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 π-calculus into standard linear π-calculus, and (2) using the encoding as the foundation of a practical toolchain for safe multiparty programming in Scala.

Our encoding is type-preserving and operationally sound and complete. Importantly for distributed applications, it preserves the choreographic nature of MPST and illuminates that multiparty sessions (and their safety properties) can be precisely represented with a decomposition into binary linear channels. Previous works have only studied the relation between (limited) multiparty sessions and binary sessions by orchestration means.

We exploit these results to implement an automated generation of Scala APIs for multiparty sessions. These APIs act as a layer on top of existing libraries for binary communication channels: this allows distributed multiparty systems to be safely implemented over binary transports, as commonly found in practice. Our implementation is also the first to support distributed multiparty delegation: our encoding yields it for free, via existing mechanisms for binary delegation.

1 Introduction

Correct design and implementation of concurrent and distributed applications is notoriously difficult. Programmers have to deal with many challenges, pertaining to both protocol conformance (do the messages being sent/received respect a given specification?) and the communication mechanics (how are the interactions actually performed?). These difficulties are exacerbated by the potential complexity of interactions between multiple participants, and in settings where the communication topology is not fixed.

As an example, consider a common scenario for a peer-to-peer multiplayer game: the clients, initially unknown to each other, first 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, expected to set the game for

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{game_server_with_3_clients.png}
\caption{Game server with 3 clients.}
\end{figure}
three players, $P_a$, $P_b$ and $P_c$. To set up a game, the server sends to each client some networking information (denoted by the $s[a]$, $s[b]$, $s[c]$ payloads of the playA/b/c messages) needed to “introduce” the clients to each other and allow them to communicate. The clients then proceed according to the main game protocol (annotated as “Game”): it consists of some initial message exchanges (InfAB), and a main game loop, where $P_a$ selects between two possible messages to send to $P_b$ (Mov1AB or Mov2AB) followed by a message from $P_b$ to $P_a$, who can choose which message send back to $P_a$.

As Fig. 1 illustrates, this applications can involve richly structured protocols, with non-trivial message dependencies between multiple roles, and a changing communication topology (initially client-to-server, eventually becoming client-to-client). Turning such a high level specification into an actual implementation is not straightforward—programmers would greatly benefit from tools and programming aids to statically assist the detection of protocol violations in source code, and correctly implement the communication topology dynamics.

**Multiparty Session Types** (MPST) [26] is a theoretical framework allowing for the precise modelling of such applications. In MPST, participants are abstracted as roles (e.g., game clients a, b, c) and implemented as session $\pi$-calculus processes, that model server/client programs. In the MPST framework, the “networking information payloads” $s[a]$, $s[b]$, $s[c]$ can be naturally modelled as multiparty channel endpoints for the game session $s$. Notably, channel endpoints can themselves be sent/received: formally, this allows to delegate part of a multiparty interaction to another process, resulting in a change of the communicating topology. In our example, the server $Q$ delegates (i.e., sends) the channel endpoint $s[b]$ to $P_a$; the latter can then use $s[b]$ to interact with the two processes that own the endpoints $s[a]$ and $s[c]$ (i.e., $P_a$ and $P_b$ after the other two delegations).

The MPST framework ensures safe interaction via session types: they formalise protocols, as structured sequences of inputs, outputs and choices. The session typing system assigns such types to channel endpoints, and type-checks the processes that use them. In our example, the channel endpoint $s[b]$ could be typed as:

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

(1)

$S_b$ says that $s[b]$ must be used to realise the Game interactions of $P_b$ in Fig. 1: first to send $\text{InfoBC}(\text{string})$ to c, then receive $\text{InfAB}$ 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 $\text{Mov1AB}(\text{Int})$ (in which case, it must be used to send $\text{Mov1BC}(\text{Int})$ to c, and loop), or $\text{Mov2AB}$ (then, it must be used to send $\text{Mov2BC}$ to c, and loop). Analogous types can be assigned to $s[a]$ and $s[c]$. The delegation actions are represented by session types like $q.?\text{PlayB}(S_c).\text{end}$, which means: from role $q$, receive a message $\text{PlayB}$ carrying a channel endpoint that must be used according to $S_b$ above; then, end the session. Session type checking ensures that, e.g., the process $P_b$ uses its channels as prescribed by the types above—thus safely implementing the expected channel dynamics and fulfilling the role of $b$ in the game.

Finally, the MPST framework allows to formalise, e.g., the whole Game protocol in Fig. 1 as a global type, and validate that it is deadlock-free; then, via typing, check whether an ensemble of processes interacts according to the global type (and is, thus, deadlock-free).

**MPSTs in practice: challenges** The above suggests that MPSTs offer a promising formal foundation for safe distributed programming, helping to develop concurrent applications whose interactions are type-safe and deadlock-free. However, bridging the gap between the abstract theory and a concrete implementation raises several challenges:
C1. Multiparty session types allow 2, 3 or more roles to interact—but in practice, communication occurs over binary channels (e.g., TCP sockets). Can multiparty channels be implemented as compositions of binary channels, while preserving their safety properties?

C2. Multiparty session types are far from the types found in “mainstream” programming languages, as demonstrated by $S_b$ in (1). Can they be represented, e.g., as objects? If so, what is their programming interface? And what are the API internals?

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

Unfortunately, the current state-of-the-art in session types has not addressed these challenges. On one hand, existing theoretical works on encoding multiparty sessions into binary sessions [7, 8] rely on a workaround by introducing 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 [26], and preclude examples such as our peer-to-peer game in Fig.1. On the other hand, there are no existing implementations of full-fledged MPST; e.g., [52, 30, 31, 40, 48, 56, 51] only support binary sessions, while none of [27, 58, 16, 19] support session delegation.

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

S1. we develop the first choreographic encoding of a “full-fledged” multiparty session $\pi$-calculus into standard 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 channels and types that cater only for binary communication, and each channel 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 linear channels—and whether multiparty session types can be represented by a decomposition into linear types. The practical payoff is that linear $\pi$-calculus channels/types are amenable for an (almost) direct object-based representation, as demonstrated in [56]: this tackles challenge C2. Moreover, linear $\pi$-calculus allows to prove whether such a decomposition is “correct”—i.e., whether it preserves type safety, and whether MPST processes can be encoded so that they only interact on binary channels, while preserving their original behaviour (thus “inheritting” 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 [28], Haskell [52, 30, 40], Links [42], Rust [31], Scala [56], ML [51]), 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 formulation of MPSTs that clearly separates the global/local typing levels. This allows us to “close the gaps” between the intricacies of the MPST theory and the (much simpler) $\pi$-calculus, while staying faithful to standard MPST literature. Via our MPST formulation, we also spot a longstanding issue with type merging [17] (Def. 2.11; §2.1 “On Consistency”) and fix it, obtaining a revised subject reduction for MPSTs (Theorem 2.16).
At the heart of our encoding there is the discovery that the type safety property of MPST is precisely characterised as a decomposition into linear $\pi$-calculus types (Theorem 6.4). Our encoding of types preserves duality and subtyping (Theorems 6.1 and 6.2); our encoding of processes is type-preserving and operationally sound and complete (Theorems 6.3 and 6.6).

We subsume the encodings of binary sessions into $\pi$-calculus [13, 14], and support recursion (§4), which was not properly handled in [12]. 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 [7, 8].

In §7, we use our encoding as formal basis for the first implementation of multiparty sessions supporting distributed multiparty delegation, over existing Scala libraries. Our implementation is available (as Open Source software) in [55].

Conventions

In derivations, we use a single/double line for inductive/coinductive rules. Recursive types $\mu t.T$ are guarded, i.e., $t$ can only appear in $T$ under a type constructor different from $\mu$. As usual, we define $\text{unf}(\mu t.T) = \text{unf}(T[\mu t.T/t])$, and $\text{unf}(T) = T$ when $T \neq \mu t.T'$. We adopt syntactic type equality, and thus distinguish a recursive type from its unfolding. Types are always closed. We write $P \rightarrow P'$ for process reductions, $\rightarrow^*$ for the reflexive+transitive closure of $\rightarrow$, and $P \not\rightarrow P'$ iff $\exists P' \text{ such that } P \rightarrow P'$. We assume a basic subtyping $\leq_B$ capturing e.g. $\text{Int} \leq_B \text{Real}$. For readability, we use blue/red for multiparty/standard $\pi$-calculus.

2 Multiparty Session $\pi$-Calculus

In this section we illustrate a multiparty session $\pi$-calculus [26] complete with recursion, subtyping [18] and type merging [61, 17]. We adopt a notation based on [10].

Definition 2.1. The syntax of multiparty session $\pi$-calculus processes and values is:

Processes $P, Q ::= 0 | P | Q | (\nu s)P$ (inaction, composition, restriction)
$c.P | \{l_i(x_i).P_i\}$ (selection towards role $p$)
$\text{def } D \text{ in } Q | X(x)$ (process definition, process call)

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

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

Values $v ::= c | \text{false} | \text{true} | 42 | \ldots$ (channel, base value)

$\text{fc}(P)$ is the set of free channels with roles in $P$, and $\text{fv}(P)$ is the set of free variables in $P$.

The inaction $0$ represents a terminated process. The parallel composition $P | Q$ represents two processes that can execute concurrently (and possibly communicate). The session restriction $(\nu s)P$ delimits the scope of a session $s$ to $P$. Process $c.P$ performs a selection (internal choice) on the channel $c$ towards role $p$: the labelled value $l(v)$ is sent, and the execution continues as process $P$. Dually, process $c.P = \{l_i(x_i).P_i\}$ waits for a branching (external choice) on the channel $c$ from role $p$. If the labelled value $l_k(v)$ is received (with $k \in I$), then the execution continues as $P_k$ (with $x_k$ holding value $v$). Note that for all $i \in I$, variable $x_i$ is bound with scope $P_i$. In both branching and selection, the labels $l_i$ ($i \in I$) are all different and their order is irrelevant. Process definition $\text{def } D \text{ in } Q$ and process call $X(x)$ model recursion, with $D$ being a process declaration: the call invokes $X$ by replacing its formal parameters with the actual ones. We postulate that process declarations are closed, i.e., in $X(x) = P$, we have $\text{fv}(P) \subseteq x$ and $\text{fc}(P) = \emptyset$. A channel
A. Scalas, O. Dardha, R. Hu, N. Yoshida

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 \( s_q \)); (c) clients directly interacting on session \( s \), after “complete” delegation.

\( c \) 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. Note that our syntax is simplified in the style of [18]: it does not have dedicated input/output prefixes, but they can be easily encoded using \& (with one branch) and \( \oplus \).

Example 2.2. The following MPST \( \pi \)-calculus process implements the scenario in Fig. 1:

\[
\text{def } Loop_b(x) = x[a] & \{ s_{\text{Notify}}(y).x[c] \oplus (s_{\text{Notify}}(y)).Loop_b(x) , s_{\text{Notify}}(z).x[c] \oplus (s_{\text{Notify}}(z)).Loop_b(x) \} \text{ in }
\text{def } Clients_b(y) = y[q] & P_{\text{PlayB}}(z).z[c] \oplus (s_{\text{Notify}}(z)).z[a] & \text{instantiate}(y).Loop_b(z) \text{ in }
(\psi_b,s_b,s_c)(Q \mid P_1 \mid P_2 \mid P_3)
\]

where: \( P_1 = Clients_b(s_b[p]) \) (for brevity, we omit the definitions of \( P_2 \) and \( P_3 \)).

In the 3\( ^{rd} \) line, \( s_a,s_b,s_c \) are the sessions between the server process \( Q \) and the clients \( P_1,P_2,P_3 \), which are composed in parallel. Each session has 2 roles: \( q \) (server) and \( p \) (client); e.g., \( s_b \) is accessed by the server (through the channel with role \( s_b[q] \)) and by the client \( P_2 \) (through \( s_b[p] \); similarly, \( s_a \) (resp. \( s_c \)) is accessed by \( P_1 \) (resp. \( P_2 \)) through \( s_a[p] \) (resp. \( s_c[p] \)), while the server owns \( s_b[q] \) (resp. \( s_c[q] \)). In the body of \( Q \), the server declares a session \( s \) (with 3 roles \( a,b,c \)) for playing the game. Note that the scope of \( s \) does not include \( P_1,P_2,P_3 \); see Fig. 2(a) for a schema of processes and sessions.

The server \( Q \) uses the channel with role \( s_b[q] \) (resp. \( s_a[q],s_c[q] \)) to concurrently send the message \( P_{\text{PlayB}} \) (resp. \( P_{\text{PlayA}},P_{\text{PlayC}} \)) and the channel with role \( s[b] \) (resp. \( s[a],s[c] \)) to \( p \); i.e., the server performs a delegation to the client process \( P_1 \) (resp. \( P_2,P_3 \)). This way, the client obtains a channel endpoint to interact in the game session \( s \), interpreting role \( b \) (resp. \( a,c \)).

The client \( P_b \) is implemented by invoking \( Clients_b(s_b[p]) \) (defined in the 2\( ^{nd} \) line). Here, \( y[q] & P_{\text{PlayB}}(z) \) means that \( y \) (that becomes \( s_b[p] \) after the invocation) is used to receive \( P_{\text{PlayB}}(z) \) from \( q \), while \( z[c] \oplus (s_{\text{Notify}}(z)) \) means that \( z \) (that becomes \( s[b] \) after the delegation is received) is used to send \( s_{\text{Notify}}(z) \) to \( c \). The game loop is implemented with the recursive process call \( Loop_b(z) \) (defined in the 1\( ^{st} \) line) — which becomes \( Loop_b(s[b]) \) after delegation.

Definition 2.3. The operational semantics of multiparty session processes is:
A Linear Decomposition of Multiparty Sessions

We omit the set of wrong (i.e., use their channels in a type-safe way), and we now illustrate the typing system for the MPST with $\nu$

2.1 Multiparty Session Typing

▶ sequences of input/output actions, specifying who is the source/target role of interaction.

Local types first discuss to use its channel endpoint. Local types, in turn, are assigned to communication channels, into a set of protocol involving various

Rules are standard: reduction can happen under parallel composition, restriction and process definition, and the reduction relation is closed under structural congruence.

Example 2.4. The process in Ex. 2.2 reduces as (see also Fig. 2(b), noting the scope of $s$):

R-Call $P \to Q$ implies $P \mid R \to Q \mid R$

R-Def $P \to Q$ implies $(\nu s)P \to (\nu s)Q$

R-Struct $P \equiv P'$ and $P \to Q$ and $P' \equiv Q$ implies $P' \to Q'$ (with $\equiv$ standard — see §A)

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 MPST framework fosters a top-down approach where a global type $G$ describes a protocol involving various roles — such as the game with roles $a, b, c$ in §1. $G$ is projected into a set of local types $S_a, S_b, S_c, \ldots$ (one per role) that specify how each role is expected to use its channel endpoint. Local types, in turn, are assigned to communication channels, and type-check the processes using them. Session typing ensures that processes (1) never go wrong (i.e., use their channels in a type-safe way), and (2) interact obeying the protocol in $G$, by respecting its local 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.

Types: Local and Partial

Multiparty 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:

$S ::= p \&_{i \in I} ?l_i(U).S_i$ (branching from role $p$ — with $I \neq \emptyset$)
$p \oplus_{i \in I} !l_i(U).S_i$ (selection towards role $p$ — with $I \neq \emptyset$)
$\mu t.S\ |\ t\ |\ \text{end}$ (recursive type, type variable, termination)

$B ::= \text{Bool} \mid \text{Int} \mid \ldots$ (base type) $U ::= B \mid S$ (closed) (payload type)

We omit $\&/\oplus$ when $I$ is a singleton: $pl_i(\text{Int}).S_i$ stands for $p \oplus_{i \in I} !l_i(\text{Int}).S_i$.

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

roles($p \oplus_{i \in I} !l_i(U).S_i$) $\triangleq$ roles($p \&_{i \in I} ?l_i(U).S_i$) $\cup$ $\{p\} \cup \bigcup_{i \in I}$ roles($S_i$)
roles($\text{end}$) $\triangleq$ $\emptyset$ roles($t$) $\triangleq$ $\emptyset$ roles($\mu t.S$) $\triangleq$ 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 \( p \&_{i \in I} ?l_i(U_i).S_i \) describes a channel that can receive a label \( l_i \) from role \( p \) (for some \( i \in I \), chosen by \( p \)), together with a payload of type \( U_i \); then, the channel must be used as \( S_i \). The selection \( p \oplus_{i \in I} U_i(U_i).S_i \), 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/selection 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 \( \text{Bool} \), \( \text{Int} \), 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 (equation (1)).

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

► Definition 2.7. Partial session types, denoted by \( H \), are:

\[
H \ ::= \ \&_{i \in I} \ ?l_i(U_i).H_i \ | \ \oplus_{i \in I} \ U_i(U_i).H_i \quad \text{branching, selection} \quad (\text{with } t \neq U_i \text{ closed})
\]

\[
\mu t.H \ | \ \mid \ | \ \text{end} \quad \text{recursive type, type variable, termination}
\]

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:

\[
\begin{align*}
\text{end} & \triangleq \text{end} & \bar{t} & \triangleq \bar{t} \\
\overline{\&_{i \in I} \ ?l_i(U_i).H_i} & \triangleq \&_{i \in I} \ ?l_i(U_i).\overline{H_i} & \overline{\oplus_{i \in I} \ U_i(U_i).H_i} & \triangleq \oplus_{i \in I} \ U_i(U_i).\overline{H_i} & \overline{\mu t.H} & \triangleq \mu t.\overline{H}
\end{align*}
\]

The dual of a select type is a branching type with dualised continuations, and vice versa. The payloads \( U_i \) are the same. Duality is the identity on \( \text{end} \) and on a type variable \( t \), and it is homomorphic on a recursive partial session type \( \mu t.H \).

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 \( q \):

\[
\begin{align*}
\text{end} \ | \ q & \triangleq \text{end} & t \ | \ q & \triangleq t & (\mu t.S) \ | \ q & \triangleq \begin{cases} \mu t.(S \ | \ q) & \text{if } S \ | \ q \neq t' \ (\forall t') \\ \text{end} & \text{otherwise} \end{cases} \\
(p \oplus_{i \in I} ?l_i(U_i).S_i) \ | \ q & \triangleq \begin{cases} \oplus_{i \in I} ?l_i(U_i).(S_i \ | \ q) & \text{if } q = p, \\ \bigcap_{i \in I} (S_i \ | \ q) & \text{if } p \neq q \end{cases} \\
(p \&_{i \in I} ?l_i(U_i).S_i) \ | \ q & \triangleq \begin{cases} \&_{i \in I} ?l_i(U_i).S_i \ | \ q & \text{if } q = p, \\ \bigcap_{i \in I} (S_i \ | \ q) & \text{if } p \neq q \end{cases}
\end{align*}
\]

where \( \bigcap \) is the merge operator for partial session types:

\[
\begin{align*}
\text{end} \ | \ \text{end} & \triangleq \text{end} & t \ | \ t & \triangleq t & \mu t.H \ | \ \mu t.H' & \triangleq \mu t.(H \ | \ H') \\
\&_{i \in I} ?l_i(U_i).H_i \ | \ \&_{i \in I} ?l_i(U_i).H'_i & \triangleq \&_{i \in I} ?l_i(U_i).(H_i \ | \ H'_i) \\
\oplus_{i \in I} U_i(U_i).H_i \ | \ \oplus_{j \in J} U_j(U_j).H'_j & \triangleq \left( \oplus_{k \in I \cup J} U_k(U_k).\left( H_k \ | \ H'_k \right) \right) + \left( \oplus_{i \in I} U_i(U_i).H_i \right) + \left( \oplus_{j \in J} U_j(U_j).H'_j \right)
\end{align*}
\]

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 \( t' \); otherwise, the projection is
We say that multiparty session typing context \( \Gamma \) is a partial mapping defined as:

\[
\Gamma ::= \emptyset \mid \Gamma, x : U \mid \Gamma, s[p] : S \text{ (with } p \notin S) \]

We say that \( \Gamma \) is consistent iff for all \( s[p] : S_p, s[q] : S_q \in \Gamma \) with \( p \neq q \), we have \( S_p \parallel S_q \). We say that \( \Gamma \) is complete iff for all \( s[p] : S_p \in \Gamma \), \( q \in S_p \) 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 \text{end}. The typing contexts composition \( \circ \) is the commutative operator with \text{end} as neutral element:

\[
\begin{align*}
\Gamma_1, c : U \circ \Gamma_2, c' : U' &\triangleq (\Gamma_1 \circ \Gamma_2), c : U \circ c' : U' \quad (\text{if } \text{dom}(\Gamma_2) \not\ni c \neq c' \not\in \text{dom}(\Gamma_1)) \\
\Gamma_1, x : B \circ \Gamma_2, x : B &\triangleq (\Gamma_1 \circ \Gamma_2), x : B
\end{align*}
\]

Note that 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, instead, can be mapped to either session or base types. The clause "\( \forall c : S \in \Gamma : S \models p \) is defined" clause is discussed 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 \( \text{end} \models q = \text{end} \not\models p \models \text{p}!U, S \models (p\text{?}l(U), S) \mid p \).

Note that our consistency in Def. 2.11 is weaker than the one in previous papers (where it is sometimes called coherence): we use \( \preceq \), instead of (syntactic) type equality \( = \), to relate dual partial projections. The reason being: if we use \( = \), and allow partial projections with type merging (Def. 2.9), subject reduction does not hold. Hence, by relaxing our definition, and proving Theorem 2.16 later on, we fix a longstanding mistake appearing e.g., in [61, 17].

Definition 2.12 (Session typing judgements). The process declaration typing context \( \Theta \) maps process variables \( X \) to \( n \)-tuples of types \( U \) (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 \( \forall c : S \in \Gamma, S \models p \) is defined \( \forall p \in S \))

for values: \( \Gamma \vdash v : U \)

for process variables: \( \Theta \vdash X : \vec{U} \)

(T-NAME) says that a channel has the type assumed in the session typing context. (T-BASIC) relates base values to their type. (T-DefCtx) says that a process name has the type assumed in the process declaration typing context. (T-Sub) is the standard subsumption rule, using \( \preceq \) (Def. 2.10). (T-Nil) says that the terminated process is well typed in any unrestricted typing context. (T-Par) says that the parallel composition of \( P \) and \( Q \) is well typed under the composition of the corresponding typing contexts, as per Def. 2.11. (T-Res) says that \( (\nu s)P \) is well typed in \( \Gamma \), if \( s \) occurs in a complete set of typed channels with roles (denoted...
with $\Gamma'$, and the open process $P$ is well typed in the “full” context $\Gamma, \Gamma'$. For convenience, we annotate the restricted $s$ with $\Gamma'$ in the process, giving $(\nu s:\Gamma') P$. $\text{(T-BRCH)}$ (resp. $\text{(T-SALL)}$) 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. $\text{(T-ASS)}$ says that a process definition $\text{def } X(\bar{x}) = P \text{ in } Q$ is well typed if both $P$ and $Q$ are well typed in their typing contexts enriched with $\bar{x}:\bar{U}$. For convenience, we annotate $\bar{x}$ with types $\bar{U}$. $\text{(T-CALL)}$ says that 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 the importance of consistency by restricting our process typing judgements to consistent typing contexts—i.e., those that allow to prove subject reduction. The clause “$\forall c: S \in \Gamma; S \vdash p$ is defined” is not usual in MPST papers, but stems naturally: by requiring the existence of partial projections, the clause rejects processes that $(a)$ use some channel with role $s[p]$: $S$ that, for some $q \in S$, cannot be (consistently) paired with $s[q]$, or $(b)$ contain some variable $x$: $S$ that, in a consistent and complete $\Gamma$, cannot be substituted by any $s[p]$: $S$. Hence, such rejected processes cannot participate in any complete session (case $(a)$), or are never-executed “dead code” (case $(b)$).

**Remark 2.13.** Unlike most MPST papers (e.g., [18, 10]), our rule $\text{(T-RES)}$ does not directly map a session $s$ to a global type: this is explained in the next section, “Global Types”.

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

$$\text{Client}_b, q?:!\text{play}!S_b ; \text{Loop}_b ; \mu a.\{\nu \text{novABC}(\text{int}), \text{c!novABC}(\text{int}), t, \ldots \} \cdot s_b[p] ; q?:!\text{play}!S_b \vdash \text{Client}_b(s_b[p])$$

It says that the channel with role $s_b[p]$ is used following type $q?:!\text{play}!S_b$. end (with a delegation of a $S_b$-typed channel); the argument of $\text{Client}_b$ has the same type; the argument of $\text{Loop}_b$ is used following the game loop. This example cannot be typed without merging $\sqcap$ (Def. 2.9): its derivation requires to compute $S_b[c = !\text{InfoBC}(\text{string}), \mu t.\{\text{!novABC}(\text{int}), t \sqcap !\text{novABC}(\text{bool}), t\} = !\text{InfoBC}(\text{string}), \mu t.\{\text{!novABC}(\text{int}), t \sqcup !\text{novABC}(\text{bool}), t\}$, which is undefined without merging.

The typing rules in Fig. 4 satisfy a subject reduction property (Theorem 2.16) based on typing context reductions: they reflect the communications required by the types in $\Gamma$.

**Definition 2.15.** The typing context reduction $\Gamma \rightarrow \Gamma'$ is:

$$s[p]:S_p, s[q]:S_q \rightarrow s[p]:S_p, s[q]:S_q' \quad \text{if} \quad \begin{cases} \text{un}(S_p) = q \text{ if } \exists i \in I, \forall U_i. S_i \quad k \in I \text{ } u_k \subseteq S_k' \\ \text{un}(S_q) = p \text{ if } \exists i \in I, \forall U_i. S_i' \text{ } u_k \subseteq S_k \end{cases}$$

$$\Gamma, c:U \rightarrow \Gamma', c:U' \text{ if } \Gamma \rightarrow \Gamma', \text{ and } U \subseteq S$$

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

**Theorem 2.16 (Subject reduction).** If $\Theta \cdot \Gamma \vdash P$ and $P \rightarrow P'$, then there exists $\Gamma'$ such that $\Gamma \Rightarrow \Gamma'$ and $\Theta \cdot \Gamma' \vdash P'$.

**Global Types** We conclude this section by discussing global types, that we mentioned in the opening of §2.1 and Remark 2.13.

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

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

$$\mu t. G \mid t \mid \text{end} \quad \text{(recursive type, type variable, termination)}$$
Type \( p \rightarrow q : \{ l(U_i).G_i \}_{i \in I} \) 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 \texttt{end} states the termination of a protocol. We omit the braces \{\ldots\} from interactions when \( I \) is a singleton: e.g., \( a \rightarrow b : l(U_i).G_i \) stands for \( a \rightarrow b : \{ l(U_i).G_i \}_{i \in \{1\}} \).

\begin{example}

The following global type formalises the Game described in §1 and Fig. 1:

\[
G_{\text{Game}} = b \rightarrow c : \text{InfoBC}(\text{string}) \cdot c \rightarrow a : \text{InfoCA}(\text{string}) \cdot a \rightarrow b : \text{InfoAB}(\text{string})
\]

\[
\mu t.a \rightarrow b : \left\{ \begin{array}{c}
\text{Mov1BC}(\text{int}).b \rightarrow c : \text{Mov1BC}(\text{int}) \cdot c \rightarrow a : \left\{ \begin{array}{c}
\text{Mov1CA}(\text{int}).t, \\
\text{Mov2CA}(\text{bool}).t
\end{array} \right\} \\
\text{Mov2BC}(\text{bool}).b \rightarrow c : \text{Mov2BC}(\text{bool}) \cdot c \rightarrow a : \left\{ \begin{array}{c}
\text{Mov1CA}(\text{int}).t, \\
\text{Mov2CA}(\text{bool}).t
\end{array} \right\}
\end{array} \right\}
\]

In MPST theory, a global type \( G \) with roles \( p_i \) \((i \in I)\) is used to project\(^1\) a set of session types \( S_i \) (one per role). E.g., projecting \( G_{\text{Game}} \) in Ex. 2.18 onto \( b \) yields the session type \( S_b \) (see (1)). 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 \textit{projected typing context} of \( G \):

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

Now, from observation (a) above, we can easily define a “strict” version of rule (T-Res) (Fig. 4) in the style of [18, 10], 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 Theorem 2.19 below, as shown e.g. in [4]: 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 (\( \nu s : \Gamma_G \)).

\begin{theorem}[Deadlock freedom]

Let \( \emptyset \cdot \emptyset \vdash P, \text{ where } P = (\nu s : G)_{i \in I} P_i \) and each \( P_i \) only interacts on \( s[p_i] \). Then, \( P \) is deadlock-free: i.e., \( P \rightarrow^* P' \not\vdash \text{ implies } P' \equiv 0. \)
\end{theorem}

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 (Theorem 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, Theorem 2.19) \textit{without} explicit references to global types.

\section{Linear \( \pi \)-Calculus}

The \( \pi \)-calculus is the canonical model for communication and concurrency based on message-passing and channel mobility. It was created towards the end of 1980’s, with the first paper published in 1992 [44], followed by various proposals for types and type systems. In this section we summarise the standard \( \pi \)-calculus with linear types [35]. The contents of this section are standard, and based on [54]; we present new \( \pi \)-calculus-related results in §4.

\begin{definition}
The syntax of \( \pi \)-calculus processes and values is:

\footnote{We use a standard projection with merging [61, 17]: for its definition (not crucial here), see §A.2.}

\begin{enumerate}
\end{enumerate}
A Linear Decomposition of Multiparty Sessions

\[ P, Q := 0 \mid P \mid Q \mid (\nu x)P \quad \text{(inaction, parallel composition, restriction)} \]
\[ *P \mid \pi(v).P \mid x(y).P \quad \text{(process replication, output, input)} \]
\[ \text{case } v \text{ of } \{ l_i(x_i) \vdash P_i \}_{i \in I} \quad \text{(variant destruct)} \]
\[ \text{with } [l_i : x_{i}]_{i \in I} \vdash v \text{ do } P \quad \text{(labelled tuple destruct)} \]
\[ u, v := x, y, w, z \mid l(v) \mid [l_i : v_i]_{i \in I} \quad \text{(name, variant value, labelled tuple value)} \]
\[ \text{false} \mid \text{true} \mid 42 \ldots \quad \text{(base value)} \]

The inaction process \( 0 \), and the parallel composition \( P \mid Q \) are straightforward, and similar to Def. 2.1. The restriction process \( (\nu x)P \) creates a new name \( x \) and binds it with scope \( P \). The replicated process \( \star P \) represents infinite replicas of \( P \), composed in parallel. The output process \( \pi(v).P \) uses the name \( x \) to send a value \( v \), and proceeds as \( P \); the input process \( x(y).P \) uses \( x \) to receive a value that will substitute \( y \) in the continuation \( P \). Process \text{case } v \text{ of } \{ l_i(x_i) \vdash P_i \}_{i \in I} \text{ pattern matches a variant value } v \text{, and if it has label } l_i \text{, substitutes } x_i \text{ and continues as } P_i \text{. Process } \text{with } [l_i : x_{i}]_{i \in I} \vdash v \text{ do } P \text{ destructs a labelled tuple } v \text{, substituting each } x_i \text{ in } P \text{. Values include names, which can be thought of as communication channels names, base values like } \text{false} \text{ or } 42 \text{, variant values } l(v) \text{ and labelled tuples } [l_i : v_i]_{i \in I} \text{. 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:

\[(\text{Rx-Com}) \quad \pi(v).P \mid x(y).Q \Rightarrow P \mid Q \{v/x\} \]
\[(\text{Rx-Case}) \quad \text{case } l_j(v) \text{ of } \{ l_i(x_i) \vdash P_i \}_{i \in I} \Rightarrow P_i \{v/x_i\} \quad (j \in I) \]
\[(\text{Rx-With}) \quad \text{with } [l_i : x_{i}]_{i \in I} \vdash [l_i : v_i]_{i \in I} \text{ do } P \Rightarrow P \{v_i/s_i\}_{i \in I} \]
\[(\text{Rx-Rs}) \quad P \Rightarrow Q \quad \text{implies } (\nu x)P \Rightarrow (\nu x)Q \]
\[(\text{Rx-Par}) \quad P \Rightarrow Q \quad \text{implies } P \mid R \Rightarrow Q \mid R \]
\[(\text{Rx-Struct}) \quad P \equiv P' \land P \Rightarrow Q \land Q' \equiv Q \quad \text{implies } P' \Rightarrow Q' \]

\( (\text{Rx-Com}) \) models communication between output and input on a name \( x \): it reduces to the corresponding continuations, with a value substitution on the receiver process. \( (\text{Rx-Case}) \) says that \text{case} applied on a variant value \( l_j(v) \) reduces to \( P_j \), with \( v \) in place of \( x_j \). \( (\text{Rx-With}) \) says that \text{with} reduces to its continuation \( P \) with \( v_i \) in place of each \( x_i \), for all \( i \in I \). By \( (\text{Rx-Rs}) \), \( (\text{Rx-Par}) \), reductions can happen under restriction and parallel composition. By \( (\text{Rx-Struct}) \), reduction is closed under structural congruence \( \equiv \): its definition is standard (see §A).

\textbf{π-Calculus Typing}  We now summarise the \( \pi \)-calculus types and typing rules.

\textbf{Definition 3.3} (\( \pi \)-types). The syntax of a \( \pi \)-calculus type \( T \) is given by:

\[ T ::= \text{Li}(T) \mid \text{Lo}(T) \mid \text{L}(T) \quad \text{(linear input, linear output, linear connection)} \]
\[ \sharp(T) \mid \bullet \quad \text{(unrestricted connection, no capability)} \]
\[ \langle l_i : T_i \rangle_{i \in I} \mid [l_i : T_i]_{i \in I} \quad \text{(variant, labelled tuple a.k.a. “record”)} \]
\[ \mu \langle T \rangle \mid \text{t} \mid \text{Int} \mid \ldots \quad \text{(recursive type, type variable, base type)} \]

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

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

\textbf{Definition 3.4} (Linear/unrestricted types). The predicate \( \text{lin} \) is inductively defined as:

\[ \text{lin}(0) \quad \text{lin}(\star P) \quad \text{lin}(\pi(v).P) \quad \text{lin}(x(y).P) \]
\[ \text{lin}(\text{case } v \text{ of } \{ l_i(x_i) \vdash P_i \}_{i \in I}) \quad \text{lin}(\text{with } [l_i : x_{i}]_{i \in I} \vdash v \text{ do } P) \]
\[ \text{lin}(u, v) \quad \text{lin}(\text{false}) \quad \text{lin}(\text{true}) \quad \text{lin}(42 \ldots) \]
\[ \text{lin}(\text{with } [l_i : v_{i}]_{i \in I}) \quad \text{lin}(\text{false}) \quad \text{lin}(\text{true}) \quad \text{lin}(42 \ldots) \]
We write \( \text{un}(T) \) iff \( \neg \text{lin}(T) \) (i.e., \( T \) is unrestricted iff \( T \) is not linear).

**Definition 3.5.** Subtyping \( \leq_{\pi} \) for \( \pi \)-types is coinductively defined as:

\[
\begin{align*}
B \leq_B B' & \quad \text{(S-LB)} \\
B \leq_{\pi} B' & \quad \text{(S-END)} \\
T \leq_{\pi} T' & \quad \text{(S-LI)} \\
\text{Lo}(T) \leq_{\pi} \text{Lo}(T') & \quad \text{(S-Lo)} \\
\forall i \in I \quad T_i \leq_{\pi} T'_i & \quad \text{(S-VARIANT)} \\
\text{L}(\langle i, T_i \rangle) \leq_{\pi} \text{L}(\langle i, T'_i \rangle) & \quad \text{(S-ALPHA)} \\
\forall i \in I \quad T'_{\mu T' / i} \leq_{\pi} T' & \quad \text{(S-LTUPLE)} \\
\mu T \leq_{\pi} T' & \quad \text{(S-LMU)}
\end{align*}
\]

Rule (S-LB) says that \( \leq_{\pi} \) includes subtyping \( \leq_B \) on base types. (S-END) relates types without I/O capabilities. Rule (S-LI) (resp. (S-Lo)) says that linear input (resp. output) subtyping is covariant (resp. contravariant) in the carried type. (S-VARIANT) says that subtyping for variant types is covariant in both carried types and number of components. (S-LTUPLE) says that subtyping for labelled tuples, a.k.a. records, is covariant in the carried types\(^2\). Rules (S-LMU)/(S-LMU)\(_{R} \) relate a recursive type \( \mu T \) to \( T' \) iff its unfolding is related to \( T' \).

**Definition 3.6 (Typing context, type combination).** The linear \( \pi \)-calculus typing context \( \Gamma \) is a partial mapping defined as:

\[
\Gamma \coloneqq \emptyset \; | \; \Gamma, x : T
\]

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

\[
\begin{align*}
\text{Li}(T) \lor \text{Lo}(T) & \triangleq \text{Li}(T) \\
\text{Lo}(T) \lor \text{Li}(T) & \triangleq \text{Lo}(T) & T \lor T & \triangleq T & \text{if \( \text{un}(T) \)} \\
(\Gamma_1 \uplus \Gamma_2)(x) & \triangleq \begin{cases} 
\Gamma_1(x) \uplus \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*}
\]

\(^2\) Subtyping on “full” records allows to add/remove entries [54, §7.3]; but here, “record” = “labelled tuple.”
The typing rules for the linear $\pi$-calculus are given in Fig. 5. Typing judgements have two forms: $\Gamma \vdash v : T$ and $\Gamma \vdash P$. (T\textsubscript{\pi-NAME}) says that a name has the type assumed in the typing context; (T\textsubscript{\pi-VAL}) relates base values to their types; both rules require unrestricted typing contexts. (T\textsubscript{\pi-LTYP}) says that a variant value $\lfloor l : v_i \rfloor_{i \in I}$ is of type $\lfloor l : T_i \rfloor_{i \in I}$ if for all $i \in I$, $v_i$ is of type $T_i$. (T\textsubscript{\pi-REC}) is the substitution rule: if $x$ has type $T$ in $\Gamma$, then it also has any subtype of $T$. (T\textsubscript{\pi-NL}) says that $0$ is well typed in every unrestricted typing context. (T\textsubscript{\pi-PAR}) says that the parallel composition of two processes is typed by combining the respective typing contexts. (T\textsubscript{\pi-Res1}) says that the restriction process $(\nu x)P$ is well typed if $P$ is well typed by augmenting the context with $x : \text{Li}(T)$. By applying Def. 3.6 (\textbf{u}), we have $x : \text{Li}(T) = x : \text{Li}(T) \sqcup \text{Lo}(T)$: this implies that $P$ owns both capabilities of linear input/output of $x$. Rule (T\textsubscript{\pi-Res2}) says that the restriction $(\nu x)P$ is well typed if $P$ is well typed and $x$ has no capabilities. (T\textsubscript{\pi-In}) (resp. (T\textsubscript{\pi-Out})) say that the input and output processes are well typed if $x$ is a (possibly standard) 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 $\sqcup$ on the premises. (T\textsubscript{\pi-RPL}) says that a replicated process $*P$ is typed in the same unrestricted context that types $P$. (T\textsubscript{\pi-Case}) says that case $v$ of $\{ l_i(x_i) \triangleright P_i \}_{i \in I}$ is well typed if the guard value $v$ has variant type, and every $P_i$ is typed assuming $x_i : T_i$, for all $i \in I$. (T\textsubscript{\pi-Won}) says that process with $\lfloor l_i : x_i \rfloor_{i \in I} = v \; \text{do} \; P$ is well typed if $v$ is of record type such that for all $i \in I$, each $v_i$ has the same type as $x_i$, i.e., $T_i$.

4 Some Typed $\pi$-Calculus Extensions and Results

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

“Let” binder, narrowing, substitution Fig. 6 shows several auxiliary definitions and typing rules. let $x = v$ in $P$ binds $x$ in $P$, and reduces by replacing $x$ with $v$ in $P$. It is a macro on other $\pi$-calculus constructs: hence, rules (T\textsubscript{\pi-Let})/(T\textsubscript{\pi-LET}) are based on the reduction/typing of its expansion (see §A). Rule (T\textsubscript{\pi-Narrow}) derives from the narrowing lemma [54, 7.2.5]. Rule (T\textsubscript{\pi-MSubst}) represents zero or more applications of the substitution lemma [54, 8.1.4].

Duality and Recursive $\pi$-Types The duality for linear $\pi$-types relates opposite but compatible input/output capabilities. Intuitively, the dual of a Li$(T)$ is Lo$(T)$ (and vice versa) [14]. Note that the carried type $T$ is the same: i.e., dual types can be combined with $\sqcup$ (Def. 3.6), yielding Li$(T)$. However, defining duality for recursive $\pi$-types is not straightforward: what is the dual of $T = \mu L$.Lo$(t)$? Is it maybe $T' = \mu L$.Li$(t)$? Since $\sqcup$ is not defined for $\mu$-types, we can check whether it is defined for the unfoldings of our hypothetical types $T$ and $T'$. Unfortunately, we have unf($T$) = Lo($\mu L$.Lo$(t)$) and unf($T'$) = Li($\mu L$.Li$(t)$): i.e., $\sqcup$ is again
undefined, so $T,T'$ 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 $\pi$-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 $\text{Lo}(T)/\text{Li}(T)$ that can be combined with $\Downarrow$ (since they carry the same $T$). We address this issue by extending the $\pi$-calculus type variables (Def.3.3) with their *dualised* counterpart, denoted with $\bar{T}$. We allow recursive types such as $\mu t.\text{Li}(\bar{T})$ (but not $\mu t\ldots$), and postulate that when unfolding, $\bar{T}$ is substituted by a “dual” type $\mu t.\text{Lo}(t)$, as formalised in Def.4.1 below. Quite interestingly, our approach “mirrors” (on $\pi$-calculus) the “logical duality” for session types [41] (we will discuss it in §8).

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

$$
\begin{align*}
\text{Li}(\bar{T}) &\triangleq \text{Lo}(T) & \bar{\bullet} &\triangleq \bullet & \bar{\langle t \rangle} &\triangleq \bar{t} & \bar{\mu t.T} &\triangleq \mu t.T\{\bar{t}/t\}
\end{align*}
$$

The substitution of $T$ for a type variable $t$ or $\bar{t}$ is: $t\{\bar{t}/t\} \triangleq T$.

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

**Lemma 4.2.** $\text{unf}(\bar{T}) = \text{unf}(T)$.

**Example 4.3.** Let $T = \mu t.\text{Li}((t,\bar{t}))$. Then:

$$
\text{unf}(T) = \text{Li}\left(\left(\mu t.\text{Li}((t,\bar{t})),\mu t.\text{Li}((t,\bar{t}))\right)\right) = \text{Li}\left(\left(\mu t.\text{Li}((t,\bar{t})),\mu t.\text{Lo}((\bar{t},t))\right)\right); \text{ and}
$$

$$
\text{unf}(\bar{T}) = \text{unf}\left(\mu t.\text{Lo}((\bar{t},t))\right) = \text{Lo}\left(\left(\mu t.\text{Li}((t,\bar{t})),\mu t.\text{Lo}((\bar{t},t))\right)\right) = \text{unf}(T)
$$

By adding dualised type variables in Def.3.3, we naturally extend the definition of $\text{fv}(T)$ (treating $\mu t\ldots$ as a binder for both $t$ and $\bar{t}$), the subtyping relation $\leq_{\pi}$ in Def.3.5 (by letting rules $(\text{s.LaL})$ and $(\text{s.LaR})$ use the substitution in Def.4.1) and ultimately the typing system in Def.3.6. This will allow us to 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.: $\bar{T}\{t'/\bar{t}\} = \bar{t'}$; (2) the equivalence $=_{\pi}$ defined as: $\leq_{\pi} \cap \leq_{\pi}^{-1}$ (see Def.C.1).

**Lemma 4.4** (Erase of $\bar{T}$). $\mu t.T =_{\pi} \mu t.T\{t'/\bar{t}/t\}$, for all $t' \notin \text{fv}(T)$.

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

$$
T =_{\pi} \mu t.\text{Li}\left(\left(t,\mu t'.\text{Li}((t,\bar{t}));t'/\bar{t}\right)\right) = \mu t.\text{Li}(t,\mu t'.\text{Lo}(t,t'))
$$

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 $\bar{T}$; i.e., any typing relation with instances of $\bar{T}$ corresponds to a $\bar{T}$-free one. As a consequence, any typing derivation using $\bar{T}$ can be turned into a $\bar{T}$-free one. Summing up: adding dualised variables preserves the standard results of typed $\pi$-calculus.

**Type Combinator $\boxcirc$** Def.4.6 introduces a type combinator that is a “relaxed” version of $\boxplus$ (Def.3.6) allowing for subtyping. We will use it to encode MPST typing contexts (Def.5.6).
The encoding of types

Encoding of Types


\( T_1 \sqcup T_2 \) is that the former combines linear inputs/outputs with the same carried type, while \( \sqcap \) is a more relaxed relation and allows one carried type to be subtype of the other, and (when defined) yields a linear connection allowing transmission of values of both carried types. This is shown in Lemma 4.7 and Ex. 4.8 below.

Lemma 4.7 says that \( T_1 \sqcap T_2 \) (when defined) is a type that, when split using \( \sqcup \), yields linear I/O types that are subtypes of the originating \( T_1, T_2 \).

Example 4.8. Let \( T_1 = \text{Li} (\text{Real}) \), \( T_2 = \text{Lo} (\text{Int}) \), and \( T = T_1 \sqcap T_2 \). We have \( T = \mathbb{L} (\text{Int}) \);

- if we let \( T_1 \sqcup T_2 = T \), then we get either (a) \( T_1 = \text{Li} (\text{Int}) \sqsubseteq_\pi T_1 \) and \( T_2 = \text{Lo} (\text{Int}) \sqsubseteq_\pi T_2 \), or (b) \( T_1 = \text{Lo} (\text{Int}) \sqsubseteq_\pi T_2 \) and \( T_2 = \text{Li} (\text{Int}) \sqsubseteq_\pi T_1 \).

5 Encoding Multiparty Session-\( \pi \) into Linear \( \pi \)-Calculus

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

Encoding of Types

Our goal is to decompose multiparty session channel endpoints into point-to-point \( \pi \)-calculus channels. One intuitive way to achieve this is to encode MPST channel endpoints as labelled tuples, such that each role involved in a session maps to a \( \pi \)-calculus name: i.e., if the labelled tuple has an entry for \( p \), it should map to a name that allows to send/receive messages to/from some other process, which in turn should be interpreting the role of \( p \) in the originating session. This suggests that an encoded MPST channel endpoint must be typed as a \( \pi \)-calculus labelled tuple; and since each name appearing in such tuple is used for communication, it should be typed with a linear input/output type.

Definition 5.1. The encoding of \( S \) into linear \( \pi \)-types is:

\[
[S] \triangleq [p; [S \mid p]]_{p \in S}
\]

where the encoding of the partial projections \( [S \mid p] \) is:

\[
\begin{align*}
\text{\texttt{\oplus}}_{i \in I} \text{\texttt{li}}(U_i).H_i & \triangleq \text{Lo}(\langle \text{\texttt{li}}(U_i).H_i \rangle)_{i \in I} \\
\text{\texttt{\&}}_{i \in I} \text{\texttt{li}}(U_i).H_i & \triangleq \text{Li}(\langle \text{\texttt{li}}(U_i).H_i \rangle)_{i \in I} \\
[B] & \triangleq B & 	ext{\texttt{\texttt{\texttt{\texttt{\end}}}}} & \triangleq \bullet & \text{\texttt{[\mu.t.H]}} & \triangleq \mu t.[H]
\end{align*}
\]

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 encoding of partial projections, in turn, adopts the basic idea of the encoding of binary, non-recursive session types [34, 14]: it is the identity on a base type \( B \), while a terminated channel type \text{\texttt{\texttt{\texttt{\texttt{\texttt{\texttt{\end}}}}}}} becomes \( \bullet \), with no capabilities. Selection \( \text{\texttt{\texttt{\texttt{\texttt{\oplus}}}}}_{i \in I} \text{\texttt{li}}(U_i).H_i \) and branching \( \text{\texttt{\texttt{\texttt{\texttt{\&}}}}}_{i \in I} \text{\texttt{li}}(U_i).H_i \) are encoded as linear output and input types, respectively, adopting a continuation-passing style (CPS). In both cases, the carried types are variants: \( \langle \text{\texttt{li}}(U_i).H_i \rangle \) for select and \( \langle \text{\texttt{li}}(U_i).H_i \rangle \) for \( \text{\texttt{\&}} \).
for branch, with the same labels as the originating partial projections. Such variants carry tuples \([\langle U_1, [H_1]\rangle, [U_2, [H_2]\rangle]\): the first element is the encoded payload type, and the second (i.e., the encoding of the continuation \(H_i\)) is the type of a continuation name: it is sent together with the encoded payload, and will be used to send/receive the next message (unless \(H_i\) is end). Note that selection sends the dual of \([H_i]\): this is because the sender is expected to 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 a type variable and a recursive type are homomorphic.

Notice that by encoding session types as labelled tuples, we untangle the order of the interactions among different roles. This order will be, however, recovered by the encoding of processes, presented later on.

\[ \textbf{Example 5.2.} \text{Consider the session type } S \triangleq p \dddot{H}_1(\text{Int}).q \dddot{I}_2(S').\text{end, where } S' \triangleq r \dddot{I}_3(\text{Bool}).q \dddot{I}_4(\text{String}).\text{end. By Def. 5.1, the encoding of } S \text{ is:} \]

\[
[S] = \langle p[S \gg p], q[S \gg q] \rangle = \langle p[H_1(\text{Int})], q[I_2(S')] \rangle = \langle p: \text{Lo}(l_1(\text{Int}, \bullet)), q: \text{Li}(l_2(\langle r: \text{Lo}(l_3(\text{Bool}, \bullet)), q: \text{Li}(l_4(\langle r: \text{Lo}(l_3(\text{String}, \bullet)), \bullet)) \rangle) \rangle \rangle
\]

\[ \text{Recursion, Continuations and Duality } \]

We now point out a subtle (but crucial) difference between Def. 5.1 and the encoding of binary, non-recursive session types in [14]. When encoding partial selections, our continuation type is the dual of the encoding of \(H\), i.e., \([H]\); in [14], instead, it is the encoding of the dual of \(H\), i.e., \([\ddot{H}]\). This difference is irrelevant for non-recursive types (Ex. 5.2); but for recursive types, using \([\ddot{H}]\) would yield the wrong continuations. Using \([\ddot{H}]\), instead, gives the expected result, by generating dualised recursion variables (cf. Def. 4.1). We explain it in Ex. 5.3 below.

\[ \textbf{Example 5.3.} \text{Let } H = \mu t.!!(\text{Bool}).t. \text{ By Def. 5.1, we have:} \]

\[
[H] = \langle \mu t.!!(\text{Bool}).t \rangle = \mu t.\text{Lo}(l_\downarrow(\langle r: \text{Lo}(l_3(\text{Bool}, \bullet)), q: \text{Li}(l_4(\langle r: \text{Lo}(l_3(\text{String}, \bullet)), \bullet)) \rangle) \rangle)
\]

Let us now unfold the encoding of \(H\). By Def. 4.1, we have:

\[
\text{unf}(\langle H \rangle) = \text{unf}(\langle \mu t.\text{Lo}(l_\downarrow(\langle r: \text{Lo}(l_3(\text{Bool}, \bullet)), q: \text{Li}(l_4(\langle r: \text{Lo}(l_3(\text{String}, \bullet)), \bullet)) \rangle) \rangle) = \text{Lo}(l_\downarrow(\langle r: \text{Lo}(l_3(\text{Bool}, \bullet)), q: \text{Li}(l_4(\langle r: \text{Lo}(l_3(\text{String}, \bullet)), \bullet)) \rangle) \rangle)
\]

This is what we want: since \(H\) requires a recursive output of \text{Booleans}, its encoding should output a \text{Boolean}, together with a recursive input name as continuation. Hence, the recipient will receive the first \text{Boolean} together with a continuation name, whose type mandates to recursively input more \text{Booleans}. If encoding continuations as in [14], instead, we would have:

\[
[H] = \nu t.\text{Lo}(l_\downarrow(\langle r: \text{Lo}(l_3(\text{Bool}, \bullet)), q: \text{Li}(l_4(\langle r: \text{Lo}(l_3(\text{String}, \bullet)), \bullet)) \rangle) \rangle)
\]

which is wrong: the recipient is required to recursively output \text{Booleans}. This wrong encoding would also prevent us from obtaining Theorem 6.1 later on.

\[ \text{Encoding of Typing Contexts } \]

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.

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

\[
\begin{align*}
\langle \emptyset \rangle & \triangleq \emptyset \\
\langle \Theta, X:U \rangle & \triangleq \langle \emptyset \rangle \cup \{X:U\} \\
\langle \Theta, \Gamma \rangle & \triangleq \langle \emptyset \rangle \cup \{\Theta, \Gamma\} \\
\{c:U\} & \triangleq \{c:U\} \\
\{s[p]\} & \triangleq \{s[p]\} \\
\{z_{s[p]}\} & \triangleq \{z_{s[p]}\} \\
\{X:U\} & \triangleq \{X:U\} \\
\{X:U_1, \ldots, U_n\} & \triangleq \{X:U_1, \ldots, U_n\}
\end{align*}
\]
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 typing context composition \(\Gamma_1 \circ \Gamma_2\) (Def. 2.11) is encoded using \(\psi\) (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 always linearly-typed.

**Encoding Typing Judgements: Overview** 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 \([P]_{\Theta \cdot \Gamma_1 + \Gamma_1}\) and \([Q]_{\Theta \cdot \Gamma_2}\) are the encoded processes yielded respectively by \([\Theta \cdot \Gamma_1 + P]\) and \([\Theta \cdot \Gamma_2 + Q]\): they exist because such typing judgements hold, by inversion of (T-Par) (Fig. 4). Similar uses of sub-processes encoded w.r.t. their typing occur in the other cases. **Process declaration** \(\text{def } X(\bar{x}: U) = P \text{ in } Q\) is encoded as a replicated \(\pi\)-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 \(\pi\)-calculus, process calls are in tail position).

**Selection** on \([p]\) is encoded using information from the session typing context: the fact that \(c\) has type \(S = p \oplus l(U).S'\) — 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**), an 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 \(l([r], z)\), 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 \(p \not\in S'\), then \(z\) is discarded, since it has type \([S'\'p] = [\text{end}] = \bullet\). After **let**, the encoding continues as \([P]\).

Symmetrically, **branching** on \([p]\) is also encoded using information from the typing context, i.e., that \(c\) has type \(S = p \& \in I \otimes l(U_i).S_i'\) — 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 **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 **let**) and the process continues as \([P_i]\). The rebinding of \([c]\) depends on \(l_i\) and the continuation type \(S_i'\); the “new” \([c]\) is a record with 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 \(S_i'\) still involves \(p\) (otherwise, if \(p \not\in S_i'\), then \(z\) has type \(\bullet\) and is discarded).

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

\[
\Gamma, s[p]: S, s[q]: S' \vdash s[p] \& [l(x).P] \mid s[q] \& [l(v) \oplus (l(v)).Q] \rightarrow P\{v/x\} \mid Q
\]
\[ [\Gamma \vdash 0] \triangleq [\Gamma] \vdash 0 \]
\[ [\Theta \cdot \Gamma_1 \circ \Gamma_2 \vdash P \mid Q] \triangleq [\Theta \cdot \Gamma_1 \circ \Gamma_2] \vdash [P]_{\Theta \circ \Gamma_1} \mid [Q]_{\Theta \circ \Gamma_1} \]
\[ [\Theta \cdot \Gamma \vdash \text{def} \ X(\vec{x} : \vec{U}) \ P \ \text{in} \ Q] \triangleq [\Theta \cdot \Gamma] \vdash (\nu [X]) \left( \forall \ (\vec{x} : \vec{U}) \cdot \text{with} \ (x_i)_{i \in \{1..n\}} = \vec{z} \ 	ext{do} \ [P]_{\Theta \cdot \Gamma \circ \vec{x} \ U \ \vec{U}} \right) \mid [Q]_{\Theta \cdot \Gamma \circ \vec{x} \ U \ \vec{U}} \]

\[ \text{Definition 5.5.} \]

The definition of typing judgements. Here, \([P]_{\Theta \cdot \Gamma} = Q\) iff \(\Theta \cdot \Gamma \vdash P = [\Theta \cdot \Gamma] \vdash Q\).

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

\[
\text{with} \ (x : z)_{x \in S} = [s[p]] \ 	ext{do} \ldots \ 	ext{with} \ (x : z)_{x \in S'} = [s[q]] \ 	ext{do} \ldots
\]

and the reason is that \([s[p]], [s[q]]\) are “just” record-typed names (respectively \(z_{s[p]}, z_{s[q]}\), as per Def. 5.4), whereas with-prefixes only reduce when applied to 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 \([\nu_1]\), such records must be suitably defined: we must ensure that the entries for \(p \) (in one record) and \(p \) (in the other) map to the 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 \(\pi\)-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., \(\pi\)-calculus names), connecting pairs of roles. We can formalise “connections” as follows.

**Definition 5.5.** The connections of \(s\) in \(\Gamma\) are: \(\text{conn}(s, \Gamma) \triangleq \{ (p, q) \mid s[p] : S_p \in \Gamma \land q \in S_q \} \)

Intuitively, two roles \(p, q\) are connected by \(s\) in \(\Gamma\) if \(p\) occurs in the type \(\Gamma(s[q])\) (but \(q\) 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, composed by \(\pi\)-calculus names; correspondingly, such names must be typed, i.e., appear in the typing context: this is addressed in Def. 5.6.

**Definition 5.6 (Reification and decomposition of MPST contexts).** The reification of a session typing context \(\Gamma_5^s\) is the substitution:
Definition 5.7 (Encoding). The encoding of session typing judgements is given in Fig. 7. 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. 18. The process $(\nu s: \Gamma') P$ is encoded by generating one delimitation for each $z(s, p, q)$ whenever $\{p, q\}$ 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 $z(s, p, q)$.

Example 5.8. Consider (2). If we delimit $s$ and encode the resulting process, we obtain a $\pi$-calculus process based on (3), enclosed by the delimitations yielded by $[[\nu s]]$, and the substitution $\sigma(s[p]: S, s[q]: S', \ldots)$. 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 withs.

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

\[
(\nu s_1, s_2, s_3)(Q \mid P_s \mid P_b \mid P_r)
\]

where

\[
Q = (\nu s)(s_a[q][p] \oplus (s_a[x]) \mid s_b[q][p] \oplus (s_b[y]) \mid s_c[q][p] \oplus (s_c[z]))
\]

Its encoding is the following process, with $s$ decomposed in 3 linear channels (see also Fig. 8):

\[
(\nu z(s_a, q_1), z(s_b, p_1), z(s_c, p_2))(Q \mid P_s \mid P_b \mid P_r)
\]

where

\[
Q = (\nu z(s_a, b), z(s_b, c), z(s_c, a))(s_a[q][p] \oplus (s_a[x]) \mid s_b[q][p] \oplus (s_b[y]) \mid s_c[q][p] \oplus (s_c[z]))
\]
6 Properties of the Encoding

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

Encoding of Types Theorem 6.1 below says that our encoding commutes the duality between session types (Def. 2.8) and π-types (Def. 4.1); Theorem 6.2 shows that it also preserves typing.

- **Theorem 6.1** (Encoding preserves duality). \( [H] = [\bar{H}] \).

- **Theorem 6.2** (Encoding preserves subtyping). If \( S \subseteq S' \), then \( [S] \leq [S'] \).

Encoding of Typing Judgements Theorem 6.3 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.

- **Theorem 6.3** (Correctness of encoding). \( \Gamma \vdash v : U \) implies \( [\Gamma] \vdash [v] : [U] \), \( \Theta \vdash X : \bar{U} \) implies \( [\Theta] \vdash [X] : [\bar{U}] \), and \( \Theta \vdash P \) implies \( \Theta \vdash \Gamma \vdash P \).

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

\[
\begin{align*}
\frac{(T\text{-Sub})}{\Theta \cdot \Gamma, c : S \vdash P, S' \subseteq S} \quad &\quad \text{implies} \quad [\Theta \cdot \Gamma, c : S \vdash P], [S'] \leq [S] \quad \text{(T-Narrow)}
\end{align*}
\]

that holds by the induction hypothesis and Theorem 6.2. 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 \( [P]_{\Theta \vdash \Gamma, \Gamma'} \); hence, to prove the theorem, we need to type such names, i.e., find a context that types \( \lbrack P \rbrack_{\Theta \vdash \Gamma, \Gamma'} \sigma(\Gamma') \). This is where \( \delta(\Gamma') \) and (T-\text{Reify}) (Def. 5.6) come into play, as we now explain.

More on Def. 5.6 and decomposition By Def. 5.6, the typing context \( \delta(\Gamma_5) \), when defined, has an entry \( z_{\langle s, p, q \rangle} \) for each \( s[p] : S_p \in \Gamma_5 \) and \( q \in S_p \). Such entries are used to type the records yielded by \( \sigma(\Gamma_5) \). The type of \( z_{\langle s, p, q \rangle} \) is based on the encoding of the unfolded partial projection \( S_p[q] \), that can be either \( \bullet \) or \( \text{Li}(T)/\text{Lo}(T) \) (for some \( T \)). Note that if there is also some \( s[q] : S_q \in \Gamma_5 \) with \( q \neq p \), the type of \( z_{\langle s, q, p \rangle} \) (when defined) is \( \lbrack \text{unf}(S_q[q]) \rbrack \cap \lbrack \text{unf}(S_p[p]) \rbrack \). This creates a deep correspondence between the consistency of \( \Gamma_5 \) and the existence of \( \delta(\Gamma_5) \), as shown in Theorem 6.4 below: it says that the precondition for type safety in MPSTs (i.e., the consistency of \( \Gamma_5 \)) can be precisely expressed in \( \pi \)-calculus, and this is captured by the linear decomposition at the roots of our encoding.

- **Theorem 6.4** (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 (T-\text{Reify}), that uses \( \delta(\Gamma'_5) \) to type a process on which \( \sigma(\Gamma'_5) \) has been applied. We explain the rule with a slight simplification. If we have \( \Gamma_5 = \{ s[p] : S_p \}_{p \in P} \), then:

\[
\delta(\Gamma'_5) = \bigcap_{p \in P} \{ z_{\langle s, p, q \rangle} : [\text{unf}(S_p[q])] \}_{s[p] : S_p} \quad \text{and} \quad \sigma(\Gamma'_5) = \{ [z_{\langle s, p, q \rangle}]_{s[p] : S_p} \}_{p \in P}
\]

(Note: \( \delta(\Gamma'_5) \) is defined iff \( \Gamma'_5 \) is consistent, by Theorem 6.4). Now, take a set of types \( \{ T_{\langle s, p, q \rangle} \}_{s[p] : S_p} \) such that \( \bigcap_{p \in P} \{ z_{\langle s, p, q \rangle} : T_{\langle s, p, q \rangle} \}_{s[p] : S_p} = \delta(\Gamma'_5) \) (note \( T_{\langle s, p, q \rangle} \), \( T_{\langle s, q, p \rangle} \) are distinct) and assume the premise of (T-\text{Reify}). The following derivation holds:
A Linear Decomposition of Multiparty Sessions

In particular, the assumptions $T_{(s,p,q)} \leq \Delta [S_p \vdash \varphi]$ hold by Lemma 4.7, since each $T_{(s,p,q)}$ has been obtained by splitting $\Delta (\Gamma_s')$ (that combines types with $\Theta$) using $\Theta$. The equivalence in the conclusion holds since $\text{dom}((\Theta \cdot \Gamma_s')) \cap \text{dom}(\Delta (\Gamma_s')) = \emptyset$. Summing up: if the $(\text{T-Redf})$ premise holds, then the above derivation holds, which proves the conclusion of $(\text{T-Redf})$.

Now, we can finish the proof of Theorem 6.3 for the case $\Theta \cdot \Gamma \vdash (\nu s: \Gamma') P$. Assuming that the judgement holds, we also have $\Theta \cdot \Gamma, \Gamma' \vdash P$ and $\Gamma'$ complete (by the premise of $(\text{T-Res})$, Fig. 4): hence, $\Gamma'$ is consistent, and $\Delta (\Gamma')$ is defined (by Theorem 6.4). Assuming that $[\Theta \cdot \Gamma, \Gamma' \vdash P]$ holds (by the induction hypothesis), we obtain:

$$
\begin{align*}
[\Theta \cdot \Gamma], [\Gamma'] \vdash P_{\Theta \cdot \Gamma, \Gamma'}.
\end{align*}
$$

Encoding and Reduction

One usual way to assess that an encoding is “behaviourally correct” (i.e., a process and its encoding behave “in the same way”) consists in proving operational correspondence. Roughly, it says that the encoding is: (1) complete, i.e., any reduction of the original process is simulated by its encoding; and (2) sound, i.e., any reduction of the encoded process matches some reduction of the original process. This is formalised in Theorem 6.6, where $\Rightarrow_{\text{with}}$ denotes a reduction induced by $(\text{Rf-Sub})$ (Def. 3.2).

**Proposition 6.5.** If $\Theta \cdot \Gamma \vdash P$ and $\Theta' \cdot \Gamma' \vdash P$, then $[P]_{\Theta \cdot \Gamma} = [P]_{\Theta' \cdot \Gamma'}$. 

**Theorem 6.6 (Operational correspondence).** If $\emptyset \cdot \emptyset \vdash P$, then:

1. (Completeness) $P \Rightarrow P'$ implies $\exists x. P''$ such that $[P] \Rightarrow (\nu x)P''$ and $P'' = [P']$;

2. (Soundness) $[P] \Rightarrow P'$ implies $\exists x. P'' P' \text{ s.t. } P \Rightarrow (\nu x)P'' P' \Rightarrow P'$ and $[P'] \Rightarrow_{\text{with}} x : P''$.

The statement of Theorem 6.6 is standard [22, §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 $[P']_{\emptyset \cdot \emptyset}$, which implies $\emptyset \cdot \emptyset \vdash P'$. 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 $\Rightarrow_{\text{with}}$, consider the following MPST process:

$$
\emptyset \cdot \Gamma, s[p] : S \vdash s[p] , \text{[l(x).P]} \quad \text{(the process is stuck)}
$$

If we encode it (and apply $\sigma(\Gamma, s[p] : S)$ as per Ex. 5.8), we get a $\tau$-calculus process that gets stuck, too — but only after firing one internal with-reduction:
Then, $J (2)$ treat the encoding of continuation-passing-style

\[ \exists \text{implies safety and distribution} \]

We can now show how our encoding directly guides the implementation of a toolchain for

As said above, $\pi$ with

This happens whenever a process is deadlocked, and even if it is closed (as in item 2 of Theorem 6.6). This is because in Fig. 7, the “atomic” branch/select operations of MPSTs are encoded with multiple steps in linear $\pi$-calculus: first with for deconstructing the tuple of linear channels, and then input/output. In general, if an MPST process is stuck, its encoding fires one with for each stuck branch/select, then blocks on a corresponding input/output.

Theorem 6.6 yields an immediate corollary pertaining deadlock freedom:

**Corollary 6.7.** $P$ is deadlock-free if and only if $[P]$ is deadlock-free, i.e.: $[P] \rightarrow P' \not\leftrightarrow$ implies $\exists Q \equiv 0$ such that $P' = [[Q]]$.

As a consequence, our encoding allows to transfer Theorem 2.19 to $\pi$-calculus processes.

**Corollary 6.8.** Let $\cdot \circ \vdash P$, where $P \equiv (\mu s: G)_{i \in I} P_i$ 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, including 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 [56]. 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 $\pi$-calculus type $\tau$-typed message: i.e., they represent channel endpoints with $\pi$-calculus types $\text{Lo}(\tau)$ and $\text{Li}(\tau)$ (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 [39]. Notably, lchannels promotes session type-safety through a continuation-passing-style encoding of binary session types [56] 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 [56, 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 Theorem 6.3 and Def 5.1 and Theorem 6.4, we know that the game player session type $S_b$ in our example (see (1)) provides the type safety guarantees of a tuple of (linear) channels, whose types are given by the encoded partial projections of $S_b$ onto $a,c$ (Def. 2.9). This suggests that, using lchannels, the delegation of an $S_b$-typed channel (as in §1) could be rendered in Scala as:

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

i.e., as a linear input channel carrying a message of type PlayB, whose payload has type $S_b$; $S_b$, in turn, is a Scala case class, which can be seen as a labelled tuple, that maps $a,c$ to I/O channels—whose types derive from $[S_b][a]$ and $[S_b][c]$ (in fact, they carry messages of type InfoAB/InfoBC). In this view, $S_b$ is our Scala rendering of the encoded session type $[S_b]$. As said above, lchannels allows to send channels remotely—hence, also allows to remotely send tuples of channels (e.g., instances of $S_b$); thus, with this simple approach, we obtain 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 type encoding $[S_b]$, the $S_b$ 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, $S_b$ so far, does not convey ordering of communications across channels, i.e., the order in which its fields, $a$ and $c$, should be used. (Indeed, $[S_b]$ may type $\pi$-processes using its separate channels in any order while preserving basic safety.) To recover the “desired” ordering of communications, and implement it correctly, we can refine our classes so that:

1. each multiparty channel class (e.g., $S_b$) exposes a send() or receive() method, according to the I/O action expected by the multiparty type ($S_b$);
2. the implementation of such method uses the binary channels as per our process encoding.

E.g., consider again $S_b$ and $S_a$. $S_b$ requires to send towards $c$, so $S_b$ could provide the API:

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

Now, $S_b$.send() behaves exactly as our process encoding in Fig.7 (case for selection $\oplus$): it picks the correct channel from the tuple (in this case, $c$), creates a new tuple $S'_b$ 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 $S'_b$ should be similar, with a receive() method that uses $a$ for input (by following the encoding of $\&$). This way, a programmer is correctly led to write, e.g.,

```scala
val x = s.send(...).receive() // using method call chaining
```

whereas attempting, e.g.,

```scala
s.receive()
```

is rejected by the Scala compiler (method undefined). These send()/receive() APIs are mechanical, and can be automatically generated: we did it by extending Scribble.

Scribble-Scala Toolchain

Scribble is a practical MPST-based language and tool for describing global protocols [57, 62]. 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 allows protocols with the syntax in Fig.9 (left), by augmenting Scribble with a projection operator $\Psi$; then, it computes the projections/encodings explained in §5, and automates the Scala API generation as outlined above (producing, e.g., the $S_b, S'_b, \ldots$ classes and their send/receive methods). This approach reminds the Java API generation in [27] — but we follow a formal foundation and target the type-safe binary channels provided by lchannels (that, as shown above, takes care of most irksome aspects — e.g., delegation). As a result, the $P_b$ 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.

Our implementation is Open Source, and is available in [55].

8 Conclusion and Related Works

We presented the first encoding of a full-fledged multiparty session $\pi$-calculus into standard $\pi$-calculus (§5), and used it as the foundation of the first implementation of multiparty sessions (based on Scala API generation) with support for distributed multiparty delegation, on top of existing libraries (§7). We proved that a consistent session typing context is characterised by a decomposition into linear $\pi$-calculus types (Theorem 6.4): i.e., the type safety property of MPSTs is precisely captured by standard $\pi$-calculus. We encode types by preserving duality and subtyping (Theorems 6.1 and 6.2); our encoding of processes is type-preserving, and operationally sound and complete (Theorems 6.3 and 6.6); hence, our encoding preserves the type-safety and deadlock-freedom properties of MPST (Cor.6.8). These results ensure the
global protocol ClientA(role p, role q) {
    PlayA(Game@a) from q to p; // Delegation payload
}

global protocol ClientB(role p, role q) {
    PlayB(Game@b) from q to p;
}

global protocol ClientC(role p, role q) {
    PlayC(Game@c) from q to p;
}

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

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

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

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

Figure 9 Game example (from §1). Left: Scribble protocols for client/server setup sessions, and main Game (matching Ex. 2.18). Right: Scala client for player b, using Scribble-generated APIs, and mimicking the processes in Ex. 2.2 (for a more natural implementation on the same API, see §A.5).

Implementations of Session Types (for Mainstream Languages) We mentioned the implementations of binary sessions for a range of “mainstream” languages in §1. Notably, [52, 30, 31, 40, 48, 56, 51] sought to realise benefits from session types in the native host language, without language extensions, to avoid hindering their use in practice. To do so, one approach (employed e.g. in [56, 51]) is the combination of static typing of I/O actions on channels, and run-time checking of linear channel usage. We adopt this idea in our implementation (§7). The Haskell-based works, instead, exploit its richer typing facilities to statically enforce linearity—but incur various expressiveness/usability compromises according to the particular strategy for embedding session types.

By contrast, implementations of multiparty sessions are, to date, limited, in part due to the intricacies of the multiparty theory (e.g., the interplay between projections, mergability and consistency), and practical issues (e.g., realising the multiparty session abstraction over binary transports, including distributed delegation), as discussed in §1. [27] proposes MPST-based API generation for Java based on communicating FSMs and has no formalisation, unlike our implementation—which follows directly from our formal encoding. [58] was the first implementation of MPST, based on extending Java with special-purpose session primitives. [16, 19] developed MPST-influenced networking APIs in Python and Erlang, respectively; [46] implemented recovery strategies in Erlang (based on Scribble). [16, 19, 46] focus on purely dynamic MPST verification by run-time monitoring. [47, 45] extended [16] with actors and timed specifications, respectively. [43] uses a dependent MPST theory for verifications of MPI programs. Crucially, none of these MPST-based implementations support delegation (nor merging of choice projections, as needed by our running Game example—cf. Ex. 2.14).

Encodings of Session Types and Processes [15] encodes binary session π-calculus into an augmented π-calculus with branch/select constructs. [14], by following [34], and [20] encode non-recursive, binary session π-calculus, respectively into linear π-calculus, and the Generic Type System for π-calculus [29], and prove correctness w.r.t. typing and reduction. All the
above works investigate binary and (except [15]) non-recursive session types, while in this paper we study the encoding of multiparty session types, subsuming binary ones; and unlike [15], we target standard π-calculus. We encode branching/selection using variants as in [14, 12], but our treatment of recursion, and the rest of the MPST theory, is novel.

The only works studying an encoding of multiparty sessions into binary sessions are [8, 7]: they adopt an orchestrated approach, by adding centralised medium/arbiter processes. Moreover, they target session calculi and not π-calculus, with a wider gap towards implementation. In [49] a restricted class of global types is used 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, thus requiring our careful treatment in §4. [6] and [1] noticed that the standard duality in [25] does not commute with the unfolding of recursion when type variables occur as payload, e.g., \( \mu t.\text{!}t.\text{end} \). To solve this issue, [6, 1] define a new notion of duality, called complement in [1], that is used in the encoding of recursive binary session types into linear π-types [12]. Unfortunately, [2] later noticed that even complement does not commute, e.g., when unfolding \( \mu t.\mu t'.\text{!}t.\text{t}' \). As said in §4, to encode recursive session types we encounter similar issues in the π-types. The reason seems quite natural: in π-calculus, types do not distinguish between “payload” and “continuation”, and, in the case of recursive linear inputs/outputs, type variables necessarily occur as “payload”, e.g. \( \mu t.\text{Lo}(t) \). Since, in the light of [2], we could not adopt the approach of [12], we proposed a solution similar to [41]: introduce dualised type variables \( \overline{t} \). [41] also sketches a property similar to our Lemma 4.4. The main difference is that, we add dualised variables to π-types (while [41] adds \( \overline{t} \) to session types). An alternative approach is given in [56]: recursive session types are encoded as non-recursive linear I/O types with recursive payloads. This avoids dualised variables (e.g., \( \text{Lo}(\mu t.\text{Li}(t)) \) instead of \( \mu t.\text{Lo}(\overline{t}) \)), but at the price of complicating Def.5.1. Most importantly, [56] tackles only the encoding of recursive types and 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., [24, 23] (capabilities) could allow to ensure that a channel endpoint is not used after being sent; [53, 59] (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 [52, 30, 31, 40, 48, 51].

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., by leveraging Cor.6.7, we could now study deadlock-freedom of processes with interleaved multiparty sessions (studied in [3, 9, 11]) by applying π-calculus deadlock detection methods to their encodings [36, 33, 60]. 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 [38, 37] and π-calculus.
References


Philipp Haller and Alexander Loiko. Lacasa: lightweight affinity and object capabilities in Scala. In OOPSLA.


Naoki Kobayashi. Type systems for concurrent programs. Extended version of [32], Tohoku University, 2007.


Luca Padovani. Deadlock and lock freedom in the linear π-calculus. online version of [50], January 2014. URL: https://hal.inria.fr/hal-00932356.


Luca Padovani. Fuse - a simple library implementation of binary sessions, 2016. URL: http://www.di.unito.it/~padovani/Software/FuSe/FuSe.html.


A Linear Decomposition of Multiparty Sessions

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


A Linear Decomposition of Multiparty Sessions

\[ P \parallel Q \equiv Q \parallel P \parallel (P \mid Q) \parallel R \equiv P \parallel (Q \parallel R) \parallel P \parallel 0 \equiv P \parallel (\nu s)0 \equiv 0 \]

The projection of global types onto session types gives branch and select types, as well as recursion.

The operational semantics of multiparty session processes is based on the notion of structural congruence \( \equiv \), given in Fig. 12. We write "s \( \notin \text{fc}(P) \)" to mean that there does not exist a \( p \) such that \( s[p] \in \text{fc}(P) \). We use \( \text{fpv}(D) \) to denote the set of free variables in \( D \). We use \( \text{dpv}(D) \) to denote the set of process variables declared in \( D \), and \( \text{fpv}(P) \) for the set of process variables which occur free in \( P \).

Most of the rules of structural congruence are standard. The first two lines in Fig. 12 show the commutativity and associativity of the relation \( w.r.t. \) parallel composition and restriction and \( 0 \) used as the neutral element \( w.r.t. \) parallel composition, restriction and process definition. The last three lines in Fig. 12 describe how a process definition can be rearranged \( w.r.t. \) restriction, parallel composition and process definition, respectively. These rules make use of well-formedness criteria on the free names and variables in the process definition, which are given as side conditions.

A.2 Global Types

We now provide the formal definition of projection of a global type onto a role.

\[ \langle p \rightarrow p' \rangle : \{ l_i(U_i), G_i \}_{i \in I} \mid q \triangleq \begin{cases} \langle p' \oplus_{i \in I} \forall_i(U_i), (G_i \mid q) \rangle & \text{if } q = p' \\ \langle p' \&_{i \in I} \exists_i(U_i), (G_i \mid q) \rangle & \text{if } q = p' \\ \bigcap_{i \in I} (G_i \mid q) & \text{if } p \neq q \neq p' \end{cases} \]

| \langle \mu t(G) \mid q \rangle \triangleq \begin{cases} \mu t.(G \mid q) & \text{if } G \mid q \neq t' \langle \forall t' \rangle \\ \text{end} & \text{otherwise} \end{cases} | \mid t \neq t \quad \text{end} \mid q \triangleq \text{end} \]

where \( \cap \) is the merge operator on session types, defined as:

\[ p \&_{i \in I} ?l_i(U_i).S_i \cap p \&_{i \in J} ?l_j(U_j).S_j' \triangleq \left( p \&_{i \in I \cap J} ?l_i(U_i).S_i \cap p \&_{i \in I \cap J} ?l_j(U_j).S_j \right) \cap p \oplus_{i \in I \cup J} ?l_i(U_i).S_i \]

The projection of global types onto session types gives branch and select types, as well as recursion and termination, which are the multiparty session types given in Def. 2.5. The intuition behind it follows the same lines as Def. 2.9.

The merge operation [61, 17], as for partial projections, makes local projections defined in more cases.
A.3 Structural Congruence for Standard $\pi$-Calculus

In order to complete the operational semantics for the $\pi$-calculus, we need the structural congruence relation, $\equiv$; it is defined as the smallest congruence relation on processes that satisfies the axioms given in Fig. 11.

$$
P | Q \equiv Q | P \\
(P | Q) | R \equiv P | (Q | R) \\
P | 0 \equiv P \\
(\nu x) 0 \equiv 0 \\
(\nu x)(\nu y) P \equiv (\nu y)(\nu x) P \\
(\nu x)P | Q \equiv (\nu x)(P | Q) \quad (x \notin \text{fv}(Q)) \\
*P \equiv P | *P
$$

Figure 11 Structural congruence for the standard $\pi$-calculus.

The first three axioms say that the parallel composition of processes is commutative, associative and uses process $0$ as the neutral element. The next three axioms involve restriction: the first of the sequence is used to collect vacuous restrictions, by saying that restriction can be removed from the terminated process, the second says that restriction is commutative and the third is the main one, scope extrusion, saying that the scope of a restriction can be extended to other parallel processes provided that no free names are captured. The last axiom states that replication can be “decomposed” into a parallel composition of a copy of the process itself and the persistent replicated process.

A.4 “Let” binder reduction and typing

The “let” binder is just a macro based on standard $\pi$-calculus constructs. Hence, its reduction and typing follow the expansion of its definition (Fig. 6):

$$
\text{let } x = v \text{ in } P = (\nu z)(\tau(v),0 | z(x),P) \\
\quad \rightarrow (\nu z)(0 | P\{v/x\}) \equiv (\nu z)0 | P\{v/x\} \equiv P\{v/x\}
$$

$$
\Gamma_1 \vdash v : T \quad \Gamma_2, x : T \vdash P \\
\Gamma_1, z : \text{Lo}(T) \vdash \tau(v),0 \\
\Gamma_2, z : \text{Li}(T) \vdash z(x),P \\
\Gamma_1 \uplus \Gamma_2 \vdash \text{let } x = v \text{ in } P = (\nu z)(\tau(v),0 | z(x),P)
$$
A.5 Multiparty API Generation for Scala

The following code shows an alternative (and more natural) implementation of the b-playing game client in Fig. 9 (right); albeit using the same Scribble-generated APIs, it does not try to mimic the processes in Ex. 2.2.

```scala
def client(s: In[binary.PlayB]) = {
  // Wrap binary chan in multiparty session obj
  val c = MPPlayB(s)
  // Receive multiparty game channel
  val g = c.receive().p
  // Send info to C; wait info from a
  val i = g.send(InfoBC(...)).receive()
  loop(i.cont) // Game loop
}

def loop(g: MPMov1ABOrMov2AB): Unit = {
  g.receive() match {
    case Mov1AB(p, cont) => {
      // cont only allows to send Mov1BC
      val g2 = cont.send(Mov1BC(p))
      loop(g2) // Keep playing
    }
    case Mov2AB(p, cont) => {
      // cont only allows to send Mov2BC
      val g2 = cont.send(Mov2BC(p))
      loop(g2) // Keep playing
    }
  }
} // If case Mov1AB or Mov2AB is missing: compiler warn
```
B. Multiparty Session Types

B.1 Structural Congruence for Multiparty Session π-Calculus

The operational semantics of multiparty session processes is based on the notion of structural congruence ≡, given in Fig. 12. We write "s \not\in Jc(P)" to mean that there does not exist a p such that s[p] \in Jc(P). We use fv(D) to denote the set of free variables in D. We use dvp(D) to denote the set of process variables declared in D, and fpv(D) for the set of process variables which occur free in P.

Most of the rules of structural congruence are standard. The first two lines in Fig. 12 show the commutativity and associativity of the relation w.r.t. parallel composition and restriction and 0 used as the neutral element w.r.t. parallel composition, restriction and process definition. The last three lines in Fig. 12 describe how a process definition can be rearranged w.r.t. parallel composition and process definition, respectively. These rules make use of well-formedness criteria on the free names and variables in the process definition, which are given as side conditions.

B.2 Global Types

We will now formally introduce global types and give the definition of projection onto a role.

Definition B.1. The syntax of global types, ranged over by \( G \), is:

\[
G ::= p \rightarrow q : \{ l_i(U_i).G_i \}_{i \in I} \quad \text{(interaction — with } U_i \text{ closed)} \\
\mu t.G \quad t \quad \text{end} \quad \text{(recursive type, type variable, termination)}
\]

Type \( p \rightarrow q : \{ l_i(U_i).G_i \}_{i \in I} \) states that role \( p \) sends one of the labels \( l_i \) for \( i \in I \), together with a payload, to role \( q \). Labels are pairwise distinct. If such label is \( l_i \), then the continuation proceeds as \( G_i \). Type \( \mu t.G \) is a recursive type, where type variables \( t, t', \ldots \) are guarded, namely they appear only under type prefixes. Finally, type end states the termination of a session. We may omit the braces \( \{ \ldots \} \) from an interaction when \( I \) is a singleton, e.g., to write \( a \rightarrow b : l_i(U_i).G_i \) instead of \( a \rightarrow b : \{ l_i(U_i).G_i \}_{i \in \{1\}} \).

The relation between global types and session types is formalised by the notion of projection, given below.

Definition B.2. The projection of \( G \) onto a role \( q \), written \( G \mid q \), is:

\[
(p \rightarrow p' : \{ l_i(U_i).G_i \}_{i \in I}) \mid q \triangleq \begin{cases} 
  p' \oplus_{s \in I} !l_i(U_i).G_i \mid q & \text{if } q = p' \\
  p' & \text{if } q = p' \\
  \bigcap_{i \in I} G_i \mid q & \text{if } p \neq q \neq p'
\end{cases}
\]

\[
(\mu t.G) \mid q \triangleq \begin{cases} 
  \mu t.(G \mid q) & \text{if } G \mid q \neq t' (\forall t') \\
  \text{end} & \text{otherwise}
\end{cases}
\]

\[
\begin{align*}
P \mid Q & \equiv Q \mid P \\
(P \mid Q) \mid R & \equiv P \mid (Q \mid R) \\
P \mid 0 & \equiv P \\
(\nu s)0 & \equiv 0 \\
(\nu s')(\nu s)P & \equiv (\nu s')(\nu s)P \\
(\nu s)P \mid Q & \equiv (\nu s)(P \mid Q) \quad \text{(if } s \notin \text{fc}(Q)\text{)} \\
\text{def } D \text{ in } 0 & \equiv 0 \\
\text{def } D \text{ in } (\nu s)P & \equiv (\nu s)\text{def } D \text{ in } P \quad \text{(if } s \notin \text{fc}(P)\text{)} \\
\text{def } D \text{ in } (P \mid Q) & \equiv \text{def } D \text{ in } P \mid Q \quad \text{(if } \text{dvp}(D) \cap \text{fpv}(Q) = \emptyset\text{)} \\
\text{def } D \text{ in } \text{def } D' \text{ in } P & \equiv \text{def } D' \text{ in } \text{def } D \text{ in } P \\
\text{def } D \text{ in } (\nu s)(\nu s')P & \equiv (\nu s')(\nu s)P \quad \text{(if } \text{dvp}(D) \cup \text{fpv}(D) \cap \text{dvp}(D') = \emptyset\text{)} \\
\end{align*}
\]

Figure 12 Structural congruence for the multiparty session π-calculus.
XX:36 A Linear Decomposition of Multiparty Sessions

where ∩ is the merge operator on session types, defined as:

\[ p \sqcap p \triangleq p \]
\[ p \sqcap \text{?} \triangleq \text{?} 
\[ p \sqcap \text{!} \triangleq \text{!} 
\[ p \sqcap (q \sqcap r) \triangleq (p \sqcap q) \sqcap r 
\[ p \sqcap \Pi_{i \in I} q_i \triangleq \Pi_{i \in I} (p \sqcap q_i) 
\[ p \sqcap \Sigma_{i \in I} q_i \triangleq \Sigma_{i \in I} (p \sqcap q_i) 
\]

The projection of global types onto session types gives branch and select types, as well as recursion and termination, which are the multiparty session types given in Def. 2.5. The intuition behind it follows the same lines as Def. 2.9.

The merge operation [61, 17], as for partial projections, makes local projections defined in more cases.

B.3 Properties of Partial Session Types

**Definition B.3 (Open Subtyping for Partial Session Types).** The relation \( \triangleright_{OP} \) between partial session types is inductively defined by the following rules:

\[
\begin{align*}
&\forall i \in I 
&H_i \triangleright_{OP} H_i' & (\text{S-PARBRCH}) \\
&\forall i \in I 
&H_i \triangleright_{OP} H_i' & (\text{S-PARSEL}) \\
&H \triangleright_{OP} H' & (\text{S-PAR}) \\
&\mu t H \triangleright_{OP} \mu t H & (\text{S-PAR}) \\
&\exists i \in I, j \in J 
&H_i \triangleright_{OP} H_i' & (\text{S-PAR})
\end{align*}
\]

**Corollary B.4.** \( \triangleright_{OP} \) is reflexive.

**Proof.** For all \( H \), we can prove \( H \triangleright_{OP} H \) by easy structural induction on \( H \). \( \Box \)

**Proposition B.5.** If \( \mu t H_1 \triangleright_{OP} \mu t H_2 \), then \( H_1 \{ x_t.H_1/s \} \triangleright_{OP} H_2 \{ x_t.H_2/s \} \).

**Proof.** Assume \( \mu t H_1 \triangleright_{OP} \mu t H_2 \). Without loss of generality, assume that all bound variables in \( \mu t H_1 \) are pairwise distinct, and similarly for \( \mu t H_2 \) (otherwise, the requirement can be met via α-conversion — i.e., this is a form of Barendregt convention). Such a relation can only hold by (S-PARH), and therefore we have some derivation \( D \) such that:

\[
D \begin{cases}
H_1 \triangleright_{OP} H_2 \\
\mu t H_1 \triangleright_{OP} \mu t H_2
\end{cases}
\]

We can inductively rewrite \( D \) by replacing each occurrence of \( H \triangleright_{OP} H' \) with \( H \{ x_t.H_1/s \} \triangleright_{OP} H' \{ x_t.H_2/s \} \). This way, we obtain a new derivation \( D' \) where:

1. each instance of the axiom (S-PAR) with \( t \triangleright_{OP} t \) in \( D \) becomes \( \mu t H_1 \triangleright_{OP} \mu t H_2 \) (which holds by hypothesis) in \( D' \);
2. the conclusion of \( D' \) is \( H_1 \{ x_t.H_1/s \} \triangleright_{OP} H_2 \{ x_t.H_2/s \} \).

Hence, \( D' \) proves the thesis. \( \Box \)

**Lemma B.6.** Let \( H, H' \) be closed partial session types. Then, \( H \triangleright_{OP} H' \) implies \( H \triangleright_{OP} H' \).

**Proof.** Consider the following relation:

\[
\begin{align*}
\mathcal{R}_1 & = \{ (H, H') \mid H, H' \text{ closed and } H \triangleright_{OP} H' \} \\
\mathcal{R}_2 & = \{ (H_1, \{ x_t.H_1/s \}, \mu t H_2, \{ x_t.H_2/s \}) \mid \mu t H_1, \mu t H_2 \text{ closed and } \mu t H_1 \triangleright_{OP} \mu t H_2 \}
\end{align*}
\]

We first prove that \( \mathcal{R} \) is closed backwards under the rules obtained from Def. 2.10, by replacing each occurrence of \( \triangleright_{OP} \) with \( \triangleright_{P} \). For each \( (H, H') \in \mathcal{R} \), we have two cases:
\( (H, H') \in R_1 \). We know that \( H, H' \) are closed and \( H \preceq_{\text{OP}} H' \). Therefore, we proceed by cases on the rule in Def. B.3 that concludes \( H \preceq_{\text{OP}} H' \):

- (S-OPPar). This case is absurd: it would imply \( H = H' = t \), which contradicts the hypothesis that \( H, H' \) are closed;

- (S-OPParEND). We have \( H = H' = \text{end} \), which satisfies rule (S-OPParEND);

- (S-OPParBrch). We have \( H = \&_{i \in I,j} ?l_i(U_i).H_i \preceq_{\text{OP}} \&_{i \in I,j} ?l_i(U_i).H'_i = H' \), and we need to show that \( (H, H') \) satisfies rule (S-OPParBrch). We observe, for all \( i \in I \):

\[
\begin{align*}
H_i, H'_i & \text{ are closed} \quad \text{(otherwise, } H \text{ or } H' \text{ would not be closed)} \\
H_i & \preceq_{\text{OP}} H'_i \quad \text{(from the premise of (S-OPParBrch))} \\
H_i & \in R H'_i \quad \text{(from (4) and (5), by definition of } R_1) \\
U_i & \subseteq S U_i \quad \text{(by reflexivity of } \subseteq S) 
\end{align*}
\]

Hence, from (6) and (7) we conclude that \( (H, H') \) satisfies rule (S-OPParBrch);

- (S-OPParSel). We have \( H = \oplus_{i \in I} U_i.H_i \preceq_{\text{OP}} \oplus_{i \in I} U_i(H_i.H'_i = H') \), and we need to show that \( (H, H') \) satisfies rule (S-OPParSel). We observe that, for all \( i \in I \):

\[
\begin{align*}
H_i, H'_i & \text{ are closed} \quad \text{(otherwise, } H \text{ or } H' \text{ would not be closed)} \\
H_i & \preceq_{\text{OP}} H'_i \quad \text{(from the premise of (S-OPParSel))} \\
H_i & \in R H'_i \quad \text{(from (8) and (9), by definition of } R_1) \\
U_i & \subseteq S U_i \quad \text{(by reflexivity of } \subseteq S) 
\end{align*}
\]

Hence, from (10) and (11) we conclude that \( (H, H') \) satisfies rule (S-OPParSel);

- (S-OPParBrch). We have \( H = \mu t.H_1 \preceq_{\text{OP}} \mu t.H_2 = H' \), and we need to show that \( (H, H') \) satisfies both rules (S-OPParL) and (S-OPParR). We observe that:

\[
\begin{align*}
(H_1 \{^t H_1 / \lambda \}, H_2 \{^t H_2 / \lambda \}) & \in R_2 \subseteq R \quad \text{(by definition of } R_2 \text{ and } R) \\
(H_1 \{^t H_1 / \lambda \}, H_2 \{^t H_2 / \lambda \}) & \in R_2 \subseteq R \quad \text{(by definition of } R_2 \text{ and } R) 
\end{align*}
\]

Therefore, we conclude that \( (H, H') \) satisfies both (S-OPParL) (by (12)) and (S-OPParR) (by (13)).

\( (H, H') \in R_2 \). We know that \( H, H' \) are closed, and either:

- \( H = H_1 \{^t H_1 / \lambda \}, H' = \mu t.H_2 \), and \( \mu t.H_1 \preceq_{\text{OP}} \mu t.H_2 \). In this case, we need to show that \( (H, H') \) satisfies rule (S-OPParR). We observe that:

\[
\begin{align*}
H_1 \{^t H_1 / \lambda \}, H_2 \{^t H_2 / \lambda \} & \text{ are closed} \quad \text{(otherwise, } H \text{ or } H' \text{ would not be closed)} \\
H_1 \{^t H_1 / \lambda \} & \preceq_{\text{OP}} H_2 \{^t H_2 / \lambda \} \quad \text{(by } \mu t.H_1 \preceq_{\text{OP}} \mu t.H_2 \text{ and Proposition B.5)} \\
(H_1 \{^t H_1 / \lambda \}, H_2 \{^t H_2 / \lambda \}) & \in R_1 \subseteq R \quad \text{(from (14), (15), and by definition of } R_1 \text{ and } R) 
\end{align*}
\]

Therefore, we conclude that \( (H, H') \) satisfies rule (S-OPParR):

- \( H = \mu t.H_1, H' = H_2 \{^t H_2 / \lambda \}, \mu t.H_1 \preceq_{\text{OP}} \mu t.H_2 \). In this case, we need to show that \( (H, H') \) satisfies rule (S-OPParL): the proof is symmetric w.r.t. the previous case.

We have shown that \( R \) is closed backwards under the rules obtained from Def. 2.10. Therefore, since \( \preceq_{\text{P}} \) is the largest relation closed backwards under such rules, we have \( R \subseteq \preceq_{\text{P}} \). We also know that for all closed \( H, H' \) such that \( H \preceq_{\text{OP}} H' \), we have \( (H, H') \subseteq R_1 \subseteq R \subseteq \preceq_{\text{P}} \); we conclude \( H \preceq_{\text{P}} H' \).

Lemma B.7. For any finite set of partial session types \( \{H_i\}_{i \in I} \), if \( H^* = \bigcap_{i \in I} H_i \) is defined, then \( \forall k \in I : H^* \preceq_{\text{P}} H_k \).
XX:38 A Linear Decomposition of Multiparty Sessions

Proof. Assuming that \( H^* \) is defined, we choose any \( H_k \) (with \( k \in I \)) and proceed by structural induction on \( H_k \):

- base case \( H_k = \mathrm{end} \). By Def. 2.9, we must have \( H^* = \mathrm{end} \) and \( \forall i \in I : H_i = \mathrm{end} \) (otherwise, \( H^* \) would be undefined). We conclude \( H^* \leq_{\mathrm{OP}} H_k \), by (S-OPAREnd);
- base case \( H_k = t \). By Def. 2.9, we must have \( H^* = t \) and \( \forall i \in I : H_i = t \) (otherwise, \( H^* \) would be undefined). We conclude \( H^* \leq_{\mathrm{OP}} H_k \), by (S-OPAR);
- inductive case \( H_k = k_{j \in J} \mu_j(U_j).H_{j} \). By Def. 2.9, we must have \( H^* = k_{j \in J} \mu_j(U_j).H_{j} \) and \( \forall i \in I : H_i = k_{j \in J} \mu_j(U_j).H_{j} \) (otherwise, \( H^* \) would be undefined). By the induction hypothesis, \( \forall j \in J : H_j \leq_{\mathrm{OP}} H_{k_j} \); we conclude \( H^* \leq_{\mathrm{OP}} H_k \), by (S-OPARBrch);
- inductive case \( H_k = \oplus_{j \in J_k} \mu_j(U_j).H_{j,k} \). By Def. 2.9, we must have:

\[
H^* = \oplus_{j \in J_k} \mu_j(U_j).(\bigotimes_{i \in I} H_i) \oplus \bigotimes_{j \in J_j} \mu_j(U_j).H_{j,k}
\]

where \( J^* = \bigcap_{i \in I} J_i \) and \( \forall i \in I : H_i = \oplus_{j \in J_i} \mu_j(U_j).H_{j,i} \) (otherwise, \( H^* \) would be undefined). By the induction hypothesis, \( \forall j \in J \cap J_k : \bigcap_{i \in I} H_i \leq_{\mathrm{OP}} H_{j,k} \); moreover, \( \forall j \in J_k \setminus J^* : H_{j,k} \leq_{\mathrm{OP}} H_{j,k} \) (by Cor.B.4). We conclude \( H^* \leq_{\mathrm{OP}} H_k \), by (S-OPARSel);

- inductive case \( H_k = \mu t. H_k' \). By Def. 2.9, we must have \( H^* = \mu t. (\bigcap_{i \in I} H_i') \) and \( \forall i \in I : H_i = \mu t. H_i' \) (otherwise, \( H^* \) would be undefined). By the induction hypothesis, \( \bigcap_{i \in I} H_i' \leq_{\mathrm{OP}} H_k' \); we conclude \( H^* \leq_{\mathrm{OP}} H_k \), by (S-OPAR);

\[\blacksquare\]

\textbf{Proposition B.8.} For all partial types \( H, H' \), \( H \leq_{P} H' \) iff \( \mathrm{unf}(H) \leq_{P} \mathrm{unf}(H') \) iff \( \mathrm{unf}(H) \leq_{P} \mathrm{unf}(H') \) iff \( \mathrm{unf}(H) \leq_{P} \mathrm{unf}(H') \).

Proof. We split the statement in three parts, and prove them separately:

- \((H \leq_{P} H' \iff \mathrm{unf}(H) \leq_{P} \mathrm{unf}(H'))\) Let \( H = \mu t_1 \ldots \mu t_m.H_0 \), with \( H_0 \neq \mu t' \ldots \). We first prove the following statement:

\[\forall n \in 0..m : \ H \leq_{P} H' \iff \ H_{\{t_1=\ldots=t_n.H_0/t_n\}} \ldots \{t_{n-1}.H_{n-1}/t_{n-1}\} \leq_{P} H' \quad \text{where} \quad H_{s} = \mu t_{n+1} \ldots \mu t_{m}.H_{s} \quad (17)\]

The proof proceeds by induction on \( n \). The base case \( n = 0 \) is trivial, and holds by reflexivity of \( \leq_{P} \). In the inductive case \( n = n' + 1 \), we have (by the induction hypothesis):

\[H \leq_{P} H' \iff \ H_{\{t_1=\ldots=t_n.H_0/t_n\}} \ldots \{t_{n-1}.H_{n-1}/t_{n-1}\} \leq_{P} H' \quad (18)\]

We can notice that, by the coinductive rule (S-PARµL) in Def. 2.10, the RHS of the “iff” in (18) holds if and only if:

\[H_{\{t_1=\ldots=t_{n-1}=\mu t'_{n-1}.H_{n-1}/t_{n-1}\}} \ldots \{t_{n-1}.H_{n-1}/t_{n-1}\} \leq_{P} H' \]

which is the thesis.

We conclude observing that, when \( n = m \) (i.e., \( H \) is completely unfolded, bringing at the top-level \( H_0 \neq \mu t' \ldots \)) we have proved \( H \leq_{P} H' \iff \ \mathrm{unf}(H) \leq_{P} \mathrm{unf}(H') \).

- \((H \leq_{P} H' \iff \ H \leq_{P} \mathrm{unf}(H'))\) The proof is symmetric w.r.t. the case above, and uses (S-PARµR);

- \((H \leq_{P} H' \iff \ \mathrm{unf}(H) \leq_{P} \mathrm{unf}(H'))\) From the previous cases we have:

\[\mathrm{unf}(H) \leq_{P} H \iff H \leq_{P} H \quad \text{and} \quad H' \leq_{P} H' \iff H' \leq_{P} \mathrm{unf}(H') \]

and by transitivity of \( \leq_{P} \), we conclude \( H \leq_{P} H' \iff \mathrm{unf}(H) \leq_{P} \mathrm{unf}(H') \).

\[\blacksquare\]

\textbf{Proposition B.9.} \( \mathrm{unf}(\overline{H}) = \overline{\mathrm{unf}(H)} \).
Proof. Let \( H = \mu t_1 \ldots \mu t_m.H_0 \), with \( H_0 \neq \mu t' \ldots \). We first prove the following statement:

\[
\forall n \in \mathbb{N} : \mu t_1 \ldots \mu t_n.H_0 = \mu t_1 \ldots \mu t_n.H_1 \ldots \mu t_n.H_n
\]

where \( H_r = \mu t_{r+1} \ldots \mu t_m.H_r \).

The proof proceeds by induction on \( n \). The base case \( n = 0 \) is trivial, while in the inductive case \( n = n' + 1 \) we have:

\[
\begin{align*}
\mu t_{n+1}.H_{n+1} &= \mu t_1 \ldots \mu t_{n+1} H_{n+1} \overset{\text{(by def. 2.8)}}{=} \mu t_1 \ldots \mu t_n.H_n \overset{\text{(by the i.h.)}}{=} \mu t_{n+1}.H_n.
\end{align*}
\]

and we obtain the thesis by further unfolding \( \mu t_{n+1} \ldots \mu t_m.H_0 \) in the LHS and RHS above.

We conclude observing that, when \( n = m \) (i.e., \( H \) is completely unfolded, bringing at the top-level \( H_0 \neq \mu t' \ldots \)) we have proved \( \text{unf}(H) = \text{unf}(H_0) \).

\[\blacksquare\]

- **Proposition B.10.** For all session types \( S \) and roles \( p \), \( \text{unf}(S) \upharpoonright p = \text{unf}(S \upharpoonright p) \).

Proof. Let \( S = \mu t_1 \ldots \mu t_m.S_0 \), with \( S_0 \neq \mu t' \ldots \). We first prove the following statement:

\[\forall n \in \mathbb{N} : \mu t_1 \ldots \mu t_n.S_0 = \mu t_1 \ldots \mu t_n.S_n\]

where \( S_n = \mu t_{n+1} \ldots \mu t_m.S_0 \).

We proceed by induction on \( n \). The base case \( n = 0 \) is trivial, while in the inductive case \( n = n' + 1 \) we have:

\[\begin{align*}
(\mu t_n.S_n \{\mu t_1 \ldots \mu t_n.S_n \upharpoonright S / t_1\} \ldots \{\mu t_n.S_n \upharpoonright S / t_n\}) \upharpoonright p &= (\mu t_n.S_n \upharpoonright p) \{\mu t_1 \ldots \mu t_n.S_n \upharpoonright S / p / t_1\} \ldots \{\mu t_n.S_n \upharpoonright S / p / t_n\} \quad \text{(by the i.h.) (19)}
\end{align*}\]

At this point we have two cases, based on the partial projection of recursive types in Def 2.9. If \( S \upharpoonright p \neq t' \) (for all \( t' \)), then \( (\mu t_n.S_n \upharpoonright p) = \mu t_n.S_n \upharpoonright p \), and we get:

\[\begin{align*}
(\mu t_n.((S_n \{\mu t_1 \ldots \mu t_n.S_n \upharpoonright S / t_1\} \ldots \{\mu t_n.S_n \upharpoonright S / t_n\}) \upharpoonright p) &= (\mu t_n.S_n \upharpoonright p) \{\mu t_1 \ldots \mu t_n.S_n \upharpoonright S / t_1\} \ldots \{\mu t_n.S_n \upharpoonright S / p / t_n\} \quad \text{(by Def. 2.9)}
\end{align*}\]

and we obtain the thesis by further unfolding \( \mu t_{n+1} \ldots \mu t_m.S_0 \) in the LHS and RHS above.

Otherwise, if \( S \upharpoonright p = t' \) (for some \( t' \)), then \( (\mu t_n.S_n \upharpoonright p) = \text{end} \). Therefore, on the RHS of (19) we have:

\[\begin{align*}
(\mu t_n.S_n \upharpoonright p) \{\mu t_1 \ldots \mu t_n.\text{end} / t_1\} \ldots \{\mu t_n.\text{end} / t_n\} &= \text{end} \{\mu t_1 \ldots \mu t_n.\text{end} / t_1\} \ldots \{\mu t_n.\text{end} / t_n\} \quad \text{(20)}
\end{align*}\]

Moreover, since in this case we must have \( t' \in \text{fv}(S_n) \), then we also have one of the following:

\[\begin{align*}
(S_n \{\mu t_1 \ldots \mu t_n.S_n \upharpoonright S / t_1\} \ldots \{\mu t_n.S_n \upharpoonright S / t_n\}) &= \text{end} \quad \text{(if } t' \neq t_n, \text{ i.e., } t'_i = t_i \text{ for some } i \in 1..n) \quad (21)
\end{align*}\]

Now, if (21) holds, we get:

\[\begin{align*}
(\mu t_n.S_n \{\mu t_1 \ldots \mu t_n.\text{end} / t_1\} \ldots \{\mu t_n.\text{end} / t_n\}) \upharpoonright p &= \text{end} \quad \text{(by Def. 2.9)}
\end{align*}\]

that, together with (20), by further (vacuously) unfolding both terms once, gives us \( \text{end} = \text{end} \) (which is our thesis).

Otherwise, if (22) holds, we get:

\[\begin{align*}
(\mu t_n.S_n \{\mu t_1 \ldots \mu t_n.\text{end} / t_1\} \ldots \{\mu t_n.S_n \upharpoonright S / t_n\}) \upharpoonright p &= \mu t_n.\text{end} \quad \text{(by Def. 2.9)}
\end{align*}\]

that, together with (20), by further unfolding both terms once, gives us \( \text{end} = \text{end} \) (which is our thesis).

We conclude observing that, when \( n = m \) (i.e., \( H \) is completely unfolded, bringing at the top-level \( H_0 \neq \mu t' \ldots \)) we have proved \( \text{unf}(S) \upharpoonright p = \text{unf}(S \upharpoonright p) \). □
Proposition B.11. \( S \preceq S' \) implies \( \text{roles}(S) = \text{roles}(S') \).

Proof. Assume \( S \preceq S' \). By contradiction, assume that \( \text{roles}(S) \neq \text{roles}(S') \), i.e., \( \exists q \in S' \text{ but } q \notin S \) (the proof for \( S \triangleright p \not\in S' \) is similar, but uses \( T\text{-Sel} \) in the following). We can observe that \( S, S' \) cannot be related by \( (S\text{-End}) \) (otherwise we would have \( S = S' = \text{end} \), and thus \( q \notin \text{roles}(S') = \emptyset \)). Moreover, \( q \) cannot be the top-level role of any rule applied in the derivation of \( S \preceq S' \) (otherwise we would have \( S' \triangleright q \in S \)). Hence, the derivation for \( S \preceq S' \) must have at least one occurrence of \( (T\text{-Branch}) \) with some (but not all) branches on the RHS containing \( q \), i.e.,

\[ p \& \forall i \in I \ ?i(U_i).S_i \preceq p \& \forall i \in I \ ?i(U_i).S_i' \quad \text{for some } k \in J, \text{ and } q \notin S_i' \quad \text{for some } i \in I \text{ (otherwise, } q \text{ would necessarily be the top-level role at some point in the derivation). Then, we have two cases: either } S_i' \mid q \text{ is not defined, or } S_i' \mid q \text{ is defined, but } S_i' \mid q \neq \text{end} \quad \text{— which implies that it cannot be merged with } S_i' \mid q = \text{end} \text{. In both cases, we obtain that } S' \mid q \text{ is not defined, which violates clause (i) of Def. 2.10, and therefore implies } S \not\preceq S' \quad \text{— contradiction.}\]

\[ \square \]

Proposition B.12. Let \( S, S' \) be closed session types. If \( S \preceq S' \), then for all \( p \) also \( S \mid p \preceq S' \mid p \).

Proof. By Def. 2.10 (clause (ii)), we already know that \( \forall p \in (\text{roles}(S) \cup \text{roles}(S')) \), we have \( S \mid p \preceq S' \mid p \). Since \( p \) in the statement is universally quantified, we are left to prove it for all \( p \notin (\text{roles}(S) \cup \text{roles}(S')) \). By Proposition B.11, we know that \( p \in S \) iff \( p \in S' \); hence, by Def. 2.9, we obtain that for all \( p \notin (\text{roles}(S) \cup \text{roles}(S')) \), \( S \mid p = \text{end} \preceq S' \mid p \).

\[ \square \]

Proposition B.13. If \( S \mid q \) is defined and closed, and either \( \text{unf}(S) = p \& \forall i \in I \ ?i(U_i).S_i \) or \( \text{unf}(S) = \exists p \oplus \forall i \in I \ ?i(U_i).S_i \) with \( p \neq q \), then \( \forall k \in I : \ S \mid q \preceq S_k \mid q \).

Proof. Assume that \( S \mid q \) is defined and closed. We have two cases:

\[ \text{unf}(S) = p \& \forall i \in I \ ?i(U_i).S_i. \quad \text{By Def. 2.9, } \text{unf}(S) \mid q = \prod_{i \in I} (S_i \mid q); \text{ moreover, by Lemma B.7, } \forall k \in I : \prod_{i \in I} (S_i \mid q) \preceq_{\text{OP}} S_k \mid q. \text{ Noticing that } \forall k \in I : S_k \mid q \text{ is closed (otherwise, } S \mid q \text{ would not be closed), by Lemma B.6 we get } \forall k \in I : \text{unf}(S) \mid q \preceq S_k \mid q, \text{ and therefore (by Proposition B.10) } \text{unf}(S) \mid q \preceq S_k \mid q; \text{ then, by Proposition B.8, we conclude } S \mid q \preceq S_k \mid q; \]

\[ = S = p \oplus \forall i \in I \ ?i(U_i).S_i. \quad \text{By Def. 2.9, } S \mid q = \prod_{i \in I} (S_i \mid q); \text{ the proof is similar to the previous case.}\]

\[ \square \]

Proposition B.14. If \( (\Gamma, x : U) \) is consistent, then \( \Gamma \) is consistent.

Proof. The proof is straightforward, by noticing that Def. 2.11 on consistency does not depend on \( x : U \).

\[ \square \]

Proposition B.15. If \( (\Gamma, s[p] : S) \) is consistent, then \( \Gamma \) is consistent.

Proof. Assume that \( \Gamma, s[p] : S \) is consistent. By Def. 2.11, it means that \( \forall q \ni s[x], s[x] \in \text{dom} (\Gamma) \setminus \{s[p] : S\} : q \neq x \implies \Gamma(s[q]) \mid x \preceq \Gamma(s[x]) \mid q. \) Since \( \text{dom} (\Gamma) = \text{dom} (\Gamma, s[p] : S) \setminus \{s[p]\} \), we also have that \( \forall q \ni s[x], s[x] \in \text{dom} (\Gamma) : q \neq x \implies \Gamma(s[q]) \mid x \preceq \Gamma(s[x]) \mid q. \) Hence, by Def. 2.11, we conclude that \( \Gamma \) is consistent.

\[ \square \]

Corollary B.16. If \( (\Gamma_1, \Gamma_2) \) is consistent, then \( \Gamma_1 \) and \( \Gamma_2 \) are consistent.

Proof. By repeatedly applying Proposition B.14 and Proposition B.15 to remove all entries of \( \Gamma_1 \) from \( (\Gamma_1, \Gamma_2) \), we prove that \( \Gamma_2 \) is consistent. With the symmetric procedure, we prove that \( \Gamma_1 \) is consistent.

\[ \square \]

Corollary B.17. If \( (\Gamma_1 \circ \Gamma_2) \) is consistent, then \( \Gamma_1 \) and \( \Gamma_2 \) are consistent.

Proof. Similar to the proof of Cor. B.16, except that we might have entries of the form \( x : B \) (which are not relevant for consistency, as per Def. 2.11) appearing in both \( \Gamma_1 \) and \( \Gamma_2 \).

\[ \square \]

Proposition B.18. If \( \Gamma, s[p] : S \) is consistent and \( S \preceq S' \), then \( \Gamma, s[p] : S' \) is consistent.

\[ \square \]
Proof. Assume that \( \Gamma, s[p]; S \) is consistent, and take any \( S' \) such that \( S \subseteq S' \). By Def. 2.11, we know that \( \forall s[q]; S_q \in \text{dom}(\Gamma): S_q \vdash p \leq s[q]. \) Moreover, by Proposition B.12, we have \( \forall q : S \mid q \leq p \mid S \mid q. \) Therefore, by transitivity of \( \leq_p \), we also have \( \forall s[q]; S_q \in \text{dom}(\Gamma): S_q \vdash p \leq S' \mid q. \) and by Def. 2.11, we conclude that \( \Gamma, s[p]; S' \) is consistent. ▶

\[\text{Corollary B.19. If } \Gamma_1, \Gamma_2 \text{ is consistent and } \Gamma_2 \leq_5 \Gamma_2', \text{ then } \Gamma_1, \Gamma_2' \text{ is consistent.} \]

Proof. By induction on the size of \( \Gamma_2 \). The base case (\( \Gamma_2 = \emptyset \)) is trivial, while the inductive case is proved by the induction hypothesis, and Proposition B.18. ▶

\[\text{Corollary B.20. If } \Gamma_1 \circ \Gamma_2 \text{ is consistent and } \Gamma_2 \leq_5 \Gamma_2', \text{ then } \Gamma_1 \circ \Gamma_2' \text{ is consistent.} \]

Proof. Similar to the proof of Cor. B.19, except that \( \Gamma_1, \Gamma_2 \) and \( \Gamma_2 \) can have (possibly shared) entries mapping some \( x \) to a basic type (which are not relevant for consistency, as per Def. 2.11). ▶

\[\text{Proposition B.21. If } \Gamma \rightarrow^{\star} \Gamma', \text{ then } \text{dom}(\Gamma) = \text{dom}(\Gamma'). \]

Proof. We first verify the following statement, by induction on the size of \( \text{dom}(\Gamma) \):

\[\Gamma \rightarrow \Gamma' \quad \text{implies} \quad \text{dom}(\Gamma) = \text{dom}(\Gamma') \quad (23)\]

Then, we can prove the main statement, by induction on the length of the sequence of reductions in \( \Gamma \rightarrow^{\star} \Gamma' \). The base case is trivial (we have 0 reductions, and \( \Gamma = \Gamma' \)), while in the inductive case, we apply the induction hypothesis and (23). ▶

\[\text{Lemma B.22. If } \Gamma \rightarrow \Gamma' \text{ and } \Gamma \text{ is consistent (resp. complete), then so is } \Gamma'. \]

Proof. Assume that \( \Gamma \) is consistent. We proceed by induction on the derivation of \( \Gamma \rightarrow \Gamma' \), as per Def. 2.15:

- base case \( \Gamma = \Gamma_1, c; U \rightarrow \Gamma_1', c; U' = \Gamma' \), with \( U \leq_5 U' \). In this case, \( c \) might be either a variable \( x \), or a channel with role \( s[x] \). If \( c = x \), the thesis holds trivially by the induction hypothesis, since \( x; U \) and \( x; U' \) are not relevant for consistency (Def. 2.11). Instead, if \( c = s[x] \), both \( U \) and \( U' \) must be session types (by Def. 2.11). Therefore, we have \( \Gamma = \Gamma_1, s[x]; S_r \rightarrow \Gamma_1', s[x]; S_r' = \Gamma', \) with:

\[\Gamma_1 \rightarrow \Gamma_1' \quad \text{(24)}\]

\[S_r \leq_5 S_r' \quad \text{(25)}\]

From (24), we can observe that \( \Gamma_1, \Gamma_1' \) must have the form:

\[\Gamma_1 = s[p]; S_p, s[q]; S_q, \Gamma_0 \quad \text{(26)}\]

\[\Gamma_1' = s[p]; S_p', s[q]; S_q', \Gamma_0' \quad \text{(27)}\]

where \( s[p]; S_p, s[q]; S_q \rightarrow s[p]; S_p', s[q]; S_q' \) and \( \Gamma_0 \leq_5 \Gamma_0' \quad \text{(28)}\]
Therefore:

\[ \Gamma = \Gamma_1, s[x]: S_r = s[p]: S_p, s[q]: S_q, \Gamma_0, s[x]: S_r \text{ is consistent} \]  
(by hypothesis and (26))

\[ s[p]: S_p, s[q]: S_q, \Gamma_0, s[x]: S_r \text{ is consistent} \]  
(from (29), (28), and Cor. B.19)

\[ \Gamma_0, s[x]: S_r \text{ is consistent} \]  
(by (30) and Cor. B.16)

\[ s[p]: S_p', s[q]: S_q', \Gamma_0' \text{ is consistent} \]  
(by (27), (24) and the induction hypothesis)

\[ s[p]: S_p', s[q]: S_q', \Gamma_0' \text{ and } s[p]: S_p', \Gamma_0 \text{ are consistent} \]  
(by (32) and Cor. B.16)

Hence, to prove that \( \Gamma' = s[p]: S_p', s[q]: S_q', \Gamma_0', s[x]: S_r \) is consistent, from (27) and (33) we can see that we are left to prove that both \( s[p]: S_p', s[q]: S_q' \) and \( s[q]: S_q', s[x]: S_r \) are consistent. By Def. 2.11, it means that we need to prove:

\[ S_p' \vdash x \leq p S'_r \mid p \quad \text{and} \quad S_q' \vdash x \leq p S'_r \mid q \]  
(34)

From (28) and Def. 2.15, we have two sub-cases:

- \( \text{unf}(S_p) = q \oplus_{i \in \mathcal{I}} \cup_i (U_i).S_i \) and \( \text{unf}(S_q) = p \&_{i \in \mathcal{I}, j \in J} \cup_i (U_i).S_i \). Then:

  for some \( k \in \mathcal{I} \), \( S_p = S_k \) and \( S_q = S_k' \) \hspace{1cm} (by Def. 2.15)

  \[ \text{unf}(S_p) \vdash x \leq p S_p \mid x \quad \text{and} \quad \text{unf}(S_q) \vdash x \leq p S_q' \mid x \]  
(by (35) and Proposition B.13)

  \[ S_p' \vdash x \leq p \text{unf}(S_p) \mid x \quad \text{and} \quad S_q' \vdash x \leq p \text{unf}(S_q) \mid x \]  
(by (36) and Proposition D.1)

  \[ \text{unf}(S_p) \vdash x \leq p S_r \mid p \quad \text{and} \quad \text{unf}(S_q) \vdash x \leq p S_r \mid q \]  
(by hypothesis (consistency of \( \Gamma \)) and Def. 2.11)

  \[ S_r \mid p \leq p S_r' \mid p \quad \text{and} \quad S_r \mid q \leq p S_r' \mid q \]  
(by (25) and Proposition B.12)

  \[ S_p' \vdash x \leq p S_r' \mid p \quad \text{and} \quad S_q' \vdash x \leq p S_r' \mid q \]  
(by (37), (38), (39) and transitivity of \( \leq p \))

- \( \text{unf}(S_p) = p \&_{i \in \mathcal{I}, j \in J} \cup_i (U_i).S_i \) and \( \text{unf}(S_q) = q \oplus_{i \in \mathcal{I}} \cup_i (U_i).S_i \). The proof is symmetric w.r.t. the previous case.

Hence, we have proved (34); from (34), (31) and (32), by Def. 2.11 we conclude that \( \Gamma' \) is consistent.

For the second part of the statement, assume that \( \Gamma \) is complete: we can prove that \( \Gamma' \) is also complete by induction on the derivation of \( \Gamma \rightarrow \Gamma' \), as per Def. 2.15. The key observation if that for each \( s[p] \in \text{dom}(\Gamma) \), we also have \( s[p] \in \text{dom}(\Gamma') \) (by Proposition B.21), and \( \text{roles}(\Gamma'(s[p])) \subseteq \text{roles}(\Gamma(s[p])) \).

\text{Corollary B.23. If } \Gamma_1, \Gamma_2 \text{ is consistent and } \Gamma_1 \rightarrow^* \Gamma'_1, \text{ then } \Gamma_1, \Gamma_2 \text{ is consistent.}

\text{Proof. Assume all the hypotheses, and let } n \text{ be the length of the sequence of reductions in } \Gamma_1 \rightarrow^* \Gamma'_1. \text{ In the base case } (n = 0) \text{ the thesis holds trivially. In the inductive case } n = n' + 1, \text{ we have: }

\[ \Gamma_1 \rightarrow \cdots \rightarrow \Gamma'_{n'} \rightarrow \Gamma'_1 \]

and by the induction hypothesis, \( \Gamma'_1, \Gamma_2 \) is consistent. This implies that \( \Gamma'_1, \Gamma_2 \) is consistent: we prove such a fact with a further induction on the size of \( \Gamma_2 \). In the base case (\( \Gamma_2 = \emptyset \)) we conclude
immediately by Lemma B.22. In the inductive case we have \( \Gamma_2 = \Gamma_0, c:U \); by applying the induction hypothesis we get that \( \Gamma'_1, \Gamma_0 \) is consistent, and we examine the shape of the additional entry \( c:U \) and its consistency w.r.t. \( \Gamma'_1, \Gamma_0 \), similarly to the inductive case in the proof of Lemma B.22. In all cases, we conclude that \( \Gamma_1, \Gamma_2 \) is consistent.

\[\text{Corollary B.24. If } \Gamma_1 \circ \Gamma_2 \text{ is consistent and } \Gamma_1 \rightarrow^* \Gamma'_1, \text{ then } \Gamma'_1 \circ \Gamma_2 \text{ is consistent.}\]

**Proof.** Similar to the proof of Cor. B.23, except that \( \Gamma_1, \Gamma'_1 \) and \( \Gamma_2 \) can have (possibly shared) entries mapping some \( x \) to a basic type (which are not relevant for consistency, Def. 2.11).

\[\text{Proposition B.25. For all multiparty session processes } P, P', \text{ if } \Theta \cdot \Gamma \vdash P \text{ and } P \equiv P', \text{ then } \Theta \cdot \Gamma \vdash P'.\]

**Proof.** The proof proceeds by induction on the structural congruence \( \equiv \), defined in Fig. 12.

\[\text{Lemma B.26 (Substitution lemma). If } \Theta \cdot \Gamma, x:U \vdash P, \Gamma' \vdash v:U \text{ and } \Gamma \circ \Gamma' \text{ is consistent, then } \Theta \cdot \Gamma \circ \Gamma' \vdash P\{v/x\}.\]

**Proof.** The proof is by induction on the typing derivations, with a case analysis on the last rule applied.

\[\text{Definition B.27 (Context subtyping). For all multiparty session typing contexts } \Gamma_5, \Gamma'_5, \text{ the relation } \Gamma_5 \leq_S \Gamma'_5 \text{ holds if dom } (\Gamma_5) = \text{dom } (\Gamma'_5) \text{ and } \forall c \in \text{dom } (\Gamma_5) : \Gamma_5(c) \leq_S \Gamma'_5(c). \text{ We define the following multiparty session typing rule, corresponding to 0 or more consecutive applications of } \text{T-MSub}:}\]

\[
\frac{\Theta \cdot \Gamma_5 \vdash P \quad \Gamma'_5 \leq_S \Gamma_5}{\Theta \cdot \Gamma'_5 \vdash P} \quad \text{(T-MSub)}
\]

Proposition B.30 below will allow us to consider only one form of “normalised” typing derivation for multiparty session processes (the first one in the statement), where (possibly vacuous) instances of \( \text{T-MSub} \) appear as premises of \( \text{T-Par} \), but not vice versa. This allows rewrite a typing derivation by “pushing” \( \text{T-MSub} \) towards the leaves, until reaching a sub-process that cannot be further decomposed using the parallel composition \( \circ \).

\[\text{Proposition B.28. If } \Theta \cdot \Gamma \vdash P_1 | P_2, \text{ then } \exists \Gamma_1, \Gamma'_1, \Gamma_2, \Gamma'_2 \text{ such that } \Gamma = \Gamma_1 \circ \Gamma_2, \Gamma_1 \leq_S \Gamma'_1, \Gamma_2 \leq_S \Gamma'_2, \Theta \cdot \Gamma'_1 \vdash P_1 \text{ and } \Theta \cdot \Gamma'_2 \vdash P_2. \text{ Moreover:}\]

\[
\frac{(\text{T-MSub}) \quad \Theta \cdot \Gamma'_1 \vdash P_1 \quad \Gamma_1 \leq_S \Gamma'_1}{\Theta \cdot \Gamma_1 \circ \Gamma_2 \vdash P_1 \big | P_2} \quad \frac{(\text{T-MSub}) \quad \Theta \cdot \Gamma'_2 \vdash P_2 \quad \Gamma_2 \leq_S \Gamma'_2}{\Theta \cdot \Gamma_1 \circ \Gamma_2 \vdash P_1 \big | P_2} \quad \text{(T-Par)}
\]

\[\iff \]

\[
\frac{(\text{T-Par}) \quad \Theta \cdot \Gamma'_1 \vdash P_1 \quad \Theta \cdot \Gamma'_2 \vdash P_2}{\Theta \cdot \Gamma'_1 \circ \Gamma'_2 \vdash P_1 \big | P_2} \quad \text{(T-MSub)}
\]

\[\iff \]

\[
\frac{\text{T-Par}}{\Theta \cdot \Gamma_1 \circ \Gamma_2 \vdash P_1 \big | P_2} \quad \Gamma_1 \circ \Gamma_2 \leq_S \Gamma_1 \circ \Gamma_2 \quad \text{(T-MSub)}
\]

\[\iff \]

\[
\frac{\text{T-Par}}{\Theta \cdot \Gamma_1 \circ \Gamma_2 \vdash P_1 \big | P_2} \quad \Gamma_1 \circ \Gamma_2 \leq_S \Gamma_1 \circ \Gamma_2 \quad \text{(T-MSub)}
\]
**Proof.** The first part of the statement is straightforward by inversion of (T-PAR) (by Def.B.27), and adding a (possibly vacuous) instance of (T-St)(u), as in the last case in the statement after “moreover.” The “iff” relations among the typing derivations are also straightforward: the hypotheses of one derivation imply all the others, and if one hypothesis is falsified, none of the derivations hold. 

**Proposition B.29.** $\Theta \cdot \Gamma_1 \vdash (\nu s: \Gamma_2)P$, then $\exists \Gamma'_1, \Gamma'_2$ such that $\Gamma_1 \subseteq S \Gamma'_1$, $\Gamma_2 \subseteq S \Gamma'_2$, and $\Gamma'_1 \circ \Gamma'_2 \vdash P$. Moreover:

\[
\begin{align*}
\text{(T-MSub) } & \Theta \cdot \Gamma'_1 \circ \Gamma'_2 \vdash P & \text{(T-Res) } & \Theta \cdot \Gamma'_1 \vdash (\nu s: \Gamma_2)P \quad \text{iff} \quad \Theta \cdot \Gamma'_1 \circ \Gamma'_2 \vdash P \quad \Theta \cdot \Gamma'_1 \vdash (\nu s: \Gamma_2)P
\end{align*}
\]

**Proof.** The first part of the statement is straightforward by inversion of (T-Res) (by Def.B.27), and adding a (possibly vacuous) instance of (T-MSub), as in the first case in the statement after “moreover.” The “iff” relations among the typing derivations are also straightforward: the hypotheses of one derivation imply the other, and if one hypothesis is falsified, none of the derivations hold.

**Proposition B.30 (Subtyping normalisation).** If $\Theta \cdot \Gamma \vdash P$, then there exist a derivation that proves the judgement by only applying rule (T-MSub) on the conclusions of (T-Brch), (T-Sel) and (T-Call).

**Proof.** Assume that we have a derivation $D$ concluding $\Theta \cdot \Gamma \vdash P$, that does not match the thesis: if it is just an instance of (T-Brch), (T-Sel) and (T-Call), we conclude by simply adding a vacuous instance of (T-MSub). Otherwise, $D$ must have one of the shapes in Proposition B.28 or Proposition B.29, and we can “push” (T-MSub) towards the leafs (where (T-Brch), (T-Sel) and (T-Call) occur) by recursively rewriting it in the first form of the statements.

**Theorem 2.16 (Subject reduction).** If $\Theta \cdot \Gamma \vdash P$ and $P \rightarrow P'$, then there exists $\Gamma'$ such that $\Gamma \rightarrow^* \Gamma'$ and $\Theta \cdot \Gamma' \vdash P'$.

**Proof.** By induction on the derivation of the reduction $P \rightarrow P'$:

- **base case (R-Comm).** We have $P = Q_1 | Q_2$, and:

\[
\begin{align*}
Q_1 &= s[p]q|k_j \in T(j(x_j), Q_{1j}) \\
Q_2 &= s[q]p \oplus \{l_k(v)\}.Q_{2} \\
P &= Q_1 | Q_2 \rightarrow Q_{1k}(p|v_k) | Q_{2} = P' \quad (k \in I)
\end{align*}
\]

(41)

Therefore, for some $k \in I$, by inversion of (T-PAR) and (T-Brch)/(T-Sel), allowing (possibly vacuous) instances of (T-MSub) as per Proposition B.30, there exist $\Gamma_1, \Gamma_2$ such that $\Gamma = \Gamma_1 \circ \Gamma_2$, and $\Gamma'_1, \Gamma'_2, \Gamma''_1, \Gamma''_2$ such that:

\[
\begin{align*}
\text{(T-Brch) } & \forall j \in I \quad \Theta \cdot \Gamma''_{1j} \circ x_j: U''_{1j}, s[p]: S'_{j} \vdash Q_{1j}'' \\
\text{(T-MSub) } & \Theta \cdot \Gamma_1 + s[p]|q|k_j \in T(j(x_j), Q_{1j}) \\
& \Gamma_2 \vdash v: U'' \quad \Theta \cdot \Gamma_2'' + s[q]: S'' \vdash Q''_2 \\
& \Theta \cdot \Gamma_1 \circ \Gamma_2 \vdash Q_1 | Q_2 = s[p]|q|k_j \in T(j(x_j), Q_{1j}) | s[q]|p \oplus \{l_k(v)\}.Q_{2} \\
& \Gamma''_1 = \Gamma''_{1j}, s[p]: S_p \quad \text{where } S_p = q|k_j \in T(j(x_j), Q_{1j}) \\
& \Gamma''_2 = \Gamma_1 \circ \Gamma''_2, s[q]: S_q \quad \text{where } S_q = p \oplus \{l_k(u''), S''\} \quad \text{for some } k \in I
\end{align*}
\]

(42) (43) (44)
Notice that:

$$\Gamma = (\Gamma'_1, s[p]; S'_k) \circ (\Gamma'_2 \circ \Gamma'_2, s[q]; S''_q)$$

where

$$\begin{align*}
\Gamma'_1 & \subseteq \Gamma'_2, S'_p \subseteq S'_p, \\
\Gamma'_2 & \subseteq \Gamma_v, \Gamma'_2 \subseteq \Gamma'_2, S''_q \subseteq S''_q
\end{align*}$$

(by (42), (43), (44))  

From the consistency of $$\Gamma = \Gamma_1 \circ \Gamma_2$$ and since $$\Gamma_1 \subseteq \Gamma'_2$$ and $$\Gamma_2 \subseteq \Gamma'_2$$ with $$\Gamma_v \vdash v: U''$$, we also have:

$$U'' \subseteq U'_k$$ (from (43) and (44))  

Now, let:

$$\Gamma' = \Gamma'_1 \circ \Gamma'_2$$

where

$$\Gamma'_1 = \Gamma'_1 \circ \Gamma_v, s[p]; S'_k$$ and $$\Gamma'_2 = \Gamma'_2 \circ \Gamma'_2$$

(47)

Before proceeding, we prove $$\Gamma \rightarrow \Gamma'$$ (and therefore, $$\Gamma \rightarrow^{*} \Gamma''$$):

1. we first observe that:

$$s[p]; S'_k, s[q]; S''_q \rightarrow s[p]; S'_k, s[q]; S''_q$$

(48)  

since $$\Gamma$$ is consistent by hypothesis, and therefore unf ($$S'_p$$) | q and unf ($$S''_q$$) | p have at least $$t_k$$ in common, with compatible payload types as per Def. 2.15);  

2. then, let:

$$\Gamma_1 = \Gamma'_1 \circ \Gamma_v \circ \Gamma'_2$$

(i.e., $$\Gamma'_1$$ and $$\Gamma''_1$$ are respectively $$\Gamma$$ and $$\Gamma''$$ without their entries for $$s[p], s[q]$$). We can prove the following statement:

$$s[p]; S'_k, s[q]; S''_q, \Gamma_1 \rightarrow s[p]; S'_k, s[q]; S''_q, \Gamma'_1$$

(and thus, $$\Gamma \rightarrow^{*} \Gamma'$$)  

(49)

by induction on the size of $$\Gamma_1$$ (which is also the size of $$\Gamma''_1$$): the base case ($$\Gamma_1 = \Gamma'_1 = \emptyset$$) follows by (48), while in the inductive case we apply the induction hypothesis, and use the subtyping relations in (45) to conclude by the inductive rule of Def. 2.15.

We can now continue proving the main statement, observing:

$$\Gamma'$$ is consistent  

(by (49) and Lemma B.22)  

(50)

$$\Theta \cdot \Gamma''_1 \cdot x_k: U_k, s[p]; S'_k \vdash Q''_{ik} (i \in k)$$

(from (42), premise of (T-Branch))  

(51)

$$\Theta \cdot \Gamma''_1 \cdot x_k: U''_k, s[p]; S'_k \vdash Q''_{ik} (i \in k)$$

(by (51), (46) and (T-Sub))  

(52)

$$\Gamma_v \vdash v: U''_k$$

(from (42), premise of (T-Select))  

(53)

$$\Gamma''_1, s[p]; S'_k \circ \Gamma_v$$ is consistent

(by (50), (47) and Cor. B.17)  

(54)

$$\Theta \cdot \Gamma''_1, s[p]; S'_k \circ \Gamma_v \vdash Q''_{ik} (i' \in k')$$

(by (52), (53), (54) and Lemma B.26)  

(55)

Therefore, by (47), using (55) and the remaining premise of (T-Select) in (42), we conclude by typing the reduct in (41) as follows:

$$\Theta \cdot \Gamma \vdash P \rightarrow \Theta \cdot \Gamma' \vdash Q''_{ik} (i' \in k') \mid Q''_2 = P'$$

(T-Par)

= base case (R-Call). We have:

$$P = \text{def } X(x_1, \ldots, x_n) = Q_X \text{ in } (X(v_1, \ldots, v_n) \mid Q) \rightarrow \text{def } X(x_1, \ldots, x_n) = Q_X \text{ in } (Q_X \{v_i/x_i\}, i \in \{1 \ldots n\} \mid Q) = P'$$
Let \( \widetilde{x} = x_1, \ldots, x_n \) and \( \widetilde{U} = U_1, \ldots, U_n \). By inversion of (T-Res), (T-Par) and (T-Call), allowing a (possibly vacuous) instance of (T-MSUB) as per Proposition B.30, we have \( \Gamma = \Gamma_1 \circ \Gamma_2 \) with \( \Gamma_1 \leq \Gamma_2 \), \( \Gamma_1 = \Gamma_1^{\top} \circ \cdots \circ \Gamma_{\top,n}^{\top} \), such that:

\[
\begin{align*}
\Theta, X: \widetilde{U} : \widetilde{x} & : U \vdash Q_X \\
\forall i \in \{1\ldots n\} & \quad \Gamma_i^{\top} : U_i \\
\Theta, X: \widetilde{U} : \Gamma \vdash X(v_1, \ldots, v_n) & \quad \Gamma_1 \leq \Gamma_2^{\top}
\end{align*}
\]

\[
\Theta, X: \widetilde{U} \cdot \Gamma_1 \vdash X(v_1, \ldots, v_n) \quad \Theta, X: \widetilde{U} \cdot \Gamma_2 \vdash Q_X \\
\Theta \cdot \Gamma \vdash \text{def} X(x_1, \ldots, x_n) = Q_X \text{ in } (X(v_1, \ldots, v_n) \mid Q)
\]

Observe that from \( \Theta, X: \widetilde{U} : \widetilde{x} : U \vdash Q_X \), by applying Lemma B.26 \( n \) times (noticing that each time we get a consistent context) we obtain \( \Theta, X: \widetilde{U} : \Gamma_1^{\top} \vdash X\{v_i/s_i\}_{i \in \{1\ldots n\}} \), and thus \( \Theta, X: \widetilde{U} \cdot \Gamma_1 \vdash X\{v_i/s_i\}_{i \in \{1\ldots n\}} \) (by \( \Gamma_1 \leq \Gamma_2^{\top} \) and (T-MSUB)), and therefore:

\[
\begin{align*}
\Theta, X: \widetilde{U} \cdot \Gamma_1^{\top} & \vdash X\{v_i/s_i\}_{i \in \{1\ldots n\}} \\
\Theta, X: \widetilde{U} \cdot \Gamma_2 \vdash Q_X \\
\Theta \cdot \Gamma \vdash \text{def} X(x_1, \ldots, x_n) = Q_X \text{ in } (X\{v_i/s_i\}_{i \in \{1\ldots n\}} \mid Q) = P'
\end{align*}
\]

and we conclude by letting \( \Gamma' = \Gamma \);

= inductive case (R-Par). We have \( P = P_1 \mid P_2 \rightarrow P'_1 \mid P_2 = P' \), with \( P_1 \rightarrow P'_1 \) (from the rule premise). By inversion of (T-Par), we have \( \Gamma = \Gamma_1 \circ \Gamma_2 \) such that:

\[
\Theta \cdot \Gamma_1 \vdash P_1 \quad \Theta \cdot \Gamma_2 \vdash P_2
\]

By the induction hypothesis, \( \exists \Gamma' \) such that \( \Gamma_1 \rightarrow^* \Gamma_1' \) and \( \Theta, \Gamma_1' \vdash P'_1 \). By Cor. B.24, we have that \( \Gamma_1 \circ \Gamma_2 \) is consistent. Hence, we conclude by letting \( \Gamma' = \Gamma_1' \circ \Gamma_2 \), obtaining:

\[
\Theta \cdot \Gamma \vdash \Gamma_1' \quad \Theta \cdot \Gamma_2 \vdash P_2
\]

= inductive case (R-Res). We have \( P = (\nu s : \Gamma') P' \rightarrow (\nu s P)' = P' \), with \( P' \rightarrow P'' \), \( \Gamma' = \{s[p]:S_p\}_{p \in I} \) (for some \( I \)), and \( \Theta \cdot \Gamma \vdash P' \) (from the rule premise). By the induction hypothesis, \( \exists \Gamma'' \) such that:

\[
\Gamma \circ \Gamma' \rightarrow^* \Gamma'' \quad \text{and} \quad \Theta \cdot \Gamma'' \vdash P''
\]

By Proposition B.21, we know that \( \text{dom} (\Gamma'') = \text{dom} (\Gamma \circ \Gamma'') = \text{dom} (\Gamma \circ \{s[p]:S_p\}_{p \in I}) \), and therefore:

\[
\text{for some } \Gamma', \Gamma'' \text{ with } \text{dom} (\Gamma') = \text{dom} (\Gamma) \text{ and } \Gamma'' = \{s[p]:\Gamma''\}_{p \in I} \quad \Gamma'' = \Gamma' \circ \Gamma''
\]

Hence, we can rewrite the typing context reduction in (56) as:

\[
\Gamma \circ \Gamma' \rightarrow^* \Gamma'' \circ \Gamma''
\]

and therefore,

\[
\Theta \cdot \Gamma' \circ \Gamma'' \vdash P'' \quad \text{by (58), (57) and (56)}
\]

By Def. 2.12, the validity of the typing judgement in (59) implies that \( \Gamma' \circ \Gamma'' \) is consistent, and therefore, by Cor. B.24, \( \Gamma' \) is consistent. Hence, we conclude by:

\[
\Theta \cdot \Gamma' \circ \Gamma'' \vdash P''
\]
inductive case (R-Def). We have $P = \text{def } X(\bar{x}) = Q_X \text{ in } Q \rightarrow \text{def } X(\bar{x}) = Q_X \text{ in } Q' = P'$, with $Q \rightarrow Q'$ (from the rule premise). By inversion of (T-Def), we get:

\[
\begin{align*}
\frac{
\Theta : X : \bar{U} \cdot \bar{x} : \bar{U} \vdash Q_X \quad \Theta : X : \bar{U} \cdot \Gamma \vdash Q
}{
\Theta \cdot \Gamma \vdash \text{def } X(\bar{x}) = Q_X \text{ in } Q}
\end{align*}
\]

By the induction hypothesis, $\exists \Gamma' : \Gamma \rightarrow^* \Gamma'$ and $\Theta : X : \bar{U} \cdot \Gamma' \vdash Q'$, and we conclude by:

\[
\begin{align*}
\frac{
\Theta : X : \bar{U} \cdot \bar{x} : \bar{U} \vdash Q_X \quad \Theta : X : \bar{U} \cdot \Gamma' \vdash Q'
}{
\Theta \cdot \Gamma' \vdash \text{def } X(\bar{x}) = Q_X \text{ in } Q' = P'}
\end{align*}
\]

inductive case (R-Struct). We have $P \equiv Q$ and $Q' \equiv P'$, with $Q \rightarrow Q'$ (from the rule premise). By Proposition B.25, $\Theta \cdot \Gamma \vdash Q$; by the induction hypothesis, $\exists \Gamma' : \Gamma \rightarrow^* \Gamma'$ and $\Theta \cdot \Gamma' \vdash Q'$; by Proposition B.25, we conclude $\Theta \cdot \Gamma' \vdash P'$.

\[\Box\]
C Proofs for § 4

Lemma 4.2. \( \text{unf}(T) = \text{unf}(\bar{\mu}T) \).

Proof. We first show that the LHS of the statement is defined iff the RHS is defined, too. Obviously, \( \bar{\mu}T \) is defined iff \( \text{unf}(\bar{\mu}T) \) is defined. Moreover, by Def. 4.1, \( \bar{\mu}T \) is defined iff \( T \) is a (possibly recursive) linear input/output type, or \( \bullet \); hence, \( \bar{\mu}T \) is defined iff \( \text{unf}(\bar{\mu}T) \) is a (non-recursive) linear input/output type, or \( \bullet \); this implies that \( \bar{\mu}T \) is defined iff \( \text{unf}(\bar{\mu}T) \) is defined. Summing up: \( \text{unf}(T) \) is defined iff \( \text{unf}(\bar{\mu}T) \) is defined.

Let us now assume that \( \bar{\mu}T \) is defined. If \( T \) is not a \( \mu \)-type, i.e., \( T \neq \mu t.T' \) (for some \( T' \)), the statement holds trivially by Def. 4.1: in fact, we have \( \text{unf}(T) = \bar{\mu}T = \text{unf}(\bar{\mu}T) \). Otherwise, when \( T = \mu t.T' \), by Def. 4.1 we have \( \bar{\mu}T = \mu t.T' = \mu t.T' \{v_k/h\} \). Let us examine the one-step unfolding of \( T \) (i.e., we do not (yet) unfold \( \bar{\mu}T \) if it is a \( \mu \)-type):

\[
\begin{align*}
\bar{\mu}T \{v_k/h\} &= \mu t.T' \{v_k/h\} \\
\bar{\mu}T' \{v_k/h\} &= \bar{\mu}T' \{v_k/h\} \\
\bar{\mu}T &= \bar{\mu}T'
\end{align*}
\]

We can observe that if we dualise the one-step unfolding of \( T = \mu t.T' \) (i.e., if we dualise \( T' \{v_k/h\} \)), we get the same result:

\[
\begin{align*}
\bar{\mu}T' \{v_k/h\} &= \bar{\mu}T' \{v_k/h\} \\
\bar{\mu}T &= \bar{\mu}T'
\end{align*}
\]

Now, if we take \( \bar{\mu}T \{v_k/h\} \) and its dual \( \bar{\mu}T' \{v_k/h\} \), we can repeat the reasoning above; we can further iterate along all the successive one-step unfoldings, until we reach a non-\( \mu \)-type: at each step, the one-step unfolding of the dualised type matches the dual of the one-step-unfolded type. Hence, we conclude \( \text{unf}(T) = \text{unf}(\bar{\mu}T) \).

Definition C.1. The relation \( =_\pi \) for \( \pi \)-types is coinductively defined as:

\[
\begin{align*}
B =_\pi B & \quad \bullet =_\pi \bullet \\
T =_\pi T' & \quad T =_\pi \bar{\mu}T \\
\Pi_i(T) =_\pi \Pi_i(T') & \quad \Pi_i(T) =_\pi \bar{\mu}T \\
\bar{\Pi}_i(T) =_\pi \bar{\Pi}_i(T') & \quad \bar{\Pi}_i(T) =_\pi \bar{\mu}T \\
\forall i \in I \ T_i =_\pi T'_i & \quad \forall i \in I \ T_i =_\pi \bar{\mu}T'_i \\
\langle T_i \rangle_{i \in I} =_\pi \langle T'_i \rangle_{i \in I} & \quad \langle T_i \rangle_{i \in I} =_\pi \bar{\mu}T'_i \\
\bar{\langle T_i \rangle}_{i \in I} =_\pi \bar{\langle T'_i \rangle}_{i \in I} & \quad \bar{\langle T_i \rangle}_{i \in I} =_\pi \bar{\mu}T'_i \\
\bar{\Pi}_i(T) =_\pi \bar{\Pi}_i(T') & \quad \bar{\Pi}_i(T) =_\pi \bar{\mu}T \\
\mu T =_\pi \mu T' & \quad \mu T =_\pi \mu T'
\end{align*}
\]

Remark C.2. Def. C.1 is actually stronger than required for Lemma 4.4: it implies \( \leq_\pi \cap \leq_{\pi^{-1}} \) (see Proposition C.4 below), but restricts unfolding of recursion so that related types can only unfold “in unison” (by rule \( \leq_{\pi \cdot \mu} \)).

Proposition C.3. \( =_\pi \) is reflexive.

Proposition C.4. If \( T =_\pi T' \), then \( T \leq_\pi T' \) and \( T' \leq_\pi T \).

Proof. We first prove the thesis for \( T \leq_\pi T' \). Consider the following relation:

\[
\begin{align*}
\mathcal{R} &= \mathcal{R}_1 \cup \mathcal{R}_2 \\
\mathcal{R}_1 &= \{ (T, T') \mid T =_\pi T' \} \\
\mathcal{R}_2 &= \{ (T \{v_k/h\}, \mu t.T'), (\mu t.T, T' \{v_k/h\}) \mid \mu t.T =_\pi \mu t.T' \}
\end{align*}
\]

We can easily prove that \( \mathcal{R} \) is closed backwards under the rules in Def. 3.5. For all \( (T_1, T_2) \in \mathcal{R} \), we have either:
We prove that $(T_1, T_2) \in R_1$. Then, we proceed by cases on the coinductive rule in Def. C.1 concluding $T_1 = T_2$. Most cases are straightforward: we show that they satisfy a corresponding rule in Def. 3.5, and the relations in the coinductive premises involve pairs of elements $(T_1, T_2)$ that belong to $R_1 \subseteq R$. The only exception is:

- $(e_{=\mu} \cdot L \cdot \mu)$. Then, $T_1 = \mu t. T$ and $T_2 = \mu t'. T'$, and we have to show that $R$ satisfies both rules $(S-L \cdot L \cdot \mu)$ and $(S-L \cdot \mu)$ in Def. 3.5; we conclude observing that the required pair $(T\{v^T\}, \mu t'. T')$ and $(\mu t. T, T'\{v^T\})$ belong to $R_2 \subseteq R$.

- $(T_1, T_2) \in R_2$. We have either:
  - $T_1 = T\{v^T\}$ and $T_2 = \mu t'. T'$. We have to satisfy rule $(S-L \cdot \mu R)$; we conclude observing that the required pair of types $(T\{v^T\}, T'\{v^T\})$ belongs to $R_1 \subseteq R$, by $(e_{=\mu} \cdot L \cdot \mu)$;
  - $T_1 = \mu t. T$ and $T_2 = T'\{v^T\}$. Similar to the previous case: we have to satisfy rule $(S-L \cdot \mu L)$, and conclude by observing that the required pair belongs to $R_1 \subseteq R$, by $(e_{=\mu} \cdot L \cdot \mu)$.

Summing up, we have shown that $R$ is closed backwards under the rules for $\leq_{\pi}$; and since $\leq_{\pi}$ is the largest relation closed backwards under such rules, we have $R \subseteq \leq_{\pi}$. Hence, since $T =_{\pi} T'$ implies $(T, T') \in R_1 \subseteq R$, we conclude that $T =_{\pi} T'$ implies $T \leq_{\pi} T'$.

The proof of the statement for $T' \leq_{\pi} T$ is symmetric. ▶

Lemma 4.4 (Erasure of $T$). $\mu t. T =_{\pi} \mu t. T\{v^T\} / t$, for all $t' \notin \text{fv}(T)$.

Proof. Let $T' = T\{v^T\} / t$ (for some $t' \notin \text{fv}(T)$), and consider the following relation:

\[
R = R_{\mu} \cup R_{\pi} \cup R_{\pi} \cup R_{\pi}
\]

\[
R_{\mu} = \{ (\mu t. T, \mu t. T'), (T\{v^T\}, T'\{v^T\}) \}
\]

\[
R_{\pi} = \{ (T\{v^T\}, T_B\{v^T\}) \mid T\{v^T\} =_{T_B} T\{v^T\} \}
\]

\[
R_{\pi} = \{ (T\{v^T\}, T_B\{v^T\}) \mid T\{v^T\} =_{T_B} T\{v^T\} \}
\]

\[
R_{\pi} = \{ (T\{v^T\}, T_B\{v^T\}) \mid T\{v^T\} =_{T_B} T\{v^T\} \}
\]

We prove that $R$ is closed backwards under the rules obtained from Def. C.1 by replacing each occurrence of $=_{\pi}$ with $R$. For each pair of types $(T_1, T_2) \in R$, we have the following cases:

- $(T_1, T_2) \in R_{\mu}$. We have the following sub-cases:
  - $T_1 = \mu t. T$ and $T_2 = \mu t'. T'$. We need to satisfy rule $(e_{=\mu} \cdot L \cdot \mu)$: we conclude by noticing that $(T\{v^T\}, T'\{v^T\}) \in R_1 \subseteq R$;
  - $T_1 = T\{v^T\}$ and $T_2 = T'\{v^T\} = T\{v^T\} / t$. Since $T\{v^T\} =_{T_B} T\{v^T\}$ (by reflexivity of $\pi$, Proposition C.3), by definition of $R_\pi$, we have $(T_1, T_2) \in R_\pi$; we study this case below;

- $(T_1, T_2) \in R_{\pi}$. We have $T_1 = T_A\{v^T\}$ and $T_2 = T_B\{v^T\} / t$, for some $T_A, T_B$ such that:

\[
T_A\{v^T\} =_{T_B} T_B\{v^T\}\quad(60)
\]

We proceed by cases on the rule in Def. C.1 concluding (60), examining the possible shapes of $T_A$ and $T_B$, and showing that $R_{\pi}$ is closed backwards under the same rule. Case $(e_{=\mu} \cdot L \cdot \mu)$ is trivial, and most other cases are simple: by the coinductive premises of the selected rule, we obtain one or more relations of the form $T_A\{v^T\} =_{T_B} T_B\{v^T\}$ (for some $T_A, T_B$), and in each case we conclude that $(T_A\{v^T\}, T_B\{v^T\}) \in R_{\pi}$ belongs to $R_{\pi}$. The only exception is when (60) is the conclusion of $(e_{=\mu} \cdot L \cdot \mu)$, and $T_A, T_B$ are either:

- $T_A = T_B = t$. In this case, we have $T_1 = \mu t. T$ and $T_2 = \mu t'. T'$, and by definition of $R_{\pi}$, $(T_1, T_2) \in R_{\mu}$; we study this case above;
We now characterise a confluent fragment of linear calculus: we will use it later on, to prove the operational correspondence of our encoding.

Definition C.5 (Quasi-linearity). The predicate \( \text{qlin}(T) \) is defined as:

\[
\begin{align*}
\text{qlin}(T) & \quad \forall i \in I \quad \text{qlin}(T_i), \text{qlin}(T_{i+1}) \quad \text{qlin}(\mu T) \\
\text{qlin}(\mu t T) & \quad \forall i \in I \quad \text{qlin}(T_i), \text{qlin}(T_{i+1}) \quad \text{qlin}(\mu t T)
\end{align*}
\]

We say that \( T \) is quasi-linear \( \iff \text{qlin}(T) \). We say that a typing judgement \( \Gamma \vdash P \) is quasi-linear \( \iff \) it has a derivation such that, for each \( \Gamma', x : T \vdash P' \) occurring in it, either: (a) \( \text{qlin}(T) \), or (b) \( T = \mu t T \) and \( P' \in \{Q, x : Q, x : Q, (Q, Q), Q, x : Q, Q, x : Q, (Q, Q)\} \) where \( x \) can occur in \( Q, Q' \) only as \( \pi(v) \) with \( x \notin \text{fv}(v) \). We say that \( P \) is quasi-linear \( \iff \exists \Gamma, P' \) such that \( P' \equiv P \) and \( \Gamma \vdash P' \) holds and is quasi-linear.
Intuitively, Def. C.5 says that if a type \( T \) is quasi-linear, then it does not harbour unrestricted communication capabilities. If a typed process \( P \) is quasi-linear, then each name \( x \) is quasi-linear (item (a)), or is unrestricted but used in a syntactically-constrained way (item (b)). The constraints of item (b) are quite standard, and ensure that \( x \) is uniformly \( \omega \)-receptive [54, §8.2]: for all synchronisations on \( x \), each transmitted value is processed immediately, and in the same way, by one process that spawns a new replica for each input on \( x \) — while \( x \) is only used for output elsewhere. Note that e.g. \( P' = x(y).0 \mid \ast(x(y).0) \) violates item (b), but is quasi-linear since \( P' \equiv \ast(x(y).0) \) (which satisfies the definition). Quasi-linearity is preserved along reductions (Proposition C.6) and implies confluence (Lemma C.7) — intuitively, because synchronisations are deterministic, as they can only involve linear names [35, Theorem 4.4.1], or \( \omega \)-receptive names.

\textbf{Proposition C.6.} If \( P \) is quasi-linear and \( P \xrightarrow{\cdot} P' \), then \( P' \) is quasi-linear.

\textbf{Lemma C.7} (Quasi-linear processes are confluent). If \( P \) is quasi-linear, \( P \xrightarrow{\cdot} P_1 \) and \( P \xrightarrow{\cdot} P_2 \), then either \( P_1 \equiv P_2 \) or \( \exists P_3 \) such that \( P_1 \xrightarrow{\cdot} P_3 \) and \( P_2 \xrightarrow{\cdot} P_3 \).

\textbf{Proposition C.6.} If \( P \) is quasi-linear and \( P \xrightarrow{\cdot} P' \), then \( P' \) is quasi-linear.

\textbf{Proof.} We first prove that:
\[
P \xrightarrow{\cdot} P' \quad \text{implies that} \quad P' \text{ is quasi-linear}
\]
We first observe that, by Def. C.5 \( P \) must be typed by some context \( \Gamma \), and there exist \( P_0 \equiv P \) such that \( \Gamma \vdash P_0 \) is quasi-linear. Since \( P_0 \xrightarrow{\alpha} P' \) (for some \( \alpha \)), by standard subject reduction on linear \( \pi \)-calculus [35, Theorem 4.3.1], there exists some \( \Gamma' \) (whose definition depends on \( \Gamma \) and \( \alpha \)) such that \( \Gamma' \equiv P' \). We then proceed by induction on the derivation of the transition \( P_0 \xrightarrow{\alpha} P' \):

\begin{itemize}
  \item in the base case of synchronisation with \( \alpha = x \) and \( x \) linear, we observe that \( \Gamma'(x) = \bullet \) and \( \forall y \in \text{dom}(\Gamma) \setminus \{x\} : \Gamma'(y) = \Gamma(y) \) (i.e., no new unrestricted communication capabilities are introduced); moreover, \( P' \) still respects item (b);
  \item in the base case of synchronisation with \( \alpha = x \) and \( x \) unrestricted, we observe that \( \Gamma' = \Gamma \) (i.e., no new unrestricted communication capabilities are introduced); then, we use item (b) of Def. C.5 to determine the shape of \( P_0 \), and verify that \( P' \) still respects item (b);
  \item in the base cases with \( \alpha \in \{\text{case, with, let}\} \), we have \( \Gamma' = \Gamma \) (i.e., no new unrestricted communication capabilities are introduced); then we verify that \( P' \) still respects item (b);
  \item the other inductive cases hold by applying the induction hypothesis.
\end{itemize}

We can now prove the main statement. Let \( n \) be the length of the sequence of transitions \( P \xrightarrow{\cdot} P' \); in the base case \( (n = 0) \) the statement holds trivially, while the inductive case \( (n = n' + 1) \) it follows by the induction hypothesis and (62). ▶

\textbf{Proposition C.8} (Linear synchronisations are confluent). If \( \Gamma, x: \mathbb{N}(T) \vdash P, P \xrightarrow{\cdot} P' \) and \( P \xrightarrow{\cdot} P'' \), then \( P' \equiv P'' \).

\textbf{Proof.} See [35, Theorem 4.4.1]. ▶

\textbf{Lemma C.9} (Quasi-linear synchronisations are confluent). If \( \Gamma, x: \mathbb{N}(T) \vdash P \) is quasi-linear, \( P \xrightarrow{\cdot} P_1 \) and \( P \xrightarrow{\cdot} P_2 \), then either \( P_1 \equiv P_2 \), or \( \exists P_3 \) such that \( P_1 \xrightarrow{\cdot} P_3 \) and \( P_2 \xrightarrow{\cdot} P_3 \).

\textbf{Proof.} We can only get a synchronisation on \( x \) if we have, by Def. C.5:
\[
P \equiv (\nu z) \left\{ x(y).Q \mid \tau(v).Q' \mid R \right\} \equiv (\nu z) \left\{ x(y).Q \mid \tau(v).Q' \mid \ast(x(y).Q) \mid R \right\}
\]
where \( x \notin z \) and \( Q, Q', R \) can only use \( x \) for output. Considering the synchronisation \( P \xrightarrow{\cdot} P_1 \), we get:
\[
P_1 \equiv (\nu z) \left\{ Q \mid \ast(x(y).Q) \mid R \right\}
\]
Let us now examine the synchronisation \( P \xrightarrow{\cdot} P_2 \). We have two cases:
if it coincides with the synchronisation $P \xrightarrow{s} P_1$, we trivially conclude $P_1 \equiv P_3$;

- otherwise, if a different synchronisation leads to $P_2$, we must have $R \equiv \pi(v').R' \mid R''$ (i.e., another output is enabled on $x$), and thus:

\[ P_1 \equiv (\nu z) (Q\{v/s\} | Q' \mid \pi(v').R' \mid R'') \equiv (\nu z) (Q\{v/s\} | \pi(v').R' \mid R'') \]

\[ P_2 \equiv (\nu z) (Q\{v'/s\} | \pi(v).Q' | R' \mid R'') \equiv (\nu z) (Q\{v'/s\} | \pi(v).Q' | R' \mid R'') \]

Therefore, letting $P_3 = (\nu z) (Q\{v/s\} | Q\{v'/s\} | Q' | R' \mid \pi(v).Q' | R' \mid R'')$, we conclude $P_1 \xrightarrow{s} P_3$ and $P_2 \xrightarrow{s} P_3$.

\[ \text{Corollary C.10 (Partial confluence). If } P \text{ is quasi-linear, } P \xrightarrow{s} P_1 \text{ and } P \xrightarrow{s} P_2 \text{ (for any } \alpha) \text{, then either } P_1 \equiv P_2 \text{ or } \exists P_3 \text{ such that } P_1 \xrightarrow{s} P_3 \text{ and } P_2 \xrightarrow{s} P_3. \]

\[ \text{Proof. (Sketch) The proof is similar to [35, Theorem 4.4.3], which assumes } x \text{ to be linearly-typed, and depends on Proposition C.8. The only difference is that in our statement, } x \text{ might be an unrestricted name; in this case, the result still holds by the quasi-linearity hypothesis, and by Lemma C.9.} \]

\[ \text{Proposition C.11. If } \alpha \in \{\text{with, case, let}\}, P \xrightarrow{s} P_1 \text{ and } P \xrightarrow{s} P_2 \text{ (for any } \beta) \text{, then either } P_1 \equiv P_2 \text{, or } \exists P_3 \text{ such that } P_1 \xrightarrow{s} P_3 \text{ and } P_2 \xrightarrow{s} P_3. \]

\[ \text{Proof. Let } \alpha = \text{with. We have: } \]

\[ P \equiv (\nu z) \left( \text{with } \left[ l_i : x_i \right]_{i \in I} = \left[ l_i : v_i \right]_{i \in I} \text{ do } P \mid Q \right) \xrightarrow{\text{with}} (\nu z) \left( P\{v_i/s_i\}_{i \in I} \mid Q \right) \equiv P_1 \]

Let us now examine the reduction $P \xrightarrow{s} P_2$. We have two cases:

- if it coincides with the reduction $P \xrightarrow{s} P_1$, we trivially conclude $P_1 \equiv P_2$;

- otherwise, if a different reduction leads to $P_2$, we must have $Q \xrightarrow{s} Q'$, and:

\[ P \xrightarrow{s} (\nu z) \left( \text{with } \left[ l_i : x_i \right]_{i \in I} = \left[ l_i : v_i \right]_{i \in I} \text{ do } P \mid Q' \right) \equiv P_2 \]

Therefore, letting $P_3 = (\nu z) \left( P\{v_i/s_i\}_{i \in I} \mid Q' \right)$, we conclude $P_1 \xrightarrow{s} P_3$ and $P_2 \xrightarrow{s} P_3$.

The proofs for $\alpha = \text{case}$ and $\alpha = \text{let}$ are similar.

\[ \text{Lemma C.7 (Quasi-linear processes are confluent). If } P \text{ is quasi-linear, } P \rightarrow P_1 \text{ and } P \rightarrow P_2, \text{ then either } P_1 \equiv P_2 \text{ or } \exists P_3 \text{ such that } P_1 \rightarrow P_3 \text{ and } P_2 \rightarrow P_3. \]

\[ \text{Proof. Assume that } P \text{ is quasi-linear. The statement follows from Cor. C.10 and Proposition C.11, which cover all possible transitions of } P. \]

\[ \text{Corollary C.12 (Quasi-linear processes are confluent (II)). If } P \text{ is quasi-linear, } P \rightarrow P_1 \text{ and } P \rightarrow P_2, \text{ then either } P_2 \rightarrow P_1, \text{ or } \exists P_3 \text{ such that } P_1 \rightarrow P_3 \text{ and } P_2 \rightarrow P_3. \]

\[ \text{Proof. Assume that } P \text{ is quasi-linear. Let } n \text{ be the length of the sequence of transitions } P \rightarrow P_1. \text{ We proceed by induction on } n:\]

- base case $n = 0$. We have $P_1 \equiv P_2$, and therefore conclude by letting $P_3 = P_2$;

- inductive case $n = n' + 1$. Take $P'_{n'}$ such that $P \rightarrow P'_{n'} \rightarrow P_1$, with $n'$ transitions in $P \rightarrow P'_{n'}$. By the induction hypothesis, either:

\[ P_2 \rightarrow P'_{n'} \text{. In this case, we conclude } P_2 \rightarrow P_1; \]

\[ \exists P'_{n'} \text{ such that } P'_{n'} \rightarrow P_2 \text{ and } P_2 \rightarrow P_3. \text{ In this case, notice that } P'_{n'} \text{ is quasi-linear (from } P \rightarrow P'_{n'} \text{ and by Proposition C.6); therefore, since } P'_{n'} \rightarrow P'_3 \text{ and } P'_3 \rightarrow P_1, \text{ by Lemma C.7 we have either:} \]

\[ * P'_3 \equiv P_1. \text{ In this case, from } P_2 \rightarrow P'_3 \text{ we conclude } P_2 \rightarrow P_1; \]

\[ * P'_3 \not\equiv P_1. \text{ In this case, from } P_2 \rightarrow P'_3 \text{ we conclude } P_2 \rightarrow P_1; \]
\* \exists P''_3 \text{ such that } P'_3 \rightarrow P''_3 \text{ and } P_1 \rightarrow P''_3. \text{ In this case, by letting } P_3 = P''_3, \text{ we conclude } P_1 \rightarrow P_3 \text{ and } P_2 \rightarrow P_3.

\begin{lemma}
\textbf{Lemma 4.7.} If \( T = T_1 \sqcap T_2 \), and \( T'_1 \sqcup T'_2 = T \), then either (a) \( T'_1 \leq_{\pi} T_1 \text{ and } T'_2 \leq_{\pi} T_2 \), or (b) \( T'_1 \leq_{\pi} T_2 \text{ and } T'_2 \leq_{\pi} T_1 \).
\end{lemma}

\textbf{Proof.} By Def. 4.6, \( T \) is defined only if either:
\begin{itemize}
\item \( T_1 = T_2 = T^* \), for some \( \text{un}(T^*) \). \text{ In this case, we also have } T = T'_1 = T'_2 = T^*, \text{ and we trivially obtain both items (a) and (b);}
\item \( T_1, T_2 \) are respectively a linear input and output type, or vice versa. \text{ Then, again by Def. 4.6, we have two sub-cases:}
\begin{itemize}
\item \( T_1 = \text{Lo}(T'_1), T_2 = \text{Li}(T'_2), \text{ and } T'_1 \leq_{\pi} T'_2. \text{ In this case, } T = \text{Li}(T'_1) \sqcup \text{Lo}(T'_2) = \text{L}(T'_1), \text{ and we have either:}
\begin{itemize}
\item \( T'_1 = \text{Lo}(T'_1) \text{ and } T'_2 = \text{Li}(T'_2). \text{ Then, we conclude } T'_1 \leq_{\pi} T_1 \text{ and } T'_2 \leq_{\pi} T_2, \text{ i.e., case (a) of the statement;}
\item \( T'_1 = \text{Li}(T'_1) \text{ and } T'_2 = \text{Lo}(T'_2). \text{ Then, we conclude } T'_1 \leq_{\pi} T_2 \text{ and } T'_2 \leq_{\pi} T_1, \text{ i.e., case (b) of the statement;}
\end{itemize}
\item \( T_1 = \text{Li}(T'_1), T_2 = \text{Lo}(T'_2), \text{ and } T = T'_2 \leq_{\pi} T'_1. \text{ The proof is similar to the previous case.}
\end{itemize}
\end{itemize}
\end{itemize}
D Properties of Encoding of Types

D.1 Auxiliary Results

 Proposition D.1. Let $H, H'$ be partial session types. Then, $H \leq_P H'$ iff $\overline{H} \leq_P \overline{H}$.
Proof. Follows by the standard properties of duality for binary session types [21].

 Definition D.2. type($p, [p : T, q : T_s]_{s \in i}$) = $T$.

D.2 Subtyping and Encoding

 Theorem 6.2 (Encoding preserves subtyping). If $S \leq_S S'$, then $[S] \leq [S']$.

Proof. By Proposition B.11, since $S \leq_S S'$, then $\text{roles}(S) = \text{roles}(S')$. We construct a relation $\mathcal{R} \triangleq \mathcal{R}_S \cup \mathcal{R}_T \cup \mathcal{R}_V \cup \mathcal{R}_P \cup \mathcal{R}_U \cup \mathcal{R}_o$, where its subcomponents are defined as follows:

$$\mathcal{R}_S \triangleq \{(S', S) \mid S \leq_S S'\}$$
$$\mathcal{R}_T \triangleq \{(T', T) \mid \exists q \text{ such that type}(q, [p : T_p]_{p \in T}) = T \text{ and type}(q, [p : T'_{p \in T}]_{p \in T}) = T'\}$$
$$\mathcal{R}_V \triangleq \{(V', U) \mid U \leq_S U'\}$$
$$\mathcal{R}_P \triangleq \{(P', H') \mid P \leq_P P'\}$$
$$\mathcal{R}_U \triangleq \{(U, T_2) \mid (Li(T_1), Li(T_2)) \in \mathcal{R}_P\}$$
$$\mathcal{R}_o \triangleq \{(T_1, T_2) \mid (Lo(T_1), Lo(T_2)) \in \mathcal{R}_P\}$$

We first prove that $\mathcal{R}$ is closed backwards under the rules of $\leq_S$, given by Def. 3.5. We examine all the elements of $\mathcal{R}$, by inspecting all its subsets.

For each pair $([U], [U']) \in \mathcal{R}_U$ we have the following cases:

- $U \leq_B U'$ meaning that types $U, U'$ are basic types. Since the encoding of basic types is the identity function, then by subtyping $\leq_B$ we conclude that the pair $([U], [U'])$ satisfies rule (S-LB).

- In all other cases $U, U'$ must be closed session types and thus $([U], [U']) \in \mathcal{R}_S$: we study this case below.

For each pair $([S], [S']) \in \mathcal{R}_S$, we know that $S \leq_S S'$, and recalling that they are closed session types, we have the following cases, depending on the coinductive rule in Def. 2.10 concluding $S \leq_S S'$:

- Case (S-End). We have $[S] = [S'] = \bullet$. Hence, we conclude that the pair $([S], [S']) = (\bullet, \bullet)$ satisfies rule (S-End).

- Case (S-µL). We have $S = \mu t.S'' \leq_S S'$; hence, by Def. 5.1, $[S] = \mu t.T''$, where $T'' = [S']$. By the premise of (S-µL) we also have $S'' \{\mu t. S'' / k\} \leq_S S'$, which implies:

$$\{[S''], \{\mu t. S'' / k\}\} \in \mathcal{R}_S \subseteq \mathcal{R} \ (63)$$

Now, we observe:

$$[S''] \{\mu t. S'' / k\} = [S''] \{\mu t. S'' / k\} \ (\text{by Lemma D.5})$$
$$= [S''] \{\mu t. S'' / k\} \ (\text{by Def. 5.1})$$
$$= T'' \{\mu t. T'' / k\} \ (\text{since } T'' = [S''])$$

From (63) we also have $\{T'', \{\mu t. T'' / k\}\} \in \mathcal{R}_S \subseteq \mathcal{R}$. Hence, the pair $([S], [S']) = (\{T'', \{\mu t. T'' / k\}\} \subseteq [S, S'])$ satisfies rule (S-µL).

- Case (S-µR). Similar to case (S-µL), except that this time we have $S' = \mu t. S''$. Then, we let $T'' = [S'']$ and we obtain the pair $([S], [S']) = ([S], \mu t. T'')$, which satisfies rule (S-Rµ).
we have that
only case to consider is rule (H
H,H

such that
A. Scalas, O. Dardha, R. Hu, N. Yoshida XX:55

By Def.5.1 and by Def.D.2 we have that
For each pair
therefore
Case (H
Case (H
Cases (H

LI
Let I = roles(S) = roles(S'). For all p ∈ I, we have (T_p, T'_p) ∈ R_T ⊆ R. We conclude that the pair ([S], [S']) satisfies rule (S-TUPLE).

For each pair (T, T') ∈ R_T there is corresponding pair ([p: T_p]_{p ∈ I}, [p: T'_p]_{p ∈ I}) ∈ R_S and there exists q such that type(q, [p: T_p]_{p ∈ I}) = T and type(q, [p: T'_p]_{p ∈ I}) = T'; hence, there are S and S' such that S ⊆ S' and

Let I = roles(S) = roles(S'). By Def.5.1 we have that for all p ∈ I,

By Def.5.1 and by Def.D.2 we have that T = [S | q] and T' = [S' | q]. By Proposition B.12, since S ⊆ S', then for all p ∈ S also S \ p ⊆ S' \ p. In particular, since q ∈ S, also S \ q ⊆ S' \ q. Then, we have that ([S] | q, [S] | q) ∈ R_P: this case is studied below.

For each pair ([H], [H']) ∈ R_P, we know that H ⊆ H'. We proceed by cases on the coinductive rule in Def.2.10 that concludes H ⊆ H':

- Case (S-ParEnd). We have H = H' = end, and therefore, [H] = [H'] = ●. We conclude that the pair ([H], [H']) = (●, ●) satisfies rule (S-END).

- Case (S-ParHist). We have H = µt.H'' ⊆ H'; hence, by Def.5.1, [H] = µt.T'', where T'' = [H'']. By the premise of (S-ParHist) we also have H''{µt.H''/t} ⊆ P H', implying:

\[(H''{µt.H''/t}, [H'']) ∈ R_P \tag{64} \]

Now, observe:

\[\begin{align*}
[H''{µt.H''/t}] & = [H'']{µt.H''/t} & \text{(by Lemma D.6)} \\
[H'']{µt.H''/t} & = [H'']{µt.T''/t} & \text{(by Def.5.1)} \\
[H'']{µt.T''/t} & = T''{µt.T''/t} & \text{(since T'' = [H''])}
\end{align*} \]

From Equation (64) we have (T''{µt.T''/t}, [H'']) ∈ R_P. Hence, the pair ([H], [H']) = (µt.T'', [H'']) satisfies rule (S-LµHist).

- Case (S-ParHistR). Symmetrical to case (S-ParHist), except that we have H' = µt.H'': we let T'' = [H''], and we obtain that the pair ([H], [H']) = ([H], µt.T'') satisfies rule (S-LµHist).

- Cases (S-ParBrch) and (S-ParSel). In these cases, we have either:

  - Case (S-ParBrch). In this case, for some T_1, T_2, we have [H] = Li(T_1) and [H'] = Li(T_2), and therefore (T_1, T_2) ∈ R_i ⊆ R. We conclude that the pair ([H], [H']) = ([H], Li(T_2)) satisfies rule (S-Li).

  - Case (S-ParSel). In this case, for some T_1, T_2, we have [H] = Lo(T_1) and [H'] = Lo(T_2), and therefore (T_1, T_2) ∈ R_o ⊆ R. We conclude that the pair ([H], [H']) satisfies rule (S-Lo).

  For each pair (T_1, T_2) ∈ R_i, there is a corresponding pair (Li(T_1), Li(T_2)) ∈ R_P, and there exist H, H' such that

\[\begin{align*}
[H] & = Li(T_1) \\
[H'] & = Li(T_2) \tag{65}
\end{align*} \]

and H ⊆ H'. Equation (65) and Def.5.1 imply that H and H' are partial branch types. So, the only case to consider is rule (S-ParBrch) and we have that:

\[\begin{align*}
H & = & \ni_{i \in I} U_i \cdot H_i & \\
H' & = & \ni_{i \in I \cup J} U_i \cdot H_i'
\end{align*} \]
By the premise of (S-ParBrch), for all \(i \in I\) it is the case that \(U_i \subseteq S U_i^\prime\) and \(H_i \leq_p H_i^\prime\). Then, for all \(i \in I\):

\[
\left(\left\{U_i^\prime, U_i^\prime\right\}\right) \in R_U \subseteq R \quad \quad \quad \quad \quad \left(\left\{H_i^\prime, H_i^\prime\right\}\right) \in R_P \subseteq R
\]

By the encoding of partial branch types:

\[
[H] = \text{Li}(T_1)\quad \quad \quad \quad \quad \quad \quad \quad H^\prime = \text{Li}(T_2)
\]

We can conclude that the pair \((T_1, T_2)\) satisfies rule (S-VARIANT).

For each pair in \((T_2, T_1)\) there is a corresponding pair \((\text{Lo}(T_1), \text{Lo}(T_2))\) \(\in R_P\) and there exist \(H, H^\prime\) such that

\[
[H] = \text{Lo}(T_1) \quad \quad \quad \quad \quad \quad H^\prime = \text{Lo}(T_2)
\]

and \(H \leq_p H^\prime\). Equation (66) and Def.5.1 imply that \(H\) and \(H^\prime\) are partial select types. So, the only case to consider is rule (S-ParSel) and we have that:

\[
H = \oplus_{i \in I,J} U_i, H_i^I \quad \quad \quad \quad \quad H^\prime = \oplus_{i \in I,J} U_i^I, H_i^I
\]

We have thus proved that \(R\) is closed backwards under the rules of \(\leq_p\) — and since \(\leq_p\) is the largest relation closed backwards under such rules, this implies \(R \subseteq \leq_p\). We prove the main statement observing that, since \(S \leq S^\prime\) implies \([S], [S^\prime]\) \(\in R_P \subseteq R \subseteq \leq_p\), then \(S \leq S^\prime\) implies \([S] \leq_p [S^\prime]\).

\begin{itemize}
  \item Corollary D.3. \(H \leq_p H^\prime\), then \([H] \leq_p [H^\prime]\).
\end{itemize}

Proof. Assume \(H \leq_p H^\prime\). The statement follows by the proof of Theorem 6.2, where the relation \(R_P\) contains the pair \(([H], [H^\prime])\); this implies \(([H], [H^\prime])\) \(\in R_P \subseteq R \subseteq \leq_p\), and therefore, \([H] \leq_p [H^\prime]\).

\begin{itemize}
  \item D.2.1 Duality and Encoding
\end{itemize}

\begin{itemize}
  \item Lemma D.4. \(H\) be a (possibly open) partial session type. Then, \(\overline{[H]}\) \(\subseteq \overline{[H]}\).
\end{itemize}

Proof. Simple induction on the structure of \(H\). We use Def. 4.1 and the substitution of dualised variables.

\begin{itemize}
  \item Theorem 6.1 (Encoding preserves duality). \(\overline{[H]} = [H]\).
\end{itemize}

Proof. We prove a more general statement: let \(H\) be a (possibly open) partial session type. Then \(\overline{[H]} = \overline{[H]}\). The case for a closed partial type \(H\) follows as a corollary by the fact that \(\text{fv}(H) = \emptyset\) (and thus, the substitution applied on \(\overline{[H]}\) is vacuous).

The proof proceeds by induction on the structure of \(H\).

\begin{itemize}
  \item \(\overline{H} = \text{end}\). By Def. 2.8 we have that \(\overline{\text{end}} = \text{end}\). We conclude by Def. 5.1 and by Def. 4.1 and the fact that \(\text{fv}(\text{end}) = \emptyset\).
\end{itemize}
\[ H = t. \]
By Def. 2.8 we have that \( \bar{t} = t. \) By Def. 5.1 we have \( \{t\} = \{t\} = t. \) By Def. 4.1 we have that \( \{t\} = \{\bar{t}\} = \bar{t}. \) Then \( \{\bar{t}\} = \bar{t}. \) which concludes this case.

\[ H = \bigoplus_{i \in I} ?!_i(U_i).H. \]
By Def. 2.5 we know that each \( U_i \) is either a base type or a closed session type; hence \( \text{fv}(H) = \cup_{i \in I} \text{fv}(H_i). \) By Def. 2.8 we have that \( \bar{H} = \bigoplus_{i \in I} ?!_i(U_i).\bar{H}. \) By Def. 5.1 we have that \( \bar{H} = \bigoplus_{i \in I} \bar{H}_i \).

By induction hypothesis for all \( i \in I, \bar{H}_i = \bigoplus_{i \in \text{fv}(H_i)} \{t\}_i \). We rewrite the above as

\[ \bar{H} = \bigoplus_{i \in \text{fv}(H_i)} \{t\}_i \]

By Lemma D.4 we conclude

\[ \bar{H} = \bigoplus_{i \in \text{fv}(H_i)} \{t\}_i \]

On the other hand, by Def. 5.1 we have that \( \bar{H} = \bigoplus_{i \in I} ?!_i(U_i).H_i \).

By Def. 4.1 we have \( \bar{H} = \bigoplus_{i \in \text{fv}(H_i)} \{t\}_i \). We rewrite the above as

\[ \bar{H} = \bigoplus_{i \in \text{fv}(H_i)} \{t\}_i \]

By comparing Equation (67) and Equation (68) we conclude this case.

\[ H = \bigoplus_{i \in I} ?!_i(U_i).H_i. \]
By Def. 2.5 we know that each \( U_i \) is either a base type or a closed session type; hence \( \text{fv}(H) = \cup_{i \in I} \text{fv}(H_i). \) By Def. 2.8we have that \( \bar{H} = \bigoplus_{i \in I} ?!_i(U_i).\bar{H}. \) By Def. 5.1 we have that \( \bar{H} = \bigoplus_{i \in I} \bar{H}_i \).

By induction hypothesis for all \( i \in I, \bar{H}_i = \bigoplus_{i \in \text{fv}(H_i)} \{t\}_i \). We rewrite the above as

\[ \bar{H} = \bigoplus_{i \in \text{fv}(H_i)} \{t\}_i \]

By comparing Equation (69) and Equation (70) we conclude this case.

\[ H = \mu t'.H'. \]
We have that \( \text{fv}(H) = \text{fv}(H') \setminus \{t'\}. \) By Def. 2.8 we have that \( \bar{H} = \mu t'.\bar{H}'. \) By Def. 5.1 we have that \( \mu t'.\bar{H}' = \mu t'.\bar{H}'. \) By induction hypothesis \( \bar{H}' = \bigoplus_{i \in \text{fv}(H')} \{t\}_i \).

This implies,

\[ \bar{H}' = \mu t'.\bar{H}'. \]

On the other hand, \( \bar{H}' = \bigoplus_{i \in \text{fv}(H')} \{t\}_i \). By Def. 4.1 we have that

\[ \bar{H}' = \bigoplus_{i \in \text{fv}(H')} \{t\}_i \]

We have the following:

\[ \bigoplus_{i \in \text{fv}(H')} \{t\}_i = (\bigoplus_{i \in \text{fv}(H')} \{t\}_i) \setminus \{t'\} \]

(by Equation (72) and the fact that \( \text{fv}(H) = \text{fv}(H') \setminus \{t'\} \))

(\text{since } \text{fv}(H') \setminus \{t'\} \cup \{t'\} = \text{fv}(H'))

(by Equation (71))

which concludes this case.
D.2.2 Substitution and Encoding

Lemma D.5. Let $S, S'$ be session types. Then, $[S\{s'/t\}] = [S]\{s'/t\}$.

Proof. By induction on the structure of $S$. ▶

Lemma D.6. Let $H, H'$ be partial session types. Then, $[[H\{u'/t\}] = [H]\{u'/t\}$.


The following lemma gives the relation between the type combinator $\uplus$ and the standard ',' operator in linear $\pi$-typing contexts.

Lemma D.7. If $\Gamma_1 \uplus \Gamma_2$ is defined and $\text{dom}(\Gamma_1) \cap \text{dom}(\Gamma_2) = \emptyset$, then also $\Gamma_1, \Gamma_2$ is defined.

Proof. Immediate by the combination of typing contexts given in Def. 3.6. ▶

The following definition is an extends Def. 2.11 to accommodate the notion of linear session typing context.

Definition D.8 (Linear and Unrestricted Session Typing Context). 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 $\text{end}$, otherwise we say that $\Gamma$ is linear, $\text{lin}(\Gamma)$.

Proposition D.9. Let $\Gamma_S$ be a session typing context and $q$ be either $\text{lin}$ or $\text{un}$. Then, $q(\Gamma_S)$ if and only if $q([\Gamma_S])$.

Proof. The result follows immediately by Def. D.8 and Def. 3.6. ▶
Figure 13 Encoding of multiparty session typing judgements (part 1).
Figure 14 Encoding of multiparty session typing judgements (part 2).
Theorem 6.3 (Correctness of encoding). \( \Gamma \vdash v : U \) implies \([\Gamma] \vdash [v] : [U] \), \( \Theta \vdash X : \tilde{U} \) implies \([\Theta] \vdash [X] : [U] \), and \( \Theta, \Gamma \vdash P \) implies \([\Theta, \Gamma] \vdash P \).

Proof. (A) The thesis for \( \Gamma \vdash v : U \) follows immediately by the encoding of rules (T-Basic) and (T-Name) in Fig. 13. Note that, in the premises of the encoded typing derivation, we use Proposition D.9.

(B) The thesis for \( \Theta \vdash X : \tilde{U} \) follows immediately by the encoding of rules (T-DefCtx) in Fig. 13. Note that the premise \( \text{un}([\Theta]) \) holds because, by Def. 5.4, \([\Theta] \) only contains names with unrestricted types.

The thesis for \( \Theta, \Gamma \vdash P \) is proved by induction on the derivation of the judgement, producing a \( \pi \)-calculus derivation that concludes \([\Theta], [\Gamma] \vdash [P] \) (for some \( \Theta', \Gamma' \) depending on the rule from Fig. 4). The possible cases are shown Fig. 13 and Fig. 14: in all cases, each encoded derivation is supported by premises that hold either by (A) or (B) above, or by the induction hypothesis. Here we discuss the crucial points of each case:

- **(T-Def) and (T-Call).** The derivations are self-explanatory. We just point out that the premise of \( \langle \pi \rangle \) in the latter holds by Proposition D.9.

- **(T-Brch).** The derivation is mostly self-explanatory, except for the topmost premises. The application of (T-Def) is needed to provide the required types to the names used to compose \( \ast \). Each relation holds because, for all \( q \in S \setminus p \), we know that \( S \vdash q \prec p \). Each holds by Proposition B.13, which implies \( S \vdash q \prec \pi S' \vdash q \) holds by Cor. D.3. The \( \langle \pi \rangle \) instance is only yielded when \( S' \vdash p \neq \text{end} \), which implies that \( z \) is used to compose \( \ast \). (note that, in this case, to avoid cluttering the notation we are omitting a premise \( \text{un}(\varnothing) \) required by \( \langle T \rangle \langle \pi \rangle \langle \text{LTup} \rangle \)); otherwise, by Def. 5.1 we have \( [S' \vdash p] = \bullet \), and the premise is replaced by \( \text{un}(z : \bullet) \), since \( z \) is not used to compose \( \ast \).

- **(T-Res1)**. The derivation is, again, mostly self-explanatory. The type of the continuation name \( z \) is either \( \bullet \) (and in this case, \( \langle \pi \rangle \langle \text{Res} \rangle \) stands for \( \langle \pi \rangle \langle \text{Res2} \rangle \)) or a linear connection type \( L_2(T) \), where \( T \) is the carried type of the encoding of the unfolded partial projection \( S' \vdash p \). In this case, \( \langle \pi \rangle \langle \text{Res} \rangle \) stands for \( \langle \pi \rangle \langle \text{Res1} \rangle \). In the second case, the unfolding ensures that \( \text{un}(S' \vdash p) \) and \( \text{un}(S' \vdash p) \) yield dual types \( Li(T)/Lo(T) \) that can be composed with \( \varnothing \) (remind that \( \varnothing \) is not defined on \( \mu \)-types). To correctly deal with such unfolding, the derivation has a branch with an instance of (T-sub) and premise \( \text{un}(S' \vdash p) \) \( S \vdash p \), that is necessary because the type of the variant being sent along \( z \) requires the type of the continuation to be exactly \( S' \vdash p \). Similarly, the derivation has another branch with (T-Sub) and premise \( \text{un}(S' \vdash p) \) \( S' \vdash p \): this is necessary because, when composing \( \ast \), the latter requires \( z \) to have exactly type \( S' \vdash p \). Similarly to the encoding of (T-Brch), the instance of (T-Name)/(T-Sub) on the right (that types \( z \) is only generated if \( S' \vdash p \neq \text{end} \), which implies that \( z \) is used to compose \( \ast \) in (T-\text{LTup}) (otherwise, \( z \) is not used and the premise of (T-\text{LTup}) is replaced by \( \text{un}(z : \bullet) \));

- **(T-Sub) and (T-Res).** These cases are discussed after the statement of Theorem 6.3 (page 21). In the latter, note that the instances of (T-Res1) can be applied because by consistency and completeness of \( \Gamma' \), and by Def. 5.6, for all \( i \in 1..n \), \( \text{un}(S_{p_i} \vdash q_i) \) \( \text{un}(S_{p_i} \vdash q_i) \) = \( \text{Li}(T) \cup \text{Lo}(T) = L_2(T) \) (for some \( T \)).
E  Operational Correspondence

Lemma E.1 says that our encoding yields quasi-linear π-calculus processes (Def.C.5). In fact, sessions are encoded as (quasi-)linearly-typed π-calculus names, and the only unrestricted names are yielded by process declarations, under the constraints of Def.C.5 (item (b)).

Lemma E.1. If Θ · Γ ⊢ P, then [[Θ · Γ ⊢ P]] is quasi-linear.

Proof. See [35, Lemma 4.1.3] and [54, Exercise 1.2.10].

Proposition E.4. For all π-calculus processes P, P′, Γ ⊢ P and P ≡ P′ implies Γ ⊢ P′.

Proof. Standard result (see e.g. [35, Lemma 4.1.1]).

Definition E.5 (Annotated transitions). Transition annotations are ranged over by α, β, ..., and are defined as:

\[ \alpha, \beta, \ldots \equiv x \mid \text{case} \mid \text{with} \mid \text{let} \mid \tau \]

We define the annotated reduction relation \( \rightarrow \) between π-calculus processes as follows:

(Rπ-ComA) \[ \pi(x).P \mid (x(y)).Q \rightarrow P \mid Q(\gamma_y) \]

(Rπ-CASEA) \[ \text{case}_{l_j} (x) \mid \{ l_i(x_i) \mid P_i \}_{i \in I} \rightarrow_{\text{case}} P_j(\gamma_{x_i}) \quad (j \in I) \]

(Rπ-WithA) \[ \text{with}_{[l_i : x_i]_{i \in I}} = [l_i : v_i]_{i \in I} \mid \text{do } P \rightarrow_{\text{with}} P(\gamma_{v_i})_{i \in I} \]

(Rπ-LET) \[ \text{let } x = v \text{ in } P \rightarrow_{\text{let}} P(\gamma_x) \]

(Rπ-RES1) \[ P \rightarrow_{\text{let}} (\nu\alpha)P \rightarrow (\nu\alpha)Q \quad (\text{if } \alpha = x) \]

(Rπ-RES2) \[ P \rightarrow_{\text{let}} (\nu\alpha)P \rightarrow (\nu\alpha)Q \quad (\text{if } \alpha \neq x) \]

(Rπ-STRUCTA) \[ P \equiv P' \land P \rightarrow_{\text{let}} Q \land Q' \equiv Q \quad \text{implies} \quad P' \rightarrow_{\text{let}} Q' \]

Lemma E.1. If Θ · Γ ⊢ P, then [[Θ · Γ ⊢ P]] is quasi-linear.

Proof. By easy analysis of Figures 13 and 14. Note that case (b) of Def.C.5 covers the encoding of (T-DEF) (which produces the \( (\nu x) \)-typed x in \( (\nu x) (\ast(x(y)).Q) | Q' \)) and (T-CALL) (which can only occur within the scope of (T-DEF), and produces the only uses of x, as outputs in Q, Q').

Remark E.6. Lemma E.1 implies that the encoded typing derivations for (T-DEF) and (T-CALL) in Fig.13 could have been further strengthened by adopting the specialised types and typing rules for ω-receptiveness [54, §8.2.2]. However, we preferred to keep the typing rules in Fig. 5 as simple as possible.

Corollary E.7. If Θ · Γ ⊢ P, then [[Θ · Γ ⊢ P]] is quasi-linear. By Def.5.6, we have:

\[ \frac{[\Theta \cdot \Gamma \vdash P]}{[\Theta], \delta(\Gamma) \vdash [P]_{\sigma(\Gamma)}} \quad (Tπ-REIFY) \]
where the conclusion is obtained by replacing all names in \(\text{dom}([\Gamma'])\) with labelled tuples of linearly-typed names, introduced in the typing context by \(\delta(\Gamma)\). (as discussed on page 21). Therefore, by Def.C.5, all names introduced in the conclusion have a quasi-linear type; moreover, the syntactic structure of \([P]_r\sigma(\Gamma)\) is the same of \([P]\): we conclude that \([P]_r\sigma(\Gamma)\) is quasi-linear, according to Def.C.5.

\[\begin{align*}
\text{Theorem E.8 (Encoding is confluent).} \quad & \text{Whenever } [P]_r\Theta\cdot\Gamma \Rightarrow^n P_1 \text{ and } [P]_r\Theta\cdot\Gamma \Rightarrow^n P_2, \text{ then there exists } P_3 \text{ such that } P_1 \Rightarrow^* P_3 \text{ and } P_2 \Rightarrow^* P_3. \\
\text{Proof.} \quad & \text{Note that, Cor.E.7, } [P]_r\Theta\cdot\Gamma \text{ is quasi-linear; and by Proposition C.6, all its reducts are quasi-linear. Letting } n \text{ be the length of the sequence of reductions in } [P]_r\Theta\cdot\Gamma \Rightarrow^n P, \text{ we proceed by induction on } n:} \\
& \quad \begin{align*}
& \text{base case } n = 0. \quad \text{We have } P_2 = [P]_r\Theta\cdot\Gamma, \text{ and we conclude by letting } P_3 = P_1; \\
& \text{inductive case } n = n' + 1. \quad \text{We have } [P]_r\Theta\cdot\Gamma \Rightarrow^n P \Rightarrow P', \text{ and by the induction hypothesis, either:} \\
& \quad \begin{align*}
& \quad \begin{cases}
& \quad \quad P_3 \equiv P_3. \quad \text{We conclude by letting } P_3 = P_3; \\
& \quad \quad \exists P'_3 \text{ such that } P_3 \Rightarrow^* P'_3 \text{ and } P_2 \Rightarrow^* P'_3. \quad \text{By Cor. C.12, we have either:} \\
& \quad \quad \quad \begin{cases}
& \quad \quad \quad \quad \quad \exists P''_3 \text{ such that } P'_3 \Rightarrow^* P''_3 \text{ and } P_2 \Rightarrow^* P''_3. \quad \text{We conclude by letting } P_3 = P''_3.
& \end{cases}
\end{cases}
\end{align*}
\end{align*}
\]

\[\begin{align*}
\text{Definition E.9 (Context narrowing).} \quad & \text{If } \Gamma \leq_\pi \Gamma' \text{ holds iff } \text{dom}(\Gamma) = \text{dom}(\Gamma') \text{ and } \forall x \in \text{dom}(\Gamma), \text{ either: (a) } \Gamma(x) \leq_\pi \Gamma'(x), \text{ or (b) } \Gamma(x) = \text{Li}(T) \uplus \text{Lo}(T) = \Gamma'(x). \\
\text{Definition E.10 (Multi-narrowing).} \quad & \text{The multi-narrowing typing rule for } \pi\text{-calculus is:} \\
& \quad \begin{align*}
& \Gamma \vdash P \quad \Gamma' \leq_\pi \Gamma \\
& \hline \\
& \Gamma' \vdash P \\
& \end{align*}
& \quad \begin{cases}
& \quad \text{(T-M\text{-\textsc{narrow})}}
\end{cases}
\end{align*}
\]

\[\begin{align*}
\text{Proposition E.11.} \quad & \text{Rule (T-M\text{-\textsc{narrow})} \text{ is sound.} \\
\text{Proof.} \quad & \text{Assume } \Gamma \vdash P \text{ and } \Gamma' \leq_\pi \Gamma. \text{ We can rewrite the proof of } \Gamma \vdash P \text{ into a proof concluding } \\
& \Gamma' \vdash P, \text{ noticing that, by Def.E.9, for all } x \in \text{dom } \Gamma, \text{ we have either:} \\
& \text{clause (a): } \Gamma'(x) \leq_\pi \Gamma(x). \text{ In this case, the proof of } \Gamma \vdash P \text{ can be adapted to use } \Gamma'(x) \text{ instead of } \\
& \text{of } \Gamma(x), \text{ by applying of the classical narrowing lemma \cite[7.2.5]{54};} \\
& \text{clause (b): } \Gamma(x) = T_1 \uplus T_2 = \Gamma'(x). \text{ In this case, the proof proof of } \Gamma \vdash P \text{ can be trivially} \\
& \text{adapted to use } \Gamma'(x) \text{ instead of } \Gamma(x). \\
\end{align*}
\]

\[\begin{align*}
\text{Definition E.12 (Context subtyping).} \quad & \text{We define the encoding of an instance of } (\text{T-M\text{-\textsc{sub})} \text{ (Def.B.27) as:} \\
& \quad \begin{align*}
& \begin{cases}
& \quad (\text{T-M\text{-\textsc{sub})} \\
& \quad \quad \Theta \cdot \Gamma_3 \vdash P \quad \Gamma'_3 \leq_\pi \Gamma_3 \\
& \quad \quad \hline \\
& \quad \quad \Theta \cdot \Gamma'_3 \vdash P \\
& \end{cases} \\
& \quad \quad \begin{cases}
& \quad \quad (\text{T-M\text{-\textsc{narrow})} \\
& \quad \quad \quad \Theta \cdot \Gamma_3 \vdash P \\
& \quad \quad \quad \quad \quad \Theta \cdot \Gamma'_3 \vdash [P]_r\Theta \cdot \Gamma_3 \\
& \quad \quad \quad \quad \quad \hline \\
& \quad \quad \quad \quad \quad \Gamma_3 \vdash [P]_r\Theta \cdot \Gamma_3 \\
& \end{cases}
\end{align*}
\end{align*}
\]

\[\begin{align*}
\text{Proposition E.13.} \quad & \text{If } \Theta \cdot \Gamma \vdash P \text{ by rule (T-M\text{-\textsc{sub})}, then } [\Theta \cdot \Gamma] \vdash P \text{ holds.} \\
\text{Proof.} \quad & \text{By induction on the typing derivation of } \Theta \cdot \Gamma \vdash P, \text{ noticing that } \text{dom}([\Theta \cdot \Gamma]) = \text{dom}([\Theta \cdot \Gamma']), \text{ and } \forall x \in \text{dom } (\Theta \cdot \Gamma)(x) \leq_\pi (\Theta \cdot \Gamma')(x) \text{ by Def.B.27 and Theorem 6.2.} \\
\end{align*}
\]

\[\begin{align*}
\text{Theorem 6.4 (Precise decomposition).} \quad & \text{\Gamma_3 \text{ is consistent if and only if } } \delta(\Gamma_3) \text{ is defined.}
\end{align*}
\]
Proof. $(\implies)$. Assume that $\Gamma_S$ is consistent. We proceed by induction on the size of $\Gamma_S$. In the base case $\Gamma_S = \emptyset$, we simply conclude noticing that $\delta(\Gamma_S) = \emptyset$. For the inductive case, for some $\Gamma'_S$ we have $\Gamma_S = \Gamma'_S, s[p] : S_p$, with $\delta(\Gamma'_S)$ defined (by the induction hypothesis), and two possibilities:

- $\exists q \neq p$ such that $s[q] \in \text{dom}(\Gamma'_S)$ (i.e., $s$ does not occur in $\Gamma'_S$). This implies $z_{(s,p,q)} \notin \text{dom}(\delta(\Gamma'_S))$; therefore, by Def. 5.6, we conclude that $\delta(\Gamma'_S)$ is defined as:
  $$\delta(\Gamma'_S) = \delta(\Gamma'_S) \cap \{\text{unf}(s[p] : q)\} = \delta(\Gamma'_S), z_{(s,p,q)} : \text{unf}(s[p] : q)$$

- $\exists q \neq p$ such that $s[q] : S_q \in \Gamma'_S$. This implies $\Gamma'_S = \Gamma'_S, s[q] : S_q$. By Def. 2.11, we also have $S_p \cap q \notin_p S_q \cap p$, and therefore:
  $$\text{unf}(S_p \cap q) \leq_p \text{unf}(S_q \cap p)$$ (by Cor. D.3)

This implies that we can have three cases:

- $\text{unf}(S_p \cap q) = \text{unf}(S_q \cap p) = \bullet$. This implies $\text{unf}(S_p \cap q) = \text{unf}(S_q \cap p) = \bullet$;

- $\text{unf}(S_p \cap q) = \text{Li}(T)$ and $\text{unf}(S_q \cap p) = \text{Li}(T')$ with $T \leq_p T'$. By Theorem 6.1, this implies $\text{unf}(S_p \cap q) = \text{Li}(T)$; and $\text{unf}(S_q \cap p) = \text{Li}(T')$ with $T' \leq_p T$. By Theorem 6.1, this implies $\text{unf}(S_p \cap q) = \text{Li}(T)$.

In all cases, we can verify that $\text{unf}(S_p \cap q) \cap \text{unf}(S_q \cap p)$ is defined, by Def. 4.6. Therefore, by Def. 5.6, noticing that $z_{(s,p,q)} = z_{(s,p,q)} \not\in \text{dom}(\delta(\Gamma'_S))$, and that $\delta(\Gamma'_S)$ is defined (by the induction hypothesis), we conclude that $\delta(\Gamma'_S)$ is defined as:

$$\delta(\Gamma'_S) = \delta(\Gamma'_S), z_{(s,p,q)} : \text{unf}(S_p \cap q) \cap \text{unf}(S_q \cap p)$$

$(\impliedby)$. We prove the contrapositive. Assume that $\delta(\Gamma_S)$ is not defined. Examining Def. 5.6, we can see that this can only occur if some application of $\cap$ is not defined, i.e., there exist some $s[p] : S_p, s[q] : S_q \in \Gamma'_S$ with $p \neq q$ such that $\text{unf}(S_p \cap q) \cap \text{unf}(S_q \cap p)$ (i.e., the type for $z_{(s,p,q)}$) is not defined. By Def. 4.6, we can have the following four cases:

- $\text{unf}(S_p \cap q) = \bullet$ and $\text{unf}(S_q \cap p) \neq \bullet$;

- $\text{unf}(S_p \cap q) \neq \bullet$ and $\text{unf}(S_q \cap p) = \bullet$;

- $\text{unf}(S_p \cap q) = \text{Li}(T)$ and $\text{unf}(S_q \cap p) \neq \text{Li}(T')$ with $T' \leq_p T$;

- $\text{unf}(S_p \cap q) = \text{Li}(T)$ and $\text{unf}(S_q \cap p) \neq \text{Li}(T')$ with $T' \leq_p T$.

In all cases, we obtain:

- $\text{unf}(S_p \cap q) \not\leq_p \text{unf}(S_q \cap p)$

- $\text{unf}(S_p \cap q) \not\leq_p \text{unf}(S_q \cap p)$ (by Lemma 4.2)

- $\text{unf}(S_p \cap q) \not\leq_p \text{unf}(S_q \cap p)$ (by the contrapositive of Cor. D.3)

- $\text{unf}(S_p \cap q) \not\leq_p \text{unf}(S_q \cap p)$ (by the contrapositive of (S-LμL) and (S-LμR))

Hence, by Def. 2.11, we conclude that $\Gamma_S$ is not consistent.

Proposition E.14. If $\Gamma_S \leq_p \Gamma'_S$, then $\sigma(\Gamma_S) = \sigma(\Gamma'_S)$.

Proof. Assume $\Gamma_S \leq_p \Gamma'_S$. Then:

- $\text{dom}(\Gamma_S) = \text{dom}(\Gamma'_S)$ and $\forall s[p] \in \text{dom}(\Gamma_S) : \Gamma_S(s[p]) \leq_p \Gamma'_S(s[p])$ (by Def. B.27)

- $s[p] \in \text{dom}(\Gamma_S) : \text{roles}(\Gamma_S(s[p])) = \text{roles}(\Gamma'_S(s[p]))$ (by Proposition B.11)

- $\text{conn}(s, \Gamma_S) = \text{conn}(s, \Gamma'_S)$ (by Def. 5.5)

Therefore, by Def. 5.6, we conclude $\sigma(\Gamma_S) = \sigma(\Gamma'_S)$.
Proposition E.15. If $\Gamma \vdash P$, then $\text{fn}(P) \subseteq \text{dom}(\Gamma)$ and $\forall x \in (\text{dom}(\Gamma) \setminus \text{fn}(P)) : \text{un}(\Gamma(x))$.

Proof. By induction on the derivation of $\Gamma \vdash P$.

Proposition E.16. If $\Theta \cdot \Gamma \vdash P$, then $\text{fc}(P) \subseteq \text{dom}(\Gamma)$ and $\forall c \in (\text{dom}(\Gamma) \setminus \text{fc}(P)) : \Gamma(c) = \text{end}$.

Proof. By induction on the derivation of $\Theta \cdot \Gamma \vdash P$.

Proposition E.17. If $\Theta \cdot \Gamma \vdash P$ and $\Theta' \cdot \Gamma' \vdash P$, then:
1. $\forall c \in (\text{dom}(\Gamma) \cap \text{dom}(\Gamma')) : \text{roles}(\Gamma(c)) = \text{roles}(\Gamma'(c))$;
2. $\forall c \in (\text{dom}(\Gamma) \setminus \text{dom}(\Gamma')) : \text{roles}(\Gamma(c)) = \emptyset$;
3. $\forall c \in (\text{dom}(\Gamma') \setminus \text{dom}(\Gamma)) : \text{roles}(\Gamma'(c)) = \emptyset$.

Proof. Item 1 is proved by induction on the derivation of $\Theta \cdot \Gamma \vdash P$, noticing that $\Theta, \Theta'$ are irrelevant for the statement, while $\Gamma, \Gamma'$ can only differ by adding/removing an instance of rule (T-SUB), so that $\Gamma(c) \subseteq \Gamma'(c)$ or $\Gamma'(c) \subseteq \Gamma(c)$; in both cases, we conclude by Proposition B.11. Items 2 and 3 are a consequence of Proposition E.16.

Definition E.19 (Guarded and normal-form processes). A multiparty session process is guarded if it has the form $s_1[p_0_0] \oplus \ldots \oplus s_n[p_0_n] \& \sigma \{ \overset{i}{l}(x_i).P_i \}$, where $\sigma = \{ \overset{i}{l}(x_i).P_i \}$. A multiparty session process $P$ is in normal form if:

$$P = \text{def } \tilde{D} \in (\nu \tilde{s}_*) \{ Q_1 | \ldots | Q_m | Y_1(\bar{v}_1) | \ldots | Y_m(\bar{v}_m) \}$$

where $Q_1, \ldots, Q_m$ are guarded.

Proposition E.20. If $\Gamma \vdash P \xrightarrow{\text{with}} P'$, then $\Gamma \vdash P'$.

Proof. Standard subject reduction property for $\pi$-calculus with linear types (see [54, Theorem 8.1.5]).

Proposition E.21. If $\Theta \cdot \Gamma \vdash P$,

(i) $x:T \in [\Theta]$ implies $T = \sharp(T')$ and

(ii) $x:T \in \delta(\Gamma)$ implies $\text{qlin}(T) \in \{ \overset{i}{l}(T'), \overset{\overset{i}{l}}{l}(T'), \overset{i}{l}(T') \}$ (for some $T'$).

Proof. Straightforward by Proposition E.18 and Def.5.6.

Proposition E.22. If $\Theta \cdot \Gamma \vdash P \equiv P'$, then $[P]_{\Theta, \Gamma} \equiv [P']_{\Theta, \Gamma}$ and $[P]_{\Theta, \Gamma, \sigma} \equiv [P']_{\Theta, \Gamma, \sigma}$.

Proof. We can prove $[P]_{\Theta, \Gamma} \equiv [P']_{\Theta, \Gamma}$ by induction on the derivation of $P \equiv P'$. Then, the last part of the statement is straightforward.

Lemma E.23. If $\Theta \cdot \Gamma \vdash P$:

$$P \xrightarrow{\text{(T} \pi\text{-REF})} \Theta \cdot \Gamma \vdash [P]_{\sigma(\Gamma)}$$

implies that $\exists \Psi, P'', \Gamma', P'$

such that $P \xrightarrow{\text{with}^*} (\nu \Xi)P''$

and $P' \xrightarrow{\text{(T} \pi\text{-REF})} \Theta \cdot \Gamma' \vdash [P']_{\sigma(\Gamma')}$.
Proof. Assume \( \Theta \cdot \Gamma \vdash P \). We first collect several facts that we will use in the proof later on. By structural equivalence [10, Proof of Theorem 1], and by Proposition B.25, we have:

\[
\Theta \cdot \Gamma \vdash \text{def } \bar{D} \in (\nu \tilde{x}_1)(Q_1 \ldots Q_m | Q_2) \equiv P \quad \text{where } \bar{D} = X_1(\tilde{x}_1) = P_{X_1} \ldots X_n(\tilde{x}_1) = P_{X_n}
\]

for some \( X_1, \ldots, X_n, P_{X_1}, \ldots, P_{X_n} \), guarded (Def.E.19) \( Q_1, \ldots, Q_m \) and \( Q_Y = Y_1(\tilde{y}_1) \ldots Y_m(\tilde{y}_m) \).

(73)

From (73), by Proposition E.22 and Def.5.7, we have:

\[
[P]_{\Theta \cdot \Gamma} \equiv (\nu [X_1] \ldots (\nu [X_n])((\bar{Q}_1)_{\Theta X_1 \Gamma_1} \ldots ((\bar{Q}_m)_{\Theta X_m \Gamma_m} | (\bar{Q}_Y)_{\Theta X_m \Gamma_Y} | P_X)
\]

where \( \Gamma_1 \circ \ldots \circ \Gamma_m \circ \Gamma_Y = \Gamma \) and \( \Theta X = \Theta, X_1 : \tilde{U}_1, \ldots, X_n : \tilde{U}_n \) and \( \forall i \in 1 .. m' : Y_i \in \text{dom}(\Theta) \)

(75)

and where \([(\nu \tilde{x}_1)]\) is a sequence of restrictions yielded by the encoding of (T-Res) (Fig.14), and

\[
P_X = \ast \left( (X_1)_{\tilde{x}_1} \vdots \text{with } (\tilde{x}_1) = z \text{ do } [P_{X_1}]_{\Theta X_1 \tilde{U}_1} \tilde{X}_1 \tilde{U}_1 \ldots \ast \left( (X_n)_{\tilde{x}_n} \vdots \text{with } (\tilde{x}_n) = z \text{ do } [P_{X_n}]_{\Theta X_n \tilde{U}_n} \tilde{X}_n \tilde{U}_n \right)
\]

(78)

i.e., \( P_X \) is a parallel composition of input-guarded replicated processes, corresponding to the encodings of \( P_{X_1}, \ldots, P_{X_n} \).

From (74) and Def.E.19, for all \( i \in 1 .. m \), we have:

\[
\text{for some } c_i, q, I : \ Q_i = c_i[q]_{k \in I} (j_x(i), Q''_{i}) \quad \text{or} \quad \text{for some } c'_i, p : \ Q_i = c'_i[p] \lor (l(v)) Q''_{i}
\]

(79)

This implies that for all \( i \in 1 .. m \) (to avoid cluttering the notation, in the following we will omit an \( i \)-index on \( S_c \) and \( S_{c'} \), which will be clear from the context):

\( c_i \) in (79) must be typed by some branching type \( S_c = q_{k \in I} ? l_j(U'_j) S'_j \)

\( c'_i \) in (79) must be typed by some selection type \( S_{c'} = p \lor ! (U'') S'' \)

Moreover, we can assume that \( \Theta X \cdot \Gamma_i \vdash Q_i \) holds by a (possibly vacuous) subtyping on \( c_i \) or \( c'_i \), as per Proposition B.30:

\[
\text{(T-MSUB)} \quad \frac{\Theta X \cdot \Gamma_i \vdash Q_i \quad \Gamma_i : \leq S \Gamma_i}{\Theta X \cdot \Gamma_i \vdash Q_i}
\]

where (by Def.B.27) \( \text{dom}(\Gamma_i) = \text{dom}(\Gamma_i) \) and

\[
\begin{align*}
\Gamma_i(c_i) & \leq S \Gamma_i(c_i) = S_c \\
\Gamma_i(c'_i) & \leq S \Gamma_i(c'_i) = S_{c'}
\end{align*}
\]

(80)

At this point, we can observe that each with-reduction in \( [P]_{\Theta \cdot \Gamma} \) can only be induced by some with-prefix occurring in \( [Q_1]_{\Theta X_1 \Gamma_1} \ldots [Q_m]_{\Theta X_m \Gamma_m} \) in (75); moreover, since \( Q_1, \ldots, Q_m \) are typed by (T-Brc)/(T-Sel), by examining Fig.13 we can see that for all \( i \in 1 .. m \), \( [Q_i]_{\Theta X_i \Gamma_i} \) has exactly one top-level with-prefix, followed by an input/output on a linearly-typed name.

We will now focus on those \( i \in 1 .. m \) where \( c_i \) and \( c'_i \) above are channels with roles: omitting some indexing on \( i \in 1 .. m \), we consider \( c_i = s[p] \) typed by some \( S_p \) (for some \( s, p \)) or \( c'_i = s'[q] \) typed by some \( S_q \) (for some \( s', q \)) (the cases where \( c_i/c'_i \) are session-typed variables are similar). Note that (80) becomes:

\[
\text{(T-MSUB)} \quad \frac{\Theta X \cdot \Gamma_i \vdash Q_i \quad \Gamma_i : \leq S \Gamma_i}{\Theta X \cdot \Gamma_i \vdash Q_i}
\]

where

\[
\begin{align*}
\Gamma_i(s[p]) & \leq S \Gamma_i(s[p]) = S_p \\
\Gamma_i(s'[q]) & \leq S \Gamma_i(s'[q]) = S_q
\end{align*}
\]

(81)
Summing up, for all \(i \in 1..m\) where \(c_i, c_i'\) in (79) are channels with roles, we have the following properties and encodings of \(Q_i\) (note that (T-MSUB) in (80) is encoded as (Tπ-MNARROW), as per Def E.12):

\[
\begin{align*}
\{Q_i\}_{i \in X, r_i} \vdash (r_i, \sigma_i(Γ_i)) = \begin{cases}
\text{for some } s, p, q, I, S_p, \quad Q_i = s[p] q \&_{j \in I} \{l_j(x_j), Q'_j\} \quad \text{with } \Gamma_i(s[p]) \subseteq S_p = q \&_{j \in I} ?l_j(U'_j), S'_j \text{ and }
[Q_i]_{s[p]} \vdash r_i \quad \text{do } z_q(y) \cdot \text{case } y \text{ of } \quad \{l_j(x_j) \triangleright \text{with } (x_j, z) = z_j \text{ do } \left(\Gamma_i \setminus \text{dom}(\sigma_i(s[p]))\right)_{\text{by } j \in I}
\end{cases}
\end{align*}
\]

where \(∀j \in I: X_j = \begin{cases}
\{q: z, r: z_j \mid z_j \in S'_j \} & \text{if } q \in S'_j \\
\{r: z \mid z \in S''\} & \text{otherwise}
\end{cases}\)

By applying the substitutions \(σ_i(Γ_i)\) in (82), for all \(i \in 1..m\) we get either (note that \([s[p]]\) and \([s'[q]]\) are rebound by let):

\[
\begin{align*}
[Q_i]_{s[p]} \vdash r_i \quad \text{do } z_q(y) \cdot \text{case } y \text{ of } \quad \{l_j(x_j) \triangleright \text{with } (x_j, z) = z_j \text{ do } \left(\Gamma_i \setminus \text{dom}(\sigma_i(s[p]))\right)_{\text{by } j \in I}
\end{align*}
\]

where \(∀j \in I: X_j = \begin{cases}
\{q: z, r: z_j \mid z_j \in S'_j \} & \text{if } q \in S'_j \\
\{r: z \mid z \in S''\} & \text{otherwise}
\end{cases}\)

Hence, there exist \(Q'_1, \ldots, Q'_m\) such that (to save some symbols, we will now redefine \(X_1\) and \(X_2\) by taking the corresponding definitions in (82) and applying the substitutions induced by with):

\[
\begin{align*}
[P]_{s[p]} \vdash (r_i, \sigma_i(Γ_i)) \equiv \left((\nu X_1) \ldots (\nu X_n) \left((\nu s') p \right) \left(\left[Q_1\right]_{s[p]} \vdash r_1 \quad \text{do } z_q(y) \cdot \text{case } y \text{ of } \quad \{l_j(x_j) \triangleright \text{with } (x_j, z) = z_j \text{ do } \left(\Gamma_i \setminus \text{dom}(\sigma_i(s[p]))\right)_{\text{by } j \in I}
\end{align*}
\]

where \(∀i \in 1..m\)

\[
\begin{align*}
Q'_i = [Q_i]_{s[p]} \vdash r_i \quad \text{do } z_q(y) \cdot \text{case } y \text{ of } \quad \{l_j(x_j) \triangleright \text{with } (x_j, z) = z_j \text{ do } \left(\Gamma_i \setminus \text{dom}(\sigma_i(s[p]))\right)_{\text{by } j \in I}
\end{align*}
\]

where \(\nu X_1 \ldots (\nu X_n) \left((\nu s') p \right) \left(\left[Q_1\right]_{s[p]} \vdash r_1 \quad \text{do } z_q(y) \cdot \text{case } y \text{ of } \quad \{l_j(x_j) \triangleright \text{with } (x_j, z) = z_j \text{ do } \left(\Gamma_i \setminus \text{dom}(\sigma_i(s[p]))\right)_{\text{by } j \in I}
\end{align*}
\]

where \(∀i \in 1..m\)

\[
\begin{align*}
Q'_i = [Q_i]_{s[p]} \vdash r_i \quad \text{do } z_q(y) \cdot \text{case } y \text{ of } \quad \{l_j(x_j) \triangleright \text{with } (x_j, z) = z_j \text{ do } \left(\Gamma_i \setminus \text{dom}(\sigma_i(s[p]))\right)_{\text{by } j \in I}
\end{align*}
\]

where \(∀i \in 1..m\)

\[
\begin{align*}
Q'_i = [Q_i]_{s[p]} \vdash r_i \quad \text{do } z_q(y) \cdot \text{case } y \text{ of } \quad \{l_j(x_j) \triangleright \text{with } (x_j, z) = z_j \text{ do } \left(\Gamma_i \setminus \text{dom}(\sigma_i(s[p]))\right)_{\text{by } j \in I}
\end{align*}
\]
We can now proceed by cases on $\alpha$ in the annotated reduction $P_0 \xrightarrow{\lambda} P^*$ (according to Def. E.5):

- $\alpha \in \{\text{case}, \text{let}\}$. These cases are impossible, and the statement hold vacuously. In fact, these reductions can only be fired by an occurrence of case or let, and we can verify that such prefixes do not appear at top-level in $Q_1, \ldots, Q_m$, nor $[Q_Y]_{x\in Y} \sigma(\Gamma_Y)$, nor $P_X$. Hence, for all $P_0$ such that $[P_0]_{\tau} \sigma(\Gamma) \xrightarrow{\lambda} P_0$, we have $P_0 \xrightarrow{\text{case}}$ and $P_0 \xrightarrow{\text{let}}$.

- $\alpha = \text{with}$. We have $P_0 \xrightarrow{\text{with}} P_*$, and thus $[P_0]_{\tau} \sigma(\Gamma) \xrightarrow{\lambda} P_*$: we conclude by letting $x = 0$, $P'' = P_*$, $\Gamma = \Gamma'$, $P = P''$.

- $\alpha = x$ (for some $x$). Since by Proposition E.20 we have $[\Theta], \delta(\Gamma) \vdash P_0$, we also know that $x \in \text{dom}(\{[\Theta], \delta(\Gamma)\})$ (by Proposition E.15). By Proposition E.21 we have two sub-cases:

  - $x : \Xi(T) \in \{\Theta\}$. This case is absurd. In fact, by (76) it would imply $x \notin \{X_j\}$ (for all $j \in 1..n$); moreover, it would require an unrestricted output on $x$, which implies $x = [Y_i]$ for some $i \in 1..m'$ (by (84), (85), and (74)). Finally, it would imply a synchronisation with a process guarded by an unrestricted input on $[Y_i]$ occurring in $P_0$ — which contradicts (84);

  - $x : \text{Li}(T) \in \delta(\Gamma)$. We have $x : \text{Li}(T) \equiv \text{Lo}(T) \in \delta(\Gamma)$, and it implies that two processes $Q'_1, Q'_2$ (from (84) and (85)) synchronise on some $x$, performing respectively an input and an output. Without loss of generality, let $i = 1$ and $j = 2$. By Def. 5.6, it means that $\sigma(\Gamma)$ replaces some $z_{x[p]}$ and $z_{x[q]}$ in $[P_0]_{\tau} \sigma(\Gamma)$ with labelled tuples of channels $v_1, v_2$ such that $x = v_1(q) = v_2(p) = z_{(s,p,q)}$, which implies $s = s'$. Correspondingly, by Def. 5.6, $\delta(\Gamma)$ combines (using $\uparrow$) the the encodings of two unfolded partial projections, that after being split with $\uparrow$, yield Li(T) and Lo(T) above. More precisely, considering that (by (76)) $\Gamma$ is split into $\Gamma_1 \uparrow \Gamma_2 \uparrow \ldots \uparrow \Gamma_m \uparrow \ldots$, we have:

  $x = \sigma(\Gamma)(s[p])(q) = \sigma(\Gamma_1)(s[p])(q) = \sigma(\Gamma_2)(s[q])(p) = \sigma(\Gamma_2)(s[q])(p) = z_{(s,p,q)}$

  (by Def. 5.6)

  (by hypothesis)

  $\Gamma_1 \circ \Gamma_2$ is consistent

  (by (86), (76) and Cor. B.17)

  (87)

  for some $s, p, q, S_p, S_q$: $\Gamma(s[p]) = \Gamma_1(s[p])$ and $\Gamma(s[q]) = \Gamma_2(s[q])$

  (by (86) and (87))

  (88)

  $\text{unf}(\Gamma_1(s[p])) \uparrow q \leq p \text{unf}(\Gamma_2(s[q])) \uparrow p$

  (by (88) and Proposition B.8)

  (89)

Now, the fact that $Q'_1$ performs an input on $z_{(s,p,q)}$ implies that $Q_1$ performs a branching on $s[p][q]$, which means (by inverting (T-Buch)) that $Q_1$ is typed by some $S$, that is a branching from $q$, and $\Gamma_1(s[p]) \leq S_p$. Symmetrically, the fact that $Q'_2$ performs an output on $z_{(s,p,q)}$ implies that $Q_2$ performs a selection on $s[q][p]$, which means (by inverting (T-Sele)) that $Q_2$ is typed by some $S_q$ that is a selection towards $p$, and $\Gamma_2(s[q]) \leq S_{q}$. From (89), and the observation that $S_p, S_q$ are respectively a branching and selection type, we have (for some $\Gamma'$):

$\text{unf}(\Gamma_1(s[p])) = q \land_{j \in I_1} \text{?}_j(U'_j), S_p \leq S_{p}$ and $\text{unf}(\Gamma_2(s[q])) = p \uplus_{j \in I_2} \text{!}_j(U'_j), S_q^* \leq S_{q}$ and $\forall j \in I' : S_j^* : q = S_j^* \downarrow p$

(90)

Now, we can notice that $S_p, S_q, \Gamma_1, \Gamma_2, \Gamma_1', \Gamma_2', Q_1, Q_2$ match the corresponding definitions in Equation (42) on page 44 (in the proof of Theorem 2.16 — subject reduction for multiparty session typed processes). In the rest of the present proof, we will thus refer to results that follow Equation (42) (in particular, (43), (44) and (46)) applying them to (85), to study the synchronisation on $x = z_{(s,p,q)}$ between $Q'_1$ and $Q'_2$.

Let $\sigma(v : U) = \emptyset$ (i.e., the empty substitution) if $U = B$ (otherwise, Def. 5.6 applies). We can see that the synchronisation on $x$ gives the following reductions, where the linear names given by the reified instantiation of $v$ is passed from $Q'_2$ to $Q'_1$ (we exploit the fact that $U'' \leq S U'_2$.}
by (46), and thus \( \sigma(v; U'_v) = \sigma(v; U'^{\prime}) \) by Proposition E.14:

\[
\begin{pmatrix}
\lbrack \Theta_X \rbrack, \delta(\Gamma_1) \cup \delta(\Gamma_2) \vdash Q'_1 \mid Q'_2 \\
\text{case } l_k([v], z) \text{ of } \{ j \mid (z_j) \uparrow \text{ with } (x_j, z) = z_j \text{ do let } [s[p]] = \mathfrak{K}_j \text{ in } \left( \begin{array}{l}
\left[ Q'_{k} \right]_{s \cdot x \cdot z \cdot s'[s]; s'_q} \sigma(\Gamma_1 \setminus [s[p]])
\end{array} \right)_{j \in I} \\
\text{let } [s[q]] = \mathfrak{K}_j \text{ in } \left( \begin{array}{l}
\left[ Q'_{k} \right]_{s \cdot x \cdot z \cdot s'[s]; s'_q} \sigma(\Gamma_2 \setminus [s[q]]) \text{ \textregistered dom } (\sigma(v; U'^{\prime}))
\end{array} \right)_{j \in I}
\end{pmatrix}
\]

Now, notice that, from Equation (85), we have:

\[
\mathfrak{K}_k = \begin{cases}
q \cdot z \cdot r \cdot z_{(s, s, t)} & \text{if } q \in S'_k \\
\Gamma \cdot z_{(s, s, t)} & \text{otherwise}
\end{cases}
\]

\[
\mathfrak{K} = \begin{cases}
p \cdot z \cdot z_{(s, s, t)} & \text{if } p \in S'' \\
q \cdot z_{(s, s, t)} & \text{otherwise}
\end{cases}
\]

If we replace \( z \) with \( z_{(s, s, t)} \) in (92), from (45), (47) and Proposition E.14 we observe that:

\[
\sigma(\Gamma_1 \setminus [s[p]]) \sigma(v; U'_v) \left( \mathfrak{K}_k \right)_{s \cdot t \cdot s'[s]; s'_q} = \sigma(\Gamma_1 \setminus [s[p]]) \sigma(v; U'_v) \left( \mathfrak{K}_k \right)_{s \cdot t \cdot s'[s]; s'_q} = \sigma(\Gamma_1)
\]

\[
\sigma(\Gamma_2 \setminus [s[q]]) \left( \mathfrak{K}_k \right)_{s \cdot t \cdot s'[s]; s'_q} = \sigma(\Gamma_2 \setminus [s[q]]) \left( \mathfrak{K}_k \right)_{s \cdot t \cdot s'[s]; s'_q} = \sigma(\Gamma_2)
\]

Moreover, by applying the substitutions in the processes in (91), by (47) and (55) we get:

\[
\left[ Q'_{k} \right]_{s \cdot x \cdot z \cdot s'[s]; s'_q} \left( \mathfrak{K}_k \right)_{s \cdot t \cdot s'[s]; s'_q} = \left[ Q'_{k} \right]_{s \cdot x \cdot z \cdot s'[s]; s'_q} = \left[ Q'_{k} \right]_{s \cdot x \cdot z \cdot s'[s]; s'_q}
\]

Now, by \( \alpha \)-renaming (\( \nu z \)) into (\( \nu z_{(s, s, t)} \)) in (91), and applying (93), (94) and (95), we get:

\[
\left[ \Theta_X \right], \delta(\Gamma_1) \cup \delta(\Gamma_2) \vdash Q'_1 | Q'_2 \rightarrow^* (\nu z_{(s, s, t)}) \left( \begin{array}{l}
\left[ Q'_{k} \right]_{s \cdot t \cdot s'[s]; s'_q} \sigma(\Gamma_1) \\
\left[ Q'_{k} \right]_{s \cdot t \cdot s'[s]; s'_q} \sigma(\Gamma_1)
\end{array} \right) | Q_3 | \cdots | Q_m | \left[ \Theta_Y \right]_{s \cdot x \cdot t \cdot s'[s]; s'_q} \sigma(\Gamma_Y) | \Phi_X
\]

Hence, if we let:

\[
P_m'' = (\nu[X_1]) \cdots (\nu[X_m]) \left( \begin{array}{l}
\left[ Q'_{k} \right]_{s \cdot t \cdot s'[s]; s'_q} \sigma(\Gamma_1) \\
\left[ Q'_{k} \right]_{s \cdot t \cdot s'[s]; s'_q} \sigma(\Gamma_1)
\end{array} \right) | Q_3 | \cdots | Q_m | \left[ \Theta_Y \right]_{s \cdot x \cdot t \cdot s'[s]; s'_q} \sigma(\Gamma_Y) \rightarrow^* (\nu z_{(s, s, t)}) P_m''
\]

we obtain:

\[
P_0 \rightarrow^* (\nu z_{(s, s, t)}) P_m''
\]
Furthermore, notice that:

\[ \Gamma_1 \circ \Gamma_2 \vdash Q_1 \mid Q_2 \quad \Gamma_1' \circ \Gamma_2' : Q'^{\prime}_{\text{Sel}} \mid Q'_2 \]  

(from (97))

\[ \Theta \cdot P \rightarrow \Theta \cdot \Gamma_1' \circ \Gamma_2' \circ \Gamma_3 \cdots \circ \Gamma_m \vdash \text{def } \bar{D} \text{ in } (\nu \tilde{s}) \big( Q'^{\prime}_{\text{Sel}} \mid Q'_2 \mid Q_3 \mid \ldots \mid Q_Y \big) \]  

(from (73), (76) and (100))

(101)

Hence, if we let:

\[ \Gamma' = \Gamma_1' \circ \Gamma_2' \circ \Gamma_3 \circ \ldots \circ \Gamma_m \]  

and \[ P' = \text{def } \bar{D} \text{ in } (\nu \tilde{s}) \big( Q'^{\prime}_{\text{Sel}} \mid Q'_2 \mid Q_3 \mid \ldots \mid Q_Y \big) \]

(102)

we obtain:

\[ (\text{T} \pi \text{-Reify}) \frac {\Theta \cdot \Gamma' \vdash P'} {\Theta, \delta(\Gamma') \vdash P' \sigma(\Gamma') \quad \text{with}_\ast \quad P''} \]  

(by (84), (85) and (99))

Summing up, we prove the statement by taking:

- \( x = \tilde{s}_{(s,p,q)} \);
- \( P'' \) as defined in (99);
- \( \Gamma' \) as defined in (102);
- \( P' \) as defined in (102);

\( \alpha = \pi \). By Def. E.5, this reduction can only be induced by a synchronisation on some delimited name \( x \). By (75), we have two cases:

- \( x \) is delimited in \( [\nu \tilde{s}] \). Therefore, there is some \( s \in \tilde{s} \), whose encoding yields a delimitation for \( x \). In this case, after opening \( (\nu s) \) by inverting \( \text{T-Res} \), we fall back into a case similar to that for \( \alpha = x \) and \( x : \tilde{s} \) above: we get an encoded \( \pi \)-calculus typing derivation where \( x \) is linearly-typed, and corresponds to a reduction between some \( s[p][q]/s[q][p] \). We prove this case as above, and conclude by re-applying the delimitation \( (\nu s) \), and taking:
  - \( \tilde{x} = \tilde{z}_{(s,p,q)} \);
  - \( P'' \) as in (99);
  - \( \Gamma' \) as in (102);
  - \( P' \) as in (102);

- \( x \) is delimited in \( (\nu [X_1]) \ldots (\nu [X_n]) \), yielded by the encodings of \( \text{def } X_1(\tilde{x}_1) = P_{X_1} \text{ in } \ldots \text{def } X_n(\tilde{x}_n) = P_{X_n} \text{ in } \ldots \) in (73), and thus by the encoding of (T-Def) in Fig. 13. Without loss of generality, let \( x = [X_1] \) (otherwise, the proof is similar). If we open the delimitation \( (\nu [X_1]) \) by looking at the premise of \( \text{T} \pi \text{-Res1} \) in the encoded derivation, we can see that \( x = [X_1] \) has an unrestricted type \( \tilde{x}_1(\tilde{U}_1) \), and is used for \text{input} by a replicated process in \( P_{X_1} \), as shown in (78); therefore, the \( \tau \)-reduction under analysis is induced by a synchronisation on \( x \) with another process that uses \( x : \tilde{x}_1(\tilde{U}_1) \) for \text{output}. By examining Fig. 13, we can see that such a process can only be produced by the encoding of (T-Call): this implies that \( Q_{Y \setminus \nu} \) in (73) and (74) contains a process \( X_1(\tilde{v}_1) \), whose arguments are typed as \( \tilde{U}_1 \). Without loss of generality, let \( Y_1 = X_1 \) in (74), which implies:

\[ \Gamma_{Y_1} = \Gamma_{X_1} \vdash \tilde{v}_1 : \tilde{U}_1 \]  

(by (75), (77) and inversion of (T-Call))

(103)

Applying these findings in (84), we obtain:

\[ [P]_{\text{def} \Gamma} \sigma(\Gamma) \quad \text{with}_\ast \quad P_0 \equiv (\nu [X_1] \ldots (\nu [X_n])[[\nu \tilde{s}]) \big( Q'_1 \mid \ldots \mid Q'_m \mid [X_1(\tilde{v}_1)]_{\text{def } X_1} \Gamma_{X_1} \big) \big( Q_{Y \setminus \nu} \text{ in } P_{X_1} \big) \]  

(104)

where \( Q_{Y \setminus \nu} = Y_2(\tilde{v}_2) \mid \ldots \mid Y_m(\tilde{v}_m) \) and \( \Gamma_{Y \setminus \nu} = \Gamma_{Y_2} \circ \ldots \circ \Gamma_{Y_m} \)
Now, letting $P_0$ synchronise, we get:

$P_0 \overset{\tau}{\rightarrow} (\nu[X_1] \ldots (\nu[X_n])[(\nu]\tilde{\nu})) \left( Q'_1 | \ldots | Q'_m | \left( \text{with } (\tilde{x}) = [\nu_1] \text{ do } [P_{X_1}]_{\tilde{x},X_1,v_{i_1},\tilde{v}_{i_1}} \sigma(T_{X_1}) \right| [Q_{\nu'}\nu_X\tau_{\nu'}]_{\nu_X}\gamma \right) \right)_{\nu}$ with $\nu$,

$(\nu[X_1] \ldots (\nu[X_n])[(\nu]\tilde{\nu})) \left( Q'_1 | \ldots | Q'_m | \left( [P_{X_1}]_{\tilde{x},X_1,v_{i_1},\tilde{v}_{i_1}} \sigma(T_{X_1}) \right| [Q_{\nu'}\nu_X\tau_{\nu'}]_{\nu_X}\gamma \right) = (\nu[X_1] \ldots (\nu[X_n])[(\nu]\tilde{\nu})) \left( Q'_1 | \ldots | Q'_m | \left( [P_{X_1}]_{\tilde{x},X_1,v_{i_1},\tilde{v}_{i_1}} \sigma(T_{X_1}) \right| [Q_{\nu'}\nu_X\tau_{\nu'}]_{\nu_X}\gamma \right) \left( \text{by (103) + Lemma B.26} \right)$

Now, from (73), and from the proof of Theorem 2.16 (case (R-Call)) we have:

$\Theta \cdot \Gamma \vdash P \rightarrow \Theta \cdot \Gamma' \vdash P' = \text{def } D \in (\nu\tilde{\nu}) \left( P_{X_1} \left\{ v_{i_1}/\tilde{v}_{i_1} \right\} | \ldots | Q_{\nu'} \right)$ with $\Gamma' = \Gamma$

(106)

Hence, we conclude by taking:

* $\tilde{x} = \emptyset$;
* $P''$ as above;
* $\Gamma' = \Gamma$;
* $P'$ as in (106).

Lemma E.24 (Operational soundness of encoding). If $\Theta \cdot \Gamma \vdash P$:

$[\Theta] \cdot [\delta(\Gamma)] \vdash [P]_{\sigma(\Gamma)} \rightarrow P''$. implies that $\exists P''', \Gamma', \nu'$ such that $P_s \rightarrow^* (\nu\tilde{\nu})P''$ and $[\Theta] \cdot [\delta(\Gamma')] \vdash [P']_{\sigma(\Gamma')} \rightarrow^* P''$

and $\Theta \cdot \Gamma \vdash P \rightarrow^* P'$.

Proof. Let $m$ be the length of the sequence of reductions $[\Theta] \cdot [\delta(\Gamma)] \vdash [P]_{\sigma(\Gamma)} \rightarrow P''$. We proceed by induction on $m$:

- base case $m = 0$. We have $\Gamma = [\Theta] \cdot [\delta(\Gamma)]$ and $P = [P]_{\sigma(\Gamma)}$. We conclude by letting $\tilde{x} = \emptyset$, $P'' = P_s$, $\Gamma' = \Gamma$, and $P' = P$;

- inductive case $m = m + 1$. Take $P''_s$ such that:

$[\Theta] \cdot [\delta(\Gamma)] \vdash [P]_{\sigma(\Gamma)} \rightarrow^* P''_s \rightarrow P''$ (107)

By the induction hypothesis, $\exists \tilde{x}_0, P''_s, \Gamma_0, P_0, n_0$ such that:

$\Theta \cdot \Gamma \vdash P \rightarrow^* \Theta \cdot \Gamma_0 \vdash P_0$ and $P''_s \rightarrow^* (\nu\tilde{\nu})P''_s$ and $\left( \text{\textcolor{red}{\textbf{T\Pi-Ref}}y} \right) \frac{\left[ \Theta \cdot \delta(\Gamma) \vdash [P]_{\sigma(\Gamma)} \rightarrow^* P''_s \right]}{\Theta \cdot \Gamma \vdash P''_s \rightarrow^* (\nu\tilde{\nu})P''}$ (108)

By Proposition C.6, $P''_s$ is quasi-linear. Therefore, from (107) and (108), by Cor. C.12, we have either:

- $P_s \rightarrow^* P''_s$. In this case, we conclude by letting $\tilde{x} = \tilde{x}_0$, $P'' = P''_s$, $\Gamma' = \Gamma_0$, and $P' = P$;

- $\exists P'''$ such that $P_s \rightarrow^* P'''$, and $(\nu\tilde{\nu})P''_s \rightarrow P'''$. In this case, the latter transition implies $P''' \equiv (\nu\tilde{\nu})P''_s$ (for some $P''_s$), and (by inverting rule (\texttt{\textbf{T\Pi-Ref}}) once per element of $\tilde{x}_0$) $P''' \rightarrow P''$. Therefore, by (108) and Lemma E.23, we know that $\exists \tilde{x}_0, \Gamma', \nu', P'''_s$ such that $\Theta \cdot \Gamma_0 \vdash P_0 \rightarrow^* \Theta \cdot \Gamma' \vdash P'$, and:

$P'' \rightarrow^* (\nu\tilde{\nu})P'' \rightarrow^* (\nu\tilde{\nu})P''$ and $\left( \text{\textcolor{red}{\textbf{T\Pi-Ref}}y} \right) \frac{\left[ \Theta \cdot \Gamma \vdash P' \right]}{\Theta \cdot \Gamma \vdash P'}$
Therefore, by letting \( \bar{x} = \bar{x}_0 \bar{x}_i \bar{x} \), we conclude \( \Theta \cdot \Gamma \vdash P \rightarrow^* \Theta \cdot \Gamma' \vdash P' \), and:

\[
P_s \rightarrow^* (\nu \bar{x})P'' \quad \text{and} \quad (T_{\pi\text{-Reify}}) \frac{[\Theta \cdot \Gamma \vdash P]}{[\Theta], \delta(\Gamma) + \left[ P \right] \sigma(\Gamma)} \rightarrow^* (\nu \bar{x})P''
\]

\( (T_{\pi\text{-Reify}}) \frac{[\Theta \cdot \Gamma \vdash P]}{[\Theta], \delta(\Gamma) + \left[ P \right] \sigma(\Gamma)} \rightarrow^* (\nu \bar{x})P'' \) and \( (T_{\pi\text{-Reify}}) \frac{[\Theta \cdot \Gamma \vdash P]}{[\Theta], \delta(\Gamma) + \left[ P \right] \sigma(\Gamma)} \rightarrow^* (\nu \bar{x})P'' \)

Lemma E.25. \( \Theta \cdot \Gamma \vdash P \rightarrow P' \) implies that \( \exists \bar{x}', P'' \) such that:

\[
(T_{\pi\text{-Reify}}) \frac{[\Theta \cdot \Gamma \vdash P]}{[\Theta], \delta(\Gamma) + \left[ P \right] \sigma(\Gamma)} \rightarrow^* (\nu \bar{x})P''
\]

**Proof.** By induction on \( \Theta \cdot \Gamma' \vdash P \rightarrow P' \). In the base cases (\( R_{\text{Comm}} \)) and (\( R_{\text{Call}} \)), the shape of \( \Gamma' \) can be determined from the proof of Theorem 2.16 (page 44), and correspondingly, \( \bar{x} \) and \( P'' \) can be determined from a simplification of the proof of Lemma E.23, so that it only contains the processes under study (without other \( \rightarrow^* \)-reducing processes). The inductive cases follow by the induction hypothesis.

Lemma E.26 (Operational completeness of encoding). \( \Theta \cdot \Gamma \vdash P \rightarrow P' \) implies that \( \exists \Gamma', \bar{x}, P'' \) such that:

\[
(T_{\pi\text{-Reify}}) \frac{[\Theta \cdot \Gamma \vdash P]}{[\Theta], \delta(\Gamma) + \left[ P \right] \sigma(\Gamma)} \rightarrow^* (\nu \bar{x})P''
\]

**Proof.** Let \( m \) be the length of the sequence of reductions in \( \Theta \cdot \Gamma \vdash P \rightarrow P' \). We prove a slightly stronger statement with the additional clause: “\( \exists n \) such that \( (\nu \bar{x}) = (\nu x_1) \ldots (\nu x_n) \)”. We proceed by induction on \( m \):

- **Base case** \( m = 0 \). We trivially conclude by letting \( \Gamma' = \Gamma, P' = P \), and \( n = 0 \);
- **Inductive case** \( m = m' + 1 \). Take \( \Gamma', \Theta, P_s \) such that:

  \[
  \Theta \cdot \Gamma \vdash P \xrightarrow{\cdots} \Theta \cdot \Gamma' \vdash P_s \rightarrow \Theta \cdot \Gamma' \vdash P'
  \]

By the induction hypothesis, for some \( n_s \), we have:

\[
(T_{\pi\text{-Reify}}) \frac{[\Theta \cdot \Gamma \vdash P]}{[\Theta], \delta(\Gamma) + \left[ P \right] \sigma(\Gamma)} \rightarrow^* (T_{\pi\text{-Reify}}) \frac{[\Theta \cdot \Gamma \vdash P_s]}{[\Theta], \delta(\Gamma_s) + \left[ P_s \right] \sigma(\Gamma_s)} \rightarrow^* (T_{\pi\text{Res}}) \times n_s
\]

Moreover, from \( \Theta \cdot \Gamma_s \vdash P_s \rightarrow \Theta \cdot \Gamma' \vdash P' \), by Lemma E.25 we get (for some \( n_{**} \)):

\[
(T_{\pi\text{-Reify}}) \frac{[\Theta \cdot \Gamma \vdash P_s]}{[\Theta], \delta(\Gamma_s) + \left[ P_s \right] \sigma(\Gamma_s)} \rightarrow^* (T_{\pi\text{-Res}}) \times n_{**}
\]

Note that each transition in (109) is preserved when fired inside the \( n_s \) delimitations taken from (109), via \( n_s \) applications of rule \( (R_{\pi\text{Res}}) \). Therefore, letting \( n = n_s + n_{**}, \ x_i = x_i' (\forall i \in 1..n_s) \) and \( x_{j+n_s} = x_j' (\forall j \in 1..n_{**}) \), we conclude:

\[
(T_{\pi\text{-Reify}}) \frac{[\Theta \cdot \Gamma \vdash P]}{[\Theta], \delta(\Gamma) + \left[ P \right] \sigma(\Gamma)} \rightarrow^* (T_{\pi\text{-Reify}}) \frac{[\Theta \cdot \Gamma \vdash P]}{[\Theta], \delta(\Gamma) + \left[ P \right] \sigma(\Gamma)} \rightarrow^* (T_{\pi\text{Res}}) \times n
\]
Items (1) and (2) of Theorem E.27 below use $\sigma(\Gamma)$ to allow reductions of encoded open channels with roles (cf. Ex. 5.8). Note that when we write $[P']_{[\Theta]_{\cdot \Gamma}}$, we imply $\Theta \cdot \Gamma \vdash P'$ (cf. Def. 5.7).

**Theorem E.27** (Open operational correspondence). If $\Theta \cdot \Gamma \vdash P$ and $\text{fv}(P) = \emptyset$:

1. **(Completeness)** $P \rightarrow^* P'$ implies $\exists x', \tilde{x}, P''$ such that $[P]_{\Theta \cdot \Gamma} \sigma(\Gamma) \rightarrow^* (\nu \tilde{x})P''$ and $P'' = [P']_{[\Theta]_{\cdot \Gamma}} \sigma(\Gamma')$;

2. **(Soundness)** $[P]_{\Theta \cdot \Gamma} \sigma(\Gamma) \rightarrow^* P$, implies $\exists \tilde{x}, P', \tilde{\gamma}, P': \Theta \vdash^* (\nu \tilde{x})P''$, $P \rightarrow^* P'$ and $[P']_{\Theta \cdot \Gamma} \sigma(\Gamma') \xrightarrow{\text{with}^*} P''$.

**Proof.** We prove the following equivalent formulation of the statement:

1. **(Completeness)** $\Theta \cdot \Gamma \vdash P \rightarrow^* P'$ implies that $\exists x', \tilde{x}, P''$ such that:

   $\frac{[\Theta], \delta(\Gamma) \vdash [P]_{\Theta \cdot \Gamma} \sigma(\Gamma)}{\Theta, \delta(\Gamma) \vdash (\nu \tilde{x})P''}$ and

   $\frac{[\Theta], \delta(\Gamma) \vdash [P']_{\Theta \cdot \Gamma} \sigma(\Gamma')}{[\Theta], \delta(\Gamma) \vdash P'} \xrightarrow{\text{with}^*} P''$

2. **(Soundness)** If $\Theta \cdot \Gamma \vdash P$;

   $\frac{[\Theta], \delta(\Gamma) \vdash [P]_{\Theta \cdot \Gamma} \sigma(\Gamma)}{\Theta, \delta(\Gamma) \vdash P'} \xrightarrow{\text{with}^*} P''$

   and $\Theta \cdot \Gamma \vdash P \rightarrow^* P'$.

   Item 1 holds by Lemma E.26. Item 2 holds by Lemma E.24.

**Theorem 6.6** (Operational correspondence). If $\emptyset \vdash P$, then:

1. **(Completeness)** $P \rightarrow_{\text{op}} P'$ implies $\exists x, P''$ such that $[P] \rightarrow^* (\nu \tilde{x})P''$ and $P'' = [P']$;

2. **(Soundness)** $[P] \rightarrow_{\text{op}} P$, implies $\exists \tilde{x}, P', P'$ s.t. $P \rightarrow^* (\nu \tilde{x})P''$, $P \rightarrow^* P'$ and $[P'] \xrightarrow{\text{with}^*} P''$.

**Proof.** Direct consequence of Theorem E.27, noticing that $\sigma(\emptyset)$ is vacuous.