Session Types Revisited

Ornela Dardha Elena Giachino Davide Sangiorgi

INRIA Focus Team / University of Bologna

{dardha, giachino, sangio}@cs.unibo.it

Abstract

Session types are a formalism to model structured communication-based programming. A session type describes communication by specifying the type and direction of data exchanged between two parties. When session types and session primitives are added to the syntax of standard π-calculus types and terms, they give rise to additional separate syntactic categories. As a consequence, when new type features are added, there is duplication of efforts in the theory: the proofs of properties must be checked both on ordinary types and on session types. We show that session types are encodable in ordinary 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 ordinary π types. The robustness of the encoding is tested on a few extensions of session types, including subtyping, polymorphism and higher-order communications.

Categories and Subject Descriptors D.3.1 [*Programming languages*]: Formal Definitions and Theory; F.1.2 [*Computation by abstract devices*]: Modes of Computation—Parallelism and concurrency; F.3.2 [*Logics and meanings of programs*]: Semantics of Programming Languages—Process models; F.3.3 [*Logics and meanings of programs*]: Studies of Program Constructs—Type structure

General Terms Languages, Theory

Keywords session types, π-calculus, linear 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* [3, 21] came into play: they describe a protocol as a type abstraction. Session types were originally designed for process calculi [7, 19, 22]. They have been studied also for other paradigms like multithreaded functional languages, object-oriented languages, Web Services and Contracts, WC3-CDL a language for choreography etc [3].

Session types are a formalism proposed as a theoretical foundation to describe and model structured communication-based programming, guaranteeing privacy as well as communication safety. They are an 'ad hoc' means to describe a *session*, namely a logical unit of data that are exchanged between two or more interacting participants.

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 and distributed scenarios.

However, they offer more flexibility than just performing inputs and outputs: they permit choice, internal and external one. 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 on hand.

As mentioned above, session types guarantee privacy and communication safety. Privacy is guaranteed since session channels are known only to the agents 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 agents. These endpoints have dual behavior and thus have dual types. So, *duality* is a fundamental 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: a client and a server communicating over a session channel. The endpoints x and y of the channel are owned by the client and the server exclusively and should have dual types. To guarantee duality of types, static checks are performed by the type system. If the type of x is $?Int.?Int.!Bool.end$ — meaning that the process listening on channel x receives an integer value followed by another integer value and then sends back a boolean value — then the type of y should be $!Int.!Int.?Bool.end$ — meaning that the process listening on channel 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 agents, is established. It is the *connection*, when a fresh (private) session channel is created and its endpoints are bound to each communicating process. The connection is also the moment when the duality, hence compliance of two session types, is verified. In order to perform a connection, primitives for session channel creation, like `accept`/`request` or (νxy), are added to the syntax of terms [7, 19, 21].

Session types and session primitives are supposed to be added to the syntax of standard π-calculus types and terms, respectively. In doing so, sessions give rise to additional separate syntactic categories. Hence, the syntax of types need to be split into separate syntactic categories, one for session types and the other for standard π-calculus types [5, 7, 19, 22] (this often introduces a duplication of type environments, as well). Common typing features, such as products, records, subtyping, polymorphism, have then to be added to both syntactic categories. Also the syntax of processes will contain both standard process constructs and session primitives (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. For instance, if a new type construct is added, the corresponding properties must be checked both on ordinary types and on session types.

In this paper we try to understand at 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 (actually, as we said above, to one of its endpoints) and 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 *linearized* channels [14], which are channels used multiple times for communication but only in a sequential manner. Linearized types can be encoded, as shown in [14], 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 [18].

Other analogies between session types and π types concern connection and duality. Connection can be seen as the *restriction* construct, since both are used to create and bind a new private session channel to the communication parties. As mentioned above, duality is checked when connection takes place. Duality describes the split of behavior of session channel endpoints. This reminds us of the split of *capabilities*: once a new channel is created by the ν construct, it can be used by two communicating processes owning the opposite capability each.

In this paper, by following Kobayashi [13], we define an interpretation of session types into π types and by exploiting this encoding, session types and all their theory are shown to be derivable from the theory of π-calculus. For instance, basic properties such as Subject Reduction and Type Safety become straightforward corollaries.

Intuitively, a session channel is interpreted as a linear channel transmitting a pair consisting of the original message and a new linear channel which is going to be used for the continuation of the communication.

Furthermore, we present an optimization 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 [13] and the constructs we use are not new (linear types and variants are well-known concepts in type theory and they are also well integrated in the π-calculus). Indeed the technical contribution of the paper may be considered minor (the main technical novelty being the optimization 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 the session types and their basic properties; and

(ii) is a robust encoding.

As evidence for (ii), in the paper we examine, besides plain session types, a few extensions of them, adding subtyping, polymorphism and higher-order features. These are non-trivial extensions, which have been studied in dedicated session types papers [4, 5, 15]. In each case we show that we can derive the main results of the papers via the encoding, as straightforward corollaries.

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 10-15 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 ordinary 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 duality plays a central role: a session is identified by two channel end-points, and these have dual types. In the ordinary 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 duality between input and output capability of channels.

The results in the paper are not however meant to say 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.

The rest of the paper is structured as follows: Section 2 gives an overview of session types and 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 consider extensions to session types: subtyping, polymorphism and higher-order, respectively and analyze the encoding w.r.t. these extensions. Section 7 presents an optimization of linear channels usage. Section 8 examines the related work and concludes the paper.

2. Background

In this section we give an overview of the main technical concepts of the two theories we will be dealing with in the next sections: sessions and π-calculus.

2.1 Session Types

Type Syntax Generally, the syntax of types is given by two separate syntactic categories: one for session types and the other for standard π types, including session types, as well. Types are presented in Figure 1.

We use S to range over session types and T to range over basic types. Session types are: *end*, the type of a terminated session; $?T.S$ and $!T.S$ indicating respectively session channel types used to *receive* and *send* a value of type T and then proceed according to type S. *Branch* and *select* are sets of labelled session types, where the order of components does not matter and labels are all distinct. They indicate external and internal choice, respectively, i.e., what is offered and what is chosen. $\&\{l_1 : S_1, \dots, l_n : S_n\}$ indicates the external choice, what is offered. Dually, select type $\oplus\{l_1 : S_1, \dots, l_n : S_n\}$ indicates the internal choice, only one of the labels will be chosen. Types T include session types, standard channel types and other standard π type constructs. They may also include other basic types meant to be the types of exchanged messages, such as ground types, classes, etc. In [21] the syntax of types is given by a unique syntactic category. In order to distinguish between session types and standard ones, [21] uses *qualifiers* `lin`-linear and `un`-unrestricted. Linear types correspond to session types whereas unrestricted types correspond to the standard π types. We have decided to adopt the syntax above, inspired by [5] as it

Types T ::=			Session Types S ::=	end	termination
	S	session type		$?T.S$	input
	$\sharp T$	channel type		$!T.S$	output
	$\cdots$	other constructs		$\&\{l_1 : S_1, \dots, l_n : S_n\}$	branch
				$\oplus\{l_1 : S_1, \dots, l_n : S_n\}$	select

Processes P ::=	$x!\langle v\rangle.P$	output	$(\nu xy)P$	session restriction
	$x?(y).P$	input	$x \triangleleft l.P$	selection
	$P \mid Q$	composition	$x \triangleright \{l_1 : P_1, \dots, l_n : P_n\}$	branching
	$\mathbf{0}$	inaction		

Values v ::=	x	variable	$true \mid false$	boolean values

Transitions	$(\nu xy)(x!\langle v\rangle.P \mid y?(z).Q) \longrightarrow (\nu xy)(P \mid Q\{v/z\})$	[*R-Com*]
	$(\nu xy)(x \triangleleft l_j.P \mid x \triangleright \{l_1 : P_1, \dots, l_n : P_n\}) \longrightarrow (\nu xy)(P \mid P_j) \quad j \in 1 \dots n$	[*R-Case*]

Figure 1. Syntax and Semantics for Sessions

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

$$\frac{}{\Gamma = \Gamma + \varnothing} \qquad \frac{\Gamma = \Gamma_1 + \Gamma_2}{\Gamma, x : S = (\Gamma_1, x : S) + \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x : S = \Gamma_1 + (\Gamma_2, x : S)}$$

Figure 2. Context split and Context update

underlines the separation of session types from standard channel types.

Language Syntax The syntax of terms presented in Figure 1 is inspired by Vasconcelos [21]. There are different ways of presenting session channel initiation and end-points, like `accept/request` [7], polarized channels [5] or by means of co-variables [21]. Standard communication (not involving sessions) is based on standard π channels [5, 7], whether in [21] it is based on co-variables, as well. In our work we have adopted the use of co-variables. Some comments on the syntax follow: We use P and Q to range over processes and v to range over values. The output process $x!\langle v\rangle.P$ sends a value v on channel x and proceeds as process P; the input process $x?(y).P$ receives on channel x a value that is going to substitute the placeholder y in P. The process $\mathbf{0}$ is the standard inaction process. $(\nu xy)P$ is the scope 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 [22]. The type system enforces that two endpoints specify dual behavior. The last two constructs represent the choices. The process $x \triangleleft l.P$ on channel x selects label l attached to process P. The process $x \triangleright \{l_1 : P_1, \dots, l_n : P_n\}$ on channel x offers a range of alternatives each labelled with a different label taken from $l_1 \dots l_n$. According to the label l_j that is selected the process P_j will be executed.

Duality Two processes willing to communicate, for example, a client and a server, must 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 action is an output one, the dual of branch(offering) is select(choice), as formalized by the following definition:

$$\overline{end} = end$$
$$\overline{?T.S} = !T.\overline{S}$$
$$\overline{!T.S} = ?T.\overline{S}$$
$$\overline{\&\{l_1 : S_1, \dots, l_n : S_n\}} = \oplus\{l_1 : \overline{S_1}, \dots, l_n : \overline{S_n}\}$$
$$\overline{\oplus\{l_1 : S_1, \dots, l_n : S_n\}} = \&\{l_1 : \overline{S_1}, \dots, l_n : \overline{S_n}\}$$

In order to guarantee that communication is safe and proceeds without any mismatch, static checks are performed by the type system. Precisely, these checks consist in controlling that the opposite endpoints of the same session channel have dual types.

Typing Let us now consider the typing rules, listed in Figure 3. We have focused only on the most important ones. Typing derivations for processes are of the form $\Gamma \vdash P$. The typing rules handle context split ($\circ$) in order to deal with linearity of session channels [21] and context update ($+$) in order to deal with the continuation type to make sure they not discarded without being used or they are not used more than once session or duplicated [21]. Context split and context update are defined in Figure 2. These rules handle only the addition of session types S, as the standard types T can be added freely to the context without any particular treatment.

Some comments on the typing rules follow. The rule that better explains duality checks is the rule for restriction [*T-Res*]. Process $(\nu xy)P$ is well-typed in Γ if P is well-typed in Γ augmented with session channel endpoints having dual types ($x : T, y : \overline{T}$). Rule [*T-In*] splits in two the context in which the input process $x?(y).P$ is well-typed: one part type checks the variable x, the other part, augmented with $y : T$ and updated with $x : U$, type checks the continuation process P. The rule for output [*T-Out*] is similar. The context is split in three parts, one to type check x, another to type check v and the last part to type check the continuation P. Similarly to the rule for input, the continuation process uses channel x with its continuation type U. Let us now consider the typing rules for branch, [*T-Brch*], and select, [*T-Sel*].

$$\frac{\Gamma, x : T, y : \overline{T} \vdash P}{\Gamma \vdash (\nu xy)P}\,[T\text{-}Res] \qquad \frac{\Gamma_1 \vdash x : ?T.U \qquad (\Gamma_2, y : T) + x : U \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash x?(y).P}\,[T\text{-}In]$$

$$\frac{\Gamma_1 \vdash x : !T.U \qquad \Gamma_2 \vdash v : T \qquad \Gamma_3 + x : U \vdash P}{\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash x!\langle v \rangle.P}\,[T\text{-}Out]$$

$$\frac{\Gamma_1 \vdash x : \&\{l_1 : T_1, \dots, l_n : T_n\} \qquad \Gamma_2 + x : T_i \vdash P_i \quad \forall i \in 1 \dots n}{\Gamma_1 \circ \Gamma_2 \vdash x \triangleright \{l_1 : P_1, \dots, l_n : P_n\}}\,[T\text{-}Brch]$$

$$\frac{\Gamma_1 \vdash x : \oplus\{l_1 : T_1, \dots, l_n : T_n\} \qquad \Gamma_2 + x : T_j \vdash P \quad j \in 1 \dots n}{\Gamma_1 \circ \Gamma_2 \vdash x \lhd l_j.P}\,[T\text{-}Sel]$$

Figure 3. Some typing rules for session processes

As these constructs are a generalization of the input and output processes, respectively, the corresponding typing rules follow the intuitions above. The branching process $x \triangleright \{l_1 : P_1, \dots, l_n : P_n\}$ is well-typed if channel x is of branch type $\&\{l_1 : T_1, \dots, l_n : T_n\}$ and every continuation process P_i is well-typed and uses x with type T_i. This rule introduces an external choice. Whilst, the rule for selection introduces an internal choice. To type check a process that selects label l_j on the channel x having type $\oplus\{l_1 : T_1, \dots, l_n : T_n\}$, we have to type check that the continuation process P_j uses x with type T_j.

Operational Semantics The operational semantics is defined as a binary relation over processes $\longrightarrow$. We present in Figure 1 only two transition rules, [*R-Com*] and [*R-Case*]. For simplicity, we do not report the transition rules for the other cases as they are standard. In rule [*R-Com*], two processes communicate on two co-variables, that are variables bound together: one sends a value v on x and the other receives it on y and substitutes the placeholder z with it. Rule [*R-Case*] follows the above rule, the communicating processes have prefixes that are co-variables. The selecting process continues as P and the offering one as Q_j, if the label l_j is selected. In order to complete the operational semantics rules, the standard structural congruence is needed.

2.2 π Types

Type Syntax We now consider the π- calculus [18]. The ordinary π-calculus types, ranged over by T, include various type constructs [18]. Here we focus on linear types and variant types, which will be used in the encoding. The syntax of the type constructs we want to discuss is presented in Figure 4. Linear types $\ell_i T$, $\ell_o T$ and $\ell_\sharp T$ are assigned to channels used *exactly once* in input to receive messages of type T, in output to send messages of type T and used once for sending and once for receiving messages of type T, respectively. The variant type $\langle l_1\_T_1 \dots l_n\_T_n \rangle$ is a labelled form of disjoint union of types. The order of the components does not matter and labels are all distinct. Product types $(T_1 \times \dots \times T_n)$ are needed to model the polyadic π-calculus. Other type constructs like ground types, recursive types etc. can be added to the syntax.

Language syntax The syntax of terms of the π-calculus is given in Figure 4. The output process $\overline{x}\langle \tilde{v} \rangle.P$ sends a tuple of values $\tilde{v}$ on channel x 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; parallel composition and **0** inaction are standard; restriction creates a new name x and binds it with scope P (the same as in sessions for shared channels). Process `case` v `of` $[l_1\_(x_1) \triangleright P_1 \dots l_n\_(x_n) \triangleright P_n]$ offers different behaviors depending on which variant value l_v it receives. Other values are boolean values and variables.

Typing Some typing rules for π processes are given in Figure 5. As in the previous section for session types, also here, there is a particular handling of typing environments in order to ensure linearity. The context is split following a split of linear types. which is defined as follows:

$$\ell_o T \uplus \ell_i T \triangleq \ell_\sharp T$$
$$T \uplus T \triangleq T$$
$$T \uplus S \triangleq (\texttt{error}) \text{ otherwise}$$

The context split is defined as follows:

$$(\Gamma_1 \uplus \Gamma_2)(x) \triangleq$$
$$\begin{array}{ll} \Gamma_1(x) \uplus \Gamma_2(x) & \text{if both } \Gamma_1(x) \text{ and } \Gamma_2(x) \text{ are defined} \\ \Gamma_1(x) & \text{if } \Gamma_1(x)\text{, but not } \Gamma_2(x)\text{, is defined} \\ \Gamma_2(x) & \text{if } \Gamma_2(x)\text{, but not } \Gamma_1(x)\text{, is defined} \\ \texttt{undef} & \text{if both } \Gamma_1(x) \text{ and } \Gamma_2(x) \text{ are undefined} \end{array}$$

Some comments on the typing rules follow. Rules for input and output processes are straightforward: on a linear input, respectively output, channel x a value v of the correct type is received, respectively sent, with a continuation process P. Rule on restriction states asserts that process $(\nu x : \ell_\sharp T)P$ is well-typed provided P is well-typed in a context extended with $x : \ell_\sharp T$. Following the definition of context split above it means $x : \ell_o T, x : \ell_i T$ and this is a fundamental feature exploited in the encoding. It is important to notice that the standard rule for restriction assigns to x some general channel type T and not necessarily : $\ell_\sharp T$. We have adopted the rule in this simplified form as it used in our encoding, presented in the next section. A variant value l_v is of type $\langle l\_T \rangle$ if v is of type T. Notice that, by means of subtyping, we can also derive that l_v is of type $\langle l_1\_T_1 \dots l\_T \dots l_n\_T_n \rangle$. Process `case` v `of` $[l_1\_(x_1) \triangleright P_1 \dots l_n\_(x_n) \triangleright P_n]$ is well-typed if value v has variant type and every process P_i is well-typed assuming x_i has type T_i.

Operational Semantics We present in Figure 4 two transition rules. Again, as for sessions, we do not present the other rules as they are standard. 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` normalization. The `case` process evolves to P_j substituting x_j with the value v, if the label l_j is chosen. Structural congruence and the corresponding rules are standard again.

3. Encoding

Session types guarantee that only the communicating parties know the corresponding endpoints of the session channel, thus providing

$$\begin{array}{lll}
\text{Types} & T ::= \ell_i T \quad | \quad \ell_o T \quad | \quad \ell_\sharp T & \text{linear input — linear output — linear connection} \\
& \langle l_1\_T \ldots l_n\_T \rangle & \text{variant type} \\
& (T \times \ldots \times T) & \text{product type} \\
& \ldots & \text{other constructs}
\end{array}$$

$$\begin{array}{llllll}
\text{Processes} & P ::= & \overline{x}\langle \tilde{v} \rangle.P & \text{output} & (\nu x)P & \text{scope restriction} \\
& & x(\tilde{y}).P & \text{input} & \texttt{case}\ v\ \texttt{of}\ [l_1\_(x_1) \triangleright P_1 \ldots l_n\_(x_n) \triangleright P_n] & \text{case} \\
& & P \mid Q & \text{composition} & & \\
& & \mathbf{0} & \text{inaction} & &
\end{array}$$

$$\begin{array}{lllll}
\text{Values} & v ::= & x & \text{variable} & true \quad | \quad false \quad \text{boolean values} \\
& & l\_v & \text{variant value} &
\end{array}$$

$$\begin{array}{lll}
\text{Transitions} & \overline{x}\langle \tilde{v} \rangle.P \mid x(\tilde{z}).Q \longrightarrow P \mid Q\{\tilde{v}/\tilde{z}\} & [R\pi\text{-}Com] \\
& \texttt{case}\ l_j\_v\ \texttt{of}\ [l_1\_(x_1) \triangleright P_1 \ldots l_n\_(x_n) \triangleright P_n] \longrightarrow P_j \quad j \in 1 \ldots n & [R\pi\text{-}Case]
\end{array}$$

Figure 4. Syntax and Semantics for π-calculus

$$\frac{\Gamma_1 \vdash x : \ell_i T \qquad \Gamma_2, y : T \vdash P}{\Gamma_1 \uplus \Gamma_2 \vdash x(y).P}\ [T\pi\text{-}In]$$

$$\frac{\Gamma_1 \vdash x : \ell_o T \qquad \Gamma_2 \vdash v : T \qquad \Gamma_3 \vdash P}{\Gamma_1 \uplus \Gamma_2 \uplus \Gamma_3 \vdash \overline{x}v.P}\ [T\pi\text{-}Out]$$

$$\frac{\Gamma, x : \ell_\sharp T \vdash P}{\Gamma \vdash (\nu x : \ell_\sharp T)P}\ [T\pi\text{-}Res] \qquad \frac{\Gamma \vdash v : T}{\Gamma \vdash l\_v : \langle l\_T \rangle}\ [T\pi\text{-}Var]$$

$$\frac{\Gamma_1 \vdash v : \langle l_1\_T_1 \ldots l_n\_T_n \rangle \qquad \Gamma_2, x_i : T_i \vdash P_i \quad \forall i}{\Gamma_1 \circ \Gamma_2 \vdash \texttt{case}\ v\ \texttt{of}\ [l_1\_(x_1) \triangleright P_1 \ldots l_n\_(x_n) \triangleright P_n]}\ [T\pi\text{-}Case]$$

Figure 5. Some typing rules for π processes

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 that are used *exactly once*. Privacy is ensured since the linear channel is used *at most once* and so it is known only to the interacting parties. Communication safety is ensured since the linear channel is used *at least once* and so the input/output actions are necessarily performed. Obviously, values transmitted should be checked if they have the right type as specified by the protocol.

In the following we provide an encoding of session types into ordinary π types and of session processes into π-calculus ones. The intuition behind our encoding is that the continuation behavior of the session channel instead of being explicitly put into the type, as in session types, is sent along with the message at each output.

3.1 Type Encoding

First we present the encoding of session types into ordinary π types, which is defined in Figure 6.

The encoding of the terminated communication channel is a linear channel with no capabilities, meaning that it cannot be used neither for input nor for output. The session channel type $?T.S$ is interpreted as the linear input channel type carrying a pair of values of type the encoding of T and of the encoding of continuation S. The encoding of $!T.S$ is a linear type used in output to carry a pair of values of type the encoding of T and of type the encoding of the dual of S. Note that in this case it is the dual of S to be sent since it is the type of a channel as seen by the receiver. The branch and the select types are generalizations of input and output types, respectively. Consequently, they are interpreted as linear input and linear output channels carrying variant types having the same labels $l_1 \ldots l_n$ and the types encodings of $S_1 \ldots S_n$ and $\overline{S_1} \ldots \overline{S_n}$, respectively. Again, the reason for duality is the same as for the output type.

As mentioned above, in order to establish a communication, the opposite endpoints of the session channel should have dual types. Consider the following dual types: $S = ?Int.!Int.end$ and $\overline{S} = !Int.?Int.end$. Their encoding is the following: $[\![S]\!] = \ell_i\,[Int, \ell_o\,[Int, \ell_\varnothing[]]]$ and $[\![\overline{S}]\!] = \ell_o\,[Int, \ell_o\,[Int, \ell_\varnothing[]]]$. It turns out that the duality of session types boils down to opposite capabilities of linear channel types. The encodings above differ only in the outermost level, that corresponds to having ℓ_i or ℓ_o types. The π channels having these types carry exactly the same messages. This happens because duality is incorporated in the output typing, where the receiver's point of view of the output type is considered, which is therefore dual w.r.t. that of the sender. The encoding simplifies the structure of the pair of dual session types. So, it can render the process of abstracting protocols as structured types easier.

To conclude, let us recall the syntax of types in Vasconcelos [21], where the qualifiers are present. What we encode are the session types (given by the syntactic category S ::=), that correspond to the *linear* types in [21] while the standard π types (given by the syntactic category T ::=) that correspond to the *unrestricted* types in [21], remain as such, i.e. they are not encoded as they are already present in the π-calculus.

3.2 Process Encoding

The encoding of session processes into π-calculus processes is defined in Figure 6.

$$\begin{aligned}
\llbracket end \rrbracket &= \ell_{\varnothing}[\,] \\
\llbracket ?T.S \rrbracket &= \ell_i[\llbracket T \rrbracket, \llbracket S \rrbracket] \\
\llbracket !T.S \rrbracket &= \ell_o[\llbracket T \rrbracket, \llbracket \overline{S} \rrbracket] \\
\llbracket \{l_1 : S_1 \&, \dots, \& l_n : S_n\} \rrbracket &= \ell_i[\langle l_1 : \llbracket S_1 \rrbracket, \dots, l_n : \llbracket S_n \rrbracket \rangle] \\
\llbracket \{l_1 : S_1 \oplus, \dots, \oplus l_n : S_n\} \rrbracket &= \ell_o[\langle l_1 : \llbracket \overline{S_1} \rrbracket, \dots, l_n : \llbracket \overline{S_n} \rrbracket \rangle]
\end{aligned}$$

$$\begin{aligned}
\llbracket P \mid Q \rrbracket_f &= \llbracket P \rrbracket_f \mid \llbracket Q \rrbracket_f \\
\llbracket x!\langle v \rangle.P \rrbracket_f &= (\nu c)\overline{f_x}\langle v, c \rangle.\llbracket P \rrbracket_{f,\{x \to c\}} \\
\llbracket x?(y).P \rrbracket_f &= f_x(y, c).\llbracket P \rrbracket_{f,\{x \to c\}} \\
\llbracket (\nu xy)P \rrbracket_f &= (\nu c)\llbracket P \rrbracket_{f,\{x \to c, y \to c\}} \\
\llbracket x \lhd l.P \rrbracket_f &= (\nu c)\overline{f_x}\langle l\_c \rangle.\llbracket P \rrbracket_{f,\{x \to c\}} \\
\llbracket x \rhd \{l_1 : P_1, \dots, l_n : P_n\} \rrbracket_f &= f_x(y).\ \texttt{case}\ y\ \texttt{of}\ [l_1\_c \Rightarrow \llbracket P_1 \rrbracket_{f,\{x \to c\}} \dots l_n\_c \Rightarrow \llbracket P_n \rrbracket_{f,\{x \to c\}}]
\end{aligned}$$

Figure 6. Encoding of types and terms

The encoding of terms differs from the encoding of types as it is parametrized in a function f that renames the linear channels involved in the communication. The reason for f is the following: since we are using linear types, 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.

Some explanations on the encoding are provided. The encoding of the output process is as follows: a new channel name c is created and is sent together with the value v along channel x renamed by function f, we denote this by f_x; the encoding of the continuation process P is parametrized in f where name x is updated to c. Similarly, the input process listens on channel f_x and receives a value, that substitutes variable y and a fresh channel c that substitutes x in the continuation process encoded in f updated.

As shown in Section 2.1, the syntax of session processes we have adopted here has a particular treatment of the binding constructor: $(\nu xy)P$ creates two fresh names and binds them in P and together as being the opposite endpoints of the same session channel. Since there is no such binding in π-calculus, the encoding is given by the creation of a new name (νc) of linear channel type with both capabilities of input/output. This may lead the reader to think of subject reduction failure, as shown in [22], since two opposite endpoints are being encoded as a single name c . However, this name substitutes x and y in the encoding of P in such a way that the capability of c matches the encoding of the type of that name. Namely, each endpoint corresponds to channel c having only one of the capabilities, input or output, whereas the other one corresponds to the opposite endpoint. So, in session types we use two names x and y, whereas in π-calculus we introduce a single name c having two capabilities.This is a static check performed by the type system. We sometimes denote c_b and $c_{\overline{b}}$ to emphasize the capabilities of c.

The last two constructs correspond to selection and branching processes. The selection process $x \lhd l.P$ is encoded as the process that first creates a new channel c and then sends on f_x a variant value, l_c where l is the label it is selecting and c is the channel created to be used for the rest of the session. Afterwards it proceeds as process P encoded in f updated. The branching process is the most complicated one: it receives on f_x a value, typically being a variant value l_c, to substitute the placeholder y in the `case` process. The value l_c, as for the selection, is composed by the label l, that the partner has chosen and the channel c to be used in the continuation processes. According to the label received one of the corresponding processes $\llbracket P_1 \rrbracket_{f,\{x\to c\}} \dots \llbracket P_n \rrbracket_{f,\{x\to c\}}$ is chosen and again the encoding is parametrized in f updated by $\{x \to c\}$. Note that the name c is bound in any process $\llbracket P_i \rrbracket_{f,\{x\to c\}}$ The encoding of other process constructs, like inaction, standard scope restriction, parallel composition etc. is a homomorphism.

For a better understanding of the encoding, let us consider a simple example: the equality test. The server and client processes are the following.

$$\begin{aligned}
server &= x?(v_1).x?(v_2).x!\langle v_1 == v_2 \rangle.\mathbf{0} \\
client &= y!\langle 3 \rangle.y!\langle 5 \rangle.y?(eq).\mathbf{0}
\end{aligned}$$

They communicate on a session channel by owning two opposite endpoints x and y, respectively. The server accepts two integer values in sequence v_1 and v_2 and sends back true or false depending on whether these values are equal or not $(v_1 == v_2)$. The client process behaves dually: it sends to the server two integer values 3 and 5 and waits for a boolean answer. After this communication, they both terminate. The encodings of server and client processes are:

$$\begin{aligned}
\llbracket server \rrbracket_f &= z(v_1, c).c(v_2, c').(\nu c'')\overline{c'}\langle v_1 == v_2, c'' \rangle.\mathbf{0} \\
\llbracket client \rrbracket_f &= (\nu c)\overline{z}\langle 3, c \rangle.(\nu c')\overline{c}\langle 5, c' \rangle.c'(eq, c'').\mathbf{0}
\end{aligned}$$

In the encoding, at the very first step, function f initializes x and y to a new name z, and after that, before every output action, a new channel c, c', c'' is created and sent to the partner together with the value.

Session types associated to channel endpoints x, y on which this interaction takes place, are as follows:

$$x : ?Int.?Int.!Bool.end \qquad y : !Int.!Int.?Bool.end$$

Following the encoding definition we have the following:

$$\begin{aligned}
\llbracket x \rrbracket &= \ell_i\,[Int, \ell_i\,[Int, \ell_o\,[Bool, unit]]] \\
\llbracket y \rrbracket &= \ell_o\,[Int, \ell_i\,[Int, \ell_o\,[Bool, unit]]]
\end{aligned}$$

3.3 Properties

The encoding presented previously can be taken as the semantics of session types and session terms. The following results show that we can derive the typing judgments and Subject Reduction and Type Safety of the session calculus.

THEOREM 1 (Type Correctness). $\Gamma \vdash P$ *if and only if* $\llbracket \Gamma \rrbracket_f \vdash \llbracket P \rrbracket_f$.

PROOF 1. *(⇒) It is proved by induction on the length of the derivation* $\Gamma \vdash P$. *Let us consider just one case of the proof.*

Case *[*T-Res*]:*

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

To prove $\llbracket \Gamma \rrbracket_f \vdash \llbracket (\nu xy)P \rrbracket_f$ *Following the encoding,* $\llbracket (\nu xy)P \rrbracket_f = (\nu z)\llbracket P \rrbracket_{f,\{x \to z, y \to z\}}$. *By IH* $\llbracket \Gamma \rrbracket_f, x : \llbracket T \rrbracket, y : \llbracket \overline{T} \rrbracket \vdash \llbracket P \rrbracket_{f,\{x,y\}}$ *The encoding of dual types is as follows* $\llbracket T \rrbracket = \ell_c[\llbracket \cdot \rrbracket]$ *and* $\llbracket \overline{T} \rrbracket = \ell_{\overline{c}}[\llbracket \cdot \rrbracket]$ *where we leave unspecified the innermost level of the type as it is not important. We will introduce a fresh name* z *having type* $\ell_\sharp[\llbracket \cdot \rrbracket]$ $\sharp = c + \overline{c}$ *i.e. a type having both the type capabilities of outermost capabilities of* T *and* $\overline{T}$ *maintaining the internal structure of the types. So, we can modify the IH as follows:* $\llbracket \Gamma \rrbracket_f, z : \ell_\sharp[\llbracket \cdot \rrbracket] \vdash \llbracket P \rrbracket_{f,\{x \to z, y \to z\}}$, *where* x *is substituted by* z *having type* $\ell_c[\llbracket \cdot \rrbracket]$ *whilst,* y *is substituted by* z *having type* $\ell_{\overline{c}}[\llbracket \cdot \rrbracket]$. *Using rule* LIN-RES *in pi, we have:*

$$\frac{\llbracket \Gamma \rrbracket_f, z : \ell_\sharp[\llbracket \cdot \rrbracket] \vdash \llbracket P \rrbracket_{f,\{x \to z, y \to z\}}}{\llbracket \Gamma \rrbracket_f \vdash (\nu z)\llbracket P \rrbracket_{f,\{x \to z, y \to z\}}}$$

($\Leftarrow$*) It is proved by induction on the structure of* P. *We consider again one case of the proof just to illustrate it.*
Case $P = x(y).P$. *Suppose* $\llbracket \Gamma \rrbracket_f \vdash \llbracket x(y).P \rrbracket_f \triangleq f_x(y,c).\llbracket P \rrbracket_{f,\{x \to c\}}$. *This is derived by using* LIN-INP *as the last rule:*

$$\frac{\llbracket \Gamma_1 \rrbracket_f \vdash f_x : \ell_i[W_1, W_2] \qquad \llbracket \Gamma_2 \rrbracket_f, y : W_1, c : W_2 \vdash \llbracket P \rrbracket_{f,\{x \to c\}}}{\llbracket \Gamma \rrbracket_f \vdash f_x(y,c).\llbracket P \rrbracket_{f,\{x \to c\}}}$$

Where $\llbracket \Gamma \rrbracket_f = \llbracket \Gamma_1 \rrbracket_f \uplus \llbracket \Gamma_2 \rrbracket_f$. *By IH* $\Gamma_1 \vdash x : ?T_1.T_2$ *and* $(\Gamma_2, y : T_1) + x : T_2 \vdash P$ *where* $W_1 = \llbracket T_1 \rrbracket$ *and* $W_2 = \llbracket T_2 \rrbracket$. *Using rule [*T-In*] we conclude the following:*

$$\frac{\Gamma_1 \vdash x : ?T_1.T_2 \qquad (\Gamma_2, y : T_1) + x : T_2 \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash x?(y).P}\,[\text{T-In}]$$

Theorem 1, and more precisely its proof, shows that the encoding can be actually used to reconstruct the typing rules of session types.

THEOREM 2 (Operational Correspondence). *If* $P \to P'$ *then* $\exists Q$ *such that* $\llbracket P \rrbracket_f \to Q$ *and* $Q \hookrightarrow \llbracket P' \rrbracket_f$ *where* $\hookrightarrow$ *is a structural congruence extended with* `case` *normalization.*

PROOF 2. *It is proved by induction on the length of the proof of the reduction* $P \to P'$. *We consider the following base cases:*
Case *(*R-Com*):*
$P = (\nu xy)(x!\langle v\rangle.P_1 \mid y?(z).P_2) \to (\nu xy)(P_1 \mid P_2\{v/z\}) = P'$.

$$\begin{aligned}
\llbracket P \rrbracket_f &= \\
&\llbracket (\nu xy)(x!\langle v\rangle.P_1 \mid y?(z).P_2) \rrbracket_f \\
&\triangleq (\nu t)\llbracket (x!\langle v\rangle.P_1 \mid y?(z).P_2) \rrbracket_{f,\{x \to t, y \to t\}} \\
&= (\nu t)\left(\llbracket x!\langle v\rangle.P_1 \rrbracket_{f,\{x \to t\}} \mid \llbracket y?(z).P_2 \rrbracket_{f,\{y \to t\}}\right) \\
&\triangleq (\nu t)\left[\left((\nu c)\overline{t}\langle v, c\rangle.\llbracket P_1 \rrbracket_{f,\{x \to t, t \to c\}}\right) \mid t(z,c).\llbracket P_2 \rrbracket_{f,\{y \to t, t \to c\}}\right] \\
&\quad \textit{(an } \alpha\textit{-conversion is performed in order to have c in the rhs)} \\
&\to (\nu t)\left[(\nu c)\left(\llbracket P_1 \rrbracket_{f,\{x \to c\}} \mid \llbracket P_2 \rrbracket_{f,\{y \to c\}}\{v/z\}\right)\right] = Q \\
&\equiv (\nu c)\left(\llbracket P_1 \rrbracket_{f,\{x \to c\}} \mid \llbracket P_2 \rrbracket_{f,\{y \to c\}}\{v/z\}\right)
\end{aligned}$$

The following also holds:

$$\begin{aligned}
\llbracket P' \rrbracket_f &\triangleq \llbracket (\nu xy)(P_1 \mid P_2\{v/z\}) \rrbracket_f \\
&\triangleq (\nu c)\llbracket P_1 \mid P_2\{v/z\} \rrbracket_{f,\{x \to c, y \to c\}} \\
&\triangleq (\nu c)(\llbracket P_1 \rrbracket_{f,\{x \to c\}} \mid \llbracket P_2 \rrbracket_{f,\{y \to c\}}\{v/z\})
\end{aligned}$$

Case *(*R-Case*):* $P = (\nu xy)(x \triangleleft l_j.R \mid y \triangleright \{l_1 : Q_1, \ldots, l_n : Q_n\}) \to (\nu xy)(R \mid Q_j) = P' j \in 1 \ldots n.$

$$\begin{aligned}
\llbracket P \rrbracket_f &\triangleq \llbracket (\nu xy)(x \triangleleft l_j.R \mid y \triangleright \{l_1 : Q_1, \ldots, l_n : Q_n\}) \rrbracket_f \\
&\triangleq (\nu z)\left(\llbracket x \triangleleft l_j.R \mid y \triangleright \{l_1 : Q_1, \ldots, l_n : Q_n\} \rrbracket_{f,\{x \to z, y \to z\}}\right) \\
&\triangleq (\nu z)\left(\llbracket z \triangleleft l_j.R \rrbracket_{f,\{x \to z\}} \mid \llbracket z \triangleright \{l_1 : Q_1, \ldots, l_n : Q_n\} \rrbracket_{f,\{y \to z\}}\right) \\
&\triangleq (\nu z)\Big((\nu c)\overline{z}\langle l_j\_c\rangle.\llbracket R \rrbracket_{f,\{x \to z, z \to c\}} \mid z(w).\ \texttt{case}\ w\ \texttt{of} \\
&\qquad [l_1\_c : \llbracket Q_1 \rrbracket_{f,\{y \to z, z \to c\}} \ldots l_n\_c : \llbracket Q_n \rrbracket_{f,\{y \to z, z \to c\}}]\Big) \\
&\quad \textit{(an } \alpha\textit{-conversion is performed in order to have c in the rhs)} \\
&= (\nu c)\Big(\overline{z}\langle l_j\_c\rangle.\llbracket R \rrbracket_{f,\{x \to c\}} \mid z(w).\ \texttt{case}\ w\ \texttt{of} \\
&\qquad [l_1\_c : \llbracket Q_1 \rrbracket_{f,\{y \to c\}} \ldots l_n\_c : \llbracket Q_n \rrbracket_{f,\{y \to c\}}]\Big) \\
&\to (\nu c)\Big(\llbracket R \rrbracket_{f,\{x \to c\}} \mid \texttt{case}\ l_j\_c\ \texttt{of} \\
&\qquad [l_1\_c : \llbracket Q_1 \rrbracket_{f,\{y \to c\}} \ldots l_n\_c : \llbracket Q_n \rrbracket_{f,\{y \to c\}}]\Big) = Q \\
&\hookrightarrow (\nu c)\left(\llbracket R \rrbracket_{f,\{x \to c\}} \mid \llbracket Q_j \rrbracket_{f,\{y \to c\}}\right)
\end{aligned}$$

The following also holds:

$$\begin{aligned}
\llbracket P' \rrbracket_f &\triangleq \llbracket (\nu xy)(R \mid Q_j) \rrbracket_f \\
&\triangleq (\nu c)\llbracket R \mid Q_j \rrbracket_{f,\{x \to c, y \to c\}} \\
&\triangleq (\nu c)(\llbracket R \rrbracket_{f,\{x \to c\}} \mid \llbracket Q_j \rrbracket_{f,\{y \to c\}})
\end{aligned}$$

The inductive step is straightforward.

By exploiting Type Correctness and Operational Correspondence we derive for free the Subject Reduction and Type Safety (absence of run-time errors) in session types.

COROLLARY 1. *If* $\Gamma \vdash P$ *and* $P \to P'$ *then* $\Gamma \vdash P'$

PROOF 3. *The result follows from the subject reduction property in* π*-calculus and Theorems 1 and 2.*

After analyzing the effectiveness of the encoding on basic session types, in the following sections we show its robustness by examining three non-trivial extensions, namely subtyping, polymorphism and higher-order.

4. Subtyping

Subtyping is a relation among channel types based on a notion of substitutability, meaning that 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. The definition of subtyping must be done carefully in order for this substitutability property to hold.

Subtyping has been studied extensively in π-calculus [17, 18]. It has also been studied in session types [5]. In this section we show that the ordinary subtyping of the π-calculus is enough to derive subtyping in session types. Subtyping rules for both systems are presented in Figure 7. We use the symbol <: for session subtyping, and $\leq$ for standard π subtyping.

Rules (*S-inp*) and (*S-out*) define subtyping relation of input and output linear π channels. These rules assert that input channels are co-variant and output channels are contra-variant in the types of values they transmit. Rule (*S-variant*) presents subtyping of variant types. It is co-variant both in depth and in breadth. Rules (*S-?*) and (*S-!*) indicate subtyping in input and output session types, respectively. As before, input operation is co-variant whilst output

$$\frac{I = \{i, \sharp\} \qquad \forall i \in 1 \dots n.\quad T_i \leq S_i}{\ell_I[T_1, \dots, T_n] \leq \ell_i[S_1, \dots, S_n]}\ (\textit{S-inp}) \qquad \frac{I = \{o, \sharp\} \qquad \forall i \in 1 \dots n.\quad S_i \leq T_i}{\ell_I[T_1, \dots, T_n] \leq \ell_o[S_1, \dots, S_n]}\ (\textit{S-out})$$

$$\frac{\forall i \in 1 \dots n.\quad T_i \leq S_i}{\langle l_1\_T_1 \dots l_n\_T_n \rangle \leq \langle l_1\_S_1 \dots l_{n+m}\_S_{n+m} \rangle}\ (\textit{S-variant})$$

$$\frac{T <: T' \qquad S <: S'}{?T.S <: ?T'.S'}\ (\textit{S-?}) \qquad \frac{T' <: T \qquad S <: S'}{!T.S <: !T'.S'}\ (\textit{S-!})$$

$$\frac{I \subseteq J \qquad \forall i \in I.\quad T_i <: S_i}{\&\{l_i : T_i\}_{i \in I} <: \&\{l_j : S_j\}_{j \in J}}\ (\textit{S-brch}) \qquad \frac{I \supseteq J \qquad \forall j \in J.\quad T_j <: S_j}{\oplus\{l_i : T_i\}_{i \in I} <: \oplus\{l_j : S_j\}_{j \in J}}\ (\textit{S-sel})$$

Figure 7. Subtyping rules for π types ($\leq$) and for session types ($<:$).

operation is contra-variant. The continuation type is co-variant in both cases. This is a difference w.r.t. the corresponding rules in π-calculus. There are two rules for labelled types, namely (*S-brch*) and (*S-sel*) being both co-variant in depth in the types of values they transmit and being co-variant and contra-variant in breadth, respectively.

In π-calculus with sessions and subtyping, one must deal both with ordinary subtyping on π types and subtyping on session types. This introduces 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 ordinary π types and on session types. In addition, on session types one also needs coinductive techniques to formalize type duality.

The encoding is used, as in the previous section, to derive basic properties of session types; in addition to Theorem 1 and 2 here we have to take into account the subtyping relation. Therefore, it is important to prove the validity of subtyping, which is necessary in order to extend Subject Reduction and Type Safety.

THEOREM 3 (Validity of Subtyping). *$T <: S$ if and only if $[\![T]\!] \leq [\![S]\!]$.*

PROOF 4. *($\Rightarrow$) The proof is by induction on the derivation of $T <: S$. To give an idea of the proof we consider the following case*
Case input*:*
Where $T = ?T_1.T_2$ and $S = ?S_1.S_2$. By induction hypothesis: $[\![T_1]\!] \leq [\![S_1]\!]$ and $[\![T_2]\!] \leq [\![S_2]\!]$. To prove $[\![?T_1.T_2]\!] \leq [\![?S_1.S_2]\!]$.

*Since $[\![?T_1.T_2]\!] = \ell_i[[\![T_1]\!], [\![T_2]\!]]$ and $[\![?S_1.S_2]\!] = \ell_i[[\![S_1]\!], [\![S_2]\!]]$, using the IH and the rule (*S-inp*) with $I = \{i\}$ we have:*

$$\frac{[\![T_1]\!] \leq [\![S_1]\!] \qquad [\![T_2]\!] \leq [\![S_2]\!]}{\ell_i[[\![T_1]\!], [\![T_2]\!]] \leq \ell_i[[\![S_1]\!], [\![S_2]\!]]}\ \text{(Sub-inp)}$$

($\Leftarrow$) The proof is by induction on the structure of session types T, S. Again, let us consider as an example one case of the proof.
Case *$T = !T_1.T_2$ and $S = !S_1.S_2$. The encodings are $[\![!T_1.T_2]\!] = \ell_o[[\![T_1]\!], [\![\overline{T_2}]\!]]$ and $[\![!S_1.S_2]\!] = \ell_o[[\![S_1]\!], [\![\overline{S_2}]\!]]$ and we suppose $[\![T]\!] \leq [\![S]\!]$. To prove $T <: S$. Using the following rule in pi we have:*

$$\frac{[\![S_1]\!] \leq [\![T_1]\!] \qquad [\![\overline{S_2}]\!] \leq [\![\overline{T_2}]\!]}{\ell_o[[\![T_1]\!], [\![\overline{T_2}]\!]] \leq \ell_o[[\![S_1]\!], [\![\overline{S_2}]\!]]}\ \text{(Sub-out)}$$

*An auxiliary Lemma gives $[\![T_2]\!] \leq [\![S_2]\!]$ if and only if $[\![\overline{S_2}]\!] \leq [\![\overline{T_2}]\!]$ and by IH we have $S_1 <: T_1$ and $T_2 <: S_2$. We conclude using rule (*S-!*).*

The other cases of the proof follow in a similar way.

Subtyping in session types has been studied in details in [5, 21]: we can derive the main results in these papers as straightforward corollaries via the encoding along the lines of what we have shown for Subject Reduction and Validity of Subtyping. Examples are: reflexivity and transitivity of subtyping, and all the auxiliary lemmas (e.g. substitution).

The above results remain valid with the addition of recursive types; in this case we can also obtain for free all the coinductive techniques for subtyping and duality in session types.

5. Polymorphism

Polymorphism is a common and useful type abstraction in programming languages as it allows operations that are generic by using an expression with several types. Polymorphism is added both on session side and on the π side. In this section we show that this duplication is not necessary: all the theory of polymorphism in session types can be derived by the corresponding theory in the π-calculus. This holds for the standard parametric polymorphism as well as for bounded polymorphism.

5.1 Parametric Polymorphism

Let us first consider (existential) parametric polymorphism. This form of polymorphism has not been studied in session types. We need to extend the syntax of types T with type variable X and polymorphic type $\langle X; T \rangle$.

Types	$T ::= S$	session type
	$\sharp T$	channel type
	X	type variable
	$\langle X; T \rangle$	polymorphic type
	$\dots$	other π types constructs

The syntax of session types remains unchanged. Modifications in the syntax of types introduce modifications in the syntax of terms, as expected. So, we add *polymorphic value* $\langle T; v \rangle$ and *unpacking process* `open` v `as` $(X; x)$ `in` P, the same constructs as in π-calculus. Note that value v and hence name x, can be tuples of values, respectively names, in order to accommodate polyadicity.

Since we added polymorphic constructs in the syntax of standard types and we left the syntax of session types unchanged, the encoding of session types is the same as before, hence the encoding of types is a homomorphism. In particular, polymorphic constructs are encoded as

$$[\![X]\!] = X$$
$$[\![\langle X; T \rangle]\!] = \langle X; [\![T]\!] \rangle$$

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

$$\begin{array}{l} [\![\langle T; v\rangle]\!]_f = \langle [\![T]\!]; [\![v]\!]_f\rangle \\ [\![\mathtt{open}\ v\ \mathtt{as}\ (X;x)\ \mathtt{in}\ P]\!]_f = \mathtt{open}\ [\![v]\!]_f\ \mathtt{as}\ [\![(X;x)]\!]\ \mathtt{in}\ [\![P]\!]_f \end{array}$$

In the case of polymorphic calculi we prove the correctness of the typing derivation, by considering *only* the polymorphic types/terms constructs and the corresponding typing rules. We also prove the operational correspondence for the new process constructs added. The typing judgments are of the form $\Delta, \Gamma \vdash P$ where Δ is the set of type variables present in P. This is the only difference w.r.t. typings introduced before. The following theorem states the correctness result for polymorphic types.

THEOREM 4. $\Delta, \Gamma \vdash P$ *if and only if* $\Delta, [\![\Gamma]\!]_f \vdash [\![P]\!]_f$.

The operational correspondence for polymorphic calculi merely adds a case to Theorem 2 stated previously. Again, Subject Reduction and Type Safety and other basic properties are derived as in the previous sections.

5.2 Bounded Polymorphism

We now consider the bounded polymorphism, studied in [4], which is a form of parametric polymorphism. This kind of polymorphism has not been studied yet in π-calculus; we add it and show how we can derive bounded polymorphism in session types passing through the π types. Bounded polymorphism in session types [4] is added only to the labels in the branch and select constructs. The two type constructs have now the following shape:

$$\begin{array}{l} \& \{l_1(X_1 \le B_1) : S_1, \ldots, l_n(X_n \le B_n) : S_n\} \\ \oplus \{l_1(X_1 \le B_1) : S_1, \ldots, l_n(X_n \le B_n) : S_n\}, \end{array}$$

where B stands for basic types (e.g. integer, boolean, $X, \ldots$) not channel types.

In order to have bounded polymorphism also in the π-calculus, we should add it to the syntax of types, precisely attached to the labels of variant types as follows:

$$\langle l_1(X_1 \le B_1) : T_1 \ldots l_n(X_n \le B_n) : T_n\rangle$$

So, 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 that satisfies the condition. The syntax of processes should be modified accordingly, by adding the bound type to the labels. The typing rules are now similar on both calculi and the same holds for the operational semantics. The encoding is once again a homomorphism and is given in Figure 8.

By using the encoding and the bounded polymorphism in π-calculus, we can derive bounded polymorphism in session types. Furthermore, all the results presented in Section 4 and 5.1 are derivable for free.

6. Higher-Order

Higher-Order π-calculus (HOπ) models mobility of processes that can be sent and received and thus can be run locally [18]. Higher-order in sessions has the same benefits as that in 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.

Let us consider higher-order sessions [15].

The syntax of types is the following. We consider standard π channel types, session types and types taken from the simply-typed λ-calculus. The syntax T of types is extended with two functional type constructs: a standard one $T \to \diamond$, assigned to a

$$\begin{array}{lll} T ::= & \sharp T & \text{standard channel type} \\ & S & \text{session type} \\ & T \to \diamond & \text{functional type} \\ & T \xrightarrow{1} \diamond & \text{linear functional type} \\ S ::= & \ldots & \text{Session Types} \end{array}$$

Figure 9. Higher-order session types

functional term that can be used without any restriction and, a linear functional type $T \xrightarrow{1} \diamond$[1] 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, π-calculus with session primitives is augmented with call-by-value λ-calculus primitives, namely *abstraction* $(\lambda x : T.P)$ and *application* (PQ).

HOπ types, given in Figure 10, include the standard functional type and its terms include abstraction and application. The only type construct missing w.r.t. higher order session types is the linear functional type, which we add in order to properly define the encoding. The syntax of terms, on the contrary, is unchanged and

$$\begin{array}{lll} T ::= & \ell_i T & \text{linear input} \\ & \ell_o T & \text{linear output} \\ & \ell_\sharp T & \text{linear connection} \\ & \langle l_1\_T_1 \ldots l_n\_T_n\rangle & \text{variant type} \\ & T \to \diamond & \text{functional type} \\ & T \xrightarrow{1} \diamond & \text{linear functional type} \end{array}$$

Figure 10. Higher-order π types

remains the one of HOπ.

The encoding is a homomorphism on the higher-order constructs added to the syntax of types and terms on both calculi. It is presented in Figure 11.

$$\begin{array}{lcl} [\![T \xrightarrow{1} \diamond]\!] & = & [\![T]\!] \xrightarrow{1} \diamond \\ [\![T \to \diamond]\!] & = & [\![T]\!] \to \diamond \end{array}$$

$$\begin{array}{lcl} [\![\lambda x : T.P]\!]_f & = & \lambda x : [\![T]\!].[\![P]\!]_f \\ [\![PQ]\!]_f & = & [\![P]\!]_f [\![Q]\!]_f \end{array}$$

Figure 11. Encoding of higher-order types and terms

Typing judgements, in π-calculus with and without sessions are of the form $\Gamma; \Sigma; \mathcal{S} \vdash P$, where Σ denotes the set of session channels typed by session types, $\mathcal{S}$ is the set of linear functional variables and Γ contains the rest in order to type P.

Regarding the higher-order calculus, the assertion of type correctness becomes:

THEOREM 5. $\Gamma; \Sigma; \mathcal{S} \vdash P$ *if and only if* $[\![\Gamma]\!]_f; [\![\Sigma]\!]_f; [\![\mathcal{S}]\!]_f \vdash [\![P]\!]_f$

The result of the operational correspondence for the higher-order is as before. Again, we derive Subject Reduction, Type Safety and other Lemmas as corollaries.

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

[1] In [15] they consider generic functional types $T \to T$. However, this does not add much to the calculi and as long as our goal for encoding is concerned $T \to \diamond$ is enough.

$$\llbracket \&\{l_i(X_i \leq T_i) : S_i\}_{i\in I}\rrbracket = \ell_i[\langle l_1(X_1 \leq T_1) : \llbracket S_1\rrbracket, \ldots, l_n(X_n \leq T_n) : \llbracket S_n\rrbracket\rangle]$$
$$\llbracket \oplus\{l_i(X_i \leq T_i) : S_i\}_{i\in I}\rrbracket = \ell_o[\langle l_1(X_1 \leq T_1) : \llbracket \overline{S_1}\rrbracket, \ldots, l_n(X_n \leq T_n) : \llbracket \overline{S_n}\rrbracket\rangle]$$

$$\llbracket x \triangleleft l(T).P\rrbracket_f = (\nu c)\overline{f_x}\langle l(T)\_c\rangle.\llbracket P\rrbracket_{f,\{x\to c\}}$$
$$\begin{aligned}\llbracket x \triangleright \{l_1(X_1 \leq T_1) : P_1, \ldots, l_n(X_n \leq T_n) : P_n\}\rrbracket_f &= f_x(y).\ \texttt{case}\ y\ \texttt{of}\\ &\qquad [l_1(X_1 \leq T_1)\_c \Rightarrow \llbracket P_1\rrbracket_{f,\{x\to c\}}\\ &\qquad\qquad \ldots\\ &\qquad l_n(X_n \leq T_n)\_c \Rightarrow \llbracket P_n\rrbracket_{f,\{x\to c\}}]\end{aligned}$$

Figure 8. Encoding of polymorphic types and terms

order to satisfy this linearity, on behalf 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 an output process transmitting some value v:

$$(\nu b)\overline{a}\langle v, b\rangle.\llbracket P\rrbracket_{\{a\to b\}} \tag{1}$$

One can argue that there is an overhead in doing so, and above all it 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 optimized approach permits to reuse the same linear channel. For example, the above process would be as follows:

$$\overline{a}\langle v, a\rangle.\llbracket P\rrbracket \tag{2}$$

This leads to a typing problem, since the process is not well-typed, as it obviously violates linearity. In order to overcome this problem, we introduce the following typing rule:

$$\frac{\Gamma_1 \vdash v : \ell_o T \qquad \Gamma_2, v : \ell_c S \vdash w : T \qquad \Gamma_3, v : \ell_{\overline{c}} S \vdash P}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash \overline{v}\langle w\rangle.P}$$

We have proved that (1) and (2) are typed strong barbed congruent. The modified rule may be seen as an optimization of linear types, allowing reuse of channel names. The optimization would make the encoding of session types simpler— a linear channel would be use 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.

8. Conclusions, Related and Future Work

This paper proposes an interpretation of session types into ordinary π types, more precisely into *linear types* and *variant types*. Linear types [14] force a channel to be used exactly once. Variant types [18] are a labeled form of disjoint union of types.

The idea of the encoding of session types into π-calculus linear types is not new. Kobayashi [13] was the first to propose such an encoding, but he did not provide any formal study of it. Demangeon and Honda [2] provide a subtyping theory for a π-calculus augmented with branch and select constructs and show an encoding of the session calculus. They prove the soundness of the encoding and the full abstraction. The main differences w.r.t. our work are: (*i*) the target language is closer to the session 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 calculus in the standard π-calculus in order to exploit its rich and well-established theory; (*ii*) we study the encoding in a systematic way as a means to formally derive session types and all 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, [2] focuses on subtyping issues).

Other expressivity results regarding session types theory include the work by Caires and Pfenning [1]. They present a type system for the π-calculus that corresponds to the standard sequent calculus proof system for dual intuitionistic linear logic. They give an interpretation of intuitionistic linear logic formulas as a form of session types. These results are complemented and strengthened with a theory of logical relations [16]. Moreover an interpretation of the simply-typed λ-calculus in sessions π-calculus is given in [20].

Igarashi and Kobayashi [9] 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 [6] define an interpretation 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 passing through GTS would be more difficult than proving them directly.

We develop Kobayashi's proposal of an encoding of session types into ordinary π types. We show that the encoding is faithful, in that it allows us to derive all the basic properties of session types, exploiting the analogous properties of π types. We then show that the encoding is robust, by analyzing a few non-trivial extensions to session types, namely subtyping, polymorphism and higher-order. Finally, we propose an optimization of linear channels permitting the reuse of the same channel for the continuation of the communication and prove a typed barbed congruence result. This optimization considerably simplifies Kobayashi's encoding, which on some terms (for example, input and output processes) becomes the identity relation (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 (Subject Reduction, Type Safety, ...) as straightforward corollaries (thus eliminating redundancy also in the proofs). Issues like opposite endpoints of a session channel and duality of types assigned to these endpoints are handled by the theory of π: 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 allow us to easily obtain extensions of the session calculus, by exploiting the theory of the π-calculus. As we have shown in Section 5.2, where we presented the bounded polymorphism, our approach makes it easy even when the intended extension was not already present in the π-calculus. In these cases one can just provide the π-calculus with the intended capability and obtain the same capability in sessions. The whole process has shown to be much easier passing through π-calculus than doing it from scratch for sessions.

We conclude that session types theory is indeed derivable from the theory of 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.

We are planning to investigate whether our approach can be taken a step further, by modifying the encoding in order to accommodate notions of *causality* needed to capture multiparty communication behavior [8] and deadlock freedom [10, 12].

References

[1] L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *CONCUR'10*, pages 222–236, 2010.

[2] R. Demangeon and K. Honda. Full abstraction in a subtyped pi-calculus with linear types. In *CONCUR'11*, pages 280–296, 2011.

[3] M. Dezani-Ciancaglini and U. de'Liguoro. Sessions and session types: An overview. In *WS-FM'09*, pages 1–28, 2009.

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

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

[6] S. J. Gay, N. Gesbert, and A. Ravara. Session types as generic process types. In *PLACES'08*, 2008.

[7] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP'98*, pages 122–138, 1998.

[8] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL'08*, pages 273–284, 2008.

[9] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23 (3):396–450, 2001.

[10] N. Kobayashi. A partially deadlock-free typed process calculus. *ACM Trans. Program. Lang. Syst.*, 20(2):436–482, 1998. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/276393.278524.

[11] N. Kobayashi. Type systems for concurrent programs. In *10th Anniversary Colloquium of UNU/IIST*, pages 439–453, 2002.

[12] N. Kobayashi. A new type system for deadlock-free processes. In *CONCUR'06*, pages 233–247, 2006.

[13] N. Kobayashi. Type systems for concurrent programs. Extended version of [11], Tohoku University, 2007.

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

[15] D. Mostrous and N. Yoshida. Two session typing systems for higher-order mobile processes. In *TLCA'07*, pages 321–335, 2007.

[16] J. A. Pérez, L. Caires, F. Pfenning, and B. Toninho. Linear logical relations for session-based concurrency. In *ESOP'12*, pages 539–558, 2012.

[17] B. C. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. In *LICS'93*, pages 376–385, 1993.

[18] D. Sangiorgi and D. Walker. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press, 2001. ISBN 978-0-521-78177-0.

[19] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *PARLE'94*, pages 398–413, 1994.

[20] B. Toninho, L. Caires, and F. Pfenning. Functions as session-typed processes. In *FoSSaCS'12*, pages 346–360, 2012.

[21] V. T. Vasconcelos. Fundamentals of session types. In *To appear in Information and Computation*, volume 217, pages 52–70, 2012.

[22] N. Yoshida and V. T. 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.