 Dominic A. Orchard and Nobuko Yoshida. Effects as sessions, sessions as effects. In *POPL*, 2016. doi:10.1145/2837614.2837634.

53 Luca Padovani. Deadlock and lock freedom in the linear π-calculus. Online version of [54], January 2014. URL: https://hal.inria.fr/hal-00932356.

54 Luca Padovani. Deadlock and lock freedom in the linear π-calculus. In *CSL-LICS*. ACM, 2014. doi:10.1145/2603088.2603116.

55 Luca Padovani. A simple library implementation of binary sessions. *Journal of Functional Programming*, 27, 2017. Website: http://www.di.unito.it/~padovani/Software/FuSe/FuSe.html. doi:10.1017/S0956796816000289.

56 Benjamin C. Pierce. *Types and programming languages*. MIT Press, MA, USA, 2002.

57 Riccardo Pucella and Jesse A. Tov. Haskell session types with (almost) no class. In *Haskell*, 2008. doi:10.1145/1411286.1411290.

58 Lukas Rytz, Martin Odersky, and Philipp Haller. Lightweight polymorphic effects. In *ECOOP*, 2012. doi:10.1007/978-3-642-31057-7_13.

59 Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

60 Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A linear decomposition of multiparty sessions for safe distributed programming. Technical Report 2, Imperial College London, 2017. URL: https://www.doc.ic.ac.uk/research/technicalreports/2017/#2.

61 Alceste Scalas and Nobuko Yoshida. Lightweight session programming in scala. In *ECOOP*, 2016. doi:10.4230/LIPIcs.ECOOP.2016.21.

62 Alceste Scalas and Nobuko Yoshida. Lightweight Session Programming in Scala (Artifact). *Dagstuhl Artifacts Series*, 2(1), 2016. doi:http://dx.doi.org/10.4230/DARTS.2.1.11.

63 Scribble homepage. http://www.scribble.org.

64 K. C. Sivaramakrishnan, Karthik Nagaraj, Lukasz Ziarek, and Patrick Eugster. Efficient session type guided distributed interaction. In *COORDINATION*, 2010. doi:10.1007/978-3-642-13414-2_11.

65 Matías Toro and Éric Tanter. Customizable gradual polymorphic effects for Scala. In *OOPSLA*, 2015. `doi:10.1145/2814270.2814315`.

66 TYPICAL. Type-based static analyzer for the pi-calculus. `http://www-kb.is.s.u-tokyo.ac.jp/~koba/typical/`.

67 Nobuko Yoshida, Pierre-Malo Deniélou, Andi Bejleri, and Raymond Hu. Parameterised multiparty session types. In *FOSSACS*, 2010. `doi:10.1007/978-3-642-12032-9_10`.

68 Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The Scribble protocol language. In *TGC*, 2013. `doi:10.1007/978-3-319-05119-2_3`.