Session Types Revisited

Ornela Dardha, Elena Giachino, Davide Sangiorgi

Session Types Revisited

Ornela Dardha[a], Elena Giachino[b], Davide Sangiorgi[b]

[a] *School of Computing Science, University of Glasgow, United Kingdom*
[b] *INRIA Focus Team / DISI, University of Bologna, Italy*

Abstract

Session types are a formalism used to model structured communication-based programming. A binary session type describes communication by specifying the type and direction of data exchanged between two parties. When session types and session processes are added to the syntax of standard π-calculus they give rise to additional separate syntactic categories. As a consequence, when new type features are added, there is duplication of effort in the theory: the proofs of properties must be checked both on standard types and on session types. We show that session types are encodable into standard π-types, relying on linear and variant types. Besides being an expressivity result, the encoding (i) removes the above redundancies in the syntax, and (ii) the properties of session types are derived as straightforward corollaries, exploiting the corresponding properties of standard π-types. The robustness of the encoding is tested on a few extensions of session types, including subtyping, polymorphism and higher-order communications.

Keywords: session types, π-calculus, linear types, variant types, encoding.

1. Introduction

In complex distributed systems, participants willing to communicate should previously agree on a protocol to follow. The specified protocol describes the types of messages that are exchanged as well as their direction. In this context *session types* [15, 28, 16] came into play: they describe a protocol as a type abstraction. Session types were originally designed for process calculi. However, they have been studied also for other paradigms, such as multi-threaded functional programming [31], component-based systems [29], object-oriented languages [10, 11, 3], Web Services and Contracts, W3C-CDL a language for choreography [5, 21] and many more. Session types are a type formalism proposed as a theoretical foundation to describe and model structured communication-based programming, guaranteeing properties like session fidelity, privacy and communication safety.

Email addresses: Ornela.Dardha@glasgow.ac.uk (Ornela Dardha), egiachino@gmail.com (Elena Giachino), davide.sangiorgi@gmail.com (Davide Sangiorgi)

Session types are defined as a sequence of input and output operations, explicitly indicating the types of messages being transmitted. This structured *sequentiality* of operations is what makes session types suitable to model protocols. However, they offer more flexibility than just performing inputs and outputs: they also permit internal and external choice. Branch and select are typical type (and term) constructs in the theory of session types, the former being the offering of a set of alternatives and the latter being the selection of one of the possible options at hand.

As mentioned above, session types guarantee session fidelity, privacy and communication safety. Session fidelity guarantees that the session channel has the expected structure. Privacy is guaranteed since session channels are known and used only by the participants involved in the communication. Such communication proceeds without any mismatch of direction and of message type. In order to achieve communication safety, a session channel is split by giving rise to two opposite endpoints, each of which is owned by one of the participants. These endpoints are used according to dual behaviours and thus have dual types, namely one participant sends what the other one is expecting to receive and vice versa. So, *duality* is a key concept in the theory of session types as it is the ingredient that guarantees communication safety.

To better understand session types and the notion of duality, let us consider a simple example: the equality test. A client and a server communicate over a session channel. The endpoints x and y of the session channel are owned by the client and the server respectively and exclusively and must have dual types. To guarantee duality of types, static checks are performed by the type system. If the type of x is

$$?\texttt{Int}.?\texttt{Int}.!\texttt{Bool}.\texttt{end}$$

— meaning that the process listening on channel endpoint x receives (?) an integer value followed by another integer value and then sends (!) back a boolean value corresponding to the equality test of the integers received — then the type of y should be

$$!\texttt{Int}.!\texttt{Int}.?\texttt{Bool}.\texttt{end}$$

— meaning that the process listening on channel endpoint y sends an integer value followed by another integer value and then waits to receive back a boolean value — which is exactly the dual type.

There is a precise moment at which a session between two participants is established. It is the *connection* phase, when a fresh (private) session channel is created and its endpoints are bound to each communicating process. The connection is also the moment when duality, hence compliance of two session types, is verified. In order to establish a connection, primitives like `accept/request` or $(\boldsymbol{\nu} xy)$, are added to the syntax of terms [28, 16, 30].

Session types and session terms are added to the syntax of standard π-calculus types and terms, respectively. In doing so, the syntax of types often needs to be split into two separate syntactic categories, one for session types and the other for standard π-calculus types [28, 16, 33, 14] (this often introduces a duplication of type environments, as well). Common typing features,

like subtyping, polymorphism, recursion are then added to both syntactic categories. Also the syntax of processes will contain both standard π-calculus process constructs and session process constructs (for example, the constructs mentioned above to create session channels). This redundancy in the syntax brings in redundancy also in the theory, and can make the proofs of properties of the language heavy. In particular, this duplication becomes more obvious in proofs by structural induction on types. Moreover, if a new type construct is added, the corresponding properties must be checked both on standard π-types and on session types. By "standard type systems" we mean type systems originally studied in depth in sequential languages such as the λ-calculus and then transplanted onto the π-calculus as types for channel names (rather than types for terms as in the λ-calculus). Such type systems may include constructs for products, records, variants, polymorphism, linearity, and so on.

In this paper we aim to understand to which extent this redundancy is necessary, in the light of the following similarities between session constructs and standard π-calculus constructs. Consider ?`Int`.?`Int`.!`Bool`.`end`. This type is assigned to a session channel endpoint and it describes a structured sequence of inputs and outputs by specifying the type of messages that it can transmit. This way of proceeding reminds us of the *linearised* channels [20], which are channels used multiple times for communication but only in a sequential manner. This paper [20] discusses the possibility of encoding linearised channel types into linear types—i.e., channel types used *exactly once*.

The considerations above deal with input and output operations and the sequentiality of session types. Let us consider branch and select. These constructs give more flexibility by offering and selecting a range of possibilities. This brings in mind an already existing type construct in the π-calculus, namely the *variant* type [26, 27]. Another analogy between the session types theory and the standard π-types theory, concerns duality. As mentioned above, duality is checked when connection takes place, in the typing rule for channel restriction. Duality describes the split of behaviour of session channel endpoints. This reminds us of the split of input and output *capabilities* of π-types: once a new channel is created via the ν construct, it can then be used by the two communicating processes owning the opposite capabilities.

In this paper, by following Kobayashi's approach [19], we define an encoding of binary session types into standard π-types and by exploiting this encoding, session types and their theory are shown to be derivable from the theory of the standard typed π-calculus. For instance, basic properties such as subject reduction and type safety become straightforward corollaries. Intuitively, a session type is interpreted as a linear channel type carrying a pair consisting of the original payload type and a new linear channel type, which is going to be used for the continuation of the communication. Furthermore, we present an optimisation of linear channels enabling the reuse of the same channel, instead of a new one, for the continuation of the communication. As stated above, the encoding we adopt follows Kobayashi [19] and the constructs we use are not new (linear types and variant types are well-known concepts in type theory and they are also well integrated in the π-calculus). Indeed the technical contribu-

tion of the paper may be considered minor (the main technical novelty being the optimisation in linear channel usage mentioned above). Rather than technical, the contribution of the paper is meant to be foundational: we show that Kobayashi's encoding

(i) does permit to derive session types and their basic properties; and
(ii) is a robust encoding.

As evidence for (ii), in this paper we examine, besides plain session types, a few extensions of them, such as subtyping, polymorphism and higher-order features in Sections 4, 5 and 6, respectively. These are non-trivial extensions, which have been studied in dedicated session types papers [14, 12, 22]. In each case we show that we can derive the main results of these papers via the encoding, as straightforward corollaries. As long as the encoding is concerned, these extensions follow the same line as the encoding of types and terms given in Section 3. We will avoid repeating technical results, when it is not necessary. Hence, Sections 4, 5 and 6 are less detailed than Section 3. While Kobayashi's encoding was generally known, its strength, robustness, and practical impact were not. This is witnessed by the plethora of papers on session types over the last 20 years, in which session types are always taken as primitives — we are not aware of a single work that explains the results on session types via an encoding of them into standard types. In our opinion, the reasons why Kobayashi's encoding had not caught attention are:

(a) Kobayashi did not prove any properties of the encoding and did not investigate its robustness;
(b) as certain key features of session types do not clearly show up in the encoding, the faithfulness of the encoding was unclear.

A good example for (b) is duality. In session types theory, duality plays a central role: a session is identified by two channel endpoints, and these must have dual types. In the standard typed π-calculus, in contrast, there is no notion of duality on types. Indeed, in the encoding, dual session types (e.g., the branch type and the select type) are mapped onto the same type (e.g., the variant type). In general, dual session types will be mapped onto linear types that are identical except for the outermost I/O tag — duality on session types boils down to the opposite input and output capabilities of channels.

The results in the paper are not however meant to imply that session types are useless, as they are very useful from a programming perspective. The work just tells us that, at least for the binary sessions and properties examined in the paper, session types and session primitives may be taken as macros. This paper is an extension of the conference version [8] and further details can be found in the first author's published Ph.D. thesis [7].

Structure: The remainder of the paper is structured as follows. Section 2 gives an overview of session types and standard π-calculus types as well as language terms, typing rules and operational semantics. Section 3 presents the encoding of both session types and session processes. Sections 4, 5 and 6 present extensions of session types: subtyping, polymorphism and higher-order

$T ::=$	S	(session type)	$S ::=$	$\texttt{end}$	(termination)
	$\sharp T$	(channel type)		$?T.S$	(receive)
	$\texttt{Unit}$	(unit type)		$!T.S$	(send)
	$\ldots$	(other constructs)		$\&\{l_i : S_i\}_{i \in I}$	(branch)
				$\oplus\{l_i : S_i\}_{i \in I}$	(select)

$P, Q ::=$	$x!\langle v \rangle.P$	(output)	$\mathbf{0}$	(inaction)
	$x?(y).P$	(input)	$P \mid Q$	(composition)
	$x \lhd l_j.P$	(selection)	$(\boldsymbol{\nu} xy)P$	(session restriction)
	$x \rhd \{l_i : P_i\}_{i \in I}$	(branching)	$(\boldsymbol{\nu} x)P$	(channel restriction)
$v ::=$	x	(name)	$\star$	(unit value)

Figure 1: Syntax of the π-calculus with session types

communication, respectively and analyse the encoding with respect to these extensions. Section 7 presents an optimisation of linear channel usage. Section 8 examines related work and Section 9 concludes the paper.

2. Background

In this section we give an overview of the two theories we will be working with: session types theory and standard typed π-calculus theory.

2.1. Session Types

Type Syntax. Types are presented at the top of Fig. 1. The syntax of types is given by two separate syntactic categories: one for session types and the other for standard π-types, which includes session types. We use S to range over session types and T to range over types. Session types are: $\texttt{end}$, the type of a terminated session endpoint; $?T.S$ and $!T.S$ indicating, respectively a session type assigned to an endpoint used to *receive* and to *send* a value of type T and then continue according to the protocol specified by session type S. Branch and select are sets of labelled session types, where the order of components does not matter and labels are all distinct. The labelled components of branch and select range over an index set I. Branch $\&\{l_i : S_i\}_{i \in I}$ indicates *external choice*, namely what is offered, and it is a generalisation of the input type. Dually, select $\oplus\{l_i : S_i\}_{i \in I}$ indicates the *internal choice*, only one of the labels will be chosen, and it is a generalisation of the output type. Types T include session types, standard channel types $\sharp T$, $\texttt{Unit}$ type and if required other standard π-type constructs, such as other ground types like $\texttt{Int}, \texttt{String}, \ldots$, or classes, data types etc.

Language Syntax. The syntax of terms is presented at the bottom of Fig. 1 and it follows [30]. There are different ways of presenting session channel initiation and endpoints, like `accept/request` [16], polarised channels [14] or by means of *co-names* [30]. Standard communication (not involving sessions) is based on standard π-calculus channels [16, 14], whereas in [30] it is based on co-names. In this paper we use co-names for session communication and standard π-calculus channels otherwise. Co-names specify the two opposite endpoints of a communication channel and are created and bound together by the restriction construct. Our results can be applied to all the different syntaxes in session types theory.

We use P, Q to range over processes, x, y over names and v to range over values. We use $\mathtt{fn}(P)$ to denote the set of free names in P, $\mathtt{bn}(P)$ to denote the bound ones and $\mathtt{n}(P) = \mathtt{fn}(P) \cup \mathtt{bn}(P)$ to denote the set of all names in P. We adopt the Barendregt name convention, namely that all names in bindings in any mathematical context are pairwise distinct and distinct from the free names. The output process $x!\langle v\rangle.P$ sends a value v on x and proceeds as process P; the input process $x?(y).P$ receives on x a value to substitute the placeholder y in the continuation process P. The selection process $x \lhd l_j.P$ on x selects label l_j and proceeds as process P. The branching process $x \rhd \{l_i : P_i\}_{i\in I}$ on x offers a range of alternatives each labelled with a different label ranging over the index set I. According to the selected label l_j the process P_j will be executed. The process $\mathbf{0}$ is the standard inaction process. The last two constructs represent restriction. $(\boldsymbol{\nu} xy)P$ is the session restriction construct; it creates a session channel, more precisely its two endpoints x and y and binds them in P. The two endpoints should be distinguished to validate subject reduction (see [33]). The type system enforces duality of behaviours on endpoints. Process $(\boldsymbol{\nu} x)P$ is the standard channel restriction; it creates a new channel x and binds it with scope P.

Duality. Two processes willing to communicate, e.g., a client and a server, must first agree on a protocol. The protocol is abstracted as a structured type, namely a session type. Intuitively, client and server should perform dual operations: when one process sends, the other receives, when one offers, the other chooses. So, the dual of an input is an output, the dual of branch is select, and vice versa. Formally, duality on session types is defined as:

$$\begin{array}{rcl}
\overline{\mathtt{end}} & \triangleq & \mathtt{end} \\
\overline{!T.S} & \triangleq & ?T.\overline{S} \\
\overline{?T.S} & \triangleq & !T.\overline{S} \\
\overline{\oplus\{l_i : S_i\}_{i\in I}} & \triangleq & \&\{l_i : \overline{S}_i\}_{i\in I} \\
\overline{\&\{l_i : S_i\}_{i\in I}} & \triangleq & \oplus\{l_i : \overline{S}_i\}_{i\in I}
\end{array}$$

In order to guarantee that communication is safe and proceeds without any mismatch, static checks are performed by the type system. These checks control that the opposite endpoints of the same session channel have dual types, as we will see shortly.

$$\frac{}{\varnothing = \varnothing \circ \varnothing} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2 \quad \mathtt{un}(T)}{\Gamma, x : T = (\Gamma_1, x : T) \circ (\Gamma_2, x : T)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2 \quad \mathtt{lin}(S)}{\Gamma, x : S = (\Gamma_1, x : S) \circ \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2 \quad \mathtt{lin}(S)}{\Gamma, x : S = \Gamma_1 \circ (\Gamma_2, x : S)}$$

Figure 2: Context split for session types

Typing Rules. A typing context Γ is a partial function from names to types. We use $\mathtt{dom}(\Gamma)$ to denote the domain of Γ. Typing judgements for values are of the form $\Gamma \vdash v : T$, reading "a value v is of type T in a typing context Γ". Typing judgements for processes are of the form $\Gamma \vdash P$, reading "a process P is well typed in a typing context Γ". In order to deal with linearity, the typing rules make use of `lin` and `un` predicates and a context split operator '$\circ$'. The `lin` and `un` predicates state when a type, or a typing context, is linear or unrestricted, respectively [30].

$\mathtt{lin}(T)$	if T is a session type and $T \neq \mathtt{end}$
$\mathtt{un}(T)$	otherwise
$\mathtt{lin}(\Gamma)$	if there is $(x : T) \in \Gamma$ such that $\mathtt{lin}(T)$
$\mathtt{un}(\Gamma)$	otherwise

The context split '$\circ$' is defined by the rules in Fig. 2. Intuitively, these rules state that a typing context is split in a way that linear names occur only in one of the halves. This does not hold for the unrestricted names.

The typing rules for the π-calculus with session types are given in Fig. 3. Rule (T-Var) states that a name x is of type T, if this is the type assumed in the typing context. Rule (T-Val) states that a unit value is of unit type. Rule (T-Inact) states that the terminated process $\mathbf{0}$ is always well typed under any Γ. Notice that in all the previous rules, the typing context Γ is unrestricted. Rule (T-Par) types the parallel composition of two processes under the combination of typing contexts by using the split operator. The rule that performs duality checks is the rule for session restriction (T-Res). Process $(\boldsymbol{\nu} xy)P$ is well typed in Γ, if P is well typed in Γ augmented with session channel endpoints having dual types, namely $x : T$ and $y : \overline{T}$. Rule (T-In) splits in two the context in which the input process $x?(y).P$ is well typed: one part typechecks x, the other part augmented with $y : T$ and updated with $x : S$, typechecks the continuation process P. The rule for output (T-Out) is similar. The context is split in three parts, one to typecheck x, another to typecheck v and the last part to typecheck the continuation process P. Similarly to the input rule, the continuation process uses channel x with its continuation type S. In addition to the typing rules for session restriction, input and output, there are also the corresponding ones for standard channel types, namely the typing rules (T-StndRes), (T-StndIn) and (StndOut). This is an example of the duplication of rules and work that is needed in presentations of the π-calculus with session types. Let us now consider

$$\text{(T-Var)}\ \frac{\mathtt{un}(\Gamma)}{\Gamma, x:T \vdash x:T} \qquad \text{(T-Val)}\ \frac{\mathtt{un}(\Gamma)}{\Gamma \vdash \star : \mathtt{Unit}} \qquad \text{(T-Inact)}\ \frac{\mathtt{un}(\Gamma)}{\Gamma \vdash \mathbf{0}} \qquad \text{(T-Par)}\ \frac{\Gamma_1 \vdash P \qquad \Gamma_2 \vdash Q}{\Gamma_1 \circ \Gamma_2 \vdash P \mid Q}$$

$$\text{(T-Res)}\ \frac{\Gamma, x:T, y:\overline{T} \vdash P}{\Gamma \vdash (\boldsymbol{\nu} xy)P} \qquad \text{(T-StndRes)}\ \frac{\Gamma, x:T \vdash P \qquad T \text{ is not a session type}}{\Gamma \vdash (\boldsymbol{\nu} x)P}$$

$$\text{(T-In)}\ \frac{\Gamma_1 \vdash x : ?T.S \qquad \Gamma_2, x:S, y:T \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash x?(y).P} \qquad \text{(T-StndIn)}\ \frac{\Gamma_1 \vdash x : \sharp T \qquad \Gamma_2, x:\sharp T, y:T \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash x?(y).P}$$

$$\text{(T-Out)}\ \frac{\Gamma_1 \vdash x : !T.S \qquad \Gamma_2 \vdash v:T \qquad \Gamma_3, x:S \vdash P}{\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash x!\langle v\rangle.P} \qquad \text{(T-StndOut)}\ \frac{\Gamma_1 \vdash x : \sharp T \qquad \Gamma_2 \vdash v:T \qquad \Gamma_3, x:\sharp T \vdash P}{\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash x!\langle v\rangle.P}$$

$$\text{(T-Brch)}\ \frac{\Gamma_1 \vdash x : \&\{l_i : T_i\}_{i\in I} \qquad \Gamma_2, x:T_i \vdash P_i \quad \forall i \in I}{\Gamma_1 \circ \Gamma_2 \vdash x \rhd \{l_i : P_i\}_{i\in I}} \qquad \text{(T-Sel)}\ \frac{\Gamma_1 \vdash x : \oplus\{l_i : T_i\}_{i\in I} \qquad \Gamma_2, x:T_j \vdash P \quad \exists j \in I}{\Gamma_1 \circ \Gamma_2 \vdash x \lhd l_j.P}$$

Figure 3: Typing rules for the π-calculus with session types

the typing rules (T-Brch) and (T-Sel). The branching process $x \rhd \{l_i : P_i\}_{i\in I}$ is well typed if channel x is of branch type $\&\{l_i : T_i\}_{i\in I}$ and every continuation process P_i is well typed and uses x with type T_i. To typecheck a process that selects label l_j on channel x of type $\oplus\{l_i : T_i\}_{i\in I}$, we typecheck the continuation process P_j that uses x with type T_j.

Operational Semantics. The operational semantics is defined as a binary relation $\rightarrow$ over processes and is given in Fig. 4. Rule (R-StndCom) is the standard communication rule. In rule (R-Com), two processes communicate on two co-names, and the value so received replaces the input placeholder. Rule (R-Case) is similar: the communicating processes have prefixes that are co-names, and the label received selects the continuation on the recipient side. Rules (R-StndRes), (R-Res), (R-Par), and (R-Struct) are standard, stating that communication can happen under restriction and parallel composition and allowing to exploit the structural congruence relation.

$$
\begin{array}{rl}
\text{(R- StndCom)} & x!\langle v\rangle.P \mid x?(z).Q \to P \mid Q[v/z] \\
\text{(R-Com)} & (\boldsymbol{\nu} xy)(x!\langle v\rangle.P \mid y?(z).Q) \to (\boldsymbol{\nu} xy)(P \mid Q[v/z]) \\
\text{(R-Case)} & (\boldsymbol{\nu} xy)(x \lhd l_j.P \mid y \rhd \{l_i : P_i\}_{i\in I}) \to (\boldsymbol{\nu} xy)(P \mid P_j) \quad j \in I \\
\text{(R-StndRes)} & P \to Q \Longrightarrow (\boldsymbol{\nu} x)P \to (\boldsymbol{\nu} x)Q \\
\text{(R-Res)} & P \to Q \Longrightarrow (\boldsymbol{\nu} xy)P \to (\boldsymbol{\nu} xy)Q \\
\text{(R-Par)} & P \to Q \Longrightarrow P \mid R \to Q \mid R \\
\text{(R-Struct)} & P \equiv P',\ P \to Q,\ Q' \equiv Q \Longrightarrow P' \to Q'
\end{array}
$$

Figure 4: Semantics for the π-calculus with session types

$$
\begin{array}{rcl}
P \mid Q & \equiv & Q \mid P \\
(P \mid Q) \mid R & \equiv & P \mid (Q \mid R) \\
P \mid \mathbf{0} & \equiv & P \\
(\boldsymbol{\nu} xy)\mathbf{0} & \equiv & \mathbf{0} \\
(\boldsymbol{\nu} x)\mathbf{0} & \equiv & \mathbf{0} \\
(\boldsymbol{\nu} xy)(\boldsymbol{\nu} zw)P & \equiv & (\boldsymbol{\nu} zw)(\boldsymbol{\nu} xy)P \\
(\boldsymbol{\nu} x)P \mid Q & \equiv & (\boldsymbol{\nu} x)(P \mid Q) \quad (x \notin \mathtt{fn}(Q)) \\
(\boldsymbol{\nu} xy)(\boldsymbol{\nu} zw)P & \equiv & (\boldsymbol{\nu} zw)(\boldsymbol{\nu} xy)P \\
(\boldsymbol{\nu} xy)P \mid Q & \equiv & (\boldsymbol{\nu} xy)(P \mid Q) \quad (x, y \notin \mathtt{fn}(Q))
\end{array}
$$

In order to complete the operational semantics, *structural congruence* relation, denoted as $\equiv$, is needed and is defined as the smallest congruence relation on processes that satisfies the above axioms. The first three axioms state that parallel composition of processes is commutative, associative and uses process $\mathbf{0}$ as the neutral element. The remaining axioms involve restriction, the main ones being *scope extrusion* stating that the scope of a restriction can be extended to other processes in parallel provided that no capture of (session co-) names occurs. To conclude, let C, D range over *contexts*. Intuitively, a context is a process with a hole [27].

Properties. We recall some basic properties of the session π-calculus [30]. The weakening lemma states that it is sound to introduce unrestricted type assumptions in a typing context.

Lemma 1 (Weakening in Sessions)**.** If $\Gamma \vdash P$ and $x \notin \mathtt{fn}(P)$ and $\mathtt{un}(T)$, then $\Gamma, x : T \vdash P$.

The strengthening lemma states somehow the opposite of weakening: it is sound to remove unrestricted names from the typing context provided that they do not occur free in the process being typed.

Lemma 2 (Strengthening in Sessions)**.** If $\Gamma, x : T \vdash P$ and $x \notin \mathtt{fn}(P)$ and $\mathtt{un}(T)$, then $\Gamma \vdash P$.

$\tau ::=$	$\ell_{\mathtt{o}}[\widetilde{\tau}]$	(linear output)	$\sharp[\widetilde{\tau}]$	(connection)	
	$\ell_{\mathtt{i}}[\widetilde{\tau}]$	(linear input)	$\langle l_i\_\tau_i\rangle_{i\in I}$	(variant type)	
	$\ell_{\sharp}[\widetilde{\tau}]$	(linear connection)	`Unit`	(unit type)	
	$\varnothing[\,]$	(no capability)	$\ldots$	(other constructs)	

$P, Q ::=$	$x!\langle\tilde{v}\rangle.P$	(output)	$\mathbf{0}$	(inaction)
	$x?(\tilde{y}).P$	(input)	$P \mid Q$	(composition)
	$(\boldsymbol{\nu} x)P$	(restriction)	$\mathbf{case}\, v\, \mathbf{of}\, \{l_i\_(x_i) \triangleright P_i\}_{i\in I}$	(case)
$v ::=$	x	(name)	$\star$	(unit value)
	l_v	(variant value)		

Figure 5: Syntax of the standard typed π-calculus

We are ready now to state the subject congruence and the subject reduction properties for the session π-calculus.

Lemma 3 (Subject Congruence for Sessions)**.** If $\Gamma \vdash P$ and $P \equiv P'$, then $\Gamma \vdash P'$.

Theorem 4 (Subject Reduction for Sessions)**.** If $\Gamma \vdash P$ and $P \to Q$, then $\Gamma \vdash Q$.

2.2. π-Types

Type Syntax. We now consider the standard typed π-calculus [27]. The syntax of the type constructs is presented at the top of Fig. 5. The standard π-types, ranged over by τ, include various type constructs. Here we focus on linear types and variant types, which will be used in the encoding. We use a tilde $\widetilde{\ }$ to indicate a sequence of elements. Linear types $\ell_{\mathtt{i}}[\widetilde{\tau}]$, $\ell_{\mathtt{o}}[\widetilde{\tau}]$ and $\ell_{\sharp}[\widetilde{\tau}]$ are assigned to channels used *exactly once* in input to receive messages of type $\widetilde{\tau}$, in output to send messages of type $\widetilde{\tau}$ and used once for sending and once for receiving messages of type $\widetilde{\tau}$, respectively. The type $\varnothing[\,]$ is assigned to a channel without any capability. We use α, β to range over the $\mathtt{i}$, $\mathtt{o}$ or $\sharp$ capabilities. Type $\sharp[\widetilde{\tau}]$ indicates a channel used for communication without any restriction. The variant type $\langle l_i\_\tau_i\rangle_{i\in I}$ is a labelled form of disjoint union of types. The order of the components does not matter and labels are all distinct. `Unit` type is standard. Other type constructs, like ground types and recursive types, can be added to the syntax.

We define a notion of duality on π-types to be the duality on the capability of the channel, via the following rules:

$$\begin{array}{rcl} \overline{\ell_{\mathtt{i}}[\widetilde{\tau}]} & = & \ell_{\mathtt{o}}[\widetilde{\tau}] \\ \overline{\ell_{\mathtt{o}}[\widetilde{\tau}]} & = & \ell_{\mathtt{i}}[\widetilde{\tau}] \\ \overline{\varnothing[\,]} & = & \varnothing[\,] \end{array}$$

Language Syntax. The syntax of terms of the π-calculus is given at the bottom of Fig. 5. The output process $x!\langle\tilde{v}\rangle.P$ sends a tuple of values $\tilde{v}$ on channel x

$$\big(\Gamma_1 \uplus \Gamma_2\big)(x) \triangleq \begin{cases} \Gamma_1(x) \uplus \Gamma_2(x) & \text{if both } \Gamma_1(x) \text{ and } \Gamma_2(x) \text{ are defined} \\ \Gamma_1(x) & \text{if } \Gamma_1(x), \text{ but not } \Gamma_2(x), \text{ is defined} \\ \Gamma_2(x) & \text{if } \Gamma_2(x), \text{ but not } \Gamma_1(x), \text{ is defined} \\ \mathtt{undef} & \text{otherwise} \end{cases}$$

Figure 6: Context combination for linear π-types

and proceeds as P; input process $x?(\tilde{y}).P$ receives on x a tuple of values that is going to substitute $\tilde{y}$ in P; restriction process $(\boldsymbol{\nu} x)P$ creates a new name x and binds it with scope P; differently from session π-calculus, here we have only one restriction process. Inaction and parallel composition are standard. Process $\mathbf{case}\, v\, \mathbf{of}\, \{l_i\_(x_i) \triangleright P_i\}_{i \in I}$ offers different behaviours depending on which variant value l_v it receives. Values include names, variant values and the unit value.

Typing Rules. The predicates `lin` and `un` on the standard π-types and typing contexts are defined as:

$\mathtt{lin}(\tau)$	if $\tau = \ell_\alpha[\widetilde{\tau}]$ or $\big(\tau = \langle l_i\_\tau_i\rangle_{i\in I}$ and for some $j \in I$. $\mathtt{lin}(\tau_j)\big)$
$\mathtt{un}(\tau)$	otherwise
$\mathtt{lin}(\Gamma)$	if there is $(x : \tau) \in \Gamma$ such that $\mathtt{lin}(\tau)$
$\mathtt{un}(\Gamma)$	otherwise

As for session types, also for linear types there is a careful handling of typing contexts in order to ensure linearity. The combination '$\uplus$' of types is a symmetric operation and is defined by the following rules, and the combination of typing contexts is defined in Fig. 2.2.

$$\begin{array}{rcll} \ell_{\mathtt{i}}[\widetilde{\tau}] \uplus \ell_{\mathtt{o}}[\widetilde{\tau}] & \triangleq & \ell_\sharp[\widetilde{\tau}] & \\ \tau \uplus \tau & \triangleq & \tau & \text{if } \mathtt{un}(\tau) \\ \tau \uplus \tau' & \triangleq & \mathtt{undef} & \text{otherwise} \end{array}$$

Typing judgements for the standard typed π-calculus have the same shape as the corresponding ones for the session typed π-calculus. The typing rules are given in Fig. 7. Rule (Tπ-Var) states that a name has type the one assumed in the typing context. Rule (Tπ-Val) states that a unit value has type `Unit`. Both rules use an unrestricted typing context. Rule (Tπ-Inact) states that the terminated process $\mathbf{0}$ is well typed in every unrestricted typing context. Rule (Tπ-Par) states that the parallel composition of two processes is well typed in the combination of typing contexts used to type each of the processes. There are two typing rules for the restriction process $(\boldsymbol{\nu} x)P$. Rule (Tπ-Res1) states that the restriction process $(\boldsymbol{\nu} x)P$ is well typed if process P is well typed under the same typing context augmented with $x : \ell_\sharp[\widetilde{\tau}]$. By applying the definition of context combination given in Fig. 2.2, we have $x : \ell_\sharp[\widetilde{\tau}] = x : \ell_{\mathtt{i}}[\widetilde{\tau}] \uplus \ell_{\mathtt{o}}[\widetilde{\tau}]$. This implies that process P owns both capabilities of input and output of channel x.

$$\frac{\mathtt{un}(\Gamma)}{\Gamma, x:\tau \vdash x:\tau}\ (\text{T}\pi\text{-Var}) \qquad \frac{\mathtt{un}(\Gamma)}{\Gamma \vdash \star : \mathtt{Unit}}\ (\text{T}\pi\text{-Val}) \qquad \frac{\mathtt{un}(\Gamma)}{\Gamma \vdash \mathbf{0}}\ (\text{T}\pi\text{-Inact})$$

$$\frac{\Gamma_1 \vdash P \qquad \Gamma_2 \vdash Q}{\Gamma_1 \uplus \Gamma_2 \vdash P \mid Q}\ (\text{T}\pi\text{-Par}) \qquad \frac{\Gamma, x : \ell_{\sharp}[\widetilde{\tau}] \vdash P}{\Gamma \vdash (\boldsymbol{\nu} x)P}\ (\text{T}\pi\text{-Res1}) \qquad \frac{\Gamma, x : \varnothing[\,] \vdash P}{\Gamma \vdash (\boldsymbol{\nu} x)P}\ (\text{T}\pi\text{-Res2})$$

$$\frac{\Gamma_1 \vdash x : \ell_{\mathtt{i}}[\widetilde{\tau}] \qquad \Gamma_2, \tilde{y} : \widetilde{\tau} \vdash P}{\Gamma_1 \uplus \Gamma_2 \vdash x?(\tilde{y}).P}\ (\text{T}\pi\text{-Inp}) \qquad \frac{\Gamma_1 \vdash x : \ell_{\mathtt{o}}[\widetilde{\tau}] \qquad \widetilde{\Gamma_2} \vdash \tilde{v} : \widetilde{\tau} \qquad \Gamma_3 \vdash P}{\Gamma_1 \uplus \widetilde{\Gamma_2} \uplus \Gamma_3 \vdash x!\langle \tilde{v} \rangle.P}\ (\text{T}\pi\text{-Out})$$

$$\frac{\Gamma \vdash v : \tau_j \qquad j \in I}{\Gamma \vdash l_j \_ v : \langle l_i \_ \tau_i \rangle_{i \in I}}\ (\text{T}\pi\text{-LVal}) \qquad \frac{\Gamma_1 \vdash v : \langle l_i \_ \tau_i \rangle_{i \in I} \qquad \Gamma_2, x_i : \tau_i \vdash P_i \quad \forall i \in I}{\Gamma_1 \uplus \Gamma_2 \vdash \mathbf{case}\, v\, \mathbf{of}\, \{l_i \_ (x_i) \triangleright P_i\}_{i \in I}}\ (\text{T}\pi\text{-Case})$$

Figure 7: Typing rules for the standard typed π-calculus

$$\begin{aligned}
&(\text{R}\pi\text{-Com}) && x!\langle \tilde{v} \rangle.P \mid x?(\tilde{z}).Q \to P \mid Q[\tilde{v}/\tilde{z}] \\
&(\text{R}\pi\text{-Case}) && \mathbf{case}\, l_j _ v\, \mathbf{of}\, \{l_i _ (x_i) \triangleright P_i\}_{i \in I} \to P_j[v/x_j] \quad j \in I \\
&(\text{R}\pi\text{-Res}) && P \to Q \Longrightarrow (\boldsymbol{\nu} x)P \to (\boldsymbol{\nu} x)Q \\
&(\text{R}\pi\text{-Par}) && P \to Q \Longrightarrow P \mid R \to Q \mid R \\
&(\text{R}\pi\text{-Struct}) && P \equiv P',\ P \to Q,\ Q' \equiv Q \Longrightarrow P' \to Q'
\end{aligned}$$

Figure 8: Semantics for the standard typed π-calculus

This is a fundamental feature used in the encoding. Rule (Tπ-Res2) states that $(\boldsymbol{\nu} x)P$ is well typed if P is well typed and x has no capabilities in P. This rule is needed in the standard typed π-calculus to prove subject reduction (see [27]), and it is needed also for our encoding. Rules (Tπ-Inp) and (Tπ-Out) state that the input and output processes are well typed if x is a linear channel used in input and output, respectively and the carried types are compatible with the types of $\tilde{y}$ and $\tilde{v}$, respectively We use $\widetilde{\Gamma}$ to denote $\Gamma_1 \uplus \ldots \uplus \Gamma_k$ such that k is the length of the sequence denoted by $\widetilde{\cdot}$. A variant value $l_j \_ v$ is of type $\langle l_i \_ \tau_i \rangle_{i \in I}$ if v is of type τ_j for $j \in I$. Process $\mathbf{case}\, v\, \mathbf{of}\, \{l_i \_ (x_i) \triangleright P_i\}_{i \in I}$ is well typed if value v has variant type and every process P_i is well typed assuming x_i has type τ_i.

Operational Semantics. The semantics of the π-calculus is presented in Fig. 8. Rule (Rπ-Com) is very similar to the corresponding one in session processes. The only difference here is that we are considering the polyadic π-calculus. Rule (Rπ-Case) is also called a *case normalisation*. The case process reduces to P_j substituting x_j with the value v, if the label l_j is chosen. Rules (Rπ-Res) and

(Rπ-Par) state that communication and case normalisation can happen under restriction and parallel composition, respectively. Rule (Rπ-Struct) states that reduction can happen under structural congruence $\equiv$, which is defined in the same way as in the previous section for session π-calculus semantics; the only difference being that there are no rules for co-names.

Properties. We recall some basic properties of the type system with linear π-types [27]. They follow the same intuition as the analogous properties for session types given in the previous section.

Definition 5 (Closed Typing Context)**.** A typing context is *closed* if for all $x \in \mathtt{dom}(\Gamma)$, then $\Gamma(x) \neq \ell_{\sharp}[\widetilde{\tau}]$.

Lemma 6 (Substitution Lemma for Linear π-calculus)**.** Let $\Gamma, x : \tau \vdash P$, and let $\Gamma \uplus \Gamma'$ be defined and $\Gamma' \vdash v : \tau$. Then, $\Gamma \uplus \Gamma' \vdash P[v/x]$.

Lemma 7 (Weakening in Linear π-calculus)**.** If $\Gamma \vdash P$ and $x \notin \mathtt{fn}(P)$ and $\mathtt{un}(\tau)$, then $\Gamma, x : \tau \vdash P$.

Lemma 8 (Strengthening in Linear π-calculus)**.** If $\Gamma, x : \tau \vdash P$ and $x \notin \mathtt{fn}(P)$ and $\mathtt{un}(\tau)$, then $\Gamma \vdash P$.

We are ready now to state the subject congruence and the subject reduction properties for the linear π-calculus.

Lemma 9 (Subject Congruence for Linear π-calculus)**.** If $\Gamma \vdash P$ and $P \equiv P'$, then $\Gamma \vdash P'$.

Theorem 10 (Subject Reduction for Linear π-calculus)**.** If $\Gamma \vdash P$ with Γ closed and $P \to P'$, then $\Gamma \vdash P'$.

By analysing and combining the definition of closed typing context with the statement of the subject reduction property for linear π-calculus, we notice that since the typing context has no linear channel owning both capabilities (condition $\neq \ell_{\sharp}[\widetilde{\tau}]$), if a process reduces it is either the result of a case normalisation or of a communication on a restricted channel owning both capabilities of input and output. The reason for adopting a closed typing context is to avoid reductions of typing contexts due to reductions of processes. This gives a simpler statement of the subject reduction property. Further details can be found in Sangiorgi and Walker [27].

We conclude the section with a lemma showing that if two structurally congruent processes reduce by consuming exactly the same prefixes, then the derivatives are again structurally congruent. To express this, we use a *marking* of the involved prefixes, as a way of pointing out the specific prefixes involved (we mark only a prefix, not the process underneath it). The marking does not otherwise affect syntax and operational semantics.

Lemma 11. Suppose P has exactly one input and one output prefix that are marked, and that $P \to P'$ in which precisely the two marked input and output

prefixes are consumed. Suppose also that $P \equiv Q$ and that $Q \to Q'$ in which, as before, precisely the two marked input and output prefixes are consumed. Then, also $P' \equiv Q'$.

Proof. Straightforward proof on the number of axioms of structural congruence applied to infer $P \equiv Q$. □

3. Encoding

Session types guarantee that only the communicating parties know the corresponding endpoints of the session channel, thus providing privacy. Moreover, the opposite endpoints should have dual types, thus providing communication safety. The interpretation of session types should take into account these fundamental issues. In order to guarantee privacy and safety of communication we adopt linear channels, which are used exactly once. Privacy is ensured since the linear channel is known only to the interacting parties. Communication safety is ensured by the type safety of linear types. Furthermore, in order to preserve the structure of a session type and the session fidelity property, our encoding is based on the *continuation-passing* principle.

3.1. Type Encoding

We present the encoding of session types into linear π-types at the top of Fig. 9. All the other types are encoded in a homomorphic way, namely $[\![\sharp T]\!] \triangleq \sharp[\![T]\!]$ and $[\![\mathtt{Unit}]\!] \triangleq \mathtt{Unit}$. The encoding of $\mathtt{end}$ is a channel with no capabilities, meaning that it cannot be used neither for input nor for output. Type $?T.S$ is interpreted as the linear input channel type carrying a pair of values whose types are the encoding of T and of S. The encoding of $!T.S$ is similar. However, in this case it is the dual of S to be sent since it is the type of a channel as used by the receiver. This will be shown later by an example of the encoding. The branch and the select types are generalisations of input and output types, respectively. Consequently, they are interpreted as linear input and linear output channels carrying variant types having the same labels l_i and, as types respectively, the encoding of S_i and of $\overline{S_i}$ for all $i \in I$. Again, the reason for duality is the same as for the output type.

3.2. Process Encoding

The encoding of session processes into π-calculus processes is defined at the bottom of Fig. 9. The encoding of terms differs from the encoding of types as it is parametrised by a function, ranging over f, g, from names to names. We use $\mathtt{dom}(f)$ to denote the domain of function f. We use f_x, f_y as an abbreviation for $f(x), f(y)$, respectively.

Let P be a session process. We say that a function f is a *renaming function for* P, if for all names $x \in \mathtt{fn}(P)$, the image f_x is either x, or it is a fresh name not included in $\mathtt{n}(P)$; and f is the identity function on all bound names of P. During the encoding of a session process, its renaming function f is updated

$$
\begin{array}{rcll}
[\![\mathtt{end}]\!] & \triangleq & \varnothing[\,] & \text{(E-End)} \\
[\![!T.S]\!] & \triangleq & \ell_{\mathtt{o}}[[\![T]\!], [\![\overline{S}]\!]] & \text{(E-Out)} \\
[\![?T.S]\!] & \triangleq & \ell_{\mathtt{i}}[[\![T]\!], [\![S]\!]] & \text{(E-Inp)} \\
[\![\oplus\{l_i : S_i\}_{i\in I}]\!] & \triangleq & \ell_{\mathtt{o}}[\langle l_i \_ [\![\overline{S_i}]\!]\rangle_{i\in I}] & \text{(E-Select)} \\
[\![\&\{l_i : S_i\}_{i\in I}]\!] & \triangleq & \ell_{\mathtt{i}}[\langle l_i \_ [\![S_i]\!]\rangle_{i\in I}] & \text{(E-Branch)}
\end{array}
$$

$$
\begin{array}{rcll}
[\![x]\!]_f & \triangleq & f_x & \text{(E-Name)} \\
[\![\star]\!]_f & \triangleq & \star & \text{(E-Star)} \\
[\![\mathbf{0}]\!]_f & \triangleq & \mathbf{0} & \text{(E-Inaction)} \\
[\![x!\langle v\rangle.P]\!]_f & \triangleq & (\boldsymbol{\nu} c) f_x!\langle [\![v]\!]_f, c\rangle.[\![P]\!]_{f,\{x\mapsto c\}} & \text{(E-Output)} \\
[\![x?(y).P]\!]_f & \triangleq & f_x?(y,c).[\![P]\!]_{f,\{x\mapsto c\}} & \text{(E-Input)} \\
[\![x \lhd l_j.P]\!]_f & \triangleq & (\boldsymbol{\nu} c) f_x!\langle l_j \_ c\rangle.[\![P]\!]_{f,\{x\mapsto c\}} & \text{(E-Selection)} \\
[\![x \rhd \{l_i : P_i\}_{i\in I}]\!]_f & \triangleq & f_x?(y).\ \mathbf{case}\, y\, \mathbf{of}\, \{l_i \_ (c) \rhd [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I} & \text{(E-Branching)} \\
[\![P \mid Q]\!]_f & \triangleq & [\![P]\!]_f \mid [\![Q]\!]_f & \text{(E-Composition)} \\
[\![(\boldsymbol{\nu} xy)P]\!]_f & \triangleq & (\boldsymbol{\nu} c)[\![P]\!]_{f,\{x,y\mapsto c\}} & \text{(E-Restriction)}
\end{array}
$$

Figure 9: Encoding of types and terms

as in $f, \{x \mapsto c\}$ or $f, \{x, y \mapsto c\}$, where names x and y are now associated to c, namely $f(x)$ and $f(y)$ are updated to c. The notion of a renaming function is extended also to values, being ground values and names, as expected. In the uses of the definition of renaming function f for P (respectively v), process P (respectively value v) will be typed in a typing context, say Γ. We will implicitly assume that the fresh names used by f (that is, the names y such that $y = f(x)$, for some $x \neq y$) are also fresh for Γ (that is, they are not in $\mathtt{dom}(\Gamma)$. We prefer to avoid the explicit mention of the typing context Γ so to ease the reading of the statements.

We explain now the reason for f. Since we are using linear channels, once a channel is used, it cannot be used again for transmission. To enable structured communications however, like session types do, the channel is renamed: a new channel is created and is sent to the partner in order to use it to continue the rest of the session. This procedure is repeated at every step of communication and the function f is updated to the new name created. This is the continuation-passing principle.

We provide some explanations on the encoding. A channel name x is encoded by using a renaming function f for x, meaning that f is defined on x. The encoding of the unit value is the unit value itself. This holds for every ground value added to the language. In the encoding of the output process, a new channel name c is created and is sent together with the encoding of the payload v along the channel f_x; the encoding of the continuation process P is parametrised in f where name x is updated to c. Similarly, the input process listens on channel

f_x and receives a value, that substitutes name y and a fresh channel c that substitutes x in the continuation process encoded in f updated with $\{x \mapsto c\}$. As indicated in Section 2.1, session restriction $(\boldsymbol{\nu} xy)P$ creates two fresh names and binds them in P as being the opposite endpoints of the same session channel. This is not needed in the standard π-calculus. The restriction construct $(\boldsymbol{\nu} x)P$ creates and binds a unique name x to P; this name identifies both endpoints of the communicating channel. The encoding of a session restriction process $(\boldsymbol{\nu} xy)P$ is a linear channel restriction process $(\boldsymbol{\nu} c)[\![P]\!]_{f,\{x,y \mapsto c\}}$ with the new name c used to substitute x and y in the encoding of P. The last two constructs correspond to selection and branching processes. Selection $x \lhd l_j.P$ is encoded as the process that first creates a new channel c and then sends on f_x a variant value l_j_c, where l_j is the selected label and c is the channel created to be used for the continuation of the session. The encoding of branching receives on f_x a value, typically being a variant value, which is the guard of the **case** process. According to the chosen label, one of the corresponding processes $[\![P_i]\!]_{f,\{x \mapsto c\}}$ for $i \in I$, will be chosen. Note that the name c is bound in any process $[\![P_i]\!]_{f,\{x \to c\}}$. The encoding of the other process constructs, like inaction, standard scope restriction, and parallel composition, is a homomorphism, namely $[\![\mathbf{0}]\!]_f \triangleq \mathbf{0}$, $[\![(\boldsymbol{\nu} x)P]\!]_f \triangleq (\boldsymbol{\nu} x)[\![P]\!]_f$, and $[\![P \mid Q]\!]_f \triangleq [\![P]\!]_f \mid [\![Q]\!]_f$.

3.3. Example: the Mathematical Server and Client

In this section we present an example of a mathematical server and a client communicating with it, from Gay and Hole [14]; the example illustrates channel interaction as well as branching and selection. We assume ground types like `Int`, `Bool` and standard mathematical operations on ground values. We present the encoding of types and processes, and the operational semantics of both the session system and its encoding. The server offers three mathematical operations as services: addition of integers; the equality test; and negation of integers. The server runs in parallel with a client, which selects among the services offered. Communication occurs along a session channel with endpoints x for the server and y for the client.

The session type S for the server endpoint x is defined as:

$$\begin{array}{rcll} S & \triangleq & \&\{ & \mathit{plus} : ?\texttt{Int}.?\texttt{Int}.!\texttt{Int}.\texttt{end}, \\ & & & \mathit{equal} : ?\texttt{Int}.?\texttt{Int}.!\texttt{Bool}.\texttt{end}, \\ & & & \mathit{neg} : ?\texttt{Int}.!\texttt{Int}.\texttt{end} \quad \} \end{array}$$

The session type for the client endpoint y must be dual to S and is thus defined as:

$$\begin{array}{rcll} \overline{S} & \triangleq & \oplus\{ & \mathit{plus} : !\texttt{Int}.!\texttt{Int}.?\texttt{Int}.\texttt{end}, \\ & & & \mathit{equal} : !\texttt{Int}.!\texttt{Int}.?\texttt{Bool}.\texttt{end}, \\ & & & \mathit{neg} : !\texttt{Int}.?\texttt{Int}.\texttt{end} \quad \} \end{array}$$

Now we move to processes. The server process is defined as:

$$\begin{array}{rcll} \mathit{server} & \triangleq & x \rhd \{ & \mathit{plus} : x?(v_1).x?(v_2).x!\langle v_1 + v_2\rangle.\mathbf{0}, \\ & & & \mathit{equal} : x?(v_1).x?(v_2).x!\langle v_1 == v_2\rangle.\mathbf{0}, \\ & & & \mathit{neg} : x?(v).x!\langle -v\rangle.\mathbf{0} \quad \} \end{array}$$

We have $x : S \vdash \mathit{server}$. The client must be typechecked by using $\overline{S}$. By rule (T-Sel) this means that the client chooses one of the possible branches specified in its type. Thus, a possible client is:

$$\mathit{client} \quad \triangleq \quad y \lhd \mathit{equal}.y!\langle 3\rangle.y!\langle 5\rangle.y?(\mathit{eq}).\mathbf{0}$$

Such a client selects the equality test, which we already mentioned in the introduction. The client sends to the server two integers 3 and 5, and waits for a boolean answer. Once all this is done, both processes terminate. The whole system is given by

$$(\boldsymbol{\nu} xy)\big(\mathit{server} \mid \mathit{client}\big)$$

which, as outlined above, reduces thus:

$$\begin{array}{rl} (\boldsymbol{\nu} xy)\big(\mathit{server} \mid \mathit{client}\big) & \to (\boldsymbol{\nu} xy)\big(x?(v_1).x?(v_2).x!\langle v_1 == v_2\rangle.\mathbf{0} \mid y!\langle 3\rangle.y!\langle 5\rangle.y?(\mathit{eq}).\mathbf{0}\big) \\ & \to (\boldsymbol{\nu} xy)\big(x?(v_2).x!\langle 3 == v_2\rangle.\mathbf{0} \mid y!\langle 5\rangle.y?(\mathit{eq}).\mathbf{0}\big) \\ & \to (\boldsymbol{\nu} xy)\big(x!\langle 3 == 5\rangle.\mathbf{0} \mid y?(\mathit{eq}).\mathbf{0}\big) \;\to\; \mathbf{0} \end{array}$$

We are ready now to present the encoding of the system. We start with session types. We have:

$$\begin{array}{rcll} [\![S]\!] & = & \ell_{\mathtt{i}}[\langle & \mathit{plus}\_\ell_{\mathtt{i}}[\mathtt{Int}, \ell_{\mathtt{i}}[\mathtt{Int}, \ell_{\mathtt{o}}[\mathtt{Int}, \varnothing[]]]] \\ & & & \mathit{equal}\_\ell_{\mathtt{i}}[\mathtt{Int}, \ell_{\mathtt{i}}[\mathtt{Int}, \ell_{\mathtt{o}}[\mathtt{Bool}, \varnothing[]]]], \\ & & & \mathit{neg}\_\ell_{\mathtt{i}}[\mathtt{Int}, \ell_{\mathtt{o}}[\mathtt{Int}, \varnothing[]]] \quad \rangle] \end{array} \tag{1}$$

and

$$\begin{array}{rcll} [\![\overline{S}]\!] & = & \ell_{\mathtt{o}}[\langle & \mathit{plus}\_\ell_{\mathtt{i}}[\mathtt{Int}, \ell_{\mathtt{i}}[\mathtt{Int}, \ell_{\mathtt{o}}[\mathtt{Int}, \varnothing[]]]] \\ & & & \mathit{equal}\_\ell_{\mathtt{i}}[\mathtt{Int}, \ell_{\mathtt{i}}[\mathtt{Int}, \ell_{\mathtt{o}}[\mathtt{Bool}, \varnothing[]]]], \\ & & & \mathit{neg}\_\ell_{\mathtt{i}}[\mathtt{Int}, \ell_{\mathtt{o}}[\mathtt{Int}, \varnothing[]]] \quad \rangle] \end{array} \tag{2}$$

When examining (1) and (2) we notice that duality on session types boils down to opposite capabilities of linear channel types. Indeed the encodings $[\![S]\!]$ and $[\![\overline{S}]\!]$ only differ in the capabilities of the outermost linear types $\ell_{\mathtt{i}}[\cdot]$ and $\ell_{\mathtt{o}}[\cdot]$. Thus checking the duality between two session types amounts to checking, in the encoding, this simple duality on capabilities.

Now we move to processes. When encoding processes, the initial renaming function is the identity function, below simply indicated as $\varnothing$:

$$\begin{array}{rcl} [\![(\boldsymbol{\nu} xy)(\mathit{server} \mid \mathit{client})]\!]_{\varnothing} & = & (\boldsymbol{\nu} z)[\![(\mathit{server} \mid \mathit{client})]\!]_{\{x,y\mapsto z\}} \\ & = & (\boldsymbol{\nu} z)\big([\![\mathit{server}]\!]_{\{x\mapsto z\}} \mid [\![\mathit{client}]\!]_{\{y\mapsto z\}}\big) \end{array}$$

where

$$\begin{array}{rcl}[\![server]\!]_{\{x\mapsto z\}} & = & z?(y).\mathbf{case}\, y\, \mathbf{of}\, \{ \\ & & \quad plus\_(s) \triangleright s?(v_1,c).c?(v_2,c').(\boldsymbol{\nu} c'')c'!\langle v_1+v_2,c''\rangle.\mathbf{0} \\ & & \quad equal\_(s) \triangleright s?(v_1,c).c?(v_2,c').(\boldsymbol{\nu} c'')c'!\langle v_1 == v_2,c''\rangle.\mathbf{0} \\ & & \quad neg\_(s) \triangleright s?(v,c).(\boldsymbol{\nu} c'')c!\langle -v,c''\rangle.\mathbf{0} \quad \}\end{array}$$

and

$$[\![client]\!]_{\{y\mapsto z\}} \quad = \quad (\boldsymbol{\nu} s)z!\langle equal\_s\rangle.(\boldsymbol{\nu} c)s!\langle 3,c\rangle.(\boldsymbol{\nu} c')c!\langle 5,c'\rangle.c'?(eq,c'').\mathbf{0}$$

The renaming function $\{x,y \mapsto z\}$ maps the session endpoints x and y to a fresh name z; after that, in every output of the session, a new channel is created and sent to the partner together with the payload. For example, the *client* creates and sends a name s together with the selected label *equal*, and afterwards it creates the channels c, c' and c'', for the rest of the communication. Note that, when a new channel is created, for example in $(\boldsymbol{\nu} c)$, it has both the input and the output capabilities. The client process sends to the server, at channel s, the payload 3 and the input capability of the new channel c, retaining for itself the output capability. Interaction then continues along such new channel c, with the sending of the payload 5 and the output capability of a new continuation channel c'. Here is how the encoded system evolves:

$$\begin{array}{l}(\boldsymbol{\nu} z)\big([\![server]\!]_{f,\{x\mapsto z\}} \mid [\![client]\!]_{f,\{y\mapsto z\}}\big) \\ \quad \to (\boldsymbol{\nu} s)\big(\mathbf{case}\, equal\_s\, \mathbf{of}\, \{\ldots\} \mid (\boldsymbol{\nu} c)s!\langle 3,c\rangle.(\boldsymbol{\nu} c')c!\langle 5,c'\rangle.c'?(eq,c'').\mathbf{0}\big) \\ \quad \to (\boldsymbol{\nu} s)\big(s?(v_1,c).c?(v_2,c').(\boldsymbol{\nu} c'')c'!\langle v_1 == v_2,c''\rangle.\mathbf{0} \mid \\ \qquad\qquad (\boldsymbol{\nu} c)s!\langle 3,c\rangle.(\boldsymbol{\nu} c')c!\langle 5,c'\rangle.c'?(eq,c'').\mathbf{0}\big) \\ \quad \to^* (\boldsymbol{\nu} c'')c'!\langle 3 == 5,c''\rangle.\mathbf{0} \mid c'?(eq,c'').\mathbf{0} \;\to\; \mathbf{0}\end{array}$$

The first reduction of the encoded maths system corresponds to the first reduction of the original system, where a label (namely *equal*) is selected. The second reduction is a *case normalisation*, where a pattern matching of the **case** guard occurs so to identify the appropriate continuation process. The case normalisation is the only reduction in the encoded system that does not have a corresponding reduction in the original system; it represents however an administrative reduction, without a real computational content. The remaining reductions of the encoded systems, which for simplicity have not been detailed, are in one-to-one correspondence with the reductions of the original system.

3.4. Properties of the Encoding

The encoding previously presented can be considered as the semantics of session types and session terms. The following results show that indeed we can derive the typing judgements and the properties of the π-calculus with sessions

$$\begin{array}{rcll} [\![\varnothing]\!]_f & \triangleq & \varnothing & \text{(E-Empty)} \\ [\![\Gamma, x:T]\!]_f & \triangleq & [\![\Gamma]\!]_f \uplus f_x : [\![T]\!] & \text{(E-Gamma)} \end{array}$$

Figure 10: Encoding of typing contexts

via the encoding and the corresponding typing judgements and properties in the standard π-calculus.

In order to prove these results, we need to extend the encoding to session typing contexts. Given a session process P, respectively a value v, such that there is a session typing context Γ with $\Gamma \vdash P$, respectively $\Gamma \vdash v : T$, and a renaming function f for P, respectively v, we use f in the encoding of Γ as defined in Fig. 10.

3.4.1. Auxiliary Results

We start this section with some auxiliary results. The following proposition states that the encoding of typing contexts, given in Fig. 10, is sound and complete with respect to predicates `lin` and `un`.

Proposition 12. Let Γ be a session typing context and q be either `lin` or `un`. Then $q(\Gamma)$ if and only if $q([\![\Gamma]\!]_f)$, for all renaming functions f for Γ.

The following two lemmas give the relation between the combination operator '$\uplus$' and the standard ',' operator in linear π-typing contexts.

Lemma 13. If $\Gamma, x:T$ is defined, then also $\Gamma \uplus x:T$ is defined.

Proof. By definition of ',' on typing contexts, it means that $x \notin \mathtt{dom}(\Gamma)$. We conclude by definition of combination of typing contexts. □

Lemma 14. If $\Gamma \uplus x:T$ is defined and $x \notin \mathtt{dom}(\Gamma)$, then also $\Gamma, x:T$ is defined.

Proof. Immediate by definition of combination of typing contexts. □

The following two lemmas give a relation between the context split operator '$\circ$' used in session typing contexts and the combination operator '$\uplus$' used in linear π-typing contexts by using the encoding of typing contexts presented in Fig. 10.

Lemma 15 (Split to Combination)**.** Let $\Gamma_1, \ldots, \Gamma_n$ be session typing contexts such that $\Gamma_1 \circ \ldots \circ \Gamma_n$ is defined, then

$$[\![\Gamma_1 \circ \ldots \circ \Gamma_n]\!]_f = [\![\Gamma_1]\!]_f \uplus \ldots \uplus [\![\Gamma_n]\!]_f$$

for some renaming function f for $\Gamma_1 \circ \ldots \circ \Gamma_n$.

Proof. It follows immediately by the definitions of the encoding on typing contexts, given in Fig. 10, and the combination on typing contexts, given in Fig. 2.2. □

Lemma 16 (Combination to Split)**.** Let Γ be a session typing context and f a renaming function for Γ and $[\![\Gamma]\!]_f = \Gamma_1^\pi \uplus \ldots \uplus \Gamma_n^\pi$. Then, for all $i \in \{1 \ldots n\}$, there exist Γ_i such that $[\![\Gamma_i]\!]_f = \Gamma_i^\pi$ and $\Gamma_1 \circ \ldots \circ \Gamma_n = \Gamma$.

Proof. It follows immediately by the encoding of typing contexts given in Fig. 10 and Fig. 2 on context split '$\circ$' for session types. □

The following lemma relates the encoding of dual session types with dual linear π-types.

Lemma 17 (Encoding of Dual Session Types)**.** If $[\![T]\!] = \tau$ then $[\![\overline{T}]\!] = \overline{\tau}$.

Proof. The proof is by induction on the structure of the session type T. We use the duality of session types defined in Section 2.1 and the duality of standard π-types defined in Section 2.2.

- $T = \mathtt{end}$
  By (E-End) we have $[\![\mathtt{end}]\!] = \varnothing[\,]$ and $\overline{T} = \mathtt{end}$. We conclude by the duality of $\varnothing[\,]$.

- $T = !T.U$
  By (E-Out) we have $[\![!T.U]\!] = \ell_{\mathtt{o}}[[\![T]\!], [\![\overline{U}]\!]]$. By the duality of session types we have $!\overline{T}.\overline{U} = ?T.\overline{U}$. By (T-In) we have $[\![?T.\overline{U}]\!] = \ell_{\mathtt{i}}[[\![T]\!], [\![\overline{U}]\!]]$. We conclude by the duality of π-types.

- $T = ?T.U$
  By (E-In) we have $[\![?T.U]\!] = \ell_{\mathtt{i}}[[\![T]\!], [\![U]\!]]$. By the duality of session types we have $\overline{?T.U} = !T.\overline{U}$. By (E-Out) we have $[\![!T.\overline{U}]\!] = \ell_{\mathtt{o}}[[\![T]\!], [\![\overline{\overline{U}}]\!]]$, which by the involution property of duality on session types is $\ell_{\mathtt{o}}[[\![T]\!], [\![U]\!]]$. We conclude by the duality of π-types.

- $T = \oplus\{l_i : T_i\}_{i \in I}$
  By (E-Select) we have $[\![\oplus\{l_i : T_i\}_{i \in I}]\!] = \ell_{\mathtt{o}}[\langle l_i \_ [\![\overline{T_i}]\!]\rangle_{i \in I}]$ By duality on session types we have $\overline{\oplus\{l_i : T_i\}_{i \in I}} = \&\{l_i : \overline{T}_i\}_{i \in I}$. By (E-Branch) we have $[\![\&\{l_i : \overline{T}_i\}_{i \in I}]\!] = \ell_{\mathtt{i}}[\langle l_i \_ [\![\overline{T_i}]\!]\rangle_{i \in I}]$ We conclude by the duality of π-types.

- $T = \&\{l_i : T_i\}_{i \in I}$
  By (E-Branch) we have $[\![\&\{l_i : T_i\}_{i \in I}]\!] = \ell_{\mathtt{i}}[\langle l_i \_ [\![T_i]\!]\rangle_{i \in I}]$ By duality on session types we have $\overline{\&\{l_i : T_i\}_{i \in I}} = \oplus\{l_i : \overline{T}_i\}_{i \in I}$. By (E-Select) we have $[\![\oplus\{l_i : \overline{T}_i\}_{i \in I}]\!] = \ell_{\mathtt{o}}[\langle l_i \_ [\![\overline{\overline{T_i}}]\!]\rangle_{i \in I}]$, which by the involution property of duality on session types means $\ell_{\mathtt{o}}[\langle l_i \_ [\![T_i]\!]\rangle_{i \in I}]$. We conclude by the duality of π-types.

□

3.4.2. Type Correctness for Values

We state the soundness and completeness of the encoding in typing derivations for values. The correctness of an encoded typing judgement on the target terms implies the correctness of the judgement on the source terms, and conversely.

Lemma 18 (Soundness)**.** If $[\![\Gamma]\!]_f \vdash [\![v]\!]_f : [\![T]\!]$ for some renaming function f for v, then $\Gamma \vdash v : T$.

Proof. The proof is done by induction on the structure of the value v:

- Case $v = x$:
  By (E-Name) we have $[\![x]\!]_f = f_x$ and assume $[\![\Gamma]\!]_f \vdash f_x : [\![T]\!]$. By rule (Tπ-Var) it means that $(f_x : [\![T]\!]) \in [\![\Gamma]\!]_f$. Hence, $(x : T') \in \Gamma$ for some type T'. By (E-Gamma) it must be $[\![T']\!] = [\![T]\!]$ which implies $T' = T$. By Proposition 12 also $\mathtt{un}(\Gamma_1)$ holds. By applying rule (T-Var) we obtain the result.

- Case $v = \star$:
  By (E-Star) we have $[\![\star]\!]_f = \star$ and assume $[\![\Gamma]\!]_f \vdash \star : \mathtt{Unit}$ and $\mathtt{un}([\![\Gamma]\!]_f)$. Then $\mathtt{un}(\Gamma)$ holds by Proposition 12. We conclude by rule (T-Val). □

Lemma 19 (Completeness)**.** If $\Gamma \vdash v : T$, then $[\![\Gamma]\!]_f \vdash [\![v]\!]_f : [\![T]\!]$ for some renaming function f for v.

Proof. The proof is done by induction on the derivation $\Gamma \vdash v : T$.

- Case (T-Var):
  $$\frac{\mathtt{un}(\Gamma)}{\Gamma, x : T \vdash x : T}$$
  By (E-Gamma) and (E-Name) we have $[\![\Gamma]\!]_f \uplus f_x : [\![T]\!] \vdash f_x : [\![T]\!]$. We conclude by Proposition 12, Lemma 14 and typing rule (Tπ-Var).

- Case (T-Val):
  $$\frac{\mathtt{un}(\Gamma)}{\Gamma \vdash \star : \mathtt{Unit}}$$
  Follows immediately from the encoding of $\star$ and $\mathtt{Unit}$, Proposition 12 and rule (Tπ-Val). □

3.4.3. Type Correctness for Processes

We state the soundness and completeness of the encoding in typing derivations for processes.

Theorem 20 (Soundness)**.** If $[\![\Gamma]\!]_f \vdash [\![P]\!]_f$ for some renaming function f for P, then $\Gamma \vdash P$.

Proof. The proof is by induction on the structure of session process P. We only present some of the cases.

- Case $\mathbf{0}$:
  By (E-Inaction) we have $[\![\mathbf{0}]\!]_f = \mathbf{0}$ and assume $[\![\Gamma]\!]_f \vdash \mathbf{0}$, where $\mathtt{un}([\![\Gamma]\!]_f)$ holds. By Proposition 12 also $\mathtt{un}(\Gamma)$ holds. We conclude by applying typing rule (T-Inact).

- Case $P \mid Q$: By (E-Composition) we have $[\![P \mid Q]\!]_f = [\![P]\!]_f \mid [\![Q]\!]_f$ and assume $[\![\Gamma]\!]_f \vdash [\![P]\!]_f \mid [\![Q]\!]_f$, which by rule (T$\pi$-Par) means:

$$\frac{\Gamma_1^\pi \vdash [\![P]\!]_f \quad \Gamma_2^\pi \vdash [\![Q]\!]_f}{\Gamma_1^\pi \uplus \Gamma_2^\pi \vdash [\![P]\!]_f \mid [\![Q]\!]_f}$$

  where $[\![\Gamma]\!]_f = \Gamma_1^\pi \uplus \Gamma_2^\pi$. By Lemma 16 $\Gamma_1^\pi = [\![\Gamma_1]\!]_f$ and $\Gamma_2^\pi = [\![\Gamma_2]\!]_f$, such that $\Gamma = \Gamma_1 \circ \Gamma_2$. By induction hypothesis we have $\Gamma_1 \vdash P$ and $\Gamma_2 \vdash Q$. By applying (T-Par) we obtain the result $\Gamma_1 \circ \Gamma_2 \vdash P \mid Q$.

- Case $x?(y).P$:
  By (E-Input) we have $[\![x?(y).P]\!]_f = f_x?(y,c).[\![P]\!]_{f,\{x\mapsto c\}}$ and assume $[\![\Gamma]\!]_f \vdash f_x?(y,c).[\![P]\!]_{f,\{x\mapsto c\}}$ which by rule (Tπ-Inp) means:

$$\frac{\Gamma_1^\pi \vdash f_x : \ell_{\mathtt{i}}[T^\pi, S^\pi] \qquad \Gamma_2^\pi, c : S^\pi, y : T^\pi \vdash [\![P]\!]_{f,\{x\mapsto c\}}}{[\![\Gamma]\!]_f \vdash f_x?(y,c).[\![P]\!]_{f,\{x\mapsto c\}}}$$

  where $[\![\Gamma]\!]_f = \Gamma_1^\pi \uplus \Gamma_2^\pi$. By Lemma 16 $\Gamma_1^\pi = [\![\Gamma_1]\!]_f$ and $\Gamma_2^\pi = [\![\Gamma_2]\!]_f$, with $\Gamma = \Gamma_1 \circ \Gamma_2$. By Lemma 18 we have $\Gamma_1 \vdash x : ?T.S$. By induction hypothesis $\Gamma_2, x : S, y : T \vdash P$ where $T^\pi = [\![T]\!]$, $S^\pi = [\![S]\!]$ and the renaming function $f, \{x \mapsto c\}$ is used in the encoding of P. By applying (T-Inp) we obtain $\Gamma_1 \circ \Gamma_2 \vdash x?(y).P$.

□

Theorem 21 (Completeness)**.** If $\Gamma \vdash P$, then $[\![\Gamma]\!]_f \vdash [\![P]\!]_f$ for some renaming function f for P.

Proof. The proof is by induction on the derivation $\Gamma \vdash P$. We present the main cases.

- Case (T-Res):

$$\frac{\Gamma, x : T, y : \overline{T} \vdash P}{\Gamma \vdash (\boldsymbol{\nu} xy)P}\ (\text{T-Res})$$

  Notice that $x, y \notin \mathtt{dom}(\Gamma)$ by typability assumptions. We distinguish the following two cases:

  – Suppose $T \neq \mathtt{end}$. By duality on session types also $\overline{T} \neq \mathtt{end}$. By induction hypothesis we have $[\![\Gamma, x : T, y : \overline{T}]\!]_{f'} \vdash [\![P]\!]_{f'}$, for some renaming function f' for P, which by (E-Gamma) means that $[\![\Gamma]\!]_{f'} \uplus f'_x : [\![T]\!] \uplus f'_y : [\![\overline{T}]\!] \vdash [\![P]\!]_{f'}$. Let $f = f'$ and update f with $\{x, y \mapsto c\}$ for a fresh name c that does not occur in the codomain of f. We will use $f, \{x, y \mapsto c\}$ as a renaming function. By Lemma 17, $[\![T]\!] = \tau$ and $[\![\overline{T}]\!] = \overline{\tau}$. Since $T \neq \mathtt{end}$ and $\overline{T} \neq \mathtt{end}$, we have $[\![T]\!] = \ell_\alpha[W]$ and $[\![\overline{T}]\!] = \ell_{\overline{\alpha}}[W]$ and by the combination of linear channel types $\ell_\alpha[W] \uplus \ell_{\overline{\alpha}}[W] = \ell_\sharp[W]$, where W denotes the pair of carried types, which are totally irrelevant for this proof. Hence, we can rewrite the

induction hypothesis as $[\![\Gamma]\!]_f \uplus c : \ell_\sharp[W] \vdash [\![P]\!]_{f,\{x,y\mapsto c\}}$. By Lemma 14 we obtain $[\![\Gamma]\!]_f, c : \ell_\sharp[W] \vdash [\![P]\!]_{f,\{x,y\mapsto c\}}$. By applying (T$\pi$-Res1) we obtain $[\![\Gamma]\!]_f \vdash (\boldsymbol{\nu} c)[\![P]\!]_{f,\{x,y\mapsto c\}}$, which concludes this case.

– Suppose $T = \mathtt{end}$. By duality on session types also $\overline{T} = \mathtt{end}$. By induction hypothesis we have $[\![\Gamma, x : \mathtt{end}, y : \mathtt{end}]\!]_{f'} \vdash [\![P]\!]_{f'}$, for some renaming function f' for P. By (E-Gamma) it means that $[\![\Gamma]\!]_{f'} \uplus f'_x : [\![\mathtt{end}]\!] \uplus f'_y : [\![\mathtt{end}]\!] \vdash [\![P]\!]_{f'}$. Let $f = f'$ and update f with $\{x, y \mapsto c\}$ for a fresh name c that does not occur in the codomain of f. We will now use $f, \{x, y \mapsto c\}$ as a renaming function for P. Hence, we can rewrite the induction hypothesis as $[\![\Gamma]\!]_f \uplus c : \varnothing[\,] \uplus c : \varnothing[\,] \vdash [\![P]\!]_{f,\{x,y\mapsto c\}}$, which by the combination of unrestricted types means $[\![\Gamma]\!]_f \uplus c : \varnothing[\,] \vdash [\![P]\!]_{f,\{x,y\mapsto c\}}$. Moreover, $c \notin \mathtt{dom}([\![\Gamma]\!]_f)$, since c is chosen fresh, then by Lemma 14 we obtain $[\![\Gamma]\!]_f, c : \varnothing[\,] \vdash [\![P]\!]_{f,\{x,y\mapsto c\}}$. We conclude by rule (T$\pi$-Res2).

- Case (T-Brch):

$$\frac{\Gamma_1 \vdash x : \&\{l_i : T_i\}_{i\in I} \quad \Gamma_2, x : T_i \vdash P_i \quad \forall i \in I}{\Gamma_1 \circ \Gamma_2 \vdash x \rhd \{l_i : P_i\}_{i\in I}}\ (\text{T-Brch})$$

By applying Lemma 19 we have that $[\![\Gamma_1]\!]_{f'} \vdash [\![x : \&\{l_i : T_i\}_{i\in I}]\!]_{f'}$, for some renaming function f' for x, which by (E-Branch) means that $[\![\Gamma_1]\!]_{f'} \vdash f'_x : \ell_{\mathtt{i}}[\langle l_i \_ [\![T_i]\!]\rangle_{i\in I}]$. By induction hypothesis and by (E-Gamma) we have that $[\![\Gamma_2]\!]_{f''} \uplus f''_x : [\![T_i]\!] \vdash [\![P_i]\!]_{f''}$ for some renaming function f'' for P_i, for all $i \in I$. Since $\Gamma_1 \circ \Gamma_2$ is defined, by definition of context split, for all $x \in \mathtt{dom}(\Gamma_1) \cap \mathtt{dom}(\Gamma_2)$ it holds that $\Gamma_1(x) = \Gamma_2(x) = T$ and $\mathtt{un}(T)$. Let $\mathtt{dom}(\Gamma_1) \cap \mathtt{dom}(\Gamma_2) = D$ and define $f'_D = f' \setminus \bigcup_{d\in D}\{d \mapsto f'(d)\}$ and $f''_D = f'' \setminus \bigcup_{d\in D}\{d \mapsto f''(d)\}$. Suppose $f''(x) = c$. Then, let $f = \bigcup_{d\in D}\{d \mapsto d'\} \cup f'_D \cup f''_D \setminus \{x \mapsto c\}$, where for all $d \in D$ we create a fresh name d' and map d to d'. Moreover, f is a function since its subcomponents act on disjoint domains. Then, by applying Lemma 6, the above can be rewritten as:

$$[\![\Gamma_1]\!]_f \vdash f_x : \ell_{\mathtt{i}}[\langle l_i \_ [\![T_i]\!]\rangle_{i\in I}] \qquad [\![\Gamma_2]\!]_f \uplus c : [\![T_i]\!] \vdash [\![P_i]\!]_{f,\{x\mapsto c\}} \text{ for all } i \in I$$

Since $x \notin \mathtt{dom}(\Gamma_2)$, then $[\![\Gamma_2, x : T_i]\!]_{f,\{x\mapsto c\}}$ can be distributed and thus optimised as $[\![\Gamma_2]\!]_f \uplus c : [\![T_i]\!]$. By rules (T$\pi$-Case), and (T$\pi$-Var) for deriving $y : \langle l_i \_ [\![T_i]\!]\rangle_{i\in I}$, and Lemma 14 we have the following derivation:

$$\dfrac{\dfrac{\text{(T}\pi\text{-Case)}\ \text{(T}\pi\text{-Var)}}{y : \langle l_i \_ [\![T_i]\!]\rangle_{i\in I} \vdash y : \langle l_i \_ [\![T_i]\!]\rangle_{i\in I}} \qquad [\![\Gamma_2]\!]_f, c : [\![T_i]\!] \vdash [\![P_i]\!]_{f,\{x\mapsto c\}}\ \forall i \in I}{[\![\Gamma_2]\!]_f, y : \langle l_i \_ [\![T_i]\!]\rangle_{i\in I} \vdash \mathbf{case}\, y\, \mathbf{of}\, \{l_i \_ (c) \rhd [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}}$$

Then, by applying (Tπ-Inp) we have:

$$(\text{T}\pi\text{-Inp})\ \frac{\begin{array}{c}[\![\Gamma_1]\!]_f \vdash f_x : \ell_{\mathrm{i}}[\langle l_i\_[\![T_i]\!]\rangle_{i\in I}] \\ [\![\Gamma_2]\!]_f, y : \langle l_i\_[\![T_i]\!]\rangle_{i\in I} \vdash \mathbf{case}\, y\, \mathbf{of}\, \{l_i\_(c) \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}\end{array}}{[\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f \vdash f_x?(y).\ \mathbf{case}\, y\, \mathbf{of}\, \{l_i\_(c) \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}}$$

By applying (E-Branching) and Lemma 15 we conclude this case. □

3.4.4. Operational Correspondence

In this section we prove the operational correspondence. This property states that the encoding of processes is sound and complete with respect to the operational semantics of the π-calculus with and without sessions.

We start with a lemma which relates the encoding of processes and name substitution.

Lemma 22. Let P be a session process and let $P[v/z]$ denote process P where name z is substituted by value v. Then,

$$[\![P[v/z]]\!]_f = [\![P]\!]_f[[\![v]\!]_f/f(z)]$$

for all renaming functions f for P and v in which, for all names x, we have $f(x) = z$ if and only if $x = z$.

Proof. It follows immediately from the encoding of processes given in Fig. 9. □

Lemma 23 (Structural Congruence and Encoding)**.** Let P and P' be session processes. Then, $P \equiv P'$ if and only if $[\![P]\!]_f \equiv [\![P']\!]_f$ for all renaming functions f for P and P'.

Proof. The proof is done by induction on the number of axioms of structural congruence applied. □

Let $\hookrightarrow$ denote $\equiv$ extended with a *case normalisation*, namely a reduction by using (Rπ-Case). We are ready now to formally state the operational correspondence.

Theorem 24 (Operational Correspondence)**.** Let P be a session process, Γ a session typing context, and f a renaming function for P such that $[\![\Gamma]\!]_f \vdash [\![P]\!]_f$. Then the following statements hold.

1. If $P \to P'$, then $[\![P]\!]_f \to\!\!\hookrightarrow [\![P']\!]_f$.
2. If $[\![P]\!]_f \to Q$, then there is a session process P' such that
   - either $P \to P'$;
   - or there are x, y such that $(\boldsymbol{\nu} xy)P \to P'$

   and $Q \hookrightarrow [\![P']\!]_f$.

Proof. Notice that, since $[\![\Gamma]\!]_f \vdash [\![P]\!]_f$, by Theorem 20 it means that $\Gamma \vdash P$. We prove separately the two assertions of the theorem.

1. The proof is done by induction on the length of the derivation $P \to P'$.

- Case (R-Com):

$$P \triangleq (\boldsymbol{\nu} xy)(x!\langle v\rangle.Q_1 \mid y?(z).Q_2) \to (\boldsymbol{\nu} xy)(Q_1 \mid Q_2[v/z]) \triangleq P'$$

By the encoding of output and input processes we have:

$$\begin{array}{rcl}
[\![P]\!]_f & = & [\![(\boldsymbol{\nu} xy)\big(x!\langle v\rangle.Q_1 \mid y?(z).Q_2\big)]\!]_f \\
& = & (\boldsymbol{\nu} c)\ \big([\![x!\langle v\rangle.Q_1 \mid y?(z).Q_2]\!]_{f,\{x,y\mapsto c\}}\big) \\
& = & (\boldsymbol{\nu} c)\ \big([\![x!\langle v\rangle.Q_1]\!]_{f,\{x,y\mapsto c\}} \mid [\![y?(z).Q_2]\!]_{f,\{x,y\mapsto c\}}\big) \\
& = & (\boldsymbol{\nu} c)\ \big((\boldsymbol{\nu} c')(c!\langle [\![v]\!]_f, c'\rangle.[\![Q_1]\!]_{f,\{x,y\mapsto c,x\mapsto c'\}}) \mid c?(z,c').[\![Q_2]\!]_{f,\{x,y\mapsto c,y\mapsto c'\}}\big) \\
& \to & (\boldsymbol{\nu} c)\ \big((\boldsymbol{\nu} c')\big([\![Q_1]\!]_{f,\{x,y\mapsto c,x\mapsto c'\}} \mid [\![Q_2]\!]_{f,\{x,y\mapsto c,y\mapsto c'\}}[[\![v]\!]_f/z]\big)\big) \\
& \equiv & (\boldsymbol{\nu} c')\big([\![Q_1]\!]_{f,\{x,y\mapsto c,x\mapsto c'\}} \mid [\![Q_2]\!]_{f,\{x,y\mapsto c,y\mapsto c'\}}[[\![v]\!]_f/z]\big)
\end{array}$$

Since P is a session-typed process, it means $x \notin \mathtt{fn}(Q_2)$ and $y \notin \mathtt{fn}(Q_1)$. Then, both $f, \{x, y \mapsto c, x \mapsto c'\}$ and $f, \{x, y \mapsto c, y \mapsto c'\}$ can be replaced by $f, \{x, y \mapsto c'\}$. We can rewrite the above as:

$$(\boldsymbol{\nu} c')\big([\![Q_1]\!]_{f,\{x,y\mapsto c'\}} \mid [\![Q_2]\!]_{f,\{x,y\mapsto c'\}}[[\![v]\!]_f/z]\big)$$

Since z is bound with scope Q_2 it means that $f_z = z$. The encoding of P' using f as a renaming function is as follows:

$$\begin{array}{rcl}
[\![P']\!]_f & = & [\![(\boldsymbol{\nu} xy)\big(Q_1 \mid Q_2[v/z]\big)]\!]_f \\
& = & (\boldsymbol{\nu} c')\big([\![Q_1]\!]_{f,\{x,y\mapsto c'\}} \mid [\![Q_2[v/z]]\!]_{f,\{x,y\mapsto c'\}}\big) \\
& = & (\boldsymbol{\nu} c')\big([\![Q_1]\!]_{f,\{x,y\mapsto c'\}} \mid [\![Q_2]\!]_{f,\{x,y\mapsto c'\}}[[\![v]\!]_{f,\{x,y\mapsto c'\}}/[\![z]\!]_{f,\{x,y\mapsto c'\}}]\big) \\
& = & (\boldsymbol{\nu} c')\big([\![Q_1]\!]_{f,\{x,y\mapsto c'\}} \mid [\![Q_2]\!]_{f,\{x,y\mapsto c'\}}[[\![v]\!]_f/f_z]\big) \\
& = & (\boldsymbol{\nu} c')\big([\![Q_1]\!]_{f,\{x,y\mapsto c'\}} \mid [\![Q_2]\!]_{f,\{x,y\mapsto c'\}}[[\![v]\!]_f/z]\big)
\end{array}$$

In order to obtain $[\![Q_2]\!]_{f,\{x,y\mapsto c'\}}[[\![v]\!]_{f,\{x,y\mapsto c'\}}/[\![z]\!]_{f,\{x,y\mapsto c'\}}]$ above, we use Lemma 22. Function f coincides with $f, \{x, y \mapsto c'\}$ when applied to value v and $f_z = z$, so we obtain $[\![Q_2]\!]_{f,\{x,y\mapsto c'\}}[[\![v]\!]_f/z]$; meaning:

$$[\![P]\!]_f \to\equiv [\![P']\!]_f$$

- Case (R-Sel):

$$P \triangleq (\boldsymbol{\nu} xy)(x \lhd l_j.Q \mid y \rhd \{l_i : P_i\}_{i\in I}) \to (\boldsymbol{\nu} xy)(Q \mid P_j) \triangleq P' \quad \text{if } j \in I$$

By the encoding of selection and branching processes we have:

$$\begin{array}{rcl}
[\![P]\!]_f & = & [\![(\boldsymbol{\nu} xy)\big(x \lhd l_j.Q \mid y \rhd \{l_i : P_i\}_{i\in I}\big)]\!]_f \\
& = & (\boldsymbol{\nu} c)\ \big([\![x \lhd l_j.Q \mid y \rhd \{l_i : P_i\}_{i\in I}]\!]_{f,\{x,y\mapsto c\}}\big) \\
& = & (\boldsymbol{\nu} c)\ \big([\![x \lhd l_j.Q]\!]_{f,\{x,y\mapsto c\}} \mid [\![y \rhd \{l_i : P_i\}_{i\in I}]\!]_{f,\{x,y\mapsto c\}}\big) \\
& = & (\boldsymbol{\nu} c)\ \big((\boldsymbol{\nu} c')\big(c!\langle l_j_c'\rangle.[\![Q]\!]_{f,\{x,y\mapsto c,x\mapsto c'\}}\big) \mid \\
& & \qquad c?(z).\mathbf{case}\, z\, \mathbf{of}\, \{l_i_(c') \rhd [\![P_i]\!]_{f,\{x,y\mapsto c,y\mapsto c'\}}\}_{i\in I}\big) \\
& \to & (\boldsymbol{\nu} c)\ \big((\boldsymbol{\nu} c')\big([\![Q]\!]_{f,\{x,y\mapsto c,x\mapsto c'\}} \mid \\
& & \qquad \mathbf{case}\, l_j_c'\, \mathbf{of}\, \{l_i_(c') \rhd [\![P_i]\!]_{f,\{x,y\mapsto c,y\mapsto c'\}}\}_{i\in I}\big)\big) \\
& \to & (\boldsymbol{\nu} c)\ \big((\boldsymbol{\nu} c')\big([\![Q]\!]_{f,\{x,y\mapsto c,x\mapsto c'\}} \mid [\![P_j]\!]_{f,\{x,y\mapsto c,y\mapsto c'\}}\big)\big) \\
& \equiv & (\boldsymbol{\nu} c')\big([\![Q]\!]_{f,\{x,y\mapsto c,x\mapsto c'\}} \mid [\![P_j]\!]_{f,\{x,y\mapsto c,y\mapsto c'\}}\big)
\end{array}$$

Since P is well typed, it means that for all $i \in I$, $x \notin \mathtt{fn}(P_i)$ and $y \notin \mathtt{fn}(Q)$. Then, both $f, \{x, y \mapsto c, x \mapsto c'\}$ and $f, \{x, y \mapsto c, y \mapsto c'\}$ can be replaced by $f, \{x, y \mapsto c'\}$. We can rewrite the above as:

$$(\boldsymbol{\nu} c')\big([\![Q]\!]_{f,\{x,y\mapsto c'\}} \mid [\![P_j]\!]_{f,\{x,y\mapsto c'\}}\big)$$

On the other hand we have:

$$\begin{array}{rcl}
[\![P']\!]_f & = & [\![(\boldsymbol{\nu} xy)\big(Q \mid P_j\big)]\!]_f \\
& = & (\boldsymbol{\nu} c')\big([\![Q]\!]_{f,\{x,y\mapsto c'\}} \mid [\![P_j]\!]_{f,\{x,y\mapsto c'\}}\big)
\end{array}$$

The above implies:

$$[\![P]\!]_f \to\!\!\!\!\to [\![P']\!]_f$$

- Case (R-Par):

$$\frac{P \to Q}{P \mid R \to Q \mid R}$$

By (E-Composition) we have $[\![P \mid R]\!]_f = [\![P]\!]_f \mid [\![R]\!]_f$. By induction hypothesis $[\![P]\!]_f \to\!\!\!\!\to [\![Q]\!]_f$. We conclude that $[\![P]\!]_f \mid [\![R]\!]_f \to\!\!\!\!\to [\![Q]\!]_f \mid [\![R]\!]_f$ by applying (Rπ-Par) and (Rπ-Struct).

- Case (R-Struct):

$$\frac{P \equiv P',\ P' \to Q',\ Q' \equiv Q}{P \to Q}$$

By induction hypothesis $[\![P']\!]_f \to\!\!\!\!\to [\![Q']\!]_f$, and f is a renaming function for P'. By Lemma 23 we have $[\![P]\!]_f \equiv [\![P']\!]_f$ and $[\![Q']\!]_f \equiv [\![Q]\!]_f$. We conclude by (R$\pi$-Struct).

2. We discuss the case in which the reduction $[\![P]\!]_f \to Q$ is due to an interaction between an input and an output prefix that represent encodings of session

endpoints. The case of unrestricted names, as well as the case in which the reduction originates from the **case** construct are simpler, and are handled along the same line.

The input and the output prefixes that are consumed in the reduction $[\![P]\!]_f \to Q$ must be top-level, in the sense that they are not underneath other prefixes. To identify such prefixes, we suppose they are marked (in the same way as we did in Lemma 11).

Consider now the corresponding input and output prefixes in the session process P (those that produce, via the encoding, the two marked input and output prefixes of $[\![P]\!]_f$). Using structural congruence we can obtain a process $R \equiv P$ in which such prefixes are in contiguous position. Precisely, using structural congruence we can make sure that R is of the form:

$$R = C[\, x!\langle v\rangle.P_1 \mid y?(z).P_2 \,]$$

where $x!\langle v\rangle.P_1$ and $y?(z).P_2$ are the mentioned input and output processes and the hole of the context is at top level.

Since $R \equiv P$, it is sufficient to prove the statement of the theorem for R in place of P, as any derivative of R is also a derivative of P. Moreover, by Lemma 23, we also have that

$$[\![P]\!]_f \;\equiv\; [\![R]\!]_f$$

where $[\![R]\!]_f$ is of the form

$$[\![R]\!]_f = D[\, [\![x!\langle v\rangle.P_1]\!]_g \mid [\![y?(z).P_2]\!]_g \,] \tag{3}$$

for some context $D[\cdot]$ with a top-level hole, and some renaming function g with $g(x) = g(y)$. Now, expanding the definition of the encoding, for some fresh name c, and using brackets for the marked prefixes, we can continue from (3) as follows:

$$\begin{aligned} &= \quad D[\, (\boldsymbol{\nu} c)\underline{g_x!\langle [\![v]\!]_g, c\rangle}.[\![P_1]\!]_{g,\{x\mapsto c\}} \mid \underline{g_y?(z,c)}.[\![P_2]\!]_{g,\{y\mapsto c\}} \,] \\ &\to \quad D[\, (\boldsymbol{\nu} c)([\![P_1]\!]_{g,\{x\mapsto c\}} \mid [\![P_2]\!]_{g,\{y\mapsto c\}}[[\![v]\!]_g/z]) \,] \triangleq Q' \end{aligned} \tag{4}$$

Appealing to Lemma 11, we know that such a reduction, having consumed the two marked prefixes yields the following equivalence:

$$Q' \equiv Q$$

We will now find the derivative P' of the assertion of theorem (as a derivative of R) and prove

$$Q' \equiv [\![P']\!]_f$$

For this we distinguish two cases, corresponding to the cases in the statement of the theorem:

(i) x and y are not restricted in R (that is $(\boldsymbol{\nu} xy)$ does not appear in the context $C[\cdot]$);

(ii) x and y are restricted session endpoints, namely co-names.

The distinction is relevant because, in the π-calculus with sessions, communications along session endpoints is only possible if such endpoints are co-names.

We consider the second case (ii) first. Since $(\boldsymbol{\nu} xy)$ appears in the context $C[\cdot]$, its encoded context $D[\cdot]$ contains a restriction $(\boldsymbol{\nu} c')$ where c' is the linear name $g(x)$ (and also $g(y)$). The reduction in (4) is along the name c', which, being linear, after the reduction does not occur anymore in the continuation process. We can therefore remove such a restriction and simply replace it with the restriction at c. If $E[\cdot]$ is the context with c' replaced by c, we therefore have the following:

$$Q' \equiv E[\, [\![P_1]\!]_{g,\{x\mapsto c\}} \mid [\![P_2]\!]_{g,\{y\mapsto c\}}[[\![v]\!]_g/z]\,] \triangleq Q''$$

Moreover, since $(\boldsymbol{\nu} xy)$ is in the context $C[\cdot]$, we can infer the reduction:

$$R = C[\, x!\langle v\rangle.P_1 \mid y?(z).P_2\,] \;\rightarrow\; C[\, P_1 \mid P_2[v/z]\,] \triangleq P' \tag{5}$$

We can show that $Q'' = [\![P']\!]_f$ as follows:

$$\begin{aligned} [\![P']\!]_f &= D[\, [\![P_1]\!]_g \mid [\![P_2[v/z]]\!]_g\,] \\ &= D[\, [\![P_1]\!]_g \mid [\![P_2]\!]_g[[\![v]\!]_g/z]\,] \end{aligned}$$

where we have applied Lemma 22, and then, by renaming c' into c, namely by using context $E[\cdot]$, we obtain Q''.

Now the case (i), which is simpler. In this case context $D[\cdot]$ does not contain a restriction on c'; indeed name c' does not appear in Q'. As $(\boldsymbol{\nu} xy)$ is not in $C[\cdot]$, to infer a reduction akin to (5) we have to add $(\boldsymbol{\nu} xy)$ as follows:

$$\begin{aligned} (\boldsymbol{\nu} xy)R = (\boldsymbol{\nu} xy)C[\, x!\langle v\rangle.P_1 \mid y?(z).P_2\,] &\rightarrow (\boldsymbol{\nu} xy)C[\, P_1 \mid P_2[v/z]\,] \\ &\equiv C[\, (\boldsymbol{\nu} xy)(P_1 \mid P_2[v/z])\,] \triangleq P' \end{aligned}$$

Then, one concludes $Q' = [\![P']\!]_f$.

Note that the statement of item 2 of the operational correspondence theorem uses $\hookrightarrow$. In case the reduction of the session process P is due to an input and output communication, as in the proof outlined above, then $\hookrightarrow$ is simply $\equiv$. Otherwise, in case the reduction of the session process P is due to a branching and selection, then $\hookrightarrow$ is $\equiv$ extended with a **case** normalisation, as we showed in Section 3.3 on the maths server and client example. □

3.5. Properties Derived from the Encoding

In this section we show how we can use the encoding and properties from the linear π-calculus to derive the analogous properties in the π-calculus with session types. We start with a lemma stating type preservation under $\equiv$ by using the encoding.

Proof of Lemma 3:. If $\Gamma \vdash P$ and $P \equiv P'$, then $\Gamma \vdash P'$.

Proof. Assume $\Gamma \vdash P$ and $P \equiv P'$. By Theorem 21 we have $[\![\Gamma]\!]_f \vdash [\![P]\!]_f$ for some renaming function f for P. By Lemma 23 $[\![P]\!]_f \equiv [\![P']\!]_f$, then by Lemma 9 we have $[\![\Gamma]\!]_f \vdash [\![P']\!]_f$. We conclude by Theorem 20. □

Now we are ready to prove subject reduction in the π-calculus with session types by using our encoding and subject reduction in the linear π-calculus.

Proof of Theorem 4:. If $\Gamma \vdash P$ and $P \to P'$, then $\Gamma \vdash P'$.

Proof. Assume $\Gamma \vdash P$ and $P \to P'$. By Theorem 21 we have $[\![\Gamma]\!]_f \vdash [\![P]\!]_f$, for some renaming function f for P and by point 1 of Theorem 24 we have $[\![P]\!]_f \to\!\!\!\!\!\rightarrow [\![P']\!]_f$. Let Q be the process such that $[\![P]\!]_f \to Q \hookrightarrow [\![P']\!]_f$. By subject reduction for the linear π-calculus, given by Theorem 10, we have $[\![\Gamma]\!]_f \vdash Q$. By subject congruence, given by Lemma 9, and by Theorem 10, we have $[\![\Gamma]\!]_f \vdash [\![P']\!]_f$. By Theorem 20 we conclude that $\Gamma \vdash P'$. □

4. Subtyping

Subtyping is a relation between syntactic types based on a notion of substitutability. In the π-calculus, the language constructs meant to act on channels of the supertype can also act on channels of the subtype. If T is a subtype of T', then any channel of type T can be safely used in a context where a channel of type T' is expected. Subtyping has been studied extensively in the standard π-calculus [25, 27]. It has also been studied for session types [14, 9]. In this section we show that subtyping on the standard π-calculus can be used to derive subtyping on session types. Subtyping rules for both calculi are presented in Fig. 11: the rules for standard π-calculus are given at the top and the rules for session types are given at the bottom of the figure. We use $<\!:$ to denote subtyping for session types and $\leq$ for standard π-types.

Rules (Sπ-Refl) and (Sπ-Trans) state that subtyping is a preorder. Rules (Sπ-`ii`) and (Sπ-`oo`) define subtyping for input and output linear channel types, respectively. These rules assert that input channels are co-variant and output channels are contra-variant in the types of values they transmit. Finally, rule (Sπ-Variant) defines subtyping of variant types. It is co-variant both in depth and in breadth. We comment now on the subtyping rules for session types. Rules (S-Unit) and (S-End) define subtyping on `Unit` type and on a terminated channel type, respectively. Rules (S-Inp) and (S-Out) define subtyping on input and output session types, respectively. As before, the input operation is co-variant whilst the output operation is contra-variant. The continuation type is co-variant in both cases. This is a difference with respect to the corresponding rules in the standard π-calculus. Rules (S-Brch) and (S-Sel) define subtyping for choice types. They are both co-variant in depth in the types of values they transmit and co-variant and contra-variant in breadth, respectively.

In the π-calculus with sessions and subtyping, one must deal both with standard subtyping on π-types and subtyping on session types. This introduces

$$\frac{}{\tau \leq \tau}\,(\text{S}\pi\text{-Refl}) \qquad \frac{\tau \leq \tau' \qquad \tau' \leq \tau''}{\tau \leq \tau''}\,(\text{S}\pi\text{-Trans})$$

$$\frac{\widetilde{\tau} \leq \widetilde{\tau}'}{\ell_{\mathtt{i}}[\widetilde{\tau}] \leq \ell_{\mathtt{i}}[\widetilde{\tau}']}\,(\text{S}\pi\text{-}\mathtt{ii}) \qquad \frac{\widetilde{\tau}' \leq \widetilde{\tau}}{\ell_{\mathtt{o}}[\widetilde{\tau}] \leq \ell_{\mathtt{o}}[\widetilde{\tau}']}\,(\text{S}\pi\text{-}\mathtt{oo})$$

$$\frac{I \subseteq J \qquad \tau_i \leq \tau'_j \quad \forall i \in I}{\langle l_i\text{-}\tau_i\rangle_{i\in I} \leq \langle l_j\text{-}\tau'_j\rangle_{j\in J}}\,(\text{S}\pi\text{-Variant})$$

$$\frac{}{\mathtt{Unit} <: \mathtt{Unit}}\,(\text{S-Unit}) \qquad \frac{}{\mathtt{end} <: \mathtt{end}}\,(\text{S-End})$$

$$\frac{T <: T' \qquad S <: S'}{?T.S <: ?T'.S'}\,(\text{S-Inp}) \qquad \frac{T' <: T \qquad S <: S'}{!T.S <: !T'.S'}\,(\text{S-Out})$$

$$\frac{I \subseteq J \qquad S_i <: S'_j \quad \forall i \in I}{\&\{l_i : S_i\}_{i\in I} <: \&\{l_j : S'_j\}_{j\in J}}\,(\text{S-Brch}) \qquad \frac{I \supseteq J \qquad S_i <: S'_j \quad \forall j \in J}{\oplus\{l_i : S_i\}_{i\in I} <: \oplus\{l_j : S'_j\}_{j\in J}}\,(\text{S-Sel})$$

Figure 11: Subtyping rules for π-types ($\leq$) and for session types ($<:$).

a duplication of effort that grows as the type syntax and type system become richer. For example, this duplication is very heavy when recursive types are included. If the type system is structural, then subtyping on recursive types is established with coinductive techniques, e.g., simulation relations. These techniques must be defined and proved sound both on standard π-types and on session types. In addition, on session types one also needs coinductive techniques to formalise type duality.

The encoding is used, as in the previous section, to derive basic properties of session types. For Theorems 20 and 21 to remain valid, we have to take the subtyping relation into account. Therefore, it is important to prove the validity of subtyping, which is necessary in order to extend subject reduction and type safety. We now state the soundness and completeness of the encoding of types with respect to subtyping in session types and linear π-types.

Theorem 25 (Soundness for Subtyping)**.** If $[\![T]\!] \leq [\![T']\!]$, then $T <: T'$.

The proof is done by induction on the structure of session types T, T'.

Theorem 26 (Completeness for Subtyping)**.** If $T <: T'$ then $[\![T]\!] \leq [\![T']\!]$.

The proof is done by induction on the last rule applied in the derivation of $T <: T'$. The full proofs of the above theorems are given in Appendix A.

To be able to use the subtyping relation, we introduce the *subsumption rule* in the type system, both on the standard typed π-calculus and the π-calculus

with session types.

$$\frac{\Gamma \vdash x : \tau \qquad \tau \leq \tau'}{\Gamma \vdash x : \tau'} \qquad \text{and} \qquad \frac{\Gamma \vdash x : T \qquad T <: T'}{\Gamma \vdash x : T'}$$

Then, we can prove the correctness of the encoding with respect to typing values and processes by using the updated type systems.

Papers that study subtyping [14, 30], prove several results related to this feature. We can derive the main results in these papers as straightforward corollaries via the encoding, in the same way as we did for the subject reduction, thus using Theorem 25 and Theorem 26. Examples of such results include: reflexivity and transitivity of subtyping, and other auxiliary lemmas (e.g., substitution).

Example 27 (Upgrading the Maths Server)**.** Suppose we want to upgrade the maths *server* presented in Section 3.3, for instance by extending the *equal* service to `Real` numbers [14]. The upgrade should not disrupt possible or ongoing communications with the *client* process, defined in Section 3.3. For this, we can exploit the session subtyping relation given at the bottom of Fig.11 combined with the subsumption rule added to the type system for the π-calculus with session types. The session type S_{real} for the endpoint x of the the upgraded $server_{up}$ is:

$$\begin{array}{rcll} S_{real} & \triangleq & \&\{ & plus : ?\texttt{Int}.?\texttt{Int}.!\texttt{Int}.\texttt{end}, \\ & & & equal : ?\texttt{Real}.?\texttt{Real}.!\texttt{Bool}.\texttt{end}, \\ & & & neg : ?\texttt{Int}.!\texttt{Int}.\texttt{end} \quad \} \end{array}$$

Assume `Int` <: `Real`, since session subtyping is co-variant in input, then by (S-Inp) we have that $S <: S_{real}$. Hence $server_{up}$ can communicate with the *client* process of Section 3.3. The encoding of the session type S_{real} gives:

$$\begin{array}{rcll} [\![S_{real}]\!] & = & \ell_{\mathtt{i}}[\langle & plus\_\ell_{\mathtt{i}}[\texttt{Int}, \ell_{\mathtt{i}}[\texttt{Int}, \ell_{\mathtt{o}}[\texttt{Int}, \varnothing[]]]], \\ & & & equal\_\ell_{\mathtt{i}}[\texttt{Real}, \ell_{\mathtt{i}}[\texttt{Real}, \ell_{\mathtt{o}}[\texttt{Bool}, \varnothing[]]]], \\ & & & neg\_\ell_{\mathtt{i}}[\texttt{Int}, \ell_{\mathtt{o}}[\texttt{Int}, \varnothing[]]] \quad \rangle] \end{array}$$

Since also linear π-calculus channel types are co-variant in input, by (Sπ-ii), we have $[\![S]\!] \leq [\![S_{real}]\!]$.

Another possible upgrade for the server is the addition of a new service, say *mult*, that multiplicates two integers. Calling S_{mult} this new type, by rule (S-Brch) we obtain $S <: S_{mult}$. The subtyping relation on the encoded types, namely $[\![S]\!] \leq [\![S_{mult}]\!]$, is obtained by using rules (Sπ-ii) and (Sπ-Variant).

5. Polymorphism

Polymorphism is a common and useful type abstraction in programming languages as allows operations that are generic by using an expression with

several types. There are two forms of polymorphism for the π-calculus with and without sessions: parametric and bounded polymorphism. In this section we study both forms. Parametric polymorphism is already present and well studied in standard π-calculus [27]. It has also been studied for the π-calculus with session types [1]. In Section 5.1 we show that, by the encoding and by adding parametric polymorphism to the syntax of types (and terms) in sessions, we obtain the properties in the polymorphic sessions for free, deriving them from the theory of the polymorphic π-calculus.

Bounded polymorphism is studied for the π-calculus with session types. In Section 5.2 we show how one can obtain bounded polymorphism in session types, by adding bounded polymorphism to the π-calculus types and by exploiting our encoding.

5.1. Parametric Polymorphism

Syntax and Semantics. Let us first consider parametric polymorphism. The following syntax is an extension of the ones presented in Section 2.1. The same extensions are performed to the syntax in Section 2.2, with the difference that τ is used instead of T.

$$
\begin{array}{rcll}
T & ::= & \ldots \mid X & \text{(session type variable)} \\
 & & \mid \langle X; T\rangle & \text{(polymorphic session type)} \\
P & ::= & \ldots \mid \mathbf{open}\ v\ \mathbf{as}\ (X;x)\ \mathbf{in}\ P & \text{(unpacking process)} \\
v & ::= & \ldots \mid \langle T; v\rangle & \text{(polymorphic session value)} \\
\Delta & ::= & \varnothing \mid \Delta, X & \text{(type variable environment)}
\end{array}
$$

We extend the syntax of session types and standard π-types with *type variable* X and *polymorphic type* $\langle X;T\rangle$ and $\langle X;\tau\rangle$, respectively. The rest of the type constructs remain unchanged. Modifications in the syntax of types trigger modifications in the syntax of terms. We add the *polymorphic value* for sessions $\langle T; v\rangle$ and for the standard π-calculus $\langle \tau; v\rangle$ and the *unpacking process* $\mathbf{open}\ v\ \mathbf{as}\ (X;x)\ \mathbf{in}\ P$ to both calculi. These constructs are native of the standard π-calculus. In addition to Γ, here we consider another typing context Δ containing polymorphic type variables.

The reduction rule for the unpacking process is given below:

$$(\text{R-Unpack}) \qquad \mathbf{open}\ \langle T; v\rangle\ \mathbf{as}\ (X;x)\ \mathbf{in}\ P \to P[T/X][v/x]$$

It states that process $\mathbf{open}\ \langle T; v\rangle\ \mathbf{as}\ (X;x)\ \mathbf{in}\ P$, with the guard being a polymorphic value $\langle T; v\rangle$, reduces to process P where two substitutions occur: type T substitutes type variable X and value v substitutes the placeholder variable x. The reduction rule (Rπ-Unpack) for the standard π-calculus is the same as above, where $\langle T; v\rangle$ is replaced by $\langle \tau; v\rangle$.

Typing Rules. Typing judgements are of the form $\Gamma; \Delta \vdash v : T$ for values and $\Gamma; \Delta \vdash P$ for processes. The typing rules for the polymorphic π-calculus with session types are given in Fig. 12 and they are straightforward. The typing rules (Tπ-PolyVal) and (Tπ-Unpack) for the standard π-calculus follow the same line, hence we omit them for simplicity.

$$\frac{\Gamma\ ;\ \Delta \vdash v : T[T'/X]}{\Gamma\ ;\ \Delta \vdash \langle T'; v\rangle : \langle X; T\rangle}\ (\text{T-PolyVal})$$

$$\frac{\Gamma_1\ ;\ \Delta \vdash v : \langle X; T\rangle \qquad \Gamma_2, x : T\ ;\ \Delta, X \vdash P}{\Gamma_1 \circ \Gamma_2\ ;\ \Delta \vdash \mathbf{open}\ v\ \mathbf{as}\ (X; x)\ \mathbf{in}\ P}\ (\text{T-Unpack})$$

Figure 12: Typing rules for parametric polymorphic constructs

Encoding. Since we added polymorphic constructs to the syntax of types and left the syntax of session types unchanged, the encoding of session types remains as before, hence the encoding of types is a homomorphism. The same holds for the terms of the calculus with or without sessions: we added the same value and process constructs on both sides and thus the encoding is again a homomorphism. We only present the new rules.

$$\begin{array}{rcll}
[\![X]\!] & \triangleq & X & (\text{E-PolyVar}) \\
[\![\langle X; T\rangle]\!] & \triangleq & \langle X; [\![T]\!]\rangle & (\text{E-PolyType}) \\
[\![\langle T; v\rangle]\!]_f & \triangleq & \langle [\![T]\!]; [\![v]\!]_f\rangle & (\text{E-PolyVal}) \\
[\![\mathbf{open}\ v\ \mathbf{as}\ (X; x)\ \mathbf{in}\ P]\!]_f & \triangleq & \mathbf{open}\ [\![v]\!]_f\ \mathbf{as}\ (X; f_x)\ \mathbf{in}\ [\![P]\!]_f & (\text{E-Unpack})
\end{array}$$

The encoding of typing contexts is given by:

$$\begin{array}{rcll}
[\![\varnothing]\!]_f & \triangleq & \varnothing & (\text{E-Empty}) \\
[\![\Gamma, x : T]\!]_f & \triangleq & [\![\Gamma]\!]_f \uplus f_x : [\![T]\!] & (\text{E-Gamma}) \\
[\![\Gamma; \Delta]\!]_f & \triangleq & [\![\Gamma]\!]_f; \Delta & (\text{E-Delta})
\end{array}$$

We encode Γ as in Fig. 10, and on Δ the encoding is the identity function, since the encoding of type variables is the identity function.

To complete Lemma 18 and Lemma 19 on the correctness of the encoding with respect to typing values, it suffices to add the case for polymorphic values. To complete Theorems 20 and 21 on the correctness of the encoding with respect to typing processes, it suffices to add the case for the unpack process. However, adding these cases requires modification in the typing judgements: previous typing judgements of the form $\Gamma \vdash v : T$ should be now written as $\Gamma; \Delta \vdash v : T$ and previous typing judgements of the form $\Gamma \vdash Q$ should be now written as $\Gamma; \Delta \vdash Q$ (with $\Delta = \varnothing$ in absence of polymorphism). The proofs of the above results for the parametric polymorphic π-calculus are given in Appendix B. Operational correspondence for bounded polymorphic processes follows the same line as Theorem 24.

5.2. Bounded Polymorphism

Syntax and Semantics. We now consider bounded polymorphism [12], which is a form of parametric polymorphism. This kind of polymorphism has not been studied yet in the π-calculus; we add it and show how we can derive

bounded polymorphism in session types by using the standard π-types. Bounded polymorphism in session types [12] is added only to the labels of branch and select constructs. In our work, we specify only upper bounds and use only basic types in the bounds. This is a simplification with respect to [12] and it is sufficient to illustrate how the encoding works. We report only on the new constructs added to the syntax of types and terms. Type B stands for basic types (e.g., `Unit`, `Bool`, ...), as opposed to channel types.

$$
\begin{array}{rlll}
S ::= & \ldots & \mid \oplus\{l_i(X_i <: B_i) : T_i\}_{i\in I} & \text{(bounded polymorphic select)} \\
 & & \mid \&\{l_i(X_i <: B_i) : T_i\}_{i\in I} & \text{(bounded polymorphic branch)} \\
P ::= & \ldots & \mid x \lhd l_j(B).P & \text{(bounded polymorphic selection)} \\
 & & \mid x \rhd \{l_i(X_i <: B_i) : P_i\}_{i\in I} & \text{(bounded polymorphic branching)}
\end{array}
$$

In order to have bounded polymorphism also in the π-calculus, we add it to the labels of variant types and the case process.

$$
\begin{array}{rlll}
\tau ::= & \ldots & \mid \langle l_i(X_i \le B_i)\_\tau_i\rangle_{i\in I} & \text{(bounded poly variant)} \\
P ::= & \ldots & \mid \mathbf{case}\, v\, \mathbf{of}\, \{l_i(X_i \le B_i)\_x_i \rhd P\}_{i\in I} & \text{(bounded poly case)} \\
v ::= & \ldots & \mid l_j(B)\_v & \text{(bounded poly variant value)}
\end{array}
$$

On both π-calculi with or without sessions, we should take into account the condition $(X_i \le B_i)$ and X_i should be instantiated with a type satisfying the condition. The syntax of processes is modified accordingly. We give now the reduction rules for bounded polymorphic processes.

$$
\begin{array}{rll}
(\text{R-BPolySel}) & (\boldsymbol{\nu} xy)(x \lhd l_j(B).P \mid y \rhd \{l_i(X_i <: B_i) : P_i\}_{i\in I}) \to & \\
 & \qquad (\boldsymbol{\nu} xy)(P \mid P_j[B/X_j]) \quad j \in I & \\
(\text{R}\pi\text{-BPolyCase}) & \mathbf{case}\, l_j(B)\_v\, \mathbf{of}\, \{l_i(X_i \le B_i)\_x_i \rhd P\}_{i\in I} \to & \\
 & \qquad P_j[B/X_j][v/x_j] \quad j \in I &
\end{array}
$$

Typing Rules. The typing rules for bounded polymorphic constructs are similar on both π-calculi and are given in Fig. 13.

Encoding. The encoding is once again a homomorphism and we present the most relevant cases.

$$
\begin{array}{lcll}
[\![\oplus\{l_i(X_i <: B_i) : T_i\}_{i\in I}]\!] & \triangleq & \ell_{\mathtt{o}}[\langle l_i(X_i \le B_i)\_[\![\overline{T_i}]\!]\rangle_{i\in I}] & (\text{E-BPolySel}) \\
[\![\&\{l_i(X_i <: B_i) : T_i\}_{i\in I}]\!] & \triangleq & \ell_{\mathtt{i}}[\langle l_i(X_i \le B_i)\_[\![T_i]\!]\rangle_{i\in I}] & (\text{E-BPolyBrch})
\end{array}
$$

$$
\begin{array}{l}
(\text{E-BPolySelection}) \\
\qquad [\![x \lhd l_j(B).P]\!]_f \triangleq (\boldsymbol{\nu} c) f_x!\langle l_j(B)\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}} \\
(\text{E-BPolyBranching}) \\
[\![x \rhd \{l_i(X_i <: B_i) : P_i\}_{i\in I}]\!]_f \triangleq f_x?(y).\, \mathbf{case}\, y\, \mathbf{of}\, \{l_i(X_i \le B_i)\_c \rhd [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}
\end{array}
$$

By using the encoding and the bounded polymorphism in the standard π-calculus, we can derive bounded polymorphism in the π-calculus with session

$$\frac{\Gamma_1 \;;\; \Delta \vdash x : \oplus\{l_i(X_i <: B_i) : T_i\}_{i\in I} \qquad \Gamma_2, x : T_j[B/X_j] \;;\; \Delta \vdash P \quad j \in I \qquad B <: B_i \quad \forall i \in I}{\Gamma_1 \circ \Gamma_2 \;;\; \Delta \vdash x \lhd l_j(B).P} \text{ (T-BPolySel)}$$

$$\frac{\Gamma_1 \;;\; \Delta \vdash x : \&\{l_i(X_i <: B_i) : T_i\}_{i\in I} \qquad \Gamma_2, x : T_i \;;\; \Delta, X_i <: B_i \vdash P_i \quad \forall i \in I}{\Gamma_1 \circ \Gamma_2 \;;\; \Delta \vdash x \rhd \{l_i(X_i <: B_i) : P_i\}_{i\in I}} \text{ (T-BPolyBrch)}$$

$$\frac{\Gamma \;;\; \Delta \vdash v : \tau_j[B/X_j] \quad j \in I \qquad B \leq B_i \quad \forall i \in I}{\Gamma \;;\; \Delta \vdash l_j(B)\_v : \langle l_i(X_i \leq B_i)\_\tau_i \rangle_{i\in I}} \text{ (T}\pi\text{-BPolyLVal)}$$

$$\frac{\Gamma_1 \;;\; \Delta \vdash v : \langle l_i(X_i \leq B_i)\_\tau_i \rangle_{i\in I} \qquad \Gamma_2, x_i : \tau_i \;;\; \Delta, X_i \leq B_i \vdash P_i \quad \forall i \in I}{\Gamma_1 \uplus \Gamma_2 \;;\; \Delta \vdash \mathbf{case}\, v\, \mathbf{of}\, \{l_i(X_i \leq B_i)\_x_i \rhd P\}_{i\in I}} \text{ (T}\pi\text{-BPolyCase)}$$

Figure 13: Typing rules for bounded polymorphic constructs

types. Furthermore, all the results presented in Section 4 and 5.1 are now derivable for free. To complete Lemma 18 and Lemma 19 on correctness of the encoding with respect to typing values, it suffices to add the cases for bounded polymorphic variables. These cases follow immediately by (E-BPolySel) and (E-BPolyBrch) and by typing rules (T-Var) and (Tπ-Var). To complete Theorem 20 and Theorem 21 on the correctness of the encoding with respect to typing processes, it suffices to add the cases for bounded branching and selection. The modifications to the typing judgements are as in parametric polymorphism. These modifications will also influence the operational correspondence. The proofs of the above theorems as well as the operational correspondence are given in Appendix B.2.

Example 28 (Another Upgrade of the Maths Sever)**.** In Example 27 we upgraded our maths *server* using subtyping (in depth, adding `Int` <: `Real`, or in breadth, adding a new service *mult*). We describe now an upgrade that employs bounded polymorphism. As shown by Gay [12], there are upgrading scenarios where subtyping alone is not sufficient, and in these cases, one can appeal to bounded polymorphism. We recall that *server* and *client* are the processes defined in Section 3.3, and S is the session type used to typecheck the *server*'s endpoint. We upgrade the service *equal* by using bounded polymorphism and we define the upgraded session type S_{bnd} as:

$$\begin{array}{rcll} S_{bnd} & \triangleq & \&\{ & \mathit{plus} : ?\texttt{Int}.?\texttt{Int}.!\texttt{Int}.\texttt{end}, \\ & & & \mathit{equal}(X <: \texttt{Real}) : ?X.?X.!\texttt{Bool}.\texttt{end}, \\ & & & \mathit{neg} : ?\texttt{Int}.!\texttt{Int}.\texttt{end} \quad \} \end{array}$$

where the type variable X has an upper bound of type `Real`. The new server,

that uses the new type S_{bnd}, can communicate with any client that, when selecting the service *equal*, sends a value of any subtype of `Real`. In particular, the server can communicate with our original *client* process of Section 3.3, assuming `Int` <: `Real`.

Typing rules and reduction rules for bounded polymorphism follow the same lines in the π-calculus with and without sessions. As a consequence, also in the encoding, communication between the upgraded server and the original client is possible. We show the encoding of session type S_{bnd}:

$$\begin{array}{rcll} [\![S_{bnd}]\!] & = & \ell_{\mathtt{i}}[\langle & plus\_\ell_{\mathtt{i}}[\mathtt{Int}, \ell_{\mathtt{i}}[\mathtt{Int}, \ell_{\mathtt{o}}[\mathtt{Int}, \varnothing[]]]] \\ & & & equal(X \leq \mathtt{Real})\_\ell_{\mathtt{i}}[X, \ell_{\mathtt{i}}[X, \ell_{\mathtt{o}}[\mathtt{Bool}, \varnothing[]]]], \\ & & & neg\_\ell_{\mathtt{i}}[\mathtt{Int}, \ell_{\mathtt{o}}[\mathtt{Int}, \varnothing[]]] \quad \rangle] \end{array}$$

6. Higher-Order π-calculus

Higher-order π-calculus (HOπ) models mobility of processes that can be sent and received and thus can be run locally [27]. Higher-order communication for sessions has the same benefits as for the π-calculus, in particular, it models code mobility in a distributed scenario. What we want to do is to use HOπ to provide sessions with higher-order capabilities by exploiting the encoding, as we did with subtyping and polymorphism.

Syntax and Semantics. The syntax of types and terms for the HOπ with sessions [22] is given by the following grammar. The syntax of types and terms for the standard HOπ is the same as the one below, with the difference that τ replaces T.

$$\begin{array}{rcll} \sigma & ::= & \ldots \mid T & \text{(general type)} \\ & & \mid \diamond & \text{(process type)} \\ T & ::= & \ldots \mid T \to \sigma & \text{(functional type)} \\ & & \mid T \stackrel{1}{\to} \sigma & \text{(linear functional type)} \\ P & ::= & \ldots \mid PQ & \text{(application)} \\ & & \mid v & \text{(values)} \\ v & ::= & \ldots \mid \lambda x : T.P & \text{(abstraction)} \end{array}$$

Let $\diamond$ denote the type of a process, and σ range over a type T or $\diamond$. The new types added to T are the functional type $T \to \sigma$, assigned to a functional term that can be used without any restriction, and the linear functional type $T \stackrel{1}{\to} \sigma$, assigned to a term that should be used exactly once. The reason for this is that a function may contain free session channels, hence it should necessarily be used at least once in order to complete the session and should not be used more than once, so not to violate session safety. Regarding terms, the π-calculus with sessions is augmented with call-by-value λ-calculus primitives, namely *abstraction* ($\lambda x : T.P$) and *application* (PQ).

$$\text{(T-HoAbs1)}\quad \frac{\Phi, x:T; \Gamma; \mathcal{S} \vdash P:\sigma \qquad \text{if } T = T' \overset{1}{\to} \sigma \text{ then } x \in \mathcal{S}}{\Phi; \Gamma; \mathcal{S} - \{x\} \vdash \lambda x:T.P : T \to \sigma}$$

$$\text{(T-HoAbs2)}\quad \frac{\Phi; \Gamma, x:T; \mathcal{S} \vdash P:\sigma}{\Phi; \Gamma; \mathcal{S} \vdash \lambda x:T.P : T \to \sigma}$$

$$\text{(T-HoApp)}\quad \frac{\Phi; \Gamma_1; \mathcal{S}_1 \vdash P : T \overset{1}{\to} \sigma \qquad \Phi; \Gamma_2; \mathcal{S}_2 \vdash Q : T \qquad \text{if } T = T' \to \sigma' \text{ then } \mathtt{un}(\Gamma_2) \text{ and } \mathcal{S}_2 = \varnothing}{\Phi; \Gamma_1 \circ \Gamma_2; \mathcal{S}_1 \cup \mathcal{S}_2 \vdash PQ : \sigma}$$

Figure 14: Typing rules for higher-order constructs

We present the new reduction rules for the HOπ with sessions which are added to the rules in Fig. 4. The reduction rules for the standard HOπ are the same as the ones below, with the difference that τ replaces T.

$$\begin{aligned}
&\text{(R-Beta)} && (\lambda x:T.P)v \to P[v/x] \\
&\text{(R-ApplLeft)} && P \to P' \Longrightarrow PQ \to P'Q \\
&\text{(R-ApplRight)} && P \to P' \Longrightarrow vP \to vP'
\end{aligned}$$

Typing Rules. Typing judgements for the higher-order π-calculus with and without sessions are of the form $\Phi; \Gamma; \mathcal{S} \vdash v : T$ and $\Phi; \Gamma; \mathcal{S} \vdash v : \tau$, respectively where Φ associates variables to value types, except session types; Γ associates variables to session types; and $\mathcal{S}$ denotes the set of linear functional variables. A typing judgement is well formed if $\mathcal{S} \subseteq \mathtt{dom}(\Phi)$ and $\mathtt{dom}(\Phi) \cap \mathtt{dom}(\Gamma) = \varnothing$. The new typing rules are presented in Fig. 14. For simplicity we omit the rules for the standard HOπ as they are the same as those in Fig. 14.

Encoding. The encoding of typing contexts is an extension of the one given in Fig. 10.

$$\begin{aligned}
[\![\varnothing]\!]_f &\triangleq \varnothing && \text{(E-Empty)} \\
[\![\Phi; \Gamma; \mathcal{S}]\!]_f &\triangleq [\![\Phi]\!]_f; [\![\Gamma]\!]_f; [\![\mathcal{S}]\!]_f && \text{(E-HOContext)} \\
[\![\Gamma, x:T]\!]_f &\triangleq [\![\Gamma]\!]_f \uplus f_x : [\![T]\!] && \text{(E-Gamma)} \\
[\![\Phi, x:T]\!] &\triangleq [\![\Phi]\!]_f, f_x : [\![T]\!] && \text{(E-Phi)}
\end{aligned}$$

The encoding of types and terms is a homomorphism on the higher-order constructs added to both the π-calculi.

$$\begin{aligned}
[\![\Diamond]\!] &\triangleq \Diamond && \text{(E-ProcType)} \\
[\![T \overset{1}{\to} \sigma]\!] &\triangleq [\![T]\!] \overset{1}{\to} [\![\sigma]\!] && \text{(E-LinFunType)} \\
[\![T \to \sigma]\!] &\triangleq [\![T]\!] \to [\![\sigma]\!] && \text{(E-FunType)} \\
[\![\lambda x:T.P]\!]_f &\triangleq \lambda x : [\![T]\!].[\![P]\!]_f && \text{(E-Abstraction)} \\
[\![PQ]\!]_f &\triangleq [\![P]\!]_f [\![Q]\!]_f && \text{(E-Application)}
\end{aligned}$$

The process type, functional types, abstraction and application in the HOπ calculus with sessions are encoded respectively as the process type, functional types, abstraction and application in the standard HOπ calculus.

We are ready now to give the results on the correctness of the encoding with respect to typing in case of HOπ. Namely, a HOπ process P is of type σ in some typing context if and only if the encoding of P is of type encoding of σ in the encoding of the same typing context.

Theorem 29 (Soundness)**.** If $[\![\Phi;\Gamma;\mathcal{S}]\!]_f \vdash [\![P]\!]_f : [\![\sigma]\!]$ for some renaming function f for P, then $\Phi;\Gamma;\mathcal{S} \vdash P : \sigma$.

Theorem 30 (Completeness)**.** If $\Phi;\Gamma;\mathcal{S} \vdash P : \sigma$, then $[\![\Phi;\Gamma;\mathcal{S}]\!]_f \vdash [\![P]\!]_f : [\![\sigma]\!]$ for some renaming function f for P.

The result of the operational correspondence for the HOπ is as before, given by Theorem 24. The proofs of the above two theorems as well as the operational correspondence for the HOπ are given in Appendix C.

7. Further considerations

As explained in the previous sections, a session type is interpreted as a linear channel type, which in turn carries a linear channel. In order to satisfy this linearity, on the side of terms, a fresh channel is created at any step of communication and is sent to the partner along with the message to be transmitted. The sent channel will be used to handle the rest of the communication. What we just said describes the encoding of the output process transmitting some value, call it v:

$$[\![x!\langle v\rangle.P]\!]_f \triangleq (\boldsymbol{\nu} c) f_x!\langle v, c\rangle.[\![P]\!]_{f,\{x\mapsto c\}} \tag{6}$$

One can argue that there is an overhead in doing so, and that is not necessary. Since the fresh names are assigned linear types, once they are used, we are guaranteed by the type system that those channels are not going to be used again. An optimised approach permits to reuse the same linear channel. For example, we can optimise the above process as:

$$[\![x!\langle v\rangle.P]\!] \triangleq x!\langle v, x\rangle.[\![P]\!] \tag{7}$$

This leads to a typing problem, since the process obviously violates linearity. In order to overcome this problem, we introduce the following typing rule:

$$\frac{\Gamma_1 \vdash x : \ell_\circ[\widetilde{T}] \qquad \widetilde{\Gamma_2}, x : \ell_\alpha[\widetilde{S}] \vdash \tilde{v} : \widetilde{T} \qquad \Gamma_3, x : \ell_{\overline{\alpha}}[\widetilde{S}] \vdash P}{\Gamma_1 \uplus \widetilde{\Gamma_2} \uplus \Gamma_3 \vdash x!\langle \tilde{v}\rangle.P} \,(\text{T}\pi\text{-NewOut})$$

We prove that (6) and (7) are *typed strong barbed congruent*. The details are shown in Appendix D. The modified rule allows reuse of channel names. The optimisation would make the encoding of session types simpler— a linear channel would be used like a session channel and therefore the function parameter f

of the encoding would not be needed. In our presentation, we have preferred not to do so in order to relate ourselves to the standard π-calculus and its theory.

The above is an optimisation on the treatment of linear channels, that came up while working on the encoding. We mentioned it because it makes the encoding even simpler, and because we think it may be useful also in other situations.

8. Related Work

The idea of the encoding of session types into π-calculus linear types is not new. Kobayashi [19] was the first to propose such an encoding, but he did not provide any formal study of it. Demangeon and Honda [9] provide a subtyping theory for a π-calculus augmented with branch and select constructs and show an encoding of the session π-calculus. They prove soundness of the encoding and full abstraction. The main differences with respect to our work are: (i) the target language is closer to the session π-calculus having branch and select constructs (instead of having just one variant construct), and a refined subtyping theory is provided, while we focus on encoding the session π-calculus in the standard π-calculus in order to exploit the rich and well-established theory of the latter; (ii) we study the encoding in a systematic way as a means to formally derive session types and their properties, in order to provide a methodology for the treatment of session types and their extensions without the burden of establishing the underlying theory (specifically, Demangeon and Honda [9] focus on subtyping issues).

Other expressiveness results regarding binary session types theory include the work by Caires and Pfenning [2]. They present a type system for the π-calculus that corresponds to the standard sequent calculus proof system for Dual Intuitionistic Linear Logic (DILL). They give an interpretation of intuitionistic linear logic formulas as a form of session types. Later on Wadler [32], by following Caires and Pfenning [2], proposes a calculus where propositions of classical linear logic correspond to session types.

Igarashi and Kobayashi [17] have developed a single generic type system (GTS) for the π-calculus from which numerous specific type systems can be obtained by varying certain parameters. A range of type systems are thus obtained as instances of the generic one. Gay, Gesbert and Ravara [13] define an encoding from session types and terms into GTS by proving operational correspondence and correctness of the encoding. However, as the authors state, the encoding they present is very complex and deriving properties of sessions by using GTS would be more difficult than proving them directly, from scratch.

Carbone et al. [4] show that one can use the encoding, together with the type system for lock freedom in the π-calculus [18], to derive the progress property in the π-calculus with sessions. Padovani [23] studies deadlock and lock-freedom in linear π-calculus and relates it to session π-calculus via our encoding. Padovani [24] uses our encoding and the theory of linear types to define type reconstruction algorithms for session types. Dardha [6] adds a new extension to our encoding, namely recursive types, to further investigate its robustness.

9. Conclusion

This paper proposes an encoding of binary session types into standard π-types, more precisely into linear types and variant types. Linear types [20, 27] force a channel to be used exactly once. Variant types [26, 27] are a labelled form of disjoint union of types. We develop Kobayashi's proposal of an encoding of session types into standard π-types. We show that the encoding is faithful, in that it allows us to derive the basic properties of session types, exploiting the analogous properties of π-types. We then show that the encoding is robust, by analysing a few non-trivial extensions to session types, namely subtyping, polymorphism and higher-order communication. Finally, we propose an optimisation of linear channels permitting to reuse the same channel for the continuation of the session and prove a typed barbed congruence result. This optimisation considerably simplifies Kobayashi's encoding, which does not need any renaming function. The encoding of session types, however is the same as before.

The benefits coming from the encoding include the elimination of the redundancy introduced both in the syntax of types and of terms, and the derivation of properties (such as subject reduction and type safety) as straightforward corollaries (thus eliminating redundancy also in the proofs). Issues like opposite endpoints of a session channel and duality of session types assigned to these endpoints are handled by the theory of the standard typed π-calculus: there is just one channel we deal with (no need to distinguish endpoints) and duality boils down to having opposite outermost capabilities of linear channel types. Moreover, the robustness of the encoding allows us to easily obtain extensions of the session π-calculus, by exploiting the theory of the standard π-calculus. As we have shown in Section 5.2 on bounded polymorphism, our approach works smoothly even when the intended extension is not already present in the π-calculus. In this case, one can just enrich the π-calculus with the intended feature and obtain the same one in sessions via the encoding, as passing through the π-calculus is simpler than developing the system from scratch for sessions.

We conclude that session types theory is indeed derivable from the theory of standard typed π-calculus. This does not mean that we believe session types are useless: on the contrary, due to their simple and intuitive structure they represent a fine tool for describing and reasoning about communication protocols in distributed scenarios. Our aim is to provide a methodology for facilitating the definition of session types and their extensions, hence encouraging their study.

References

[1] Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. Behavioral polymorphism and parametricity in session-based communication. In *ESOP*, volume 7792 of *LNCS*, pages 330–349. Springer, 2013.

[2] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, volume 6269 of *LNCS*, pages 222–236. Springer, 2010.

[3] Sara Capecchi, Mario Coppo, Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, and Elena Giachino. Amalgamating sessions and methods in object-oriented languages with generics. *Theor. Comput. Sci.*, 410(2-3):142–167, 2009.

[4] Marco Carbone, Ornela Dardha, and Fabrizio Montesi. Progress as compositional lock-freedom. In *COORDINATION*, volume 8459 of *LNCS*, pages 49–64. Springer, 2014.

[5] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In *ESOP*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007.

[6] Ornela Dardha. Recursive session types revisited. In *BEAT*, volume 162 of *EPTCS*, pages 27–34, 2014.

[7] Ornela Dardha. *Type Systems for Distributed Programs: Components and Sessions*, volume 7 of *Atlantis Studies in Computing*. Atlantis Press, July 2016.

[8] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *PPDP*, pages 139–150, New York, NY, USA, 2012. ACM.

[9] Romain Demangeon and Kohei Honda. Full abstraction in a subtyped pi-calculus with linear types. In *CONCUR*, volume 6901 of *LNCS*, pages 280–296. Springer, 2011.

[10] Mariangiola Dezani-Ciancaglini, Elena Giachino, Sophia Drossopoulou, and Nobuko Yoshida. Bounded session types for object oriented languages. In *FMCO*, volume 4709 of *LNCS*, pages 207–245. Springer, 2007.

[11] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session types for object-oriented languages. In *ECOOP*, volume 4067 of *LNCS*, pages 328–352. Springer, 2006.

[12] Simon J. Gay. Bounded polymorphism in session types. *Mathematical Structures in Computer Science*, 18(5):895–930, 2008.

[13] Simon J. Gay, Nils Gesbert, and António Ravara. Session types as generic process types. In *EXPRESS / SOS*, volume 160 of *EPTCS*, pages 94–110, 2014.

[14] Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Inf.*, 42(2-3):191–225, 2005.

[15] Kohei Honda. Types for dyadic interaction. In *CONCUR*, volume 715 of *LNCS*, pages 509–523. Springer, 1993.

[16] Kohei Honda, Vasco Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.

[17] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. In *POPL*, volume 36(3), pages 128–141, New York, NY, USA, 2001. ACM Press.

[18] Naoki Kobayashi. A type system for lock-free processes. *Inf. Comput.*, 177(2):122–159, 2002.

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

[20] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.*, 21(5):914–947, 1999.

[21] Fabrizio Montesi and Nobuko Yoshida. Compositional choreographies. In *CONCUR*, volume 8052 of *LNCS*, pages 425–439. Springer, 2013.

[22] Dimitris Mostrous and Nobuko Yoshida. Two session typing systems for higher-order mobile processes. In *TLCA*, volume 4583 of *LNCS*, pages 321–335. Springer, 2007.

[23] Luca Padovani. Deadlock and Lock Freedom in the Linear π-Calculus. In *CSL-LICS*, pages 72:1–72:10. ACM, 2014.

[24] Luca Padovani. Type reconstruction for the linear π-calculus with composite regular types. *LMCS*, 11(4), 2015.

[25] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *LICS*, pages 376–385. IEEE Computer Society, 1993.

[26] Davide Sangiorgi. An interpretation of typed objects into typed pi-calculus. *Inf. Comput.*, 143(1):34–73, 1998.

[27] Davide Sangiorgi and David Walker. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press, 2001.

[28] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *PARLE*, volume 817 of *LNCS*, pages 398–413. Springer, 1994.

[29] Antonio Vallecillo, Vasco Thudichum Vasconcelos, and António Ravara. Typing the behavior of software components using session types. *Fundam. Inform.*, 73(4):583–598, 2006.

[30] Vasco T. Vasconcelos. Fundamentals of session types. *Information Computation*, 217:52–70, 2012.

[31] Vasco Thudichum Vasconcelos, Simon J. Gay, and António Ravara. Type checking a multithreaded functional language with session types. *Theor. Comput. Sci.*, 368(1-2):64–87, 2006.

[32] Philip Wadler. Propositions as sessions. In *ICFP*, pages 273–286. ACM, 2012.

[33] Nobuko Yoshida and Vasco Thudichum Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electr. Notes Theor. Comput. Sci.*, 171(4):73–93, 2007.

Appendix A. Proofs for Subtyping

We start with a lemma relating subtyping and duality of session types, stating that two encoded session types are in a subtyping relation if the encoded dual types are in an inverse subtyping relation.

Lemma 31 (Subtyping on dual types)**.** If $[\![T]\!] \leq [\![T']\!]$, then $[\![\overline{T'}]\!] \leq [\![\overline{T}]\!]$.

Proof. The lemma follows immediately by the definition of encoding, the duality function for session types and the subtyping rules for standard π-calculus types presented in Fig. 11. □

We now present the proofs of the correctness of the encoding of types with respect to subtyping.

Proof of Theorem 25. If $[\![T]\!] \leq [\![T']\!]$, then $T <: T'$.

Proof. The proof is done by induction on the structure of session types T, T'. We present some of the cases.

- Case $T = T' = \mathtt{end}$:
  By (E-End) we have $[\![T]\!] = [\![T']\!] = \varnothing[\,]$. By rule (S$\pi$-Refl) we have $\varnothing[\,] \leq \varnothing[\,]$. By applying rule (S-End) we obtain the result.

- Case $T = ?T_1.S_1$ and $T' = ?T_2.S_2$:
  Assume that $[\![?T_1.S_1]\!] \leq [\![?T_2.S_2]\!]$, which by the encoding of input means that $\ell_{\mathtt{i}}[[\![T_1]\!], [\![S_1]\!]] \leq \ell_{\mathtt{i}}[[\![T_2]\!], [\![S_2]\!]]$. The last rule applied is (Sπ-ii), which by its premise asserts that $[\![T_1]\!] \leq [\![T_2]\!]$ and $[\![S_1]\!] \leq [\![S_2]\!]$. By induction hypothesis we have $T_1 <: T_2$ and $S_1 <: S_2$. By applying rule (S-Inp) on the induction hypothesis we obtain $?T_1.S_1 <: ?T_2.S_2$.

- Case $T = !T_1.S_1$ and $T' = !T_2.S_2$:
  Assume that $[\![!T_1.S_1]\!] \leq [\![!T_2.S_2]\!]$, which by encoding of output means that $\ell_{\mathtt{o}}[[\![T_1]\!], [\![\overline{S_1}]\!]] \leq \ell_{\mathtt{o}}[[\![T_2]\!], [\![\overline{S_2}]\!]]$. The last rule applied is (Sπ-oo), which by its premise asserts that $[\![T_2]\!] \leq [\![T_1]\!]$ and $[\![\overline{S_2}]\!] \leq [\![\overline{S_1}]\!]$. By Lemma 31, $[\![S_1]\!] \leq [\![S_2]\!]$. By induction hypothesis we have $T_2 <: T_1$ and $S_1 <: S_2$. By applying rule (S-Out) we obtain $!T_1.S_1 <: !T_2.S_2$.

□

Proof of Theorem 26. If $T <: T'$ then $[\![T]\!] \leq [\![T']\!]$.

Proof. The proof is by induction on the last rule applied in the derivation of $T <: T'$. We present some of the cases.

- Case (S-Inp):
$$\frac{T <: T' \qquad S <: S'}{?T.S <: ?T'.S'}$$
By induction hypothesis we have $[\![T]\!] \leq [\![T']\!]$ and $[\![S]\!] \leq [\![S']\!]$. We need to prove that $[\![?T.S]\!] \leq [\![?T'.S']\!]$. By applying (E-Inp) we obtain $[\![?T.S]\!] = \ell_{\mathtt{i}}[[\![T]\!], [\![S]\!]]$ and $[\![?T'.S']\!] = \ell_{\mathtt{i}}[[\![T']\!], [\![S']\!]]$. By applying rule (S$\pi$-ii) on the induction hypothesis we obtain the result.

- Case (S-Out):
$$\frac{T' <: T \qquad S <: S'}{!T.S <: !T'.S'}$$
By induction hypothesis we have $[\![T']\!] \leq [\![T]\!]$ and $[\![S]\!] \leq [\![S']\!]$. We need to prove that $[\![!T.S]\!] \leq [\![!T'.S']\!]$. By applying (E-Out) we obtain $[\![!T.S]\!] = \ell_{\mathtt{o}}[[\![T]\!], [\![\overline{S}]\!]]$ and $[\![!T'.S']\!] = \ell_{\mathtt{o}}[[\![T']\!], [\![\overline{S'}]\!]]$. By Lemma 31 we obtain $[\![\overline{S'}]\!] \leq [\![\overline{S}]\!]$. By applying rule (S$\pi$-oo) on the induction hypothesis we obtain the result.

- Case (S-Brch):
$$\frac{I \subseteq J \qquad S_i <: S'_j \quad \forall i \in I}{\&\{l_i : S_i\}_{i\in I} <: \&\{l_j : S'_j\}_{j\in J}}$$
By induction hypothesis we have $[\![S_i]\!] \leq [\![S'_j]\!]$ for all $i \in I$. We need to prove that $[\![\&\{l_i : S_i\}_{i\in I}]\!] \leq [\![\&\{l_j : S'_j\}_{j\in J}]\!]$. By (E-Branch) we obtain $[\![\&\{l_i : S_i\}_{i\in I}]\!] = \ell_{\mathtt{i}}[\langle l_i \_ [\![S_i]\!]\rangle_{i\in I}]$ and $[\![\&\{l_j : S'_j\}_{j\in J}]\!] = \ell_{\mathtt{i}}[\langle l_j \_ [\![S'_j]\!]\rangle_{j\in J}]$. By applying rules (S$\pi$-Variant) and (S$\pi$-ii) on the induction hypothesis we obtain the result.

- Case (S-Sel):
$$\frac{I \supseteq J \qquad S_i <: S'_j \quad \forall j \in J}{\oplus\{l_i : S_i\}_{i\in I} <: \oplus\{l_j : S'_j\}_{j\in J}}$$
By induction hypothesis we have $[\![S_i]\!] \leq [\![S'_j]\!]$ for all $j \in J$. We need to prove that $[\![\oplus\{l_i : S_i\}_{i\in I}]\!] \leq [\![\oplus\{l_j : S'_j\}_{j\in J}]\!]$. By (E-Select) we obtain $[\![\oplus\{l_i : S_i\}_{i\in I}]\!] = \ell_{\mathtt{o}}[\langle l_i \_ [\![\overline{S_i}]\!]\rangle_{i\in I}]$ and $[\![\oplus\{l_j : S'_j\}_{j\in J}]\!] = \ell_{\mathtt{o}}[\langle l_j \_ [\![\overline{S'_j}]\!]\rangle_{j\in J}]$. By Lemma 31 we obtain $[\![\overline{S'_j}]\!] \leq [\![\overline{T}_j]\!]$ for all $j \in J$. By (Sπ-Variant) and (Sπ-oo) on the induction hypothesis we obtain the result.

□

Appendix B. Proofs for Polymorphism

We start with a lemma relating the encoding of types and substitution of a type for a type variable.

Lemma 32. Let T be a session type and let $T[T'/X]$ denote type T where type variable X is substituted by type T'. Then,

$$[\![T[T'/X]]\!] = [\![T]\!][[\![T']\!]/X]$$

Proof. It follows directly from the encoding of polymorphic session types into polymorphic linear types and the definition of type substitution. □

Appendix B.1. Parametric Polymorphism

Proof of Lemma 18 and Lemma 19 for Parametric Polymorphic Values.

1. If $[\![\Gamma\ ;\ \Delta]\!]_f \vdash [\![v]\!]_f : [\![T]\!]$ for some renaming function f for v, then $\Gamma\ ;\ \Delta \vdash v : T$.
2. If $\Gamma\ ;\ \Delta \vdash v : T$, then $[\![\Gamma\ ;\ \Delta]\!]_f \vdash [\![v]\!]_f : [\![T]\!]$ for some renaming function f for v.

Proof. We split the proof as follows.

1. The proof is done by induction on the structure of the value v.
   We consider only the case for polymorphic values, namely $v = \langle T'; v' \rangle$. By applying (E-PolyVal) we have $[\![\langle T'; v' \rangle]\!]_f = \langle [\![T']\!]; [\![v']\!]_f \rangle$ and assume $[\![\Gamma\ ;\ \Delta]\!]_f \vdash \langle [\![T']\!]; [\![v']\!]_f \rangle : \langle X; [\![T]\!] \rangle$, which means that the last typing rule applied must have been (Tπ-PolyVal).

   $$\frac{[\![\Gamma\ ;\ \Delta]\!]_f \vdash [\![v']\!]_f : [\![T]\!][[\![T']\!]/X]}{[\![\Gamma\ ;\ \Delta]\!]_f \vdash \langle [\![T']\!]; [\![v']\!]_f \rangle :\ \langle X; [\![T]\!] \rangle}$$

   By induction hypothesis and by Lemma 32 we have $\Gamma; \Delta \vdash v' : T[T'/X]$. We conclude by applying (T-PolyVal).
2. The proof is done by induction on the derivation for $\Gamma\ ;\ \Delta \vdash v : T$.
   We consider only the case for (T-PolyVal).

   $$\frac{\Gamma\ ;\ \Delta \vdash v' : T[T'/X]}{\Gamma\ ;\ \Delta \vdash \langle T'; v' \rangle : \langle X; T \rangle}$$

   By induction hypothesis and by Lemma 32, there is f' such that $[\![\Gamma\ ;\ \Delta]\!]_{f'} \vdash [\![v']\!]_{f'} : [\![T]\!][[\![T']\!]/X]$. By choosing $f = f'$ and by applying (Tπ-PolyVal), (E-PolyType) and (E-PolyVal), we obtain the result.

□

Proof of Theorem 20 and Theorem 21 for Parametric Polymorphic Processes.

1. If $[\![\Gamma;\Delta]\!]_f \vdash [\![Q]\!]_f$ for some renaming function f for Q, then $\Gamma;\Delta \vdash Q$.
2. If $\Gamma;\Delta \vdash Q$, then $[\![\Gamma;\Delta]\!]_f \vdash [\![Q]\!]_f$ for some renaming function f for Q.

Proof. We split the proof as follows.

1. The proof is done by induction on the structure of session process Q.
   We consider only the case for the unpack process. By (E-Unpack) we have that $[\![\Gamma\ ;\ \Delta]\!]_f \vdash \mathbf{open}\ [\![v]\!]_f\ \mathbf{as}\ (X; f_x)\ \mathbf{in}\ [\![P]\!]_f$. This means that the last rule applied must be (Tπ-Unpack):

$$\frac{[\![\Gamma]\!]_f\ ;\ \Delta \vdash [\![v]\!]_f : \langle X; [\![T]\!]\rangle \qquad [\![\Gamma]\!]_f, f_x : [\![T]\!]\ ;\ \Delta, X \vdash [\![P]\!]_f}{[\![\Gamma]\!]_f\ ;\ \Delta \vdash \mathbf{open}\ [\![v]\!]_f\ \mathbf{as}\ (X; f_x)\ \mathbf{in}\ [\![P]\!]_f}$$

   By soundness of the encoding with respect to typing parametric polymorphic values we have $\Gamma\ ;\ \Delta \vdash v : \langle X; T\rangle$. By induction hypothesis we have $\Gamma, x : T\ ;\ \Delta, X \vdash P$. We conclude by applying (T-Unpack).
2. The proof is done by induction on the derivation $\Gamma\ ;\ \Delta \vdash Q$.
   We consider only the case when (T-Unpack) is applied:

$$\frac{\Gamma_1\ ;\ \Delta \vdash v : \langle X; T\rangle \qquad \Gamma_2, x : T\ ;\ \Delta, X \vdash P}{\Gamma_1 \circ \Gamma_2\ ;\ \Delta \vdash \mathbf{open}\ v\ \mathbf{as}\ (X; x)\ \mathbf{in}\ P}$$

   By completeness of the encoding with respect to typing parametric polymorphic values we have $[\![\Gamma]\!]_{f'}\ ;\ \Delta \vdash [\![v]\!]_{f'} : \langle X; [\![T]\!]\rangle$, for some function f'. By induction hypothesis we have $[\![\Gamma, x : T]\!]_{f''}\ ;\ \Delta, X \vdash [\![P]\!]_{f''}$, for some function f''. By (E-Gamma) it means $[\![\Gamma]\!]_{f''} \uplus f''_x : [\![T]\!]\ ;\ \Delta, X \vdash [\![P]\!]_{f''}$. Since $\Gamma_1 \circ \Gamma_2$ is defined, then for all $x \in \mathtt{dom}(\Gamma_1) \cap \mathtt{dom}(\Gamma_2)$ it holds that $\Gamma_1(x) = \Gamma_2(x) = T$ and $\mathtt{un}(T)$. Let $\mathtt{dom}(\Gamma_1) \cap \mathtt{dom}(\Gamma_2) = D$ and define $f'_D = f' \setminus \bigcup_{d \in D}\{d \mapsto f'(d)\}$ and $f''_D = f'' \setminus \bigcup_{d \in D}\{d \mapsto f''(d)\}$. Let $f = \bigcup_{d \in D}\{d \mapsto d'\} \cup f'_D \cup f''_D$, such that for all $d \in D$ we create a fresh name d' and associate $d \mapsto d'$. Moreover, f is a function since its subcomponents act on disjoint domains. By Lemma 6 and since $x \notin \Gamma_2$, by Lemma 14 we have the following:

$$[\![\Gamma]\!]_f\ ;\ \Delta \vdash [\![v]\!]_f : \langle X; [\![T]\!]\rangle \qquad\qquad [\![\Gamma]\!]_f, f_x : [\![T]\!]\ ;\ \Delta, X \vdash [\![P]\!]_f$$

   By applying (E-Unpack) and rule (Tπ-Unpack) we obtain the result. □

Appendix B.2. Bounded Polymorphism

We present the proofs of the main results for bounded polymorphism. We present only some of the cases. We begin with the correctness of the encoding with respect to typing bounded polymorphic processes.

Proof of Theorem 20 for Bounded Polymorphic Processes. If $[\![\Gamma;\Delta]\!]_f \vdash [\![Q]\!]_f$ for some renaming function f for Q, then $\Gamma;\Delta \vdash Q$.

Proof. The proof is done by induction on the structure of session process Q. We present the case for selection.

- Case $Q = x \lhd l_j(B).P$:
  By (E-BPolySelection) $[\![x \lhd l_j(B).P]\!]_f = (\boldsymbol{\nu} c) f_x!\langle l_j(B)\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}$ and assume $[\![\Gamma]\!]_f;\Delta \vdash (\boldsymbol{\nu} c) f_x!\langle l_j(B)\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}$. Since c is a restricted channel name in the encoding of Q, then either rule (Tπ-Res1) or (Tπ-Res2) must be applied. We consider only the case when rule (Tπ-Res1) is applied, as the one for (Tπ-Res2) is symmetrical. Then, by (Tπ-Res1) and (Tπ-Out) we have the derivation:

$$\dfrac{\dfrac{\begin{array}{c}(\text{T}\pi\text{-Res1})\\(\text{T}\pi\text{-Out})\end{array}\quad \begin{array}{c}\Gamma_1^\pi;\Delta \vdash f_x : \ell_\circ[\langle l_i(X_i \le B_i)\_T_i^\pi\rangle_{i\in I}]\\ \Gamma_2^\pi, c : \overline{T_j^\pi}[B/X_j];\Delta \vdash [\![P]\!]_{f,\{x\mapsto c\}}\\ c : T_j^\pi[B/X_j];\Delta \vdash l_j(B)\_c : \langle l_i(X_i \le B_i)\_T_i^\pi\rangle_{i\in I}\end{array}}{[\![\Gamma]\!]_f, c : \ell_\sharp[W][B/X_j];\Delta \vdash f_x!\langle l_j(B)\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}}}{[\![\Gamma]\!]_f;\Delta \vdash (\boldsymbol{\nu} c) f_x!\langle l_j(B)\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}}$$

  and $[\![\Gamma]\!]_f = \Gamma_1^\pi \uplus \Gamma_2^\pi$. By Lemma 16 $\Gamma_1^\pi = [\![\Gamma_1]\!]_f$, and $\Gamma_2^\pi = [\![\Gamma_2]\!]_f$ such that $\Gamma = \Gamma_1 \circ \Gamma_2$. By applying (T$\pi$-Var) and (T$\pi$-BPolyLval) for some $j \in I$, we have the derivation:

$$\dfrac{\dfrac{\begin{array}{c}(\text{T}\pi\text{-BPolyLVal})\\(\text{T}\pi\text{-Var})\end{array}\quad \dfrac{}{c : T_j^\pi[B/X_j];\Delta \vdash c : T_j^\pi[B/X_j]}}{c : T_j^\pi[B/X_j];\Delta \vdash l_j(B)\_c : \langle l_i(X_i \le B_i)\_T_i^\pi\rangle_{i\in I}}\quad B \le B_i \quad \forall i \in I$$

  Name c has type $\ell_\sharp[W][B/X_j]$, which is $T_j^\pi[B/X_j] \uplus \overline{T_j^\pi}[B/X_j]$. One capability of c is sent along $l_j(B)\_c$, whereas the other one is used in the continuation $[\![P]\!]_{f,\{x\mapsto c\}}$. In the case where (T$\pi$-Res2) is applied, c is of type $\varnothing[\,] \uplus \varnothing[\,]$. The correctness of the encoding with respect to typing bounded polymorphic values implies $\Gamma_1;\Delta \vdash x : \oplus\{l_i(X_i \le B_i) : T_i\}_{i\in I}$, which by (E-BPolySel) means $[\![\oplus\{l_i(X_i \le B_i) : T_i\}_{i\in I}]\!] = \ell_\circ[\langle l_i(X_i \le B_i)\_[\![T_i^\pi]\!]\rangle_{i\in I}]$ and $T_i^\pi = [\![\overline{T_i}]\!]$ for all $i \in I$. By induction hypothesis we have that $\Gamma_2, x : T_j[B/X_j];\Delta \vdash P$. By Theorem 25 we obtain $B <: B_i$ for all $i \in I$. By applying rule (T-BPolySel) we conclude $\Gamma_1 \circ \Gamma_2;\Delta \vdash x \lhd l_j(B).P$. □

Proof of Theorem 21 for Bounded Polymorphic Processes. If $\Gamma;\Delta \vdash Q$, then $[\![\Gamma;\Delta]\!]_f \vdash [\![Q]\!]_f$ for some renaming function f for Q.

Proof. The proof is done by induction on the derivation $\Gamma;\Delta \vdash Q$. We examine only the case where (T-BPolyBrch) is applied.

- Case (T-BPolyBrch):

$$\frac{\Gamma_1;\Delta \vdash x : \&\{l_i(X_i \leq B_i) : T_i\}_{i\in I} \qquad \Gamma_2, x : T_i; \Delta, X_i <: B_i \vdash P_i \quad \forall i \in I}{\Gamma_1 \circ \Gamma_2; \Delta \vdash x \rhd \{l_i(X_i \leq B_i) : P_i\}_{i\in I}} \text{(T-BPolyBrch)}$$

By correctness of the encoding with respect to typing bounded polymorphic values, we have that $[\![\Gamma_1]\!]_{f'}; \Delta \vdash f'_x : \ell_{\mathtt{i}}[\langle l_i(X_i \leq B_i)\_[\![T_i]\!]\rangle_{i\in I}]$ for some function f'. By induction hypothesis, (E-Gamma) and Theorem 26 we have that $[\![\Gamma_2]\!]_{f''} \uplus f''_x : [\![T_i]\!]; \Delta, X_i \leq B_i \vdash [\![P_i]\!]_{f''}$ for all $i \in I$ and for some function f''. Since $\Gamma_1 \circ \Gamma_2$ is defined, it means that for all $x \in \mathtt{dom}(\Gamma_1) \cap \mathtt{dom}(\Gamma_2)$ it holds that $\Gamma_1(x) = \Gamma_2(x) = T$ and $\mathtt{un}(T)$. Let $\mathtt{dom}(\Gamma_1) \cap \mathtt{dom}(\Gamma_2) = D$. Then, we define $f'_D = f' \setminus \bigcup_{d\in D}\{d \mapsto f'(d)\}$ and $f''_D = f'' \setminus \bigcup_{d\in D}\{d \mapsto f''(d)\}$. Suppose $f''(x) = c$. Then, let $f = \bigcup_{d\in D}\{d \mapsto d'\} \cup f'_D \cup f''_D \setminus \{x \mapsto c\}$, where for all $d \in D$ we create a fresh name d' and associate $d \mapsto d'$. Moreover, f is a function since its subcomponents act on disjoint domains. We now have:

$$[\![\Gamma_1]\!]_f; \Delta \vdash f_x : \ell_{\mathtt{i}}[\langle l_i(X_i \leq B_i)\_[\![T_i]\!]\rangle_{i\in I}]$$

and for all $i \in I$,

$$[\![\Gamma_2]\!]_f \uplus c : [\![T_i]\!]; \Delta, X_i \leq B_i \vdash [\![P_i]\!]_{f,\{x\mapsto c\}}$$

Since $x \notin \mathtt{dom}(\Gamma_2)$, then $[\![\Gamma_2, x : T_j]\!]_{f,\{x\mapsto c\}}$ can be optimised and distributed as $[\![\Gamma_2]\!]_f \uplus c : [\![T_j]\!]$. By (T$\pi$-Var), used to derive $y : \langle l_i(X_i \leq B_i)\_[\![T_i]\!]\rangle_{i\in I}$, (T$\pi$-BPolyCase) and Lemma 14 we have the derivation:

$$\frac{\dfrac{}{y : \langle l_i(X_i \leq B_i)\_[\![T_i]\!]\rangle_{i\in I}; \Delta \vdash y : \langle l_i(X_i \leq B_i)\_[\![T_i]\!]\rangle_{i\in I}} \text{(T}\pi\text{-Var)} \qquad [\![\Gamma_2]\!]_f, c : [\![T_i]\!]; \Delta, X_i \leq B_i \vdash [\![P_i]\!]_{f,\{x\mapsto c\}} \quad \forall i \in I}{[\![\Gamma_2]\!]_f, y : \langle l_i(X_i \leq B_i)\_[\![T_i]\!]\rangle_{i\in I}; \Delta \vdash \mathbf{case}\, y\, \mathbf{of}\, \{l_i(X_i \leq B_i)\_c \rhd [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}} \text{(T}\pi\text{-BPolyCase)}$$

Then, by applying (Tπ-Inp) we conclude as follows:

$$\frac{[\![\Gamma_1]\!]_f; \Delta \vdash f_x : \ell_{\mathtt{i}}[\langle l_i(X_i \leq B_i)\_[\![T_i]\!]\rangle_{i\in I}] \qquad [\![\Gamma_2]\!]_f, y : \langle l_i(X_i \leq B_i)\_[\![T_i]\!]\rangle_{i\in I}; \Delta \vdash \mathbf{case}\, y\, \mathbf{of}\, \{l_i\_(c(X_i \leq B_i)) \rhd [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}}{[\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f; \Delta \vdash f_x?(y).\ \mathbf{case}\, y\, \mathbf{of}\, \{l_i(X_i \leq B_i)\_c \rhd [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}}$$

□

We now prove the operational correspondence for bounded polymorphic processes.

Proof of Theorem 24 for Bounded Polymorphic Processes. Let P be a session process, Γ, Δ session typing contexts, and f a renaming function for P such that $[\![\Gamma;\Delta]\!]_f \vdash [\![P]\!]_f$. Then, the following statements hold.

1. If $P \to P'$, then $[\![P]\!]_f \to\!\!\!\!\to [\![P']\!]_f$.
2. If $[\![P]\!]_f \to Q$, then there is a session process P' such that
   - either $P \to P'$;
   - or there are x, y such that $(\boldsymbol{\nu} xy)P \to P'$

   and $Q \hookrightarrow [\![P']\!]_f$.

Proof. Since $[\![\Gamma;\Delta]\!]_f \vdash [\![P]\!]_f$, then by Theorem 20 for bounded polymorphic processes, given earlier in this section, it is the case that $\Gamma;\Delta \vdash P$.

1. We consider only the case where rule (R-BPolySel) is applied.

   $$P \triangleq (\boldsymbol{\nu} xy)(x \lhd l_j(B).Q \mid y \rhd \{l_i(X_i \leq B_i) : P_i\}_{i\in I}) \to (\boldsymbol{\nu} xy)(Q \mid P_j[B/X_j]) \triangleq P'$$

   where $j \in I$. By the encoding of bounded polymorphic processes we have

   $$\begin{array}{rcl}
   [\![P]\!]_f & = & [\![(\boldsymbol{\nu} xy)(x \lhd l_j(B).Q \mid y \rhd \{l_i(X_i \leq B_i) : P_i\}_{i\in I})]\!]_f \\
   & = & (\boldsymbol{\nu} c)\left([\![x \lhd l_j(B).Q]\!]_{f,\{x,y\mapsto c\}} \mid [\![y \rhd \{l_i(X_i \leq B_i) : P_i\}_{i\in I}]\!]_{f,\{x,y\mapsto c\}}\right) \\
   & = & (\boldsymbol{\nu} c)\left((\boldsymbol{\nu} c')\big(c!\langle l_j(B)_c'\rangle.[\![Q]\!]_{f,\{x,y\mapsto c,x\mapsto c'\}}\big) \mid \right. \\
   & & \qquad\qquad \left. c?(z).\ \mathbf{case}\, z\, \mathbf{of}\, \{l_i(X_i \leq B_i)_c' \rhd [\![P_i]\!]_{f,\{x,y\mapsto c,y\mapsto c'\}}\}_{i\in I}\right) \\
   & \to & (\boldsymbol{\nu} c)\left((\boldsymbol{\nu} c')\big([\![Q]\!]_{f,\{x\mapsto c,c\mapsto c'\}} \mid \right. \\
   & & \qquad\qquad \left. \mathbf{case}\, l_j(B)_c'\, \mathbf{of}\, \{l_i(X_i \leq B_i)_c' \rhd [\![P_i]\!]_{f,\{y\mapsto c,c\mapsto c'\}}\}_{i\in I}\big)\right) \\
   & \to & (\boldsymbol{\nu} c)\left((\boldsymbol{\nu} c')\big([\![Q]\!]_{f,\{x,y\mapsto c,x\mapsto c'\}} \mid [\![P_j]\!]_{f,\{x,y\mapsto c,y\mapsto c'\}}[B/X_j]\big)\right) \\
   & \equiv & (\boldsymbol{\nu} c')\big([\![Q]\!]_{f,\{x,y\mapsto c,x\mapsto c'\}} \mid [\![P_j]\!]_{f,\{x,y\mapsto c,y\mapsto c'\}}[B/X_j]\big)
   \end{array}$$

   Notice that since P is a well-typed session process, it means that for all $i \in I$, $x \notin \mathtt{fn}(P_i)$ and $y \notin \mathtt{fn}(Q)$. Then, function $f, \{x, y \mapsto c, x \mapsto c'\}$ and function $f, \{x, y \mapsto c, y \mapsto c'\}$ can both be subsumed by $f, \{x, y \mapsto c'\}$. We can rewrite the above as:

   $$(\boldsymbol{\nu} c')\big([\![Q]\!]_{f,\{x,y\mapsto c'\}} \mid [\![P_j]\!]_{f,\{x,y\mapsto c'\}}\big)$$

   On the other hand we have:

   $$\begin{array}{rcl}
   [\![P']\!]_f & = & [\![(\boldsymbol{\nu} xy)(Q \mid P_j[B/X_j])]\!]_f \\
   & = & (\boldsymbol{\nu} c')\big([\![Q]\!]_{f,\{x,y\mapsto c'\}} \mid [\![P_j]\!]_{f,\{x,y\mapsto c'\}}[B/X_j]\big)
   \end{array}$$

   We use Lemma 32 to obtain $[\![P_j]\!]_{f,\{x,y\mapsto c'\}}[B/X_j]$. The above implies:

   $$[\![P]\!]_f \to\!\!\!\!\to [\![P']\!]_f$$

2. Case $P = P_1 \mid P_2 = x \triangleleft l_j(B).P_1' \mid y \triangleright \{l_i(X_i \leq B_i) : P_i''\}_{i\in I}$. Following Theorem 24, we can obtain $R \equiv P$ such that R is of the form:

$$R = C[\, x \triangleleft l_j(B).P_1' \mid y \triangleright \{l_i(X_i \leq B_i) : P_i''\}_{i\in I}\,]$$

where $x \triangleleft l_j(B).P_1'$ and $y \triangleright \{l_i(X_i \leq B_i) : P_i''\}_{i\in I}$ are the session processes such that the corresponding encodings are the marked input and output standard π-calculus processes, and the hole of the context is at top level. Since $R \equiv P$, it is sufficient to prove the statement of the theorem for R in place of P, as any derivative of R is also a derivative of P. Moreover, by Lemma 23, we also have

$$[\![P]\!]_f \;\equiv\; [\![R]\!]_f$$

where $[\![R]\!]_f$ is of the form

$$[\![R]\!]_f = D[\, [\![x \triangleleft l_j(B).P_1']\!]_g \mid [\![y \triangleright \{l_i(X_i \leq B_i) : P_i''\}_{i\in I}]\!]_g\,] \tag{B.1}$$

for some context $D[\cdot]$ with a top-level hole, and some renaming function g with $g(x) = g(y)$. Now, expanding the definition of the encoding, for some fresh name c, and using brackets for the marked prefixes, we can continue from (B.1) as follows:

$$\begin{aligned}
&= D[\, (\boldsymbol{\nu} c)\underline{g_x!\langle l_j(B)_c\rangle}.[\![P_1']\!]_{g,\{x\mapsto c\}} \mid \\
&\qquad \underline{g_y?(z)}.\ \mathbf{case}\, z\, \mathbf{of}\, \{l_i(X_i \leq B_i)_c \triangleright [\![P_i'']\!]_{g,\{y\mapsto c\}}\}_{i\in I}\,] \\
&\rightarrow D[\, (\boldsymbol{\nu} c)([\![P_1']\!]_{g,\{x\mapsto c\}} \mid \\
&\qquad \mathbf{case}\, l_j(B)_c\, \mathbf{of}\, \{l_i(X_i \leq B_i)_c \triangleright [\![P_i'']\!]_{g,\{y\mapsto c\}}\}_{i\in I}[B/X_j])\,] \\
&\triangleq Q'
\end{aligned} \tag{B.2}$$

Appealing to Lemma 11, we know that such a reduction, having consumed the two marked prefixes yields the following equivalence:

$$Q' \equiv Q$$

We will now find the derivative P' of the assertion of theorem (as a derivative of R) and prove

$$Q' \hookrightarrow [\![P']\!]_f$$

We will consider only the case where x and y are restricted session endpoints, namely co-names. Since $(\boldsymbol{\nu} xy)$ appears in the context $C[\cdot]$, its encoded context $D[\cdot]$ contains a restriction $(\boldsymbol{\nu} c')$ where c' is the linear name $g(x)$ (and also $g(y)$). The reduction in (B.2) is along the name c', which, being linear, after the reduction does not occur anymore in the continuation process. We can therefore remove such a restriction and simply

replace it with the restriction at c. If $E[\cdot]$ if the context with c' replaced by c, we therefore have the following:

$$\begin{array}{rcl} Q' & \equiv & E[\,[\![P_1']\!]_{g,\{x\mapsto c\}} \mid \\ & & \quad \mathbf{case}\, l_j(B)\_c\, \mathbf{of}\, \{l_i(X_i \leq B_i)\_c \rhd [\![P_i'']\!]_{g,\{y\mapsto c\}}\}_{i\in I}[B/X_j]\,] \\ & \to & E[\,[\![P_1']\!]_{g,\{x\mapsto c\}} \mid [\![P_j'']\!]_{g,\{y\mapsto c\}}[B/X_j]\,] \mid \\ & \triangleq & Q'' \end{array}$$

Moreover, since $(\boldsymbol{\nu} xy)$ is in the context $C[\cdot]$, we can infer the following reduction:

$$\begin{array}{rcl} R & = & C[\, x \lhd l_j(B).P_1' \mid y \rhd \{l_i(X_i \leq B_i) : P_i''\}_{i\in I}\,] \\ & \to & C[\, P_1' \mid P_j''[B/X_j]\,] \triangleq P' \end{array}$$

We can easily show that $Q'' = [\![P']\!]_f$ by Lemma 22 and by renaming c' into c, namely by using context $E[\cdot]$. □

Appendix C. Proofs for the HOπ

Lemma 33. Let $\mathcal{S}_1, \ldots, \mathcal{S}_n$ be sets of linear functional variables such that their union is defined. Then,

$$[\![\mathcal{S}_1 \cup \ldots \cup \mathcal{S}_n]\!]_f = [\![\mathcal{S}_1]\!]_f \cup \ldots \cup [\![\mathcal{S}_n]\!]_f$$

for some renaming function for $\mathcal{S}_1 \cup \ldots \cup \mathcal{S}_n$.

Proof. The proof follows immediately by applying any renaming function on the disjoint union of sets of linear session functional variables. □

Lemma 34.

- Let $\mathcal{S}$ be a set of linear functional variables and f a renaming function for $\mathcal{S}$ and $[\![\mathcal{S}]\!]_f = \mathcal{S}_1^\pi \cup \ldots \cup \mathcal{S}_n^\pi$. Then, $\mathcal{S} = \mathcal{S}_1 \cup \ldots \cup \mathcal{S}_n$ and for all $i \in 1 \ldots n$, $\mathcal{S}_i^\pi = [\![\mathcal{S}_i]\!]_f$.
- Let $\mathcal{S}^\pi = [\![\mathcal{S}_1]\!]_f \cup \ldots \cup [\![\mathcal{S}_n]\!]_f$ and f a renaming function for all $\mathcal{S}_i$ for $i \in 1 \ldots n$. Then, $\mathcal{S} = \mathcal{S}_1 \cup \ldots \cup \mathcal{S}_n$ and $\mathcal{S}^\pi = [\![\mathcal{S}]\!]_f$.

Proof. The proof follows immediately from the definition of the encoding of $\mathcal{S}$ and the disjoint union of subsets of $\mathcal{S}$. □

Lemma 35 (Substitution Lemma for Linear HOπ-Calculus)**.** Let P be a standard HOπ process.

- If $\Phi\ ;\ \Gamma, x : \tau\ ;\ \mathcal{S} \vdash P$ or
- If $\Phi, x : \tau\ ;\ \Gamma\ ;\ \mathcal{S} \vdash P$ or

- If $\Phi, x : \tau \ ; \ \Gamma \ ; \ \mathcal{S}, \{x\} \vdash P$ and

$\Phi' \ ; \ \Gamma' \ ; \ \mathcal{S}' \vdash v : \tau$ and Φ, Φ', and $\Gamma \uplus \Gamma'$ and $\mathcal{S}, \mathcal{S}'$ are defined, then it holds $\Phi, \Phi' \ ; \ \Gamma \uplus \Gamma' \ ; \ \mathcal{S}, \mathcal{S}' \vdash P[v/x]$.

Proof. Immediate generalisation of Lemma 6. □

We now give the proofs of soundness and completeness of the encoding of higher-order terms with respect to typing.

Proof of Theorem 29. If $[\![\Phi \ ; \ \Gamma \ ; \ \mathcal{S}]\!]_f \vdash [\![P]\!]_f : [\![\sigma]\!]$ for some renaming function f for P, then $\Phi \ ; \ \Gamma \ ; \ \mathcal{S} \vdash P : \sigma$.

Proof. The proof is by induction on the structure of P.

- Case $\lambda x : T.P$:
  By applying (E-Abstraction) and (E-FunType) we have

$$[\![\lambda x : T.P]\!]_f = \lambda x : [\![T]\!].[\![P]\!]_f$$

  and $[\![T \to \sigma]\!] = [\![T]\!] \to [\![\sigma]\!]$. Since x is bound with scope P, then $f_x = x$. Assume

$$[\![\Phi]\!]_f ; [\![\Gamma]\!]_f ; [\![\mathcal{S}]\!]_f \vdash \lambda x : [\![T]\!].[\![P]\!]_f : [\![T]\!] \to [\![\sigma]\!]$$

  This implies that either rule (Tπ-HoAbs1) or rule (Tπ-HoAbs2) is applied. We consider both cases in the following:

  - Rule (Tπ-HoAbs1) is applied:

$$\frac{[\![\Phi]\!]_f, x : [\![T]\!] ; [\![\Gamma]\!]_f ; \mathcal{S}_1^\pi \vdash [\![P]\!]_f : [\![\sigma]\!] \qquad \text{if } [\![T]\!] = T_1^\pi \xrightarrow{1} \sigma_1^\pi \text{ then } x \in \mathcal{S}_1^\pi}{[\![\Phi]\!]_f ; [\![\Gamma]\!]_f ; \mathcal{S}_1^\pi - \{x\} \vdash \lambda x : [\![T]\!].[\![P]\!]_f : [\![T]\!] \to [\![\sigma]\!]}$$

    where $[\![\mathcal{S}]\!]_f = \mathcal{S}_1^\pi - \{x\}$, which implies $\mathcal{S}_1^\pi = [\![\mathcal{S}]\!]_f \cup \{x\}$. By Lemma 34 we have $[\![\mathcal{S}_1]\!]_f = \mathcal{S}_1^\pi$ and thus $\mathcal{S} = \mathcal{S}_1 - x$. By induction hypothesis we have $\Phi, x : T ; \Gamma ; \mathcal{S}_1 \vdash P : \sigma$. We conclude by (T-HoAbs1).
  - Rule (Tπ-HoAbs2) is applied:

$$\frac{[\![\Phi]\!]_f ; [\![\Gamma]\!]_f, x : [\![T]\!] ; [\![\mathcal{S}]\!]_f \vdash [\![P]\!]_f : [\![\sigma]\!]}{[\![\Phi]\!]_f ; [\![\Gamma]\!]_f ; [\![\mathcal{S}]\!]_f \vdash \lambda x : [\![T]\!].[\![P]\!]_f : [\![T]\!] \to [\![\sigma]\!]}$$

    By induction hypothesis $\Phi ; \Gamma, x : T ; \mathcal{S} \vdash P : \sigma$. Then, we obtain the result by applying rule (T-HoAbs1).

- Case PQ:
  By (E-Application) we have $[\![PQ]\!]_f = [\![P]\!]_f [\![Q]\!]_f$ and assume

$$[\![\Phi]\!]_f ; [\![\Gamma]\!]_f ; [\![\mathcal{S}]\!]_f \vdash [\![P]\!]_f [\![Q]\!]_f : [\![\sigma]\!]$$

Then, rule (Tπ-HoApp) is applied:

$$\frac{\llbracket\Phi\rrbracket_f;\Gamma_1^\pi;\mathcal{S}_1^\pi \vdash \llbracket P\rrbracket_f : T^\pi \xrightarrow{1} \llbracket\sigma\rrbracket \quad \llbracket\Phi\rrbracket_f;\Gamma_2^\pi;\mathcal{S}_2^\pi \vdash \llbracket Q\rrbracket_f : T^\pi \quad \text{if } T^\pi = T_1^\pi \to \sigma_1^\pi \text{ then } \mathtt{un}(\Gamma_2^\pi) \text{ and } \mathcal{S}_2^\pi = \varnothing}{\llbracket\Phi\rrbracket_f;\Gamma_1^\pi \uplus \Gamma_2^\pi;\mathcal{S}_1^\pi \cup \mathcal{S}_2^\pi \vdash \llbracket P\rrbracket_f\llbracket Q\rrbracket_f : \llbracket\sigma\rrbracket}$$

We have $\llbracket\Gamma\rrbracket_f = \Gamma_1^\pi \uplus \Gamma_2^\pi$ and $\llbracket\mathcal{S}\rrbracket_f = \mathcal{S}_1^\pi \cup \mathcal{S}_2^\pi$. By Lemma 16 we have $\Gamma_1^\pi = \llbracket\Gamma_1\rrbracket_f$ and $\Gamma_2^\pi = \llbracket\Gamma_2\rrbracket_f$, such that $\Gamma = \Gamma_1 \circ \Gamma_2$. By Lemma 34 we have $\llbracket\mathcal{S}_1\rrbracket_f = \mathcal{S}_1^\pi$ and $\llbracket\mathcal{S}_2\rrbracket_f = \mathcal{S}_2^\pi$ such that $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$. By induction hypothesis $\Phi;\Gamma_1;\mathcal{S}_1 \vdash P : T \xrightarrow{1} \sigma$ where $T^\pi = \llbracket T\rrbracket$, and $\Phi;\Gamma_2;\mathcal{S}_2 \vdash Q : T$. Then, the result follows immediately by applying rule (T-HoApp) on the induction hypothesis.

□

Proof of Theorem 30. If $\Phi;\Gamma;\mathcal{S} \vdash P : \sigma$, then $\llbracket\Phi;\Gamma;\mathcal{S}\rrbracket_f \vdash \llbracket P\rrbracket_f : \llbracket\sigma\rrbracket$ for some renaming function f for P.

Proof. The proof is by induction on the derivation $\Phi;\Gamma;\mathcal{S} \vdash P : \sigma$, by analysing the last typing rule applied.

- Case (T-HoFun):

$$\frac{\mathtt{un}(\Gamma)}{\Phi, x : T \xrightarrow{1} \sigma;\Gamma;\{x\} \vdash x : T \xrightarrow{1} \sigma}$$

We need to prove that $\llbracket\Phi\rrbracket_f, f_x : \llbracket T \xrightarrow{1} \sigma\rrbracket;\llbracket\Gamma\rrbracket_f;\{f_x\} \vdash f_x : \llbracket T \xrightarrow{1} \sigma\rrbracket$. By Proposition 12 we obtain $\mathtt{un}(\llbracket\Gamma\rrbracket_f)$. By (E-LinFunType) and by applying rule (Tπ-HoFun) we conclude the case.

- Case (T-HoAbs1):

$$\frac{\Phi, x : T;\Gamma;\mathcal{S} \vdash P : \sigma \quad \text{if } T = T' \xrightarrow{1} \sigma \text{ then } x \in \mathcal{S}}{\Phi;\Gamma;\mathcal{S} - \{x\} \vdash \lambda x : T.P : T \to \sigma}$$

By induction hypothesis $\llbracket\Phi\rrbracket_{f'}, f'_x : \llbracket T\rrbracket;\llbracket\Gamma\rrbracket_{f'};\llbracket\mathcal{S}\rrbracket_{f'} \vdash \llbracket P\rrbracket_{f'} : \llbracket\sigma\rrbracket$ for some renaming function f' for P. If $\llbracket T\rrbracket = \llbracket T' \xrightarrow{1} \sigma\rrbracket$, then $f'_x \in \llbracket\mathcal{S}\rrbracket_{f'}$. Since f' is a renaming function for P and $x \in \mathtt{fn}(P)$, then $x \notin \mathtt{dom}(\llbracket\Phi\rrbracket_{f'})$ and $x \notin \mathtt{dom}(\llbracket\Gamma\rrbracket_{f'})$. We distinguish two cases, according to the shape of type T. If $T \neq T' \xrightarrow{1} \sigma$, then also $\llbracket T\rrbracket \neq \llbracket T' \xrightarrow{1} \sigma\rrbracket$. By typing rule (T$\pi$-HoVar) we have $x : \llbracket T\rrbracket;\varnothing;\varnothing \vdash x : \llbracket T\rrbracket$. Otherwise, if $T = T' \xrightarrow{1} \sigma$, then also $\llbracket T\rrbracket = \llbracket T' \xrightarrow{1} \sigma\rrbracket$. By typing rule (T$\pi$-HoFun) we have $x : \llbracket T\rrbracket;\varnothing;\{x\} \vdash x : \llbracket T\rrbracket$. Then, $\llbracket\Phi\rrbracket_{f'}, x : \llbracket T\rrbracket;\llbracket\Gamma\rrbracket_{f'};\llbracket\mathcal{S}\rrbracket_{f'}[x/f'_x]$ is defined. By Lemma 35 $\llbracket\Phi\rrbracket_{f'}, x : \llbracket T\rrbracket;\llbracket\Gamma\rrbracket_{f'};\llbracket\mathcal{S}\rrbracket_{f'}[x/f'_x] \vdash \llbracket P\rrbracket_{f'}[x/f'_x] : \llbracket\sigma\rrbracket$. Let $f = f', \{x \mapsto x\}$. It holds that if $\llbracket T\rrbracket = \llbracket T' \xrightarrow{1} \sigma\rrbracket$ then $x \in \llbracket\mathcal{S}\rrbracket_f$. Then, we write the

induction hypothesis as $[\![\Phi]\!]_f, x : [\![T]\!]; [\![\Gamma]\!]_f; [\![\mathcal{S}]\!]_f \vdash [\![P]\!]_f$. By applying (E-Abstraction) and (E-FunType) and by (Tπ-HoAbs1) and Lemma 33 on the induction hypothesis, we obtain the result.

- Case (T-HoApp):

$$\frac{\Phi; \Gamma_1; \mathcal{S}_1 \vdash P : T \xrightarrow{1} \sigma \qquad \Phi; \Gamma_2; \mathcal{S}_2 \vdash Q : T \quad \text{if } T = T' \to \sigma' \text{ then } \mathtt{un}(\Gamma_2) \text{ and } \mathcal{S}_2 = \varnothing}{\Phi; \Gamma_1 \circ \Gamma_2; \mathcal{S}_1 \cup \mathcal{S}_2 \vdash PQ : \sigma}$$

By induction hypothesis $[\![\Phi]\!]_{f'}; [\![\Gamma_1]\!]_{f'}; [\![\mathcal{S}_1]\!]_{f'} \vdash [\![P]\!]_{f'} : [\![T]\!] \xrightarrow{1} [\![\sigma]\!]$ for some renaming function f' for P and $[\![\Phi]\!]_{f''}; [\![\Gamma_2]\!]_{f''}; [\![\mathcal{S}_2]\!]_{f''} \vdash [\![Q]\!]_{f''} : [\![T]\!]$ for some renaming function f'' for Q. Since $\Gamma_1 \circ \Gamma_2$ is defined, then for all $x \in \mathtt{dom}(\Gamma_1) \cap \mathtt{dom}(\Gamma_2)$ it holds that $\Gamma_1(x) = \Gamma_2(x) = T$ and $\mathtt{un}(T)$. Let $\mathtt{dom}(\Gamma_1) \cap \mathtt{dom}(\Gamma_2) = D$ and let $f'_D = f' \setminus \bigcup_{d \in D} \{d \mapsto f'(d)\}$ and $f''_D = f'' \setminus \bigcup_{d \in D} \{d \mapsto f''(d)\} \setminus \bigcup_{q \in \Phi} \{q \mapsto f''(q)\}$. Hence, for all $d \in D$ we are not making any assumption on $f'(d)$ and $f''(d)$. Let $f = \bigcup_{d \in D} \{d \mapsto d'\} \cup f'_D \cup f''_D$, where for all $d \in D$ we create a fresh name d' and associate $d \mapsto d'$. Moreover, f is a function since its subcomponents act on disjoint domains. By applying Lemma 35, the induction hypothesis can be rewritten as follows:

$$[\![\Phi]\!]_f; [\![\Gamma_1]\!]_f; [\![\mathcal{S}_1]\!]_f \vdash [\![P]\!]_f : [\![T]\!] \xrightarrow{1} [\![\sigma]\!]$$

and

$$[\![\Phi]\!]_f; [\![\Gamma_2]\!]_f; [\![\mathcal{S}_2]\!]_f \vdash [\![Q]\!]_f : [\![T]\!]$$

By (E-Application), (Tπ-HoApp), by Lemma 15 and Lemma 33 we obtain $[\![\Phi]\!]_f; [\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f; [\![\mathcal{S}_1]\!]_f \cup [\![\mathcal{S}_2]\!]_f \vdash [\![P]\!]_f [\![Q]\!]_f : [\![\sigma]\!]$.

□

Proof of Theorem 24 for Higher-Order Terms. Let P be a session process, $\Phi, \Gamma, \mathcal{S}$ session typing contexts, and f a renaming function for P such that $[\![\Phi]\!]_f; [\![\Gamma]\!]_f; [\![\mathcal{S}]\!]_f \vdash [\![P]\!]_f$. Then, the following statements hold.

1. If $P \to P'$, then $[\![P]\!]_f \to\!\!\!\!\to [\![P']\!]_f$.
2. If $[\![P]\!]_f \to Q$, then there is a session process P' such that
   - either $P \to P'$;
   - or there are x, y such that $(\boldsymbol{\nu} xy)P \to P'$

   and $Q \hookrightarrow [\![P']\!]_f$.

Proof. Since $[\![\Phi]\!]_f; [\![\Gamma]\!]_f; [\![\mathcal{S}]\!]_f \vdash [\![P]\!]_f$, then by Theorem 29 it is the case that $\Phi; \Gamma; \mathcal{S} \vdash P$. We consider both cases in the following.

1. The proof is done by induction on the derivation $P \to P'$.

- Case (R-Beta):
$$P \triangleq (\lambda x : T.Q)v \to Q[v/x] \triangleq P'$$
By the encoding of abstraction in HOπ with session types we have:
$$\begin{array}{rcl} [\![P]\!]_f & = & [\![(\lambda x : T.Q)v]\!]_f \\ & = & (\lambda x : [\![T]\!].[\![Q]\!]_f)[\![v]\!]_f \\ & \to & [\![Q]\!]_f[[\![v]\!]_f/x] \end{array}$$
Notice that x is bound with scope Q, hence $f_x = x$. On the other hand, by the encoding of P' and by using Lemma 22 we have:
$$[\![P']\!]_f = [\![Q[v/x]]\!]_f = [\![Q]\!]_f[[\![v]\!]_f/f_x] = [\![Q]\!]_f[[\![v]\!]_f/x]$$
This implies that $[\![P]\!]_f \to \equiv [\![P']\!]_f$.

- Case (R-ApplLeft):
$$\frac{P \to P'}{PQ \to P'Q}$$
By induction hypothesis $[\![P]\!]_f \to\hookrightarrow [\![P']\!]_f$. We conclude by context closure of structural congruence and by applying rules (Rπ-ApplLeft) and (Rπ-Struct).

- Case (R-ApplRight):
$$\frac{P \to P'}{vP \to vP'}$$
This case is symmetrical to the previous one. By induction hypothesis $[\![P]\!]_f \to\hookrightarrow [\![P']\!]_f$. We conclude by context closure of structural congruence and by applying rules (Rπ-ApplRight) and (Rπ-Struct).

2. We discuss the case in which the reduction $[\![P]\!]_f \to Q$ is due to an application of a λ-abstraction to a value. Then, by the encoding of processes also P is an application. Let $P = (\lambda x : T.Q')v$. Then, by (E-Application) and (E-Abstraction) we have $[\![P]\!]_f = \lambda x : [\![T]\!].[\![Q']\!]_f[\![v]\!]_f \to Q$ and since x is bound with scope Q', then $f_x = x$. By (R-Beta) we have
$$P = (\lambda x : T.Q')v \ \to \ Q'[v/x] \triangleq P'$$
Then, $[\![P']\!]_f = [\![Q'[v/x]]\!]_f = [\![Q']\!]_f[[\![v]\!]_f/f_x]$, where we apply Lemma 22. One can easily conclude that $Q \equiv [\![Q']\!]_f[[\![v]\!]_f/f_x]$. □

Appendix D. Proofs for the Optimisation of the Encoding

In this appendix we give the proofs for Section 7. In particular, we show that $(\boldsymbol{\nu} c)f_x!\langle v, c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}$ and $x!\langle v, x\rangle.[\![P]\!]$ as well as $(\boldsymbol{\nu} c)f_x!\langle l_j\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}$ and $x!\langle l_j\_x\rangle.[\![P]\!]$ are *typed strong barbed congruent*. We recall a few definitions [27] that lead us to the required result.

Appendix D.1. Auxiliary results

Definition 36 (Context)**.** A *context* in the π-calculus is obtained when the hole $[\cdot]$ replaces an occurrence of the terminated process $\mathbf{0}$ in a process term produced by the grammar in Section 2.2.

We give the definition of *strong barbed bisimilarity*, being the equivalence relation used in the remainder of this section.

Definition 37 (Strong Barbed Bisimilarity)**.** *Strong barbed bisimilarity* is the largest, symmetric relation $\sim$, such that whenever $P \sim Q$,

1. For all x, if P performs an input/output action on x, then Q performs an input/output action on x.
2. $P \to P'$ implies $Q \to Q'$ for some process Q' with $P' \sim Q'$.

Two processes P, Q are *strong barbed bisimilar* if $P \sim Q$.

We are ready now to define the congruence relation based on strong barbed bisimilarity.

Definition 38 (Strong Barbed Congruence)**.** Two processes are *strong barbed congruent* if they are strong barbed bisimilar for every arbitrary context they are placed into.

We pass now from the definition of strong barbed congruence to the typed version of it. Intuitively, a (Γ/Δ)-context, is a context such that when filled with a well-typed process in Δ becomes a well-typed process in Γ. We refer to [27] for the formal definition and further details.

Definition 39 (Typed Strong Barbed Congruence)**.** Let $\Delta \vdash P$ and $\Delta \vdash Q$. We say that processes P, Q are *strong barbed congruent at* Δ, denoted $\Delta \triangleright P \simeq^{\mathsf{c}} Q$, if they are strong barbed congruent for every (Γ/Δ)-context, with Γ closed.

An important result, which will act as a proof technique in the following, is the Context Lemma for the typed strong barbed congruence.

Definition 40. Suppose $\Delta \vdash P$ and $\Delta \vdash Q$. We write $\Delta \triangleright P \simeq^{\mathsf{s}} Q$ if for every closed Γ that extends Δ, for every Δ-to-Γ substitution σ and every process R such that $\Gamma \vdash R$, it holds that $R \mid \sigma(P)$ is strong barbed bisimilar to $R \mid \sigma(Q)$.

Lemma 41 (Context Lemma)**.** Suppose $\Delta \vdash P$ and $\Delta \vdash Q$. $\Delta \triangleright P \simeq^{\mathsf{s}} Q$ if and only if $\Delta \triangleright P \simeq^{\mathsf{c}} Q$.

The Context Lemma and its proof can be found in Sangiorgi and Walker [27].

Appendix D.2. Equivalence Results for the Encoding

Concerning the result for typed strong barbed congruence, we report the cases for the encoding of the output and the selection processes.

Output. Let

$$\Gamma \triangleq x : \ell_{\circ}[T, \ell_{\alpha}[\widetilde{S}]], v : T, \Gamma'$$

$$P \triangleq (\boldsymbol{\nu} c) x!\langle v, c\rangle.[\![R]\!]_{f,\{x \mapsto c\}}$$

$$Q \triangleq x!\langle v, x\rangle.[\![R]\!]$$

$\Gamma' \vdash [\![R]\!]_{f,\{x \mapsto c\}}$ and $\Gamma', x : \ell_{\overline{\alpha}}[\widetilde{S}] \vdash [\![R]\!]$.
Then

$$\Gamma \triangleright P \simeq^{\mathsf{c}} Q \tag{D.1}$$

Selection. Let

$$\Gamma \triangleq x : \ell_{\circ}[\langle l_i \_ T_i\rangle_{i \in I}], \Gamma'$$

$$P \triangleq (\boldsymbol{\nu} c) x!\langle l_j \_ c\rangle.[\![R]\!]_{f,\{x \mapsto c\}}$$

$$Q \triangleq x!\langle l_j \_ x\rangle.[\![R]\!]$$

$\Gamma' \vdash [\![R]\!]_{f,\{x \mapsto c\}}$ and $\Gamma', x : \overline{T_j} \vdash [\![R]\!]$.
Then

$$\Gamma \triangleright P \simeq^{\mathsf{c}} Q \tag{D.2}$$

Above, P is the encoding of output (selection, respectively) by following the rules in Fig. 9 and Q is the encoding of output (selection, respectively) by following the rules in Section 7. By using the typing context Γ for output (respectively selection), (D.1) and (D.2) follow by Lemma 41.