A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming

Alceste Scalas\textsuperscript{1}, Ornela Dardha\textsuperscript{2}, Raymond Hu\textsuperscript{1}, and Nobuko Yoshida\textsuperscript{1}

\textsuperscript{1} Imperial College London, UK
\{alceste.scalas, raymond.hu, n.yoshida\}@imperial.ac.uk
\textsuperscript{2} University of Glasgow, UK
ornela.dardha@glasgow.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 $\pi$-calculus into linear $\pi$-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.

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

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

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

* 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.
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 $P_a$.

Fig. 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 multi-party interaction to another process, thus changing the communicating topology. In Fig. 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 $s[b]$ could have type:

$$S_b = \quad c!\text{InfoBC}(\text{String}) . \quad a?\text{InfoAB}(\text{String}) . \quad \mu t. (a \& \{ ?\text{Mov1AB}(\text{Int}).c!\text{Mov1BC}(\text{Int}).t , ?\text{Mov2AB}(\text{Bool}).c!\text{Mov2BC}(\text{Bool}).t \} )$$

$S_b$ says that $s[b]$ must be used to realise the Game interactions of $P_b$ in Fig. 1: first to send InfoBC(String) to $c$, then receive InfoAB from $a$, then enter the recursive game “loop” $\mu t. (\ldots )$. Inside the recursion, $a \& \{ \ldots \}$ is a branching from a: depending on a’s choice, the channel will deliver either Mov1AB(Int) (in which case, it must be used to send Mov1BC(Int) to $c$, and loop), or Mov2AB (then, it must be used to send Mov2BC to $c$, and loop). Analogous types can be assigned to $s[a]$ and $s[c]$. **Delegation** is represented by types like $q!\text{PlayB}(S_b).\text{end}$, meaning: from role $q$, receive a message PlayB carrying a channel that must be used according to $S_b$ above; then, end the session. Session type checking ensures that, e.g., process $P_b$ uses its channels abiding by the types above — thus safely implementing the expected channel dynamics and fulfilling role $b$ in the game. Finally, MPST can formalise the whole Game protocol in Fig. 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 $S_2$ 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 Fig. 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 $\pi$-calculus;

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

By step S1, we formally address challenge C1. Linear $\pi$-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) $\pi$-calculus terms. This endeavour was not tackled before, and its feasibility was unclear. Its practical payoff is that linear $\pi$-calculus channels/types are amenable for an (almost) direct object-based representation (shown in [61]): this tackles challenge C2. Further, using $\pi$-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 (§5) of a full multiparty session $\pi$-calculus (§2) into standard $\pi$-calculus with linear, labelled tuple and variant types (§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) $\pi$-calculus, and spot a longstanding issue with type merging [18] (Def. 2.9, §2.1 “On Consistency”). We fix it, with a revised subject reduction (Thm. 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 $\pi$-calculus types (Thm. 6.3). Our encoding of types preserves duality and subtyping (Thm. 6.1); our encoding of processes is type-preserving and operationally sound and complete (Thm. 6.2 and Thm. 6.5).

- We subsume the encodings of binary sessions into $\pi$-calculus [14, 15], and support recursion (§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 §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).

Conventions Derivations use single/double lines for inductive/coinductive rules. Recursive types µt.T are always closed, and guarded: e.g., µt₁⋯µtₙ.t, is not a type. We define unf(µt.T)=unf(T(µₜₙ.T/ₜₙ)), and unf(T)=T if T ≠ µt.T'. Type equality is syntactic: µt.T is not equal to unf(µt.T). We write P→P' for process reductions, →* for the reflexive+transitive closure of →, and P rediff P' such that P→P'. We assume a basic subtyping ≤_{B} capturing e.g. Int ≤_{B} 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] (Def. 2.1), and its typing system — including recursion, subtyping [19] and type merging [67, 18] (§ 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 linear π-calculus (§3), that model binary channels. We provide various examples based on the scenario in §1.

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

Processes P, Q ::= 0 | P | Q | (νs)P (inaction, composition, restriction)
               c[P] ⊕ (l(v)).P (selection towards role p)
               c[P] &_{i∈I} {l_i(x_i).P_i} (branching from role p — with I ≠ ∅)
               def D in Q | X(˜x) (process definition, process call)

Declarations D ::= X(˜x) = P (process declaration)

Channels c ::= x | s[p] (variable, channel with role p)

Values v ::= c | false | true | 42 | ... (channel, base value)

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[p], i.e., a multiparty communication endpoint whose user impersonates role p in the session s. Values v can be variables, or channels with roles, or base values. The inaction 0 represents a terminated process. The parallel composition P | Q represents two processes that can execute concurrently, and potentially communicate. The session restriction (νs)P declares a new session s with scope limited to process P. Process c[P] ⊕ (l(v)).P performs a selection (internal choice) towards role p, using the channel c: the labelled value l(v) is sent, and the execution continues as process P. Dually, process c[P] &_{i∈I} {l_i(x_i).P_i} uses channels c to wait for a branching (external choice) from role p: if the labelled value l_k(v) is received (for some k ∈ I), then the execution continues as P_k (with x_k holding value v). Note that for all i ∈ I, variable x_i is bound with scope P_i. In both branching and selection, the labels l_i (i ∈ I) are all different and their order is irrelevant. Process definition def D in Q and process call X(˜x) model recursion, with D being a process declaration X(˜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(˜x) = P, we have fv(P) ⊆ ˜x and fc(P) = ∅. 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 ⊕.

Example 2.2. The following MPST π-calculus process implements the scenario in Fig. 1:
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 $s\{b\}$ (and end of session $s_2$); (c) clients directly interacting on session $s$, after “complete” delegation.

```
def LoopB(x) = x[a] & \{inv1B(y).x[c] \oplus (inv2B(z)).LoopB(x). \{inv2B(z).x[c] \oplus (inv2B(z)).LoopB(x)\}\} in
def Clients(y) = \{\{\{ InvA(z) \}. z[c] \oplus (InvA("...") \}. z[a] \& InvA(y) \}. LoopB(z) in
(\nu_{s_a, s_b, s_c})\{ Q | P_2 | P_2 | P_2 \}
```

where: $P_b = Clients(s_b[p])$ (for brevity, we omit the definitions of $P_c$ and $P_1$)

$$Q = (\nu s)\{ s_a[q][p] \oplus (InvA(s[a])) | s_b[q][p] \oplus (InvA(s[b])) | s_c[q][p] \oplus (InvA(s[c])) \}$$

In the 3rd line, $s_a, s_b, s_c$ are the sessions between the server process $Q$ and the processes $P_a, P_b, P_c$, which are composed in parallel with $|$. Each sessions has 2 roles: $q$ (server) and $p$ (client); e.g., $s_a$ is accessed by the server (through the channel with role $s_b[p]$) and by the client $P_b$ (through $s_b[p]$); similarly, $s_b$ (resp. $s_c$) is accessed by $P_a$ (resp. $P_c$) through $s_a[p]$ (resp. $s_c[p]$), while the server owns $s_a[q]$ (resp. $s_c[q]$). The body of the server process $Q$ defines a session $s$ (with 3 roles $a, b, c$) for playing the game. Note that the scope of $s$ does not include $P_1, P_2, P_c$; see Fig. 2(a) for a schema of processes and sessions.

The server $Q$ uses the channel with role $s_a[q]$ (resp. $s_a[q], s_c[q]$) to send the message $PlayB$ (resp. $PlayA, PlayC$) carrying the channel with role $s_b$ (resp. $s[a], s[c]$) to $p$. The result is a delegation of the channel to the client process $P_b$ (resp. $P_a, P_c$). This way, each client obtains a channel endpoint to interact in the game session $s$, interpreting a role among $a, b, c$.

The client $P_b$ is implemented by invoking $Clients(s_b[p])$ (defined in the 2nd line). Here, $y[q] \& PlayB(z)$ means that $y$ (that becomes $s_b[p]$ after the invocation) is used to receive $PlayB(z)$ from $q$, while $z[c] \oplus (InvA("..."))$ means that $z$ (that becomes $s[b]$ after the delegation is received) is used to send $InvA("...")$ to $c$. The game loop is implemented with the recursive process call $LoopB(z)$ (defined in the 1st line) — which becomes $LoopB(s[b])$ after delegation.

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

1. **(R-Com)***: $s[p][q]. K_{i \in I} \{ l_i(x_i). P_i \} | s[q][p] \oplus (l_j(v)). Q \rightarrow P_j\{y/\tilde{y}\} | Q$ (if $j \in I$ and $fv(v) = \emptyset$)

2. **(R-Call)**: $def \ X(\tilde{x}) = P \rightarrow Q$ in $X(\tilde{v})$ | $Q \rightarrow def \ X(\tilde{x}) = P \rightarrow Q$ in $P(\tilde{v}/\tilde{x})$ | $Q$

3. **(R-Par)**: $P \rightarrow Q$ implies $P | R \rightarrow Q | R$

4. **(R-Res)**: $P \rightarrow Q$ implies $(\nu s)P \rightarrow (\nu s)Q$

5. **(R-Deg)**: $P \rightarrow Q$ implies $def \ D in P \rightarrow def \ D in Q$

6. **(R-Struct)**: $P \equiv P'$ and $P \rightarrow Q$ and $Q' \equiv Q$ implies $P' \rightarrow Q'$ (with $\equiv$ standard — see [60])

Rule (R-Com) 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 $p$ and $q$ (i.e., via $s[p]$ and $s[q]$) and targeting each other (i.e., $s[p]$ is used to branch from $q$, and $s[q]$ is used to select towards $p$) reduces to the corresponding continuations, with a value substitution on the receiver side. (R-Call) says that a process call $X(\tilde{v})$ in the scope of
def $X(\overline{x}) = P$ is... reduces by expanding $X(\overline{y})$ into $P$, and replacing the formal parameters ($\overline{x}$) with the actual ones ($\overline{y}$). The remaining rules are standard: reduction can happen under parallel composition, restriction and process definition. By (R-STRUCT), reduction is closed under a structural congruence [60] stating, e.g., that $|$ is commutative and associative, and has 0 as neutral element (i.e., $P | Q \equiv Q | P$, $P | (Q | R) \equiv (P | Q) | R$ and $P | 0 \equiv P$).

Example 2.4. The process in Ex. 2.2 reduces as (see also Fig. 2(b), noting the scope of $s$):
$$(\nu s_4,s_3) (Q \mid P_s \mid P_b \mid P_c) \rightarrow (\nu s_3) (\{s_4,q|p\|\pi_2(s[a])\} \mid s_3,q|p\|\pi_2(s(c))) \mid s(b|c)\|\text{InfStC(}"\ldots"\ldots\}) \mid P_s \mid P_c)$$

2.1 Multiparty Session Typing

We now illustrate the typing system for the MPST $\pi$-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 fosters a top-down approach: a global type $G$ describes a protocol involving various roles — e.g., the game with roles $a,b,c$ in §1; $G$ is projected into a set of (local) session types $S_a, S_b, S_c, \ldots$ (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:

$$S ::= p \& \iota \mid \l_i(U_i).S_i$$

(branching from role $p$ — with $I \neq \emptyset$)

$$p \oplus \iota \mid \l_i(U_i).S_i$$

(selection towards role $p$ — with $I \neq \emptyset$)

$$\mu t.S \mid t \end$$

(recursive type, type variable, termination)

$$B ::= \text{Bool} \mid \text{Int} \ldots$$

(base type)

$$U ::= B \mid S$$

(closed) (payload type)

We omit $\&$/ $\oplus$ when $I$ is a singleton: $p \l_i(\text{Int}).S_i$ stands for $p \oplus \iota \mid \l_i(\text{Int}).S_i$.

The set of roles in $S$, denoted as roles$(S)$, is defined as follows:

$$\text{roles}(p \oplus \iota \mid \l_i(U_i).S_i) \hat{=} \text{roles}(p \& \iota \mid \l_i(U_i).S_i) \hat{=} \{p\} \cup \bigcup_{i \in I} \text{roles}(S_i)$$

roles$(\text{end}) \hat{=} \emptyset$

roles$(t) \hat{=} \emptyset$

roles$(\mu t.S) \hat{=} \text{roles}(S)$

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

The branching type $p \& \iota \mid \l_i(U_i).S_i$ 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 $p \oplus \iota \mid \l_i(U_i).S_i$, 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 $\mu t.S$ and type variable $t$ model infinite behaviours. end is the type of a terminated channel (often omitted). Base types $B, B', \ldots$ can be types like $\text{Bool}, \text{Int}, \text{etc.}$ 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 §1 (p. 2).

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

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

$$H ::= \kappa_{i \in I} ?t_i(U_i).H_i | \oplus_{i \in I} !t_i(U_i).H_i$$

(branching, selection) (with $I \neq \emptyset$, $U_i$ closed)

$\mu t.H | t | \text{end}$

(recursive type, type variable, termination)

A partial session type $H$ is either a branching, a selection, a recursion, a type variable, or a terminated channel type. Unlike Def. 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:

$$\overline{\oplus_{i \in I} !t_i(U_i).H'_i} \triangleq \kappa_{i \in I} ?t_i(U_i).H'_i, \quad \kappa_{i \in I} ?t_i(U_i).H'_i \triangleq \overline{\oplus_{i \in I} !t_i(U_i).H'_i}$$

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 $\text{end}$ and $t$, and homomorphic on $\mu t.H$.

Multiparty session types can be projected onto a role $q$ (Def. 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 Def. 2.11 later on.

Definition 2.9. $S | q$ is the partial projection of $S$ onto $q$:

$$\text{end} | q \triangleq \text{end} \quad t | q \triangleq t \quad \mu t.S | q \triangleq \begin{cases} \mu t.(S | q) & \text{if } S | q \neq t' (\forall t') \\ \text{end} & \text{otherwise} \end{cases}$$

$$(p \oplus_{i \in I} !t_i(U_i).S_i) | q \triangleq \begin{cases} \oplus_{i \in I} !t_i(U_i).(S_i | q) & \text{if } q = p, \\ \prod_{i \in I} (S_i | q) & \text{if } p \neq q \end{cases}$$

$$(p \&_{i \in I} ?t_i(U_i).S_i) | q \triangleq \begin{cases} \kappa_{i \in I} ?t_i(U_i).S_i | q & \text{if } q = p, \\ \prod_{i \in I} (S_i | q) & \text{if } p \neq q \end{cases}$$

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

$$\text{end} \cap \text{end} \triangleq \text{end} \quad t \cap t \triangleq t \quad \mu t.H \cap \mu t.H' \triangleq \mu t.(H \cap H')$$

$$\kappa_{i \in I} ?t_i(U_i).H_i \cap \kappa_{i \in I} ?t_i(U_i).H'_i \triangleq \kappa_{i \in I} ?t_i(U_i).(H_i \cap H'_i)$$

$$\oplus_{i \in I} !t_i(U_i).H_i \cap \oplus_{i \in I} !t_i(U_i).H'_i \triangleq \left(\oplus_{k \in I} !t_k(U_k).H_k \cup \prod_{i \in I} \|k \in I \! ?t_k(U_k).H'_k\right) \cup \left(\oplus_{i \in I} !t_i(U_i).H_i \cup \prod_{i \in I} \|k \in I \! ?t_k(U_k).H'_k\right)$$

The projection of $\text{end}$ or a type variable $t$ onto any role is the identity. Projecting a recursive type $\mu t.S$ onto $q$, means projecting $S$ onto $q$, if $S | q$ is not some $t'$, for all possible recursive variables $t'$; otherwise, the projection is $\text{end}$. The projection of a selection $p \oplus_{i \in I} !t_i(U_i).S_i$ (resp. branching $p \&_{i \in I} ?t_i(U_i).S_i$) on role $p$, produces a partial selection type $\oplus_{i \in I} !t_i(U_i).(S_i | p)$ (resp. branching $\&_{i \in I} ?t_i(U_i).S_i | p$) with the continuations projected on $p$. Otherwise, if projecting on $q \neq p$, the select/branch is “skipped”, and the projection is the merging of the continuations, i.e., $\prod_{i \in I} (S_i | q)$. The $\cap$ 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 Def. 2.11 and Ex. 2.14 later on). Crucially, $\cap$ can compose different internal choices, but not external choices (because this could break type safety).

Subtyping. The subtyping relation (Def. 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 (Def. 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 \( \preceq_S \) on multiparty session types is the largest relation such that (i) if \( S \preceq_S S' \), then \( \forall p \in (\text{roles}(S) \cup \text{roles}(S')) \, S[p] \preceq_S S'[p] \), and (ii) is closed backwards under coinductive rules at the top of Fig. 3. The subtyping \( \preceq_P \) on partial session types is coinductively defined by the rules at the bottom of Fig. 3.

Def. 2.10 uses coinduction to support recursive types [56, § 20 and § 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 (Def. 2.11). The gist of Def. 2.10 lies in clause (ii). Rules (s-Banch)/(s-Sel) define subtyping on branch/select types. Both rules are covariant in the continuation types, i.e., they require \( S_i \preceq_S S'_i \). (s-Banch) is covariant also in the number of branches offered, whereas (s-Sel) is contravariant. (s.B) relates base types, if they are related by \( \preceq_B \). (s-End) relates terminated channel types. (s-µL) and (s-µR) are standard under coinduction: they say that a recursive session type \( \mu S \) is related to \( S' \), iff its unfolding is related, too. The subtyping \( \preceq_P \) 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 (Def. 2.12), we need to formalise the notions of typing context and typing judgement, defined below.

Definition 2.11. A session typing context \( \Gamma \) is a partial mapping defined as:

\[
\Gamma ::= \emptyset \mid \Gamma, x:U \mid \Gamma, s[p]:S \quad (\text{with } p \not\in S)
\]

We say that \( \Gamma \) is consistent iff for all \( s[p]:S, s[q]:S \in \Gamma \) with \( p \neq q \), we have \( S[p][q] \not\preceq_S S[q][p] \). We say that \( \Gamma \) is complete iff for all \( s[p]:S \in \Gamma \), \( q \in S \) implies \( s[q] \in \text{dom}(\Gamma) \). We say that \( \Gamma \) is unrestricted, \( \text{un}(\Gamma) \), iff for all \( c \in \text{dom}(\Gamma) \), \( \Gamma(c) \) is either a base type or end. The typing contexts composition \( \circ \) is the commutative operator with \( \emptyset \) as neutral element:

\[
\Gamma_1, c:U \circ \Gamma_2, c':U' \triangleq (\Gamma_1 \circ \Gamma_2), c:U, c':U' \quad (\text{if } \text{dom}(\Gamma_2) \ni c \neq c' \not\in \text{dom}(\Gamma_1))
\]

\[
\Gamma_1, x:B \circ \Gamma_2, x:B \triangleq (\Gamma_1 \circ \Gamma_2), x:B
\]

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

On Consistency In Def. 2.11, and in the rest of this work, we emphasise the importance of consistency of the context \( \Gamma \) for session typing: this condition is, in fact, necessary to prove subject reduction, and will be central for our encoding (§ 5 and § 6). As an example of
non-consistent typing context, consider \(s[p]: \text{end}, s[q]: \text{p?}(U).S\): we have \(\text{end} \upharpoonright q = \text{end} \nless_p \text{p?}(U).S \upharpoonright p\).

Note that our consistency in Def. 2.11 is weaker than the one in previous papers (where it is sometimes called coherency): we use \(\leq_p\), instead of (syntactic) type equality \(=\), to relate dual partial projections. The reason being: if we use \(=\), and adopt partial projections with type merging (Def. 2.9), subject reduction does not hold. Hence, by relaxing our definition, and proving Thm. 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 \(\Theta\) maps process variables \(X\) to \(n\)-tuples of types \(\vec{U}\) (one per argument of \(X\)), and is defined as:

\[
\Theta := \emptyset \mid \Theta, X: \vec{U}
\]

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

- for processes: \(\Theta \cdot \Gamma \vdash P\) (with \(\Gamma\) consistent, and \(\forall v : S \in \Gamma, S \upharpoonright p\) is defined \(\forall p \in S\))
- for values: \(\Gamma \vdash X: \vec{U}\)

The judgement \(\Theta \cdot \Gamma \vdash P\) reads: “process \(P\) is well-typed in \(\Theta\) and \(\Gamma\)”. \(\Theta\) and \(\Gamma\), in turn, type respectively process variables (judgement \(\Theta \vdash X: \vec{U}\)) 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 \(\leq_S\) (Def. 2.10). By (T-NL), 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 Def. 2.11. By (T-RES), \(\nu s.P\) is well typed in \(\Gamma\), if \(s\) occurs in a complete set of typed channels with roles (denoted with \(\Gamma^\prime\)), and the open process \(P\) is well typed in the “full” context \(\Gamma, \Gamma^\prime\). For convenience, we annotate the restricted \(s\) with \(\Gamma^\prime\) in the process, giving \((\nu s : \Gamma^\prime).P\). (T-BRCH) (resp. (T-SEL)) state that branching (resp. selection) process on \(c[p]\) is well typed if \(c[p]\) 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 \(\text{def } X(\bar{x}) = P \in Q\) is well typed if both \(P\) and \(Q\) are well typed in their typing contexts enriched with \(\bar{x} : \vec{U}\). For convenience, we annotate \(\bar{x}\) with types \(\vec{U}\). By (T-CALL), process call \(X(v_1, \ldots, v_n)\) 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 “\(\forall c : S \in \Gamma \mid S \upharpoonright p\) is defined” is unusual in MPST works, but arises naturally: by requiring the existence of partial projections, it rejects processes containing \(\langle a \rangle\) a channel with role \(s[p]: S\).
that, for some $q \in S$, cannot be (consistently) paired with $s[q]$, or (b) a variable $x:S$ that, in a consistent and complete $\Gamma$, cannot be substituted by any $s[p]:S$. Rejected processes cannot join any complete session (case (a)), or are never-executed “dead code” (case (b)).

\textbf{Remark 2.13.} Unlike most MPST papers (e.g., [19, 11]), our rule (T-Rss) does not directly map a session $s$ to a global type: this is explained in the next section, “Global Types”.

\textbf{Example 2.14.} Consider the session type $S_b$ in §1 (p. 2), and the client process $P_b = Client_b(s_b[p])$ from Ex. 2.2. By Def. 2.12, the following typing judgement holds:

$$\Gamma \vdash Client_b(s_b[p])$$

It says that the channel with role $s_b[p]$ is used following the game loop. This example cannot be typed without merging $\cap$ (Def. 2.9): its derivation requires to compute $s_b[p] : (\text{String}, \mu t) . (\text{Move}(\text{Bool}), t) = \text{Move}(\text{String}, \mu t) . (\text{Move}(\text{Bool}), t)$, which is undefined without merging.

The typing rules in Fig. 4 satisfy a subject reduction property (Thm. 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 $\Gamma$.

\textbf{Definition 2.15 (Typing context reduction).} The reduction $\Gamma \rightarrow \Gamma'$ is:

$$s[p]:S, s[q]:S \rightarrow s[p]:S_k, s[q]:S_k'$$

if $\\Gamma, c:U \rightarrow \Gamma', c:U'$

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

\textbf{Theorem 2.16 (Subject reduction).} If $\Theta \vdash P$ and $P \rightarrow P'$, then $\exists \Gamma': (\Theta, \Gamma') \vdash P'$.

\section*{Global Types}

We conclude this section with global types, mentioned in §2 and Remark 2.13.

\textbf{Definition 2.17.} The syntax of global types, ranged over by $G$, is:

$$G ::= p \rightarrow q : \{l_i(U_i).G_i\}_{i \in I} \quad (\text{interaction with } U_i \text{ closed})$$

$$\mu t. G \mid t \mid \text{end} \quad (\text{recursive type, type variable, termination})$$

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

\textbf{Example 2.18.} The following global type formalises the $Game$ described in §1 and Fig. 1:

$$G_{Game} = b \rightarrow c : \text{Move}(\text{String}), c \rightarrow a : \text{Info}(\text{String}), a \rightarrow b : \text{Info}(\text{String})$$

$$\mu t. a \rightarrow b : \begin{cases} \text{Move}(\text{Int}), b \rightarrow c : \text{Move}(\text{Int}), c \rightarrow a : \begin{cases} \text{Move}(\text{Int}), t, \\ \text{Move}(\text{Bool}), t \end{cases} \\ \text{Move}(\text{Bool}), b \rightarrow c : \text{Move}(\text{Bool}), c \rightarrow a : \begin{cases} \text{Move}(\text{Int}), t, \\ \text{Move}(\text{Bool}), t \end{cases} \end{cases}$$

In MPST theory, a global type $G$ with roles $p_i$ ($i \in I$) is used to project\footnote{We use a standard projection with merging [67, 18]: for its definition (not crucial here), see [60].} a set of session types $S_i$ (one per role). E.g., projecting $G_{Game}$ in Ex. 2.18 onto $b$ yields the session type $S_b$...
(p. 2). When all such projections $S_i$ are defined, and all partial projections of each $S_i$ are defined (as per Def. 2.9), then we can define the projected typing context of $G$:

$$\Gamma_G = \{ s[p_i]: S_i \}_{i \in I}$$

where $\forall i \in I : S_i$ is the projection of $G$ onto $p_i$ and $\Gamma_G$ can be shown to be: (a) consistent and complete, i.e., can be used to type the session $s$ by rule (T-Res) (Fig. 4), and (b) deadlock-free, i.e.: $\Gamma_G \rightarrow^* \Gamma_G \not\vdash$ implies $\forall i \in I : \Gamma_G(s[p_i]) = \text{end}$. Similarly, it can be shown that $\Gamma_G$ reduces as prescribed by $G$.

Now, from observation (a) above, we can easily define a “strict” version of rule (T-Res) (Fig.4) in the style of [19, 11], where (1) the clause “T’ complete” is replaced with “T’ is the projected typing context of some $G$”, and (2) in the conclusion, the annotation ($\nu_s: \Gamma'$) is replaced with ($\nu_s: G$). Further, observation (b) allows to prove Thm. 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 Fig. 4, the annotation ($\nu_s: G$) would be ($\nu_s: \Gamma_G$)).

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

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 Fig. 4 do not require it, nor mention $G$. The main property of such rules is ensuring type safety (Thm. 2.16). We will exploit this insight (obtained by our separation of global/local typing) in our encoding (§5), preserving semantics and types (and thus, Thm. 2.19) without explicit references to global types.

## 3 Linear $\pi$-Calculus

The $\pi$-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 $\pi$-calculus with linear types [37], adopting a standard formulation and well-known results from [59]. We will present new $\pi$-calculus-related results in §4.

\textbf{Definition 3.1.} The syntax of $\pi$-calculus processes and values is:

$$P, Q :: | P | Q | (\nu x) P \quad \text{(inaction, parallel composition, restriction)}$$

$$*P | \pi(v).P \quad | x(y).P \quad \text{(process replication, output, input)}$$

$$\text{case } v \text{ of } \{ l_i(x_i) \triangleright P_i \}_{i \in I} \quad \text{(variant destruct)}$$

$$\text{with } [l_i : x_i]_{i \in I} = v \text{ do } P \quad \text{(labelled tuple destruct)}$$

$$u, v ::= x, y, w, z \mid \{ l(v) \mid \{ l_i : v_i \}_{i \in I} \mid false \mid true \mid 42 \mid \ldots \quad \text{(base value)}$$

In $\pi$-calculus, names $x, y, \ldots$ can be intuitively 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 \texttt{false} or 42, variant values $l(v)$ and labelled tuples $[l_i : v_i]_{i \in I}$. The inaction \texttt{0} and the parallel composition $P | Q$ are similar to Def. 2.1. The restriction $(\nu x)P$ 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 $\pi(v).P$ uses the name $x$ to send a value $v$, and proceeds as $P$; the input $\langle x(y) \rangle.P$ uses $x$ to receive a value that will substitute $y$ in the continuation $P$. Process \texttt{case } v \text{ of } \{ l_i(x_i) \triangleright P_i \}_{i \in I} \text{ pattern matches } a \text{ variant value } v$, and if it has label $l_i$, substitutes $x_i$ and continues as $P_i$. Process \texttt{with } [l_i : x_i]_{i \in I} = v \text{ do } P \text{ destructs a labelled tuple } v$, substituting each $x_i$ in $P$. For brevity, we will often write “record” instead of “labelled tuple”.

\textbf{Definition 3.2.} The $\pi$-calculus operational semantics is the relation $\rightarrow$ defined as:
A Linear Decomposition of Multiparty Sessions

(Res-Com) \[ \pi(v).P \mid x(y).Q \rightarrow P \mid Q \{v/y \} \]
(Res-Case) \[ \text{case } l_j(v) \text{ of } \{l_i(x_i) \triangleright P_i \}_{i \in I} \rightarrow P_j \{v/x_j \} \quad (j \in I) \]
(Res-With) \[ \text{with } [l_i : x_i]_{i \in I} = [l_i : v_i]_{i \in I} \text{ do } P \rightarrow P \{v/x_i \}_{i \in I} \]
(Res-RS) \[ P \rightarrow Q \text{ implies } (l x)P \rightarrow (l x)Q \]
(Res-Par) \[ P \rightarrow Q \text{ implies } P \mid R \rightarrow Q \mid R \]
(Res-Struct) \[ P \equiv P' \land P \rightarrow Q \land Q' \equiv Q \implies P' \rightarrow Q' \]

Rule (Res-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. (Res-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 (Res-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 (Res-RS) and (Res-Par), reductions can happen under restriction and parallel composition, respectively. By (Res-Struct), reduction is closed under the structural congruence \( \equiv \), whose definition is standard (see [59, Table 1.1] and [60]).

\( \pi \)-Calculus Typing We now summarise the \( \pi \)-calculus types, subtyping, and typing rules.

**Definition 3.3** (\( \pi \)-types). The syntax of a \( \pi \)-calculus type \( T \) is given by:
\[
T ::= \text{Li}(T) \mid \text{Lo}(T) \mid \text{L}^\langle T \rangle \mid \text{\#}(T) \mid \bullet \quad \text{(linear input, linear output, linear connection)}
\]
\[
\hat{T} \mid \{l_i : T_i\}_{i \in I} \mid [l_i : T_i]_{i \in I} \quad \text{(variant, labelled tuple a.k.a. “record”)}
\]
\[
\mu T \mid t \mid \text{Bool} \mid \text{Int} \mid \ldots \quad \text{(recursive type, type variable, base type)}
\]

Linear types \( \text{Li}(T), \text{Lo}(T) \) denote, respectively, names used exactly once to input/output a value of type \( T \). \( \text{L}^\langle T \rangle \) denotes a name used once for sending, and once for receiving, a message of type \( T \). \( \hat{T} \) 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. \([l_i : T_i]_{i \in I}\) 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 \text{Bool}, are standard.

The predicate \( \text{lin}(T) \) (Def. 3.4 below) holds if \( T \) has some linear input/output component.

**Definition 3.4 (Linear/unrestricted types).** The predicate \( \text{lin} \) is inductively defined as:
\[
\text{lin}(\text{Li}(T)) \land \text{lin}(\text{Lo}(T)) \land \exists j \in I : \text{lin}(T_j) \land \exists j \in I : \text{lin}(T_j) \land \text{lin}(\mu T) \land \text{lin}(\mu T)
\]

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

**Definition 3.5.** Subtyping \( \leq \pi \) for \( \pi \)-types is coinductively defined as:
\[
B \leq_{\pi} B' \quad \hat{B} \leq_{\pi} B' \quad \bullet \leq_{\pi} \bullet \quad T \leq_{\pi} T' \quad \text{Li}(T) \leq_{\pi} \text{Li}(T') \quad \text{L}^\langle T \rangle \leq_{\pi} \text{L}^\langle T \rangle' \quad \text{Lo}(T) \leq_{\pi} \text{Lo}(T') \quad T \leq_{\pi} T' \quad T' \leq_{\pi} T \quad T' \leq_{\pi} T
\]
\[
\forall i \in I \quad T_i \leq_{\pi} T'_i \quad \forall i \in I \quad T_i \leq_{\pi} T'_i \quad [l_i : T_i]_{i \in I} \leq_{\pi} [l_i : T'_i]_{i \in I}
\]
\[
\text{(S-LB)} \quad \text{(S-LEND)} \quad \text{(S-Li)} \quad \text{(S-Lo)} \quad \text{(S-variant)} \quad \text{(S-Ltuple)} \quad \text{(S-LmuL)}
\]

By rule (S-LB), \( \leq_{\pi} \) includes basic subtyping \( \leq_{\pi} \). (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-LmuL) (and its symmetric, omitted) relates a recursive type \( \mu T \) to \( T' \) iff its unfolding is related to \( T' \).
We introduce some definitions and results on typed π-calculus. As we target standard typed π-calculus (§3), we write $\Gamma \vdash v:T$ to state our encoding and its properties.

### Defining $\Gamma$:

The **typing context** $\Gamma$ is a partial mapping defined as:

$$\Gamma := \emptyset \mid \Gamma, x:T$$

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

$$\text{lin}(\Gamma) \uplus \text{lin}(\Gamma') = \text{lin}(\Gamma \uplus \Gamma')$$

$$\text{un}(\Gamma) \uplus \text{un}(\Gamma') = \text{un}(\Gamma \uplus \Gamma')$$

We say that $\Gamma$ is well typed in an unrestricted typing context $\Gamma'$. By $\text{T-Sub}()$, 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 $T_i$.

The **subsumption rule**: if $x$ has type $T$ in $\Gamma$, then it also has any supertype of $T$. By $\text{T-Now}$, $\emptyset$ is well typed in every unrestricted typing context. By $\text{T-Par}$, the parallel composition of two processes is typed by combining the respective typing contexts. By $\text{T-Res1}$, the restriction process $(\nu x)P$ is well typed if $P$ is typed by augmenting the context with $x: \text{L}_x(T)$, or $x: \text{T}$. In the first case, by applying Def. 3.6 ($\uplus$), we have $x: \text{L}_x(T) = x: \text{L}(T) \uplus \text{Lo}(T)$; this implies that $P$ owns both capabilities of linear input/output of $x$. By $\text{T-Res2}$, the restriction $(\nu x)P$ is typed if $P$ is typed and $x$ has no capabilities. By $\text{T-Ins}$ (resp. $\text{T-Out}$), 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 $\text{T-Res}$, a replicated process $P^*$ is typed in the same unrestricted context that types $P$.

### Figures and Tables

**Figure 5** Typing rules for the linear π-calculus.

**Definition 3.6 (Type combination)**. The **linear π-calculus typing context** $\Gamma$ is a partial mapping defined as:

$$\Gamma := \emptyset \mid \Gamma, x:T$$

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

$$\text{lin}(\Gamma) \uplus \text{lin}(\Gamma') = \text{lin}(\Gamma \uplus \Gamma')$$

$$\text{un}(\Gamma) \uplus \text{un}(\Gamma') = \text{un}(\Gamma \uplus \Gamma')$$

We say that $\Gamma$ is well typed in an unrestricted typing context $\Gamma'$. By $\text{T-Sub}()$, 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 $T_i$.

The **subsumption rule**: if $x$ has type $T$ in $\Gamma$, then it also has any supertype of $T$. By $\text{T-Now}$, $\emptyset$ is well typed in every unrestricted typing context. By $\text{T-Par}$, the parallel composition of two processes is typed by combining the respective typing contexts. By $\text{T-Res1}$, the restriction process $(\nu x)P$ is well typed if $P$ is typed by augmenting the context with $x: \text{L}_x(T)$, or $x: \text{T}$. In the first case, by applying Def. 3.6 ($\uplus$), we have $x: \text{L}_x(T) = x: \text{L}(T) \uplus \text{Lo}(T)$; this implies that $P$ owns both capabilities of linear input/output of $x$. By $\text{T-Res2}$, the restriction $(\nu x)P$ is typed if $P$ is typed and $x$ has no capabilities. By $\text{T-Ins}$ (resp. $\text{T-Out}$), 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 $\text{T-Res}$, a replicated process $P^*$ is typed in the same unrestricted context that types $P$.

### 4 Some Typed π-Calculus Extensions and Results

We introduce some definitions and results on typed π-calculus: we will need them in §5 and §6, to state our encoding and its properties. As we target standard typed π-calculus (§3),
The dual of a type variable is

The dual of a linear input type is

with

we can check whether it is defined for the substitution

yielding

Unfortunately, we have

Note that the carried type

ible input/output capabilities. Intuitively, the dual of a

type is

reminds of the “logical duality” for session types [43], but we study it in the context of types such as

types are dual, and we unfold them, we get a dual pair

However, defining duality for

types is convolutive. The dual of

µ

t cannot be considered duals. Solving this issue is crucial: in §5, we will need to encode recursive partial types, preserving their duality (Def. 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

that can be combined with

(Def. 3.6), yielding

. However, defining duality for recursive π-types is not straightforward: what is the dual of

? Is it maybe

? Since

is not defined for

-types, we can check whether it is defined for the unfoldings of our hypothetical duals

and

. Unfortunately, we have

and

: i.e.,

is again undefined, so

cannot be considered duals. Solving this issue is crucial: in §5, we will need to encode recursive partial types, preserving their duality (Def. 2.8) in linear π-types.

all our extensions are conservative, so to preserve standard results (e.g., subject reduction).
By adding dualised type variables in Def.3.3, we naturally extend the definition of \( \text{fv}(T) \) (with \( \mu \ldots \) binding both \( t \) and \( \tilde{t} \)), the subtyping relation \( \leq_{\pi} \) in Def. 3.5 (by letting rules (s-L-L) and (s-L-R) use the substitution in Def. 4.1) and ultimately the typing system in Def. 3.6. Using these extensions, we will obtain a rather simple encoding of recursive session types (Def. 5.1), and solve a subtle issue involving duality, recursion and continuations (Ex. 5.3).

The reader might be puzzled about the impact of dualised variables in the \( \pi \)-calculus theory. We show that dualised variables do not increase the expressiveness of linear \( \pi \)-types, and do not unsafely enlarge subtyping \( \leq_{\pi} \): this is proved in Lemma 4.4, that allows to erase dualised variables from recursive \( \pi \)-types. It uses (1) a substitution that only replaces dualised variables, i.e.: \( \tilde{t} \{ t' / \tilde{t} \} = t' \); and (2) the equivalence \( =_{\pi} \) defined as: \( \leq_{\pi} \cap \leq_{\pi}^{-1} \).

\begin{itemize}
  \item \textbf{Lemma 4.4} (Erasure of \( \tilde{t} \)). \( \mu t . T =_{\pi} \mu t . T \{ t' / t \}, \) for all \( t' \mathcal{\notin} \text{fv}(T) \).
  \item \textbf{Example 4.5} (Application of erasure). Take \( T \) from Ex. 4.3. By Lemma 4.4, we have: \( T =_{\pi} \mu t . \text{Li}(\tilde{t}, \mu t' . \text{Li}(t, \tilde{t}) \{ t' / \tilde{t} \})) = \mu t . \text{Li}(t, \mu t' . \text{Lo}(t, t')) \).
\end{itemize}

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

\textbf{Type Combinator} \( \otimes \)  
Def. 4.6 introduces a type combinator that is a “relaxed” version of \( \odot \) (Def.3.6) extended with subtyping. We will use it to encode MPST typing contexts (Def.5.6).

\begin{itemize}
  \item \textbf{Definition 4.6.} The \( \pi \)-calculus type combinator \( \otimes \) is defined on \( \pi \)-types as follows (and undefined in other cases), and naturally extended to typing contexts:
    \[
    \begin{align*}
      \text{Lo}(T) \otimes \text{Li}(T') & \triangleq \text{Li}(T) \odot \text{Lo}(T) \\
      \text{Li}(T') \otimes \text{Lo}(T) & \triangleq \text{Li}(T) \odot \text{Lo}(T) \\
      (\Gamma_1 \otimes \Gamma_2)(x) & \triangleq \begin{cases} 
        \Gamma_1(x) \otimes \Gamma_2(x) & \text{if } x \in \text{dom}(\Gamma_1) \cap \text{dom}(\Gamma_2) \\
        \Gamma_1(x) & \text{if } x \in \text{dom}(\Gamma_1) \setminus \text{dom}(\Gamma_2)
      \end{cases}
    \end{align*}
    \]
  \end{itemize}

The difference between \( \odot \) and \( \otimes \) is that the former combines linear inputs/outputs with the same carried type, while \( \otimes \) is more relaxed: it allows a carried type to be of a 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 Ex.4.8 below.

\begin{itemize}
  \item \textbf{Lemma 4.7.} If \( T =_{\pi} T_1 \otimes T_2 \) and \( T_1' \odot T_2' = T \), then either (a) \( T_1 \leq_{\pi} T_1' \) and \( T_2 \leq_{\pi} T_2' \), or (b) \( T_1' \leq_{\pi} T_2 \) and \( T_2' \leq_{\pi} T_1 \).
\end{itemize}

Lemma 4.7 says that \( T_1 \otimes T_2 \) (when defined) is a type that, when split using \( \odot \), yields linear I/O types that are subtypes of the originating \( T_1, T_2 \). Intuitively, it means that \( \otimes \) 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 \( \odot \) and \( (\pi_{\text{sub}}) \) (Fig.5).

\begin{itemize}
  \item \textbf{Example 4.8.} Let \( T_1 = \text{Li}(\text{Real}) \), \( T_2 = \text{Lo}(\text{Int}) \), and \( T = T_1 \otimes T_2 \). We have \( T = \text{Li}(\text{Int}) \); if we let \( T_1' \odot T_2' = T \), then we get either (a) \( T_1' = \text{Li}(\text{Int}) \leq_{\pi} T_1 \) and \( T_2' = \text{Lo}(\text{Int}) \leq_{\pi} T_2 \), or (b) \( T_1' = \text{Lo}(\text{Int}) \leq_{\pi} T_2 \) and \( T_2' = \text{Li}(\text{Int}) \leq_{\pi} T_1 \).
\end{itemize}

## 5 Encoding Multiparty Session-\( \pi \) into Linear \( \pi \)-Calcus

We now present our encoding of MPST \( \pi \)-calculus into linear \( \pi \)-calculus. It consists of an \textit{encoding of types} and an \textit{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 \( \pi \)-calculus channels. This leads to the main intuition behind our approach: \emph{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 \( p \), then the corresponding \( \pi \)-calculus record should have a label \( p \), mapping to a name that can send/receive messages to/from the process that plays the role \( p \). This suggests the type of an encoded MPST channel endpoint: it should be a \( \pi \)-calculus record — and since each name appearing in such record is used to communicate, it should have an input/output type.

\begin{definition}
The encoding of session type \( S \) into linear \( \pi \)-types is: 
\[ [S] \triangleq [p : [S \mid p]] \] for \( p \in S \) where the encoding of the partial projections \( [S \mid p] \) is:
\[
\begin{align*}
\oplus_{i \in I} \lll_i(U_i).H_i & \triangleq \Lo(\lll_i((U_i), H_i))_{i \in I} \\
\&_{i \in I} \lll_i(U_i).H_i & \triangleq \Li(\lll_i((U_i), H_i))_{i \in I} \\
[B] & \triangleq B \\
[end] & \triangleq \bullet
\end{align*}
\]
\end{definition}

The encoding of a session type \( S \), namely \([S]\), is a record that maps each role \( p \in S \) to the encoding of the partial projection \([S \mid p]\). 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 \( end \) becomes \( \bullet \), with no capabilities. Selection \( \oplus_{i \in I} \lll_i(U_i).H_i \) and branching \( \&_{i \in I} \lll_i(U_i).H_i \) are encoded as linear output and input types, respectively, adopting a continuation-passing style (CPS). In both cases, the carried types are variants: \( \lll_i((U_i), H_i) \) for select and \( \lll_i((U_i), H_i) \) for branch, with the same labels as the originating partial projections. Such variants carry tuples \( (U_i), (H_i) \) 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 \emph{continuation name}: it is sent together with the encoded payload, and will be used to send/receive the next message (unless \( H_i \) is \( end \)). 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. Def 4.1). E.g., if \( [H_i] \) requires to send a message, the recipient of \( [H_i] \) 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.

\begin{example}
Consider the session type \( S \triangleq \pl\lll_1(\Int).\q\lll_2(\text{String}).\text{end} \) by Def 5.1, the encoding of \( S \) is:
\[
[S] = [p : [S \mid p], q : [S \mid q]] = [\pl \lll_1(\Int), \q : \lll_2(\text{String})] = [p : \Lo(\lll_1(\Int)), \q : \Lo(\lll_2(\text{String}))]
\]
\end{example}

Recursion, Continuations and Duality We now point out a subtle (but crucial) difference between Def 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., \([H_i] \); in [15], instead, it is the encoding \emph{of the dual of the} \( H_i \), i.e., \([\overline{H_i}] \). This difference is irrelevant for non-recursive types (Ex. 5.2); but for recursive types, using \([\overline{H_i}] \) would yield the wrong continuations. Using \([H_i] \), instead, gives the expected result, by generating \emph{dualised recursion variables} (cf. Def 4.1). We explain it in Ex. 5.3 below.

\begin{example}
Let \( H = \mu t.\lll(\Boo).t \) by Def 5.1, we have:
\[
[H] = [\mu t.\lll(\Boo).t] = [\mu t.\Lo(\lll(\Boo), t)] = \mu t.\Lo(\lll(\Boo, t))
\]
Let us now unfold the encoding of \( H \). By Def 4.1, we have:
\[
\text{unf}([H]) = \text{unf}(\mu t.\Lo(\lll(\Boo, t))) = \Lo(\lll(\Boo, \mu t.\Li(\lll(\Boo, t))))
\]
This is what we want: since \( H \) requires a recursive output of \( \Boo \), its encoding should output a \( \Boo \), together with a \emph{recursive input name} as continuation. Hence, the recipient
will receive the first Boolean together with a continuation name, whose type mandates to recursively input more Booleans. If encoding continuations as in [15], instead, we would have:

\[ |H| = \mu t. L o((l_\ldots([\text{Bool}], [\emptyset])) = \mu t. L o((l_\ldots([\text{Bool}, t]))) \quad \text{(t is not dualised)} \]

\[ \text{unf}(|H|) = L o((l_\ldots([\text{Bool}, \mu t. L o((l_\ldots([\text{Bool}, t])))))) \]

which is wrong: the recipient is required to recursively output Booleans. This wrong encoding would also prevent us from obtaining Thm.6.1 later on.

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

\[ \text{Definition 5.4. The encoding of a session typing context is:} \]

\[ \Theta X = \sigma \quad \Theta \cdot \Gamma = [\Theta], \Gamma \quad [c:U] = [c]:[U] \quad [x] = x \quad [X] = z_X \]

When encoding typing contexts, variables \((x)\) keep their name, while process variables \((X)\) and channels with roles \((s[p])\) are turned into distinguished names with a subscript: e.g., \(X\) becomes \(z_X\). The composition \(\Gamma_1 \circ \Gamma_2\) (Def. 2.11) is encoded using \(\bigcirc\) (Def. 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 Thm. 6.5 later. Moreover, this will lead to the implementation discussed in §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 Fig. 7 (but we postpone the formal statement to Def. 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 \([P]_{\bigcirc} \Gamma_1\) and \([Q]_{\bigcirc} \Gamma_2\) are the encoded processes yielded respectively by \([\Theta \cdot \Gamma_1]_{\bigcirc} P\) and \([\Theta \cdot \Gamma_2]_{\bigcirc} Q\): they exist because such typing judgements hold, by inversion of \((T\text{-Par})\) (Fig. 4). Similar uses of sub-processes encoded w.r.t. their typing occur in the other cases. Process declaration \(\text{def } X(\bar{x}:U) = P\) in \(Q\) is encoded as a replicated π-calculus process that inputs a value \(z\) on a name \([X] = z_X\) (matching Def. 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 \(X(\bar{v})\) is encoded as a process that sends the encoded values \(\bar{v}\) on \(z_X\) and ends (in MPST π-calculus, process calls are in tail position).

Selection on \(d[p]\) is encoded using information from the session typing context: the fact that \(c\) has type \(S = p \oplus \llbracket U \rrbracket . S'\) — i.e., \([S]\) is a record type with one entry \(q : z_q\) for each \(q \in S\). Therefore, the encoding first deconstructs \([c]\) (using with), then uses the (linear) name in its \(p\)-entry to output on \(z_p\). Before performing the output, however, a new name \(z\) is created: it is the continuation of the interaction with \(p\). Then, one endpoint of \(z\) is sent through \(z_p\) 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 \([c]\) (using let) with a “new” record, consisting in all the entries of the “original” \([c]\), except \(z_p\) (which has been used for output). More in detail, the
“new” \([c]\) has an entry for \(p\) (mapping \(p\) to \(z\)) iff \(S'\) still involves \(p\) (otherwise, if \(p \notin S'\), then \(z\) is discarded, since it has type \([S'][p]=\textbf{end}=\bullet\). After \texttt{let}, the encoding continues as \([P]\).

Symmetrically, branching on \(c[p]\) is also encoded using information from the typing context, i.e., that \(c\) has type \(S=p \& z \in S\) — and therefore, \([S]\) is a record type with one entry \(q:z_q\) for each \(q \in S\). As above, the encoded process deconstructs \([c]\) (using \texttt{with}), an then uses the (linear) name in its \(p\)-entry to perform an input \(z_p(y)\); \(y\) 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 \([c]\) is rebound (using \texttt{let}) and the process continues as \([P]\). The rebinding of \([c]\) depends on \(l_i\) and the continuation type \(S'\): the “new” \([c]\) is a record with \textit{all} the linear names of the “original” \([c]\), except \(z_p\) (which has been used for input); as above, an entry for \(p\) will exist (and map \(p\) to \(z\)) iff \(S'\) still involves \(p\) (otherwise, if \(p \notin S'\), then \(z\) has type \(\bullet\) and is discarded).

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

\[
\Gamma, s[p]:S, s[q]:S' \vdash s[p][q] \& [l(x).P] | s[q][p] \oplus [l(v).Q] \rightarrow P\{\gamma/z\} | Q
\]

We would like its encoding to reduce and communicate, too — but it is \textit{not} the case:

\[
\text{with } \{x:z_x \in S\} \text{ do } \ldots \text{ with } \{x:z_x \in S\} \text{ do } \ldots
\]

and the reason is that \([s[p]], [s[q]]\) are “just” record-typed \textit{names} (respectively \(z_{s[p]}, z_{s[q]}\), as per Def. 5.4), whereas \texttt{with}-prefixes only when applied to \texttt{record values} (cf. Def. 3.2). Hence, to let our encoded terms reduce, we must first substitute \([s[p]], [s[q]]\) with two records; moreover, to let the two encoded processes synchronise and exchange \([v]\), 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 \textit{same} (linear) name. In the following, we show how \(\sigma (\Gamma')\) handles this issue.

#### Reification of Multiparty Sessions

By simply translating a channel with role \(s[p]\) into a π-calculus name \(z_{s[p]}\), 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 \textit{pairs of roles}. We can formalise “connections” as follows.

\begin{definition}
\textbf{Definition 5.5.} The \textit{connections} of \(s\) in \(\Gamma\) are:

\(\text{conn}(s, \Gamma) \triangleq \{ p,q \mid s[p]: S_p \in \Gamma \land q \in S_p \}\)

\end{definition}

Intuitively, two roles \(p,q\) are connected by \(s\) in \(\Gamma\) if \(p\) occurs in the type \(\Gamma (s[q])\) (but \(q\) \textit{might not} occur in \(\Gamma (s[p])\); note, however, that \(q\) will always occur if \(\Gamma\) is consistent).

Now, as anticipated above, we want to substitute each \([s[p]]\) 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 Def. 5.6.

\begin{definition}[Reification and decomposition of MPST contexts]
\textbf{Definition 5.6} (Reification and decomposition of MPST contexts). The \textit{reification} of a session typing context \(\Gamma_5\) is the substitution:

\[
\sigma (\Gamma_5) = \{ [a:z_{(s,p,a)}] \}_{s,p,a \in s} \}
\]

The \textit{linear decomposition} of \(\Gamma_5\) is the π-calculus typing context \(\delta (\Gamma_5)\), defined as:

\[
\delta (\Gamma_5) = \{ z_{(s,p,q)} : \text{\texttt{unf}} (S_p | q) \} \}
\]

The π-calculus \textit{reification typing rule} is (note that \(\Gamma_5, \Gamma'_5\) are MPST typing contexts):

\[
\frac{\}{\Theta, \Gamma_5 \vdash P} (\Pi_5\text{-REIFY})
\]
where $\sigma$ is a connection of $\Gamma$. Then, $P$ is encoded, and the substitution $\sigma(\Gamma')$ is applied: it replaces each $[s[p]]$, $[s[q]]$ in $[P]$ with records based on the delimited $z_{s,p,q}$.  

\begin{itemize}
\item \textbf{Example 5.8.} Consider (1). If we delimit $s$ and encode the resulting process, we obtain a $\pi$-calculus process based on (2), enclosed by the delimitations yielded by $[[\langle\nu s\rangle]]$, and the substitution $\sigma(s,p,s[q];S',\ldots)$. Since the latter replaces $[s[p]], [s[q]]$ with records whose entries reflect roles($S$) and roles($S'$), the encoding can now reduce, firing the two \textbf{w}iths.
\item \textbf{Example 5.9.} Consider the main server/clients parallel composition in Ex. 2.2:
\begin{equation}
Q = (\nu s_1, s_2, s_3) \left( \begin{array}{c}
Q | P_s | P_{s_1} | P_{s_2} | P_{s_3} \\
\end{array} \right) \quad \text{where}
\end{equation}

The simplest part of Def.5.6 is $\sigma(\Gamma_S)$: it is a substitution that, for each $s[p] : S_p \in \Gamma_S$, replaces $[s[p]]$ with a record containing one entry $q : z_{s,p,q}$ for each $q \in S_p$. Note that if there is also some $s[q] : S_q \in \Gamma_S$ with $p \in S_q$, then the corresponding record (replacing $[s[q]]$) has an entry $p : z_{s,q,p} = z_{s,p,q}$; i.e., $p$ (in one record) and $q$ (in the other) map to the same name. This realises the intuition of “multipartty sessions as records of interconnected binary channels”.

The definition of $\sigma(\Gamma_S)$ was the last ingredient needed to formalise our encoding, presented in Def.5.7 below. The rest of Def.5.6 will be used later on, to prove its correctness (Thm.6.2): hence, we postpone its explanation to page 20.

\begin{itemize}
\item \textbf{Definition 5.7 (Encoding).} The \textit{encoding of session typing judgements} is given in Fig. 7.
\end{itemize}

We define $[P]_{\Theta, \Gamma} = Q$ iff $[\Theta \cdot \Gamma \vdash P] = [\Theta \cdot \Gamma] \vdash Q$. Sometimes, we write $[P]$ for $[P]_{\Theta, \Gamma}$ when $\Theta, \Gamma$ are empty, or clear from the context.

We conclude by explaining the last case in Fig. 7, which was not addressed on p.17. The process $(\nu s : \Gamma')P$ is encoded by generating one delimitation for each $z_{s, p, q}$ whenever $\{p_i, q_i\}$ is a connection of $s$ in $\Gamma'$ (Def.5.5). Then, $P$ is encoded, and the substitution $\sigma(\Gamma')$ is applied: it replaces each $[s[p]]$, $[s[q]]$ in $[P]$ with records based on the delimited $z_{s,p,q}$.  

\begin{itemize}
\item \textbf{Example 5.8.} Consider (1). If we delimit $s$ and encode the resulting process, we obtain a $\pi$-calculus process based on (2), enclosed by the delimitations yielded by $[[\langle\nu s\rangle]]$, and the substitution $\sigma(s,p,s[q];S',\ldots)$. Since the latter replaces $[s[p]], [s[q]]$ with records whose entries reflect roles($S$) and roles($S'$), the encoding can now reduce, firing the two \textbf{w}iths.
\end{itemize}
6 Properties of the Encoding

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

Encoding of Types  Thm.6.1 below says that our encoding (1) commutes the duality between partial session types (Def.2.8) and $\pi$-types (Def.4.1), and (2) also preserves subtyping.

$\blacktriangleright$ Theorem 6.1 (Duality/subtyping preservation). $[\mathcal{H}]=[[\mathcal{H}]]$; if $U \leq_U U'$, then $[U] \leq_\pi [U'].$

Encoding of Typing Judgements  Thm.6.2 shows that the encoding of session typing judgements into $\pi$-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 $\pi$-calculus.

$\blacktriangleright$ Theorem 6.2 (Correctness of encoding). $\Gamma \vdash v:U$ implies $[[\Gamma]]\vdash[[v]][[U]]$, $\Theta \vdash X:U$ implies $[[\Theta]] \vdash [[X]] : [[U]]$, and $\Theta \vdash P$ implies $[[\Theta]] \vdash P$.

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

$$\begin{align*}
(T\text{-}Sub) & \quad \Theta \cdot \Gamma, c:U \vdash P; \quad U' \leq_\pi U \\
\Theta \cdot \Gamma, c:U' \vdash P & \implies \Theta \cdot \Gamma, c:U' \vdash P \\
\Theta \cdot \Gamma, c:U' \vdash P & \implies \Theta \cdot \Gamma, c:U' \vdash P
\end{align*}$$

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

More on reification and decomposition  By Def.5.6, the typing context $\delta(\Gamma_\Sigma)$, when defined, $\delta(\Gamma_\Sigma)$ has an entry for each role of each channel in $\Gamma_\Sigma$; more precisely, an entry $z_{(s,p,q)}$ for each $s[p]:S_p \in \Gamma_\Sigma$ and $q \in S_p$. Such entries are used to type the records yielded by $\sigma(\Gamma_\Sigma)$.

The type of $z_{(s,p,q)}$ is based on the encoding of the unfolded partial projection $S_p \vdash q$, that can be either $\ast$, or $\text{Li}(T)/\text{Lo}(T)$ (for some $T$). Note that if there is also some $s[q]:S_q \in \Gamma_\Sigma$ with $p \in S_q$, the type of $z_{(s,p,q)} = z_{(s,p,q)}$ (when defined) is $[[\text{unf}(S_p \vdash q)] \cap [[\text{unf}(S_q \vdash p)]]].$

This creates a deep correspondence between the consistency of $\Gamma_\Sigma$ and the existence of $\delta(\Gamma_\Sigma)$, shown in Thm.6.3: the precondition for $\text{MPST type safety}$ (i.e., consistency of $\Gamma_\Sigma$) is precisely characterised in $\pi$-calculus by the linear decomposition at the roots of our encoding.
Theorem 6.3 (Precise decomposition). \( \Gamma_5 \) is consistent if and only if \( \delta(\Gamma_5) \) is defined.

The final part of Def.5.6 is the \( \pi \)-calculus typing rule (\( \text{T}_\pi \text{-Reify} \)), that uses \( \delta(\Gamma_5) \) to type a process on which \( \sigma(\Gamma_5) \) has been applied. Intuitively, \( \delta(\Gamma_5) \) provides a typing context that types each record yield by \( \sigma(\Gamma_5) \). We now explain how the rule works and why it is sound (with a slight simplification). Let \( \Gamma_5 = \{[s:p]:S_p\}_{p \in I} \), for some \( I \). Then, by Def.5.6:

\[
\delta(\Gamma_5) = \bigcap_{p \in I} \{z_{[s:p],q}:\text{unf}(S_p \vdash q)\}_{[p,q] \in \text{conn}(s,\Gamma_5)} \quad \sigma(\Gamma_5) = \{q_{z_{[s:p],q}}\}_{[p,q] \in \text{conn}(s,\Gamma_5)}
\]

(Note: \( \delta(\Gamma_5) \) is defined iff \( \delta(\Gamma_5) \) is consistent, by Thm.6.3). Take the I/O types yielded by \( \delta(\Gamma_5) \), i.e., \( \{T_{[s:p],q}\} \}_{[p,q] \in \text{conn}(s,\Gamma_5)} \) such that \( \delta(\Gamma_5) = [\Theta \cdot \Gamma_5] \) (\( \text{T}_\pi \text{-Reify} \)) (note \( T_{[s:p],q}, T_{[s:p],p} \) are distinct). If we assume \([\Theta \cdot \Gamma_5], [\Gamma_5] \vdash P\), this derivation holds:

\[
[T_\pi \text{-Name}] \quad \forall q \in S_p \quad z_{[s:p],q} : T_{[s:p],q} \vdash z_{[s:p],q} : T_{[s:p],q} \quad T_{[s:p],q} \leq \Xi S_p \vdash q \quad (T_\pi \text{-Sub})
\]

\[
\{z_{[s:p],q} : \Xi S_p \vdash q\}_{q \in S_p} \quad \{q_{z_{[s:p],q}}\}_{q \in S_p} \quad (T_\pi \text{-Rec})
\]

\[
[T_\pi \text{-Reify}] \quad [\Theta \cdot \Gamma_5], [\Gamma_5] \vdash P \quad (\text{T}_\pi \text{-Sub})
\]

In particular, the assumptions \( T_{[s:p],q} \leq \Xi S_p \vdash q \) hold by Lemma 4.7, since each \( T_{[s:p],q} \) is obtained by splitting \( \delta(\Gamma_5) \) (that combines types with \( \ominus \)) using \( \oplus \). The equivalence in the conclusion holds since \( \text{dom}([\Theta \cdot \Gamma_5]) \cap \text{dom}(\delta(\Gamma_5)) = \emptyset \). Hence: if the (\( \text{T}_\pi \text{-Reify} \)) premise \([\Theta \cdot \Gamma_5], [\Gamma_5] \vdash P\) holds, the above derivation holds, proving the conclusion of (\( \text{T}_\pi \text{-Reify} \)).

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

\[
[T_\pi \text{-Reify}] \quad \Theta \cdot \Gamma, [\Gamma'] \vdash P_{[\Theta \cdot \Gamma, \Gamma'] \sigma(\Gamma')} \quad (\text{T}_\pi \text{-Reify})
\]

where \( \delta(\Gamma') \) types all the names \( z_{[s:p,q]} \) in \( \sigma(\Gamma') \), that are also delimited by \( [\nu s] \). We can conclude by applying (\( \text{T}_\pi \text{-Res} \)) to type such delimitations (cf. Fig.5 — this is allowed by (\( \text{T}_\pi \text{-Sub} \), Fig.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., \( S \leq S' \) implies \( \text{roles}(S) = \text{roles}(S') \)).

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 Thm.6.5, where \( \text{with} \) denotes a reduction induced by (\( \text{R}_\pi \text{-Wru} \)) (Def.3.2).

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

1. (Completeness) \( P \rightarrow P' \) implies \( \exists \nu \varnothing, P'' \) such that \( \text{with} \rightarrow (\nu \varnothing)P'' \) and \( P'' = [P] \).
2. (Soundness) \( \text{with} \rightarrow P \) implies \( \exists \nu \varnothing, P'' \) s.t. \( P \rightarrow P''(\nu \varnothing)P'' \) and \( P'' = [P] \).
The statement of Thm. 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 $[P']_{\alpha \cdot \alpha}$, which implies $\alpha \cdot \alpha \vdash P'$ (cf. Def. 5.7). The restricted variables $\bar{x}$ in items 1-2 are generated by the encoding of selection (Fig. 7): it creates a (delimited) linear name to continue the session. To see why item 2 uses $\xRightarrow{\text{with}}$, consider the following MPST process:

$$\alpha \cdot \Gamma, s[p]:S \vdash s[p][q] & \{l(x).P\} \not\overset{\text{with}}{\rightarrow} \star$$

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

$$\text{with } [x:z]_{r \in S} = [x:z_{(s,p,r)}]_{r \in S} \text{ do } z_q(y) \ldots \text{ with } z_{(s,p,q)}(y) \ldots \not\rightarrow$$

This happens whenever a process is deadlocked, because in Fig. 7, the “atomic” MPST branch/select actions are encoded with multiple $\pi$-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.

Thm. 6.5 yields a corollary on deadlock freedom (Cor. 6.6), that in turn allows to transfer deadlock freedom (Thm. 2.19) from MPST to $\pi$-calculus processes (Cor. 6.7 below).

**Corollary 6.6.** $P$ is deadlock-free if and only if $[P]$ is deadlock-free, i.e.: $[P] \not\overset{\star}{\rightarrow} P' \not\rightarrow$ implies $\exists Q \equiv 0 \text{ such that } P' = [Q]$.

**Corollary 6.7.** Let $\alpha \cdot \alpha \vdash P$, where $P \equiv (\nu s:G)_{i \in I} P_i$ and each $P_i$ only interacts on $s[p_i]$. 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 §1, focusing on player $b$: we sketch the API generation and an implementation of a client, following the results in §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 $\pi$-typed message: i.e., they represent channel endpoints with $\pi$-calculus types $\text{Lo}(T)$ and $\text{Li}(T)$ (Def. 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 Def. 5.1). Further, lchannels allows to send/receive `In[T]/Out[T]` instances for binary session delegation [61, Ex. 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 Thm. 6.2, Def. 5.1 and Thm. 6.3, we know that the game player session type $S_b$ in our example (see §1, p. 2) provides the type safety
guarantees of a tuple of (linear) channels, whose types are given by the encoded partial projections of \( S_b \) onto \( a \) and \( c \) (Def. 2.9). This suggests that, using \texttt{lchannels}, the delegation of an \( S_b \)-typed channel (as seen in §1) could be rendered in Scala as:

```scala
case class PlayB(payload: \( S_b \))
```

i.e., as a linear input channel carrying a message of type \( \text{PlayB} \), whose \texttt{payload} has type \( \text{S}_b \); \( \text{S}_b \), in turn, is a Scala \texttt{case class}, which can be seen as a labelled tuple, that maps \( a, c \) to I/O channels — whose types derive from \([S_b]a\) and \([S_b]c\) (in fact, they carry messages of type \( \text{InfoAB}/\text{InfoBC} \)). In this view, \( \text{S}_b \) is our Scala rendering of the encoded session type \([S_b]\).

As said above, \texttt{lchannels} allows to send channels remotely — hence, also allows to remotely send \textit{tuples} of channels (e.g., instances of \( \text{S}_b \)); thus, with this simple approach, we obtain \textit{type-safe distributed multiparty delegation} of an \([S_b]\)-typed channel tuple “for free”.

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

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

E.g., consider again \( \text{S}_b \) and \( \text{S}_b \). \( \text{S}_b \) requires to \texttt{send} towards \( c \), so \( \text{S}_b \) could provide the API:

```scala
case class S_b(a: InfoAB, c: InfoBC)

def send(v: String) = { // v is the payload of InfoBC message
val c′ = c \(_!\) InfoBC(v) // \texttt{lchannels} method: send \( v \), and return continuation
S_b(a, c′) } // return a “continuation object”
```

Now, \( \text{S}_b \).\texttt{send}() behaves \textit{exactly} as our process encoding in Fig. 7 (case for selection \( \texttt{@} \)): it picks the correct channel from the tuple (in this case, \( c \)), creates a new tuple \( \text{S}_b′ \), 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 \( \text{S}_b′ \) should be similar, with a \texttt{receive()} method that uses a for input (by following the encoding of \( \texttt{&} \)). This way, a programmer is correctly led to write, e.g., \texttt{val x = s.send(...).receive()} (using method call chaining) — whereas attempting, e.g., \texttt{...receive()} is rejected by the Scala compiler (method undefined). These \texttt{send()}/\texttt{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 §2, including, e.g., projection, type merging and delegation (not previously supported). Our extension supports protocols with the syntax in Fig. 9 (left), by augmenting Scribble with a \textit{projection operator} \( \& \); then, it computes the projections/encodings explained in §5, and automates the Scala API generation as outlined above (producing, e.g., the \( \text{S}_b \), \( \text{S}_b′ \),... classes and their \texttt{send}/\texttt{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 \texttt{lchannels} (that, as shown above, takes care of most irksome aspects — e.g., delegation). As a result, the \( P_b \) client in Fig. 1 can be written as in Fig. 9 (right); and although conceptually programmed as Fig. 2, the networking mechanisms of the game will concretely follow Fig. 8.
A Linear Decomposition of Multiparty Sessions

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 {
        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)))
        }
    }
}

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

8 Conclusion and Related Works

We presented the first encoding of a full-fledged multiparty session π-calculus into standard π-calculus (§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 (§7). We proved that the type safety property of MPST is precisely characterised by our decomposition into linear π-calculus (Thm.6.3). We encode types by preserving duality and subtyping (Thm.6.1); our encoding of processes is type-preserving, and operationally sound and complete (Thm.6.2 and Thm.6.5); hence, our encoding preserves the type-safety and deadlock-freedom properties of MPST (Cor.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 §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 (§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 §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 CFMSs [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. Ex. 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 §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., µt.Łt.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 µt.μt′.Łt′t. As observed in §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., µt.Łt(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 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 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., Łt(µt.Łt(t)) instead of µt.Łt(t)), but if adopted, would complicate Def.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 Cor.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


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


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


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