The Concurrency Column

BY

Nobuko Yoshida

Department of Computing
Imperial College London
180 Queen's Gate, London, SW7 2AZ
`n.yoshida@imperial.ac.uk`, `http://mrg.doc.ic.ac.uk/`

Type Systems for Distributed Programs: Session Communication

Ornela Dardha

School of Computing Science, University of Glasgow
Ornela.Dardha@glasgow.ac.uk

Abstract

Distributed systems are everywhere around us and guaranteeing their correctness is of paramount importance. It is natural to expect that these systems interact and communicate among them to achieve a common task.

In this work, we develop techniques based on types and type systems for the verification of correctness, consistency and safety properties related to communication in complex distributed systems. We study advanced safety properties related to communication, like deadlock or lock freedom and progress. We study *session types* in the π- calculus describing distributed systems and communication-centric computation. Most importantly, we define an encoding of the session π- calculus into the standard typed π- calculus in order to understand the expressive power of these concurrent calculi. We show how to derive in the session π- calculus basic properties, like type safety or complex ones, like progress, by exploiting this encoding.

1 Introduction

Context and Motivation Complex software systems, in particular distributed ones, are everywhere around us and are at the basis of our everyday activities.

These systems are highly *mobile* and *dynamic*: programs or devices may move and may often execute in networks owned and operated by other parties; new devices or pieces of software may be added; the operating environment or the software requirements may change over time.

These systems are also *heterogeneous* and *open*: the pieces that form a system may be quite different from each other, built by different people or industries, even using different infrastructures or programming languages; the constituents of a system only have a partial knowledge of the overall system, and may only know, or be aware of, a subset of the entities that operate in the system.

The computational units of a software system, often referred to as *components*, are supposed to interact and communicate with each other following some predefined patterns or protocols. Hence, it is important to understand how correctness and safety criteria can be enforced. In the communication setting, the notion of safety comes as a collection of several requirements, including basic properties like *privacy*, guaranteeing that the communication means is owned only by the communicating parties, or *communication safety*, guaranteeing that the protocol has the expected structure. Stronger safety properties related to communication may be desirable such as *deadlock freedom*, guaranteeing that the system does not get stuck, or *progress*, guaranteeing that every engaged communication or protocol satisfies all the requested interactions. Enforcing each of the previous safety requirements is a difficult task, which becomes even more difficult if one wants to enforce a combination of them. In many distributed systems, in particular, safety critical systems, a combination of these properties is required.

Goals and Methodology The goal of this work is to develop powerful techniques based on formal methods for the verification of correctness, consistency and safety properties related to communication in complex distributed systems.

In particular, static analysis techniques based on types and type systems appear to be an adequate methodology, as they stand at the formal basis of useful programming tools. Before using them in a practical setting, a rigorous development of such techniques is needed, which is more easily done on models and core languages, such as concurrent calculi.

The reason why we have adopted types in our work is twofold.

i) Type systems are an adequate means to guarantee *safety properties*. Their benefits are well-known in sequential programming, starting from early detection of programming errors to facilitating code optimisation and readability. In concurrent and distributed programming the previous benefits still hold and in addition other properties, typical of these systems, can be guaranteed by using types and type systems. In particular, there has been a considerable effort over the last 20 years in the development of types for processes, mainly in the π-calculus [28, 33, 27, 35, 37] or variants of it, which is the calculus mostly used to model concurrent and distributed scenarios. For instance, types have been proposed to ensure termination, so that when we interrogate a well-typed process we are guaranteed that an answer is eventually produced [36, 26], or deadlock freedom, ensuring that a well-typed process never reaches a deadlocked state, meaning that communications will eventually succeed, unless the whole process diverges [25, 26], or a stronger property, that of lock freedom [23, 26] ensuring that communication of well-typed processes will succeed, (under fair scheduling), even if the whole process diverges.

ii) There are several types and type system proposals for *communication*, starting from the standard channel types in the typed π- calculus to the *behavioural types* [19, 38, 20, 18, 5, 40], generally defined for (variants) of the π-calculus. The standard channel types are foundational. They are simple, expressively powerful and robust and they are well-studied in the literature. Moreover, they are at the basis of behavioural types, which were defined later in time. In this paper we concentrate on the standard channel types, especially variant types and linear channel types [35, 27, 37] and on the *session types*, the latter being a formalism used to describe and model protocols for distributed systems as type abstractions. We focus on session types because they guarantee several safety properties, such as privacy of the communication channel, communication safety and session fidelity, ensuring that the type of the transmitted data and the structure of the session type are as expected. However, as previously stated, we are also interested in studying stronger properties, such as deadlock and lock freedom of communicating participants and progress of a session. Again, these properties can be guaranteed by using session types.

Contributions (and Structure of the Paper) The contributions of this paper are as follows.

- We present an encoding (§ 3) of the session typed π- calculus into the standard typed π- calculus, by showing that the type and term primitives of the former can be obtained by using the primitives of the latter. The goal of the encoding is to understand the expressive power of session types and to which extent they are more sophisticated and complex than the standard π-calculus types. The importance of the encoding is foundational, since
  - The encoding is proved faithful as it allows the derivation of properties of the session π- calculus, for e.g., subject reduction, by exploiting the theory of the standard typed π- calculus.
  - The encoding is proved robust by extending it to handle non trivial features like, subtyping (in § 4), polymorphism (in § 5) and higher-order communication (in § 6), and by using it to derive new properties in the session π- calculus due to these new features from the corresponding ones in the standard typed π- calculus.
  - The encoding is an expressiveness result for the standard π- calculus. There are many more expressiveness results in the untyped settings as opposed to expressiveness results in the typed ones.
- We study advanced safety properties related to communication in complex distributed systems. We concentrate on (dyadic) session types and study

properties like deadlock freedom, lock freedom and progress (§ 7). We study the relation among these properties (§ 7.1) and present a type system for guaranteeing the progress property by exploiting our encoding (§ 8).

The rest of this paper is organised as follows: we start with a background on the π-calculus in§ 2. We first present session types (§ 2.1) and then the standard types (§ 2.2). In § 9 we discuss the related work and we conclude in § 10.

Origin of the Results This work is based on the author's PhD thesis titled "*Type Systems for Distributed Programs: Components and Sessions*" [1] [10], and on previous published papers, which are joint works with Elena Giachino and Davide Sangiorgi [12] and Marco Carbone and Fabrizio Montesi [6]. The complete proofs of all the results presented here can be found in [10]. The author's PhD thesis studies session communication, presented in the remainder of this paper, and dynamic reconfiguration, which is not included here for space limits. The problem of dynamic reconfiguration, i.e., changing at runtime the communication patterns, is based on a joint work with Elena Giachino and Michaël Lienhardt [11]. We design a type system for a component-based, concurrent object-oriented calculus to statically ensure consistency and reliability of dynamic reconfigurations. The type system statically tracks runtime information about the objects, and it can be seen as a technique that can be applied to other calculi and frameworks for purposes related to tracking runtime information at compile time.

2 Background on the π-calculus

In this section we give an overview of the theory of the π-calculus with session types and standard types, focusing on linear channel types and variant types.

2.1 Session Types

Type Syntax Types are produced by two separate syntactic categories: one for session types and the other for standard π-types, including session types and presented in Fig. 1. Let S range over session types and T over types. Session types can be: end, the type of a terminated session; $?T.S$ and $!T.S$ indicating respectively session types used to receive and send a value of type T and proceed according to type S. Branch and select are sets of labelled session types. $\&\{l_i : S_i\}_{i\in I}$ indicates the external choice. Dually, $\oplus\{l_i : S_i\}_{i\in I}$ indicates the internal choice,

[1]Winner of the "Best Italian PhD thesis in Theoretical Computer Science 2015" awarded by the Italian Chapter of EATCS.

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

$$
\begin{array}{rll|ll}
P, Q ::= & x!\langle v\rangle.P & \text{(output)} & \mathbf{0} & \text{(inaction)} \\
 & x?(y).P & \text{(input)} & P \mid Q & \text{(composition)} \\
 & x \triangleleft l_j.P & \text{(selection)} & (\nu xy)P & \text{(session restriction)} \\
 & x \triangleright \{l_i : P_i\}_{i\in I} & \text{(branching)} & & \\
v ::= & x & \text{(variable)} & \star & \text{(unit value)}
\end{array}
$$

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

only one of the labels must be chosen. Types T include session types, standard channel types, `Unit` type and possibly other type constructs.

Language Syntax The syntax of terms is given in Fig. 1. Session communication occurs on *co-variables* [40], specifying 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. Let P, Q range over processes and v over values. A process can be an output $x!\langle v\rangle.P$ which sends a value v on channel x and proceeds as P; an input $x?(y).P$ which receives on channel x a value for the placeholder y in P; a selection $x \triangleleft l_j.P$ on x of label l_j and continuation P; an offering $x \triangleright \{l_i : P_i\}_{i\in I}$ on x of a set of processes labelled with labels l_i for $i \in I$; the terminated process $\mathbf{0}$; the parallel composition of two processes and the session restriction $(\nu xy)P$.

Duality A key notion of session types is *duality*, which relates opposite (i.e., complementary) behaviours. Duality stands at the basis of communication safety and session fidelity. Given a session type T, its dual type $\overline{T}$ is defined as follows:

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

Typing Rules Typing contexts, ranged over by Γ, Γ', are sets of typing assignments of the form $x : T$. We let $\mathtt{dom}(\Gamma)$ denote the domain of Γ. Given a typing context Γ and a process P, a typing judgement is of the form $\Gamma \vdash P$, meaning that P

$$\text{(T-Nil)}\ \frac{}{x : \mathsf{end} \vdash \mathbf{0}} \qquad \text{(T-Par)}\ \frac{\Gamma_1 \vdash P \qquad \Gamma_2 \vdash Q}{\Gamma_1 \circ \Gamma_2 \vdash P \mid Q} \qquad \text{(T-Res)}\ \frac{\Gamma, x : T, y : \overline{T} \vdash P}{\Gamma \vdash (\nu xy)P}$$

$$\text{(T-In)}\ \frac{\Gamma_1 \vdash x : ?T.S \qquad \Gamma_2, x : S, y : T \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash x?(y).P} \qquad \text{(T-Out)}\ \frac{\begin{array}{c}\Gamma_1 \vdash x : !T.S \\ \Gamma_2 \vdash v : T \qquad \Gamma_3, x : S \vdash P\end{array}}{\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash x!\langle v\rangle.P}$$

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

Figure 2: Typing rules for the π-calculus with session types

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

Figure 3: Semantics for the π-calculus with session types

is well typed in Γ. Typing rules are given in Fig. 2. Rule (T-Nil) states that the terminated process is well typed under a terminated channel. Rule (T-Par) types the parallel composition of two processes under the composition of the corresponding typing contexts, by using the context split operator $\circ$, which performs a combination of types and deals with linearity of session types [40]. Rule (T-Res) types the restriction process under the assumption that the endpoints of the restricted channel have dual types. Rules (T-In) and (T-Out) type the receiving and sending of a value over a channel x, respectively. Finally, rules (T-Brch) and (T-Sel) are generalisations of input and output over a labelled set of processes.

Operational Semantics The operational semantics is a binary relation $\longrightarrow$ over processes and is given in Fig. 3. Rule (R-Com) states that two processes communicate on two co-variables, and the value received replaces the input placeholder. Rule (R-Case) is similar: the communicating processes have prefixes that

$T ::=$	$\ell_{\mathtt{o}}\,[\widetilde{T}]$	(linear output)	$\langle l_i : T_i \rangle_{i\in I}$	(variant type)
	$\ell_{\mathtt{i}}\,[\widetilde{T}]$	(linear input)	$\sharp[\widetilde{T}]$	(connection)
	$\ell_{\sharp}\,[\widetilde{T}]$	(linear connection)	`Unit`	(unit type)
	$\emptyset[]$	(no capability)	$\dots$	(other constructors)

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

Figure 4: Syntax for the standard typed π-calculus

are co-variables, and the label received selects the continuation on the recipient side. Rules (R-Res), (R-Par), and (R-Struct) are standard, stating that communication can happen respectively, under restriction, parallel composition, and by using the *structural congruence* relation, which can be found in [40, 37].

Properties We recall some basic properties of the session type system [40]. The first lemma states type preservation of a process under structural congruence and the second theorem states type preservation of a process under reduction.

Lemma 1 (Type Preservation for $\equiv$). *If* $\Gamma \vdash P$ *and* $P \equiv P'$, *then* $\Gamma \vdash P'$.

Theorem 1 (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 polyadic π-calculus [37] and focus on linear types and variant types, which are used in the encoding. The syntax of types, ranged over by T, is given in Fig. 4. Linear types $\ell_{\mathtt{i}}\,[\widetilde{T}]$ and $\ell_{\mathtt{o}}\,[\widetilde{T}]$ denote channels used *exactly once* to receive and send values of type $\widetilde{T}$, respectively. $\ell_{\sharp}\,[\widetilde{T}]$ denotes both sending and receiving once values of type $\widetilde{T}$. We use α,β to range over the i, o or $\sharp$ capabilities. The variant type $\langle l_i : T_i \rangle_{i\in I}$ is a labelled form of disjoint union of types. Type $\emptyset[]$ denotes a channel that cannot be used for communication and type $\sharp T$ denotes a standard channel type. Other type constructs, like ground types and recursive types, can be added to the syntax.

$$\text{(T}\pi\text{-Inact)}\ \frac{}{x : \emptyset[] \vdash \mathbf{0}} \qquad \text{(T}\pi\text{-Par)}\ \frac{\Gamma_1 \vdash P \qquad \Gamma_2 \vdash Q}{\Gamma_1 \uplus \Gamma_2 \vdash P \mid Q} \qquad \text{(T}\pi\text{-Res1)}\ \frac{\Gamma, x : \ell_\sharp\,[\widetilde{T}] \vdash P}{\Gamma \vdash (\nu x)P} \qquad \text{(T}\pi\text{-Res2)}\ \frac{\Gamma \vdash P}{\Gamma \vdash (\nu x)P}$$

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

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

Figure 5: Typing rules for the standard typed π-calculus

We define a notion of duality on π-types by the following rules, which will be used in § 3.

$$\overline{\ell_{\mathtt{i}}\,[\widetilde{T}]} = \ell_{\mathtt{o}}\,[\widetilde{T}] \qquad \overline{\ell_{\mathtt{o}}\,[\widetilde{T}]} = \ell_{\mathtt{i}}\,[\widetilde{T}] \qquad \overline{\emptyset[]} = \emptyset[]$$

Language syntax The syntax of terms of the π-calculus is given in Fig. 4. A process is an output $x!\langle\tilde{v}\rangle.P$ which sends a tuple of values $\tilde{v}$ on channel x and proceeds as P; an input $x?(\tilde{y}).P$ which receives on x a tuple of values to substitute $\tilde{y}$ in P; a channel restriction $(\nu x)P$ which creates a new name x and binds it with scope P; the terminated process $\mathbf{0}$; the parallel composition of two processes P, Q and $\mathbf{case}\ v\ \mathbf{of}\ \{l_i\_(x_i) \triangleright P_i\}_{i\in I}$ which offers different behaviours depending on which variant value is received. Values are the unit value, names and variant values l_v .

Typing Rules The typing rules are given in Fig. 5. They make use of a commutative combination operator $\uplus$ which is defined on types as follows.

$$\begin{aligned} \ell_{\mathtt{i}}\,[\widetilde{T}] \uplus \ell_{\mathtt{o}}\,[\widetilde{T}] &\triangleq \ell_\sharp\,[\widetilde{T}] \\ T \uplus T &\triangleq T \\ T \uplus S &\triangleq \mathtt{undef} \quad \text{otherwise} \end{aligned}$$

The operation lifts naturally to typing contexts.

Rule (Tπ-Inact) states that the terminated process is well typed under a terminated channel. 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. Rule (Tπ-Res1) states that $(\nu x)P$ is well typed if P is well typed under the

$$\begin{array}{ll} (\text{R}\pi\text{-Com}) & x!\langle\tilde{v}\rangle.P \mid x?(\tilde{z}).Q \to P \mid Q[\tilde{v}/\tilde{z}] \\ (\text{R}\pi\text{-Case}) & \textbf{case}\, l_j\_v\, \textbf{of}\, \{l_i\_(x_i) \triangleright P_i\}_{i\in I} \to P_j[v/x_j] \quad j \in I \end{array}$$

Figure 6: Semantics for the standard typed π-calculus

same typing context augmented with $x : \ell_\sharp\, [\widetilde{T}]$. Rule (T$\pi$-Res2) states that $(\nu x)P$ is well typed if P is well typed under the same typing context. 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 $\tilde{v}$. A variant value l_v is of type $\langle l\_T\rangle$ if v is of type T. Process **case** v **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 T_i.

Operational Semantics The operational semantics for the standard π-calculus is given in Fig. 6. Rule (Rπ-Com) describes communication under the same channel x. Rule (Rπ-Case) states that **case** l_j_v **of** $\{l_i\_(x_i) \triangleright P_i\}_{i\in I}$ reduces to P_j, substituting x_j with the value v, if the label l_j is chosen. There are also rules (Rπ-Res) (Rπ-Par) and (Rπ-Struct) that capture reduction under restriction, parallel composition and structural congruence, respectively. Since they are standard we omit them.

Properties We recall some basic properties of the type system with linear π-types, taken from [37].

Lemma 2 (Type preservation for $\equiv$)**.** *If* $\Gamma \vdash P$ *and* $P \equiv P'$*, then* $\Gamma \vdash P'$.

Definition 1 (Closed typing context)**.** *A typing context is* closed *if* $\Gamma(x) \neq \ell_\sharp\, [\widetilde{T}]$*, for all* $x \in \mathtt{dom}(\Gamma)$.

Theorem 2 (Subject reduction for Linear Processes)**.** *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-types, we notice that since the typing context has no linear channel owning both capabilities (condition $\neq \ell_\sharp\, [\widetilde{T}]$), if a process reduces it is either due to a case reduction or a communication on a restricted channel owning both capabilities of input and output.

3 Encoding session types into standard π-types

Session types guarantee privacy, communication safety and session fidelity. The interpretation of session types into standard π-types should take into account these

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

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

Figure 7: Encoding of types and terms

fundamental properties. In order to guarantee privacy and communication safety 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. Session fidelity is guaranteed by adopting the so called *continuation-passing* style. We start by giving first the encoding of types and then the encoding of terms. We conclude the section by proving the correctness of the encoding w.r.t typing and reduction.

3.1 Encoding of Session Types

The encoding of session types into linear π-types is given in Fig. 7. The encoding of standard π-types is an homomorphism, i.e., $[\![\sharp T]\!] \triangleq \sharp[\![T]\!]$ and $[\![\mathtt{Unit}]\!] \triangleq \mathtt{Unit}$. Type end is a interpreted as a channel type with no capabilities, meaning that it cannot be used further for communications. $?T.S$ 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. In this case the continuation S is dualised for it to match the type of 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 $l_1 \ldots l_n$ and the encoding of $S_1 \ldots S_n$ and of $\overline{S_1} \ldots \overline{S_n}$, respectively. Again, the reason for duality is the same as for the output type.

Consider the dual types S and $\overline{S}$ from the introduction. Their encoding is

$$[\![S]\!] = \ell_{\mathtt{i}}\,[\mathtt{Int}, \ell_{\mathtt{i}}\,[\mathtt{Int}, \ell_{\mathtt{o}}\,[\mathtt{Unit}, \emptyset[]]]]$$

and

$$[\![\overline{S}]\!] = \ell_{\mathtt{o}}\,[\mathtt{Int}, \ell_{\mathtt{i}}\,[\mathtt{Int}, \ell_{\mathtt{o}}\,[\mathtt{Unit}, \emptyset[]]]]$$

Notice that duality on session types boils down to opposite capabilities of linear types only at the outermost level. The carried types are exactly the same due to the dualisation of the continuation type in the encoding of output and select.

3.2 Encoding of Session Processes

The encoding of the session π- calculus processes into the standard π-calculus processes is defined in Fig. 7. It is parametrised by a function f used to rename linear channels. Once a linear π- channel is used, it cannot be used again for transmission. To enforce session structure via the continuation-passing principle, the linear channels are renamed at every step of communication: a new channel is created and sent to continue the rest of the session. We use $\mathtt{dom}(f)$ to denote the domain of function f. We use f_x as an abbreviation for $f(x)$. Formally, the update of a function f is defined as follows:

$$f, \{x \mapsto c\} \triangleq \begin{cases} f \cup \{x \mapsto c\} & \text{if } x \notin \mathtt{dom}(f) \\ (f \setminus \{x \mapsto f_x\}) \cup \{x \mapsto c\} & \text{otherwise} \end{cases}$$

In the encoding of the output process, a new channel c is created and sent together with the payload v along the channel f_x; the encoding of the continuation process P is parametrised by 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 with x renamed to c. The encoding of a session restriction process $(\boldsymbol{\nu}xy)P$ is a linear channel restriction process $(\boldsymbol{\nu}c)[\![P]\!]_{f,\{x\mapsto c\}}$ with the new name c used to substitute x and y in the encoding of P. The selection process $x \triangleleft l.P$ 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 selected label. Channel c is going to be used for the next interaction. The encoding of a branching process receives on f_x a value, typically a variant value l_c, to substitute the placeholder 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. Note that the name c is bound in any process $[\![P_i]\!]_{f,\{x\to c\}}$ The encoding of the other processes is a homomorphism, namely $[\![\mathbf{0}]\!]_f \triangleq \mathbf{0}$ and $[\![P \mid Q]\!]_f \triangleq [\![P]\!]_f \mid [\![Q]\!]_f$.

We now consider a simple example: the equality test

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

The system is given by

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

The *server* and the *client* communicate on a session channel by owning an endpoint x and y, respectively. The *server* receives two integers in sequence v_1 and v_2 and sends back either $\star$ or `false` depending on whether these values are equal or not. The *client* behaves dually: it sends to the *server* two integers values 3 and 5 and waits for a boolean answer. After this communication, they both terminate. The encodings of *server* and *client* processes are:

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

Initially, by (E-Restriction) function f renames x and y to a new name z, and after that, before every output action, a new channel is created and sent to the partner together with the payload: first channel c, then c' and at the end c'' are created to accommodate the continuation of the communication. The endpoints x and y are respectively typed with S and $\overline{S}$ previously introduced and encoded.

3.3 Properties of the Encoding

In this section we show how to derive properties in the session π-calculus, like subject reduction, by using our encoding and the corresponding properties in the standard typed π-calculus. We first extend the encoding to typing contexts in the expected way:

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

We first present the soundness and completeness of the encoding w.r.t typing processes: a session process P is well typed in a session typing context Γ, if and only if the encoding of P is also well typed in the encoding of Γ.

Theorem 3. $[\![\Gamma]\!]_f \vdash [\![P]\!]_f$ *if and only if* $\Gamma \vdash P$.

We will show now the operational correspondence. This property states that the encoding of processes is sound and complete w.r.t the operational semantics of the π-calculus with and without sessions. Before stating the theorem, we introduce the notion of *evaluation context*, which is defined as follows.

Definition 2 (Evaluation Context)**.** *An* evaluation context *is a process with a hole* $[\cdot]$ *and is produced by the following grammar:*

$$\mathcal{E}[\cdot] ::= [\cdot] \mid (\nu xy)\,\mathcal{E}[\cdot]$$

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 property. It is given by the following theorem.

Theorem 4 (Operational Correspondence)**.** *Let P be a session process. The following hold.*

1. *If $P \to P'$ then $[\![P]\!]_f \to\hookrightarrow [\![P']\!]_f$,*
2. *there are $P', \mathcal{E}[\cdot]$ such that $\mathcal{E}[P] \to \mathcal{E}[P']$ and $Q \hookrightarrow [\![P']\!]_{f'}$, for some f' and $f_x = f_y$ for all x, y such that (νxy) appears in $\mathcal{E}[\cdot]$.*

The proofs of the above theorems can be found in [10].

3.4 Session Communication by Encoding

We now show how we can use the encoding and the properties from the linear π-calculus to derive the analogous properties in the π- calculus with session types, e.g., the subject reduction property. We start with an auxiliary lemma, stating that the encoding of processes is sound and complete w.r.t to structural equivalence.

Lemma 3. *$P \equiv P'$ if and only if $[\![P]\!]_f \equiv [\![P']\!]_f$.*

The following state how type preservation under $\equiv$ and subject reduction are obtained by using the encoding.

Proof of Lemma 1: If $\Gamma \vdash P$ and $P \equiv P'$, then $\Gamma \vdash P'$.

Proof. By Theorem 3, we have $[\![\Gamma]\!]_f \vdash [\![P]\!]_f$. By Lemma 3, $[\![P]\!]_f \equiv [\![P']\!]_f$ and, by Lemma 2, $[\![\Gamma]\!]_f \vdash [\![P']\!]_f$. We conclude by Theorem 3. □

Proof of Theorem 1: If $\Gamma \vdash P$ and $P \to P'$, then $\Gamma \vdash P'$.

Proof. Assume $\Gamma \vdash P$ and $P \to P'$. By Theorem 3 on completeness of encoding we have $[\![\Gamma]\!]_f \vdash [\![P]\!]_f$. By point *1.* of Theorem 4 we have $[\![P]\!]_f \to\hookrightarrow [\![P']\!]_f$. □

In the following we test our encoding on a few extensions: subtyping, parametric and bounded polymorphism and higher-order communication and show its robustness.

$$\frac{\widetilde{T} \leq \widetilde{T'}}{\ell_{\mathtt{i}}\ [\widetilde{T}] \leq \ell_{\mathtt{i}}\ [\widetilde{T'}]}\ (\text{S}\pi\text{-}\mathtt{ii}) \qquad \frac{\widetilde{T'} \leq \widetilde{T}}{\ell_{\mathtt{o}}\ [\widetilde{T}] \leq \ell_{\mathtt{o}}\ [\widetilde{T'}]}\ (\text{S}\pi\text{-}\mathtt{oo})$$

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

Figure 8: Subtyping rules for π types ($\leq$) and for session types ($<:$).

4 Subtyping

Subtyping relation on channel types has been studied for the standard π-types [33, 37] as well as session types [18]. In this section we show that the ordinary subtyping of the π-calculus is enough to derive subtyping in session types. Some of the subtyping rules for both systems are presented in Fig. 8. Rules (Sπ-ii) and (Sπ-oo) state that input channels are co-variant and output channels are contra-variant in the types of values they transmit. Rules (S-Inp) and (S-Out) state subtyping in input and output session types, respectively. As for linear π- types, the input is co-variant whilst the output is contra-variant.

In the session π-calculus with subtyping, one must deal both with subtyping on standard π-types and subtyping on session types. This introduces overhead in the theory, which becomes even more noticeable in the presence of recursive types, for example. We use the encoding, as in the previous section, to derive basic properties of session types, and remove the overhead in the theory. For Theorem 3 to remain valid, we have to take the subtyping relation into account. Therefore, it is important to prove the validity of subtyping w.r.t the encoding of types.

The following theorem states the soundness and completeness of the encoding of types w.r.t subtyping in session types and linear π- types.

Theorem 5. *$T <: T'$ if and only if $[\![T]\!] \leq [\![T']\!]$.*

The proof can be found in [10]. We can derive the main results on subtyping, such as subtyping being a preorder, as corollaries by using our encoding and Theorem 5, in the same way as we did with subject reduction in the previous section.

5 Polymorphism

In this section we study two forms of polymorphisms: parametric and bounded. Let us first consider parametric polymorphism.

5.1 Parametric Polymorphism

This form of polymorphism has not been studied in session types. The following syntax is an extension of the one presented in § 2.1.

$$
\begin{array}{rcll}
T & ::= & \dots \;\mid X & \text{(type variable)} \\
 & & \mid \langle X; T \rangle & \text{(polymorphic type)} \\
P & ::= & \dots \;\mid \textbf{open } v \textbf{ as } (X; x) \textbf{ in } P & \text{(unpack process)} \\
v & ::= & \dots \;\mid \langle T; v \rangle & \text{(polymorphic value)} \\
\Delta & ::= & \Delta, X \mid \emptyset & \text{(type variable environment)}
\end{array}
$$

We extend the syntax of types with the type variable X and the polymorphic type $\langle X; T \rangle$. The syntax of session types remains unchanged. We introduce the polymorphic value $\langle T; v \rangle$ and the unpack process **open** v **as** $(X; x)$ **in** P, the same constructs as in the polymorphic π-calculus [34, 37]. We consider an additional typing context Δ containing polymorphic type variables. We extend the syntax of types and terms of the standard typed π-calculus, presented in § 2.2, with the same polymorphic type and term constructs as above. The typing rules are similar on both calculi and the same holds for the operational semantics. They can be checked in [37, 10].

Since the syntax of session types is unchanged, the encoding of session types remains as before. The encoding of the added types and terms is an homomorphism and is given in the following.

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

The encoding of typing contexts is given by:

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

We encode Γ as in § 3.3 and on Δ the encoding is the identity function, since the encoding of type variables is the identity function.

To complete Theorem 3 it suffices to add the case for the polymorphic values and the unpack of processes using the following lemmas.

Lemma 4 (Typing Polymorphic Values by Encoding)**.** $\Gamma; \Delta \vdash \langle T'; v \rangle : \langle X; T \rangle$ *if and only if* $[\![\Gamma; \Delta]\!]_f \vdash [\![\langle T'; v \rangle]\!]_f : [\![\langle X; T \rangle]\!]$.

Lemma 5 (Typing Unpack by Encoding)**.** $\Gamma; \Delta \vdash$ **open** v **as** $(X; x)$ **in** P *if and only if* $[\![\Gamma; \Delta]\!]_f \vdash [\![\textbf{open } v \textbf{ as } (X; x) \textbf{ in } P]\!]_f$.

To complete Theorem 4 about the operational correspondence, it suffices to add the case for unpack in the expected way.

5.2 Bounded Polymorphism

We now consider bounded polymorphism, studied in [16]. It has not been studied in the standard typed π-calculus yet, hence we add it and show how we can derive bounded polymorphism in session types passing through π types. In [16] lower and upper bounds are added to branch and select labels. In this paper we adopt a simpler version and add only upper bounds to branch and select labels. Type B stands for basic types e.g., integer, boolean..., as opposed to channel types.

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

In order to have bounded polymorphism also in the standard π-calculus, we add upper bounds to labels in the variant type and the case process, as shown in the following.

$$\begin{array}{rcll}
T & ::= & \ldots & \mid\ \langle l_i(X_i \leq B_i)\_T_i\rangle_{i\in I} \quad \text{(bounded polymorphic variant)} \\
P & ::= & \ldots & \mid\ \mathbf{case}\ v\ \mathbf{of}\ \{l_i(X_i \leq B_i)\_x_i \triangleright P\}_{i\in I} \quad \text{(bounded polymorphic case)} \\
v & ::= & \ldots & \mid\ l(B)\_v \quad \text{(bounded polymorphic variant value)}
\end{array}$$

The typing rules are similar on both calculi and the same holds for the operational semantics. They can be checked in [16, 10]. The encoding is once again a homomorphism and the relevant cases are given in the following.

$$\llbracket \oplus\{l_i(X_i <: B_i) : T_i\}_{i\in I} \rrbracket \triangleq \ell_{\mathtt{o}}\ [\langle l_i(X_i \leq B_i)\_\overline{\llbracket T_i \rrbracket}\rangle_{i\in I}] \qquad \text{(E-BPolySel)}$$

$$\llbracket \&\{l_i(X_i <: B_i) : T_i\}_{i\in I} \rrbracket \triangleq \ell_{\mathtt{i}}\ [\langle l_i(X_i \leq B_i)\_\llbracket T_i \rrbracket\rangle_{i\in I}] \qquad \text{(E-BPolyBrch)}$$

(E-BPolySelection)

$$\llbracket x \triangleleft l_j(B).P \rrbracket_f \triangleq (\nu c) f_x!\langle l_j(B)\_c\rangle.\llbracket P \rrbracket_{f,\{x\mapsto c\}}$$

(E-BPolyBranching)

$$\llbracket x \triangleright \{l_i(X_i <: B_i) : P_i\}_{i\in I} \rrbracket_f \triangleq f_x?(y).\ \mathbf{case}\ y\ \mathbf{of}\ \{l_i(X_i \leq B_i)\_c \triangleright \llbracket P_i \rrbracket_{f,\{x\mapsto c\}}\}_{i\in I}$$

By using the encoding and the bounded polymorphism in the standard π-calculus, we can derive bounded polymorphism in the session π- calculus. Again, properties like subject reduction and others related to polymorphism are derived as corollaries. To complete Theorem 3 and Theorem 4, it suffices to add the case for bounded branching and selection. The modifications to the typing judgements are as in parametric polymorphism.

Lemma 6 (Soundness)**.** *If* $[\![\Gamma;\Delta]\!]_f \vdash [\![Q]\!]_f$*, then* $\Gamma;\Delta \vdash Q$*, where either* $Q = x \triangleleft l_j(B).P$*, or* $Q = x \triangleright \{l_i(X_i \leq B_i) : P_i\}_{i\in I}$*.*

Lemma 7 (Completeness)**.** *If* $\Gamma;\Delta \vdash Q$*, then* $[\![\Gamma;\Delta]\!]_f \vdash [\![Q]\!]_f$*, where either* $Q = x \triangleleft l_j(B).P$*, or* $Q = x \triangleright \{l_i(X_i \leq B_i) : P_i\}_{i\in I}$

6 Higher-Order Communication

Higher-Order π-calculus (HOπ) models mobility of processes that can be sent and received and thus can be run locally [37]. Higher-order session π-calculus has the same benefits. In this section we use HOπ to provide sessions with higher-order communication by exploiting the encoding, as we did in the previous sections.

The syntax of types and terms for the HOπ with sessions [29] and without sessions [37] is given by the following grammar.

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

We 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 $T \xrightarrow{1} \sigma$, assigned to a term that should be used exactly once. The reason for linear functional types is that a function may contain free session channels. We extend the syntax of processes with call-by-value λ-calculus primitives, namely abstraction ($\lambda x : T.P$) and application (PQ).

The encoding of types and terms is a homomorphism on the higher-order constructs added on both calculi. Note that there is also a modification in the rule (E-Output). A value can be an abstraction, hence it needs to be encoded.

$$\begin{array}{rcll}
[\![T \xrightarrow{1} \sigma]\!] & \triangleq & [\![T]\!] \xrightarrow{1} [\![\sigma]\!] & \text{(E-LinFunType)} \\
[\![T \to \sigma]\!] & \triangleq & [\![T]\!] \to [\![\sigma]\!] & \text{(E-FunType)} \\
[\![\lambda x : T.P]\!]_f & \triangleq & \lambda x : [\![T]\!].[\![P]\!]_f & \text{(E-Abstraction)} \\
[\![PQ]\!]_f & \triangleq & [\![P]\!]_f[\![Q]\!]_f & \text{(E-Application)} \\
[\![x!\langle v\rangle.P]\!]_f & \triangleq & (\nu c)f_x!\langle [\![v]\!]_f, c\rangle.[\![P]\!]_{f,\{x\mapsto c\}} & \text{(E-Output)}
\end{array}$$

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

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

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

Figure 9: Typing rules for the HOπ with and without sessions

Typing judgements, in π-calculus with and without sessions are of the form $\Phi; \Gamma; \mathcal{S} \vdash v : T$, where Φ associates variables to value types, except session types; Γ associates variables to session types; $\mathcal{S}$ denotes the set of linear functional variables. A typing judgement is well-formed if $\mathcal{S} \subseteq \mathtt{dom}(\Phi)$ and $\mathtt{dom}(\Phi) \cap \mathtt{dom}(\Gamma) = \emptyset$. The new typing rules are presented in Fig. 9. We present the encoding of typing contexts in the following.

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

The following result gives the correctness of the encoding w.r.t typing for higher-order communication. The proof can be found in [10].

Theorem 6. $[\![\Phi; \Gamma; \mathcal{S}]\!]_f \vdash [\![P]\!]_f : [\![\sigma]\!]$ *if and only if* $\Phi; \Gamma; \mathcal{S} \vdash P : \sigma$.

The result of the operational correspondence for the higher-order communication is, as before, given by Theorem 4.

7 Progress and Lock Freedom for Sessions

The notion of *progress* is fundamental for safe programs. Intuitively, it means that a safe program never gets "stuck". The most basic property related to progress in concurrency is *deadlock freedom*: a process is deadlock-free if all its communications are eventually performed, unless the whole process diverges [25, 26].

Observe that in a deadlock-free process some subprocesses can get stuck. For instance, consider the following process:

$$P \;=\; (\boldsymbol{\nu} x)(x?(y).\mathbf{0} \mid \Omega)$$

where Ω is a diverging process executing an infinite series of internal actions. Even though the subterm $x?(y).\mathbf{0}$ will never reduce, process P is deadlock-free. In order to cope with this limitation of the deadlock freedom property, *lock freedom* has been proposed as a stronger property that requires every input/output action to be eventually executed under fair process scheduling, even if the whole process diverges [23, 26]. Different techniques have been proposed for guaranteeing deadlock- and lock freedom, mostly based on type systems [23, 25, 26].

In this section we study progress and lock freedom in the session π-calculus and their relation. Later on, we will use our encoding in order to obtain progress from the standard π-calculus. We start with some definitions. Below, we assume that reduction sequences are *fair*, as formalised in [23].

Definition 3 (Lock-Freedom for Session π-Calculus)**.** *A process P_0 is* lock-free *if for any fair reduction sequence $P_0 \to P_1 \to P_2 \to \ldots$, we have*

1. $P_i \equiv (\boldsymbol{\nu}\widetilde{xy})(\overline{x}!\langle v\rangle.Q \mid R)$, *for* $i \geq 0$, *implies that there exists* $n \geq i$ *such that* $P_n \equiv (\boldsymbol{\nu}\widetilde{x'y'})(\overline{x}!\langle v\rangle.Q \mid y?(z).R_1 \mid R_2)$ *and* $P_{n+1} \equiv (\boldsymbol{\nu}\widetilde{x'y'})(Q \mid R_1[v/z] \mid R_2)$*;*
2. $P_i \equiv (\boldsymbol{\nu}\widetilde{xy})(x \triangleleft l_j.Q \mid R)$, *for some* $i \geq 0$, *implies that there exists* $n \geq i$ *such that* $P_n \equiv (\boldsymbol{\nu}\widetilde{x'y'})(x \triangleleft l_j.Q \mid y \triangleright \{l_k : R_k\}_{k \in I \cup \{j\}} \mid S)$ *and* $P_{n+1} \equiv (\boldsymbol{\nu}\widetilde{x'y'})(Q \mid R_j \mid S)$.

For simplicity, above we have omitted the cases for input and branching, which have the expected definitions.

In order to give the formal definition for the progress property, we need some auxiliary notions. An *evaluation context* is a process with a hole $[\cdot]$ and is produced by the following grammar:

$$\mathcal{E}[\cdot] ::= \quad [\cdot] \mid P \quad \mid \quad (\boldsymbol{\nu} xy)\, \mathcal{E}[\cdot] \quad \mid \quad \mathcal{E}[\cdot] \mid \mathcal{E}[\cdot]$$

Given a type T, its *characteristic process* $[\![T]\!]^x$ is the simplest process that can inhabit a type and is inductively defined on the structure of T. We present some cases in the following (the full definition is given in [7, 6]).

$$\begin{array}{rl}
(\textsc{inVal}) & [\![?\texttt{Unit}.S]\!]^x = x?(y).[\![S]\!]^x \\
(\textsc{inSess}) & [\![?T.S]\!]^x = x?(y).([\![S]\!]^x \mid [\![T]\!]^y) \\
(\textsc{outSess}) & [\![!T.S]\!]^x = (\boldsymbol{\nu} zw)(x!\langle z\rangle.([\![S]\!]^x \mid [\![\overline{T}]\!]^w))
\end{array}$$

Finally, a *catalyser* is a context with only characteristic processes.

Definition 4 (Catalyser). *A* catalyser $\mathcal{C}[\cdot]$ *is a context produced by the following grammar:*

$$\mathcal{C}[\cdot] ::= [\cdot] \quad | \quad (\nu xy)\,\mathcal{C}[\cdot] \quad | \quad \mathcal{C}[\cdot] \mid [\![T]\!]^x$$

We illustrate the catalysers by the following example.

Example 1. Consider

$$\begin{aligned} \mathcal{C}[\cdot] &= (\nu wx)(\nu uy)([\cdot] \mid P_1 \mid P_2) \\ P_1 &= x?(z).(z!\langle\star\rangle.\mathbf{0} \mid \mathbf{0}) \\ P_2 &= y \triangleleft l_2.y!\langle\star\rangle.\mathbf{0} \end{aligned}$$

The context $\mathcal{C}[\cdot]$ is a catalyser obtained by composing the characteristic processes P_1 and P_2 of the session types $T_1 = ?(!\texttt{Unit}.\mathsf{end}).\mathsf{end}$ and $T_2 = \oplus\{l_1 : \mathsf{end}, l_2 : !\texttt{Unit}.\mathsf{end}\}$, respectively. □

Finally, we define $\bowtie_{\{x,y\}}$, a binary relation over processes which relates processes prefixed by co-actions. This operator, differently from the original one in [3], is parametrised by a pair of variables $\{x, y\}$, which are co-variables.

Definition 5 ($\bowtie_{\{x,y\}}$). *The duality* $\bowtie_{\{x,y\}}$ *between input and output processes is defined as follows:*

$$\begin{aligned} &x!\langle v\rangle.P \bowtie_{\{x,y\}} y?(z).Q \\ &x \triangleleft \{l_i : P_i\}_{i\in I} \bowtie_{\{x,y\}} y \triangleright \{l_i : Q_i\}_{i\in I} \end{aligned}$$

We are now ready to give the formal definition of progress, based on [3, 7].

Definition 6 (Progress). *A process P has* progress *if for all $\mathcal{C}[\cdot]$ such that $\mathcal{C}[P]$ is well typed, $\mathcal{C}[P] \rightarrow^* \mathcal{E}[R]$ (where R is an input or an output) implies that there exist $\mathcal{C}'[\cdot]$, $\mathcal{E}'[\cdot][\cdot]$ and R' such that $\mathcal{C}'[\mathcal{E}[R]] \rightarrow^* \mathcal{E}'[R][R']$ and $R \bowtie_{\{x,y\}} R'$ for some x and y such that (νxy) is a restriction in $\mathcal{C}'[\mathcal{E}[R]]$.*

7.1 Progress meets Lock Freedom

In this section we study the relation between progress and lock freedom for the session π-calculus: first for closed terms and later for open ones.

Properties of Closed Terms Lock freedom and progress are tightly related for closed terms, i.e., processes with no free variables, as the following shows.

Theorem 7 (Lock freedom and Progress). *Let P be a session-typed closed process. Then, P lock-free if and only if P has progress.*

It follows as a corollary from Theorem 7 that lock freedom and progress properties coincide for closed terms.

Corollary 1 (Progress ⇔ Lock freedom)**.** *Let P be a well-typed closed process. Then P is lock-free if and only if P has progress.*

Properties of open terms Differently from the case of the closed terms, lock freedom and progress intuitively do not coincide in the case of open terms.

For example, consider the following process:

$$P \; = \; x!\langle\star\rangle.x?(z).\mathbf{0}$$

where x is an open session with a missing participant. Process P has progress, by following Definition 6 but it is locked since it does not respect Definition 3.

Although the two properties do not coincide in the case of open terms, we can still relate progress to lock freedom. The idea is to use catalysers in order to reduce the problem of checking progress for open terms to the problem of checking progress (and lock freedom) for closed terms. The intuition for using catalysers is that when a process is open, its type can provide us with some information about how such a process can be put in a context such that the final composition is closed. We formalise this idea with the notion of *closure* given below.

Definition 7 (Closure)**.** *Let P be such that* $\Gamma \vdash P$*. Then, the* closure *of P, denoted by* $\mathtt{close}(P)$*, is the process* $C[P]$ *where*

$$C[\cdot] = (\boldsymbol{\nu}\widetilde{xy})(\,[\cdot] \mid \prod_{x_i:T_i\in\Gamma} [\![T_i]\!]^{y_i})$$

Notice that, in the definition above all x_i in the sequence $\widetilde{xy}$ correspond exactly to the domain of Γ. The y_i in $\widetilde{xy}$ are all different from x_i and are the variables used to create the characteristic processes from every type T_i. Below, we give an example of how the closure of a process works.

Example 2. Consider the open process previously shown

$$P = x!\langle\star\rangle.x?(z).\mathbf{0}$$

We can type P in a typing context $\Gamma = x : !\mathtt{Unit}.?\mathtt{Unit}.\mathtt{end}$. Then, the closure of P is defined as:

$$\mathtt{close}(P) = (\boldsymbol{\nu}xy)([P] \mid y?(z).y!\langle\star\rangle.\mathbf{0})$$

□

We now establish that checking the progress property for a process P is equivalent to checking the progress property for its closure:

Theorem 8 (Closure Progress ⇔ Progress). *If P is a session-typed process, then* `close`(P) *has progress if and only if P has progress.*

Our main result is that the progress property of a process P and the lock freedom property of the closure of P coincide:

Theorem 9. (Progress ⇔ Closed Lock-Free) *If P is well typed then P has progress if and only if* `close`(P) *is lock-free.*

Proof. It follows immediately from Theorem 8 and Corollary 1. □

8 Progress by Encoding and Lock Freedom

In this section we will describe a new way of guaranteeing progress in session π-calculus, namely by using our encoding of session types into linear types and the type system for lock freedom in the standard π-calculus [23, 26]. We start by giving a background on the new types and type system for lock freedom.

Usage Types by Kobayashi

$U ::=$	$\mathsf{i}^{o}_{\kappa}.U$	(used in input)	$\emptyset$	(not usable)
	$\mathsf{o}^{o}_{\kappa}.U$	(used in output)	$(U_1 \mid U_2)$	(used in parallel)
$T ::=$	$U[\widetilde{T}]$	(channel types)	$\langle l_i : T_i\rangle_{i\in I}$	(variant type)

Let α range over input i or output o actions, where for simplicity, we have removed ℓ. The usage $\emptyset$ describes a channel that cannot be used at all, and we will often omit it. Usages $\mathsf{i}^{o}_{\kappa}.U$ and $\mathsf{o}^{o}_{\kappa}.U$ describe channels that can be used once for input and output, respectively and then used according to the continuation usage U. The *obligation* o and *capability* κ range over the set of natural numbers. The usage $U_1 \mid U_2$ describes a channel that is used according to U_1 by one process and U_2 by another processes in parallel.

Obligation and capability describe inter-channel dependencies. Citing [23, 25, 26], their relation may be described as:

- An obligation of level n must be fulfilled by using only capabilities of level *less than n*. Said differently, an action of obligation n must be prefixed by actions of capabilities less than n.

- For an action with capability of level n, there must exist a co-action with obligation of level *less than or equal to n*.

Before commenting on the typing rules, we present some auxiliary notions. First, the composition operation on types, denoted $\mid$, is based on the composition of usages and is defined as follows:

$$\langle l_i : T_i \rangle_{i\in I} \mid \langle l_i : T_i \rangle_{i\in I} = \langle l_i : T_i \rangle_{i\in I} \qquad U_1[\widetilde{T}] \mid U_2[\widetilde{T}] = (U_1 \mid U_2)[\widetilde{T}]$$

The generalisation of $\mid$ to typing contexts, denoted $(\Gamma_1 \mid \Gamma_2)(x)$, is defined as expected. The unary operation $\uparrow^t$ applied to usage U lifts its obligation level *up to t*; it is inductively defined as:

$$\uparrow^t \emptyset = \emptyset \qquad \uparrow^t \alpha^o_\kappa.U = \alpha^{max(o,t)}_\kappa.U \qquad \uparrow^t (U_1 \mid U_2) = (\uparrow^t U_1 \mid \uparrow^t U_2)$$

The $\uparrow^t$ is extended to types and typing contexts as expected. The notion of *duality* on usage types is the same as the one presented in § 2.2 on linear types. Operator " ; " in $\Delta = x : [T]\alpha^o_\kappa \,;\, \Gamma$, used in rules (Tπ-In) and (Tπ-Out), is such that the following hold, where $y \neq x$:

$$\mathtt{dom}(\Delta) = \{x\} \cup \mathtt{dom}(\Gamma) \qquad \Delta(x) = \begin{cases} \alpha^o_\kappa.U[\widetilde{T}] & \text{if } \Gamma(x) = U[\widetilde{T}] \\ \alpha^o_\kappa[\widetilde{T}] & \text{if } x \notin dom(\Gamma) \end{cases} \qquad \Delta(y) = \uparrow^{\kappa+1} \Gamma(y)$$

The final required notion is that of a *reliable usage*. It builds upon the following definition:

Definition 8. *Let U be a usage. The input and output* obligation levels *(resp.* capability levels*) of U, written $\mathsf{ob_i}(U)$ and $\mathsf{ob_o}(U)$ (resp. $\mathsf{cap_i}(U)$ and $\mathsf{cap_o}(U)$), are defined as follows:*

$$\begin{array}{rclcrcl} \mathsf{ob}_\alpha(\alpha^o_\kappa.U) & = & o & \quad & \mathsf{ob}_\alpha(U_1 \mid U_2) & = & min(\mathsf{ob}_\alpha(U_1), \mathsf{ob}_\alpha(U_2)) \\ \mathsf{cap}_\alpha(\alpha^o_\kappa.U) & = & \kappa & & \mathsf{cap}_\alpha(U_1 \mid U_2) & = & min(\mathsf{cap}_\alpha(U_1), \mathsf{cap}_\alpha(U_2)) \end{array}$$

The definition of reliable usages depends on a reduction relation on usages, noted $U \rightarrow U'$. Intuitively, $U \rightarrow U'$ means that if a channel of usage U is used for communication, then after the communication occurs, the channel should be used according to usage U'. Thus, e.g., $\mathsf{i}^o_\kappa.U_1 \mid \mathsf{i}^{o'}_{\kappa'}.U_2$ reduces to $U_1 \mid U_2$.

Definition 9 (Reliability)**.** *We write $\mathsf{con}_\alpha(U)$ when $\mathsf{ob}_{\overline{\alpha}}(U) \leq \mathsf{cap}_\alpha(U)$. We write $\mathsf{con}(U)$ when $\mathsf{con_i}(U)$ and $\mathsf{con_o}(U)$ hold. Usage U is* reliable*, noted $\mathsf{rel}(U)$, if $\mathsf{con}(U')$ holds $\forall U'$ such that $U \rightarrow^* U'$.*

Typing Rules The typing rules for the standard π- calculus with usage types are in Fig. 10, and are taken from Kobayashi's work [26]. Rule (Uπ-Res) states that process $(\nu x)P$ is well typed if the usage for x is reliable (cf. Definition 9). Rule (Uπ-Par) states that the parallel composition of processes is well typed in the

$$\text{(U}\pi\text{-R}\text{ES)}\quad \frac{\Gamma, x : U[\widetilde{T}] \vdash P \quad \mathsf{rel}(U)}{\Gamma \vdash (\nu x)P} \qquad \text{(U}\pi\text{-P}\text{AR)}\quad \frac{\Gamma_1 \vdash P \qquad \Gamma_2 \vdash Q}{\Gamma_1 \mid \Gamma_2 \vdash P \mid Q}$$

$$\text{(U}\pi\text{-I}\text{N)}\quad \frac{\Gamma, \widetilde{y} : \widetilde{T} \vdash P}{x : \mathsf{i}^0_\kappa[\widetilde{T}] \,;\, \Gamma \vdash x?(\tilde{y}).P} \qquad \text{(U}\pi\text{-O}\text{UT)}\quad \frac{\Gamma_1 \vdash \tilde{v} : \widetilde{T} \qquad \Gamma_2 \vdash P}{x : \mathsf{o}^0_\kappa[\widetilde{T}] \,;\, (\Gamma_1 \mid \Gamma_2) \vdash x!\langle\tilde{v}\rangle.P}$$

Figure 10: Typing rules for the π-calculus with usage types

composition of their corresponding typing contexts. Rules (Uπ-In) and (Tπ-Out) type input and output processes in a typing context where the " ; " operator is used in order to increase the obligation level of the channels in continuation P. The rest of the rules is the same as in Fig. 5.

The next theorems imply that well-typed processes by the type system in Fig. 10 are deadlock-free.

Theorem 10 (Type Preservation for Usage Types)**.** *If* $\Gamma \vdash P$ *and* $P \rightarrow Q$*, then* $\Gamma' \vdash Q$ *for some* Γ' *such that* $\Gamma \rightarrow \Gamma'$.

Theorem 11 (Lock Freedom)**.** *If* $\emptyset \vdash P$*, then* $P \rightarrow Q$ *for some* Q.

Corollary 2. *If* $\emptyset \vdash P$*, then* P *is lock-free.*

A Type System for Progress In this section we use the encoding of session types into usage types, the results of the previous section relating progress and lock freedom and the type system for lock freedom by Kobayashi to guarantee progress in the π-calculus with sessions. The encoding of session types into usage types is the same as in Fig. 7.

We start with an auxiliary lemma which follow directly from the encoding and the definition of lock freedom in the π-calculus with and without sessions.

Lemma 8. *A session process* P *is lock-free if and only if* $[\![P]\!]_f$ *is lock-free.*

Theorem 12 (Progress in Sessions)**.** *Let* P *be a session process such that* $\Gamma \vdash P$*. If* $\emptyset \vdash [\![\texttt{close}(P)]\!]_f$*, then* P *has progress.*

Proof. Let $\Gamma \vdash P$ and $\emptyset \vdash [\![\texttt{close}(P)]\!]_f$. Then, by Theorem 11 and Corollary 2 this means that $[\![\texttt{close}(P)]\!]_f$ is lock-free. By Lemma 8 also $\texttt{close}(P)$ is lock-free. By Theorem 9 we have that P has progress. □

To conclude, we consider below a corner-case example to illustrate our technique and also to compare it with already existing works in the literature.

Example 3. Consider the session process

$$(\nu a_1 a_2)(\nu b_1 b_2)\Big(\; a_1?(x).\, b_1!\langle x\rangle.\, b_1?(y).\, a_1!\langle y\rangle \mid a_2!\langle\star\rangle.\, b_2?(z).\, b_2!\langle\star\rangle.\, a_2?(z)\;\Big)$$

This process satisfies the progress property, but it is rejected by the type systems in [3] and [7]. This is because, in the two processes in parallel, there is a circular dependency between channels that such type systems cannot handle. Let us now consider its encoding in the π-calculus, given as the process:

$$(\nu a)(\nu b)\left(\begin{array}{l} a?(x,c_1).\,(\nu c_2)\big(\, b!\langle x,c_2\rangle.\, c_2?(y).\, c_1!\langle y\rangle\,\big) \mid \\ (\nu c_1)\big(a!\langle\star,c_1\rangle.\, b?(z,c_2).\, c_2!\langle\star\rangle.\, c_1?(z)\big) \end{array}\right)$$

This process is well typed in Kobayashi's type system. The types assigned to the channels are: $a : [\mathtt{Unit}, T_1]\ \mathsf{i}^0_0 \mid \mathsf{o}^0_0$ and $b : [\mathtt{Unit}, T_2]\ \mathsf{o}^1_1 \mid \mathsf{i}^1_1$ such that $T_1 = [\mathtt{Unit}]\ \mathsf{o}^1_3 \mid \mathsf{i}^3_1$ and $T_2 = [\mathtt{Unit}]\ \mathsf{i}^2_0 \mid \mathsf{o}^0_2$. Hence, we conclude that it has progress. □

9 Related Work

Encoding and Expressiveness Results The idea of encoding session types into π-calculus linear types is not new. Kobayashi [26] was the first to propose such an encoding, but he did not prove any properties and did not investigate its robustness; moreover, as certain key features of session types do not clearly show up in the encoding, like duality, the faithfulness of the encoding was unclear. Later on, Dardha et al. [12] studied such encoding by showing its soundness and completeness w.r.t typing and reduction. Advanced features, such as subtyping, polymorphism and higher-order communication are introduced to prove the robustness of the encoding. In [10, 9], the author investigates also recursion. The interesting part of [9] is the use of the *complement* function as opposed to the *inductive duality* function $\bar{\cdot}$ since the latter does not commute with the unfolding of recursive session types, as shown in [2, 1]. Demangeon and Honda [14] 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 and full abstraction of the encoding. The main differences w.r.t our work are: *i*) the target language is closer to the session π-calculus having branch and select constructs, and a refined subtyping theory is provided; instead, we use the variant construct and focus on the encoding of the session π-calculus into the standard typed π-calculus in order to exploit the 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 need to establish the underlying theory.

Other expressiveness results regarding session types include the work by Caires and Pfenning [4]. 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 propositions as a form of session types. Another work is Wadler's [41] that, following [4], proposes a calculus where propositions of classical linear logic correspond to session types.

Igarashi and Kobayashi [22] have developed a generic type system (GTS) for the π-calculus from which several type systems can be obtained as instances of the generic one, by varying certain parameters. Gay, Gesbert and Ravara [17] 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.

Progress, Deadlock Freedom and Lock Freedom Progress for session π-calculus has been studied by several works [15, 3, 7, 30, 8, 6]. In [30] the author defines a session type system for the progress property by using Kobayashi's obligation and capability levels. In [15, 3, 8] progress is studied for multiparty session types. Padovani studies the linear π-calculus and the encoding of session types into linear π-calculus types to define a type reconstruction algorithm for session types in [32] and to guarantee deadlock freedom for session π-processes in [31]. A very recent work [13] studies the formal relationship between the class of deadlock free session processes induced by the correspondence of session types with linear logic [4] and the class of deadlock free session processes induced by the encoding and Kobayashi's type system for deadlock freedom [25].

10 Conclusions and Future Work

This paper is based on the works by Dardha et al. [10, 12, 6]. It first studies an encoding of the session π-calculus into the standard π-calculus. This encoding was first proposed by Kobayashi [26] and later on studied by Dardha et al. [12, 10]. It uses *linear types*, *variant types* and the *continuation-passing* principle. Linear types [27] force a channel to be used exactly once. Variant types [35, 37] are a labelled form of disjoint union of types. We show that the encoding is faithful, in that it allows us to derive properties of the session π-calculus by exploiting the corresponding ones of the linear π-calculus. 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 test our encoding on more advanced properties, by following [6, 10], such as progress and lock freedom. In this matter, we start by proving that, for closed terms, i.e., terms with no free variables, lock freedom and progress coincide on the session π- calculus. For open terms, i.e., terms containing free variables, we show that these notions do not coincide. However, we define a procedure for *closing* a process by using the notions of catalyser and characteristic process. Then, we prove that progress and lock freedom coincide for $\texttt{close}(P)$, which implies progress for P, thus making progress a compositional form of lock freedom. Ultimately, we use our encoding and the type system for lock freedom in standard π-calculus [23], to obtain a more accurate analysis of progress in the session π-calculus, as shown in [6, 10].

The benefits of the encoding include the elimination of the redundancy introduced both in the syntax of types and terms in the session π- calculus, and the derivation of properties (such as subject reduction) as straightforward corollaries by exploiting the corresponding ones in the standard typed π-calculus, thus eliminating also the redundancy in the proofs. 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, which permits large reusability. In particular, this holds for more advanced properties like progress and lock freedom.

As future work, we would like to investigate whether our approach can be taken a step further, by accommodating the notion of *causality* needed to capture multiparty communication behaviour [21]. Next, we would like to implement a tool that given in input a session process, performs the encoding into standard typed π- calculus process and uses Kobayashi's tool TyPiCal [39] to check for lock freedom. Hence, by our theoretical results on the matter, we can conclude if the original process satisfies the progress property or not.

Acknowledgements I am grateful to Davide Sangiorgi, who supervised my PhD thesis on which this paper is based. The research for this work was carried out while I was a PhD student at the University of Bologna and part of the INRIA Focus Team, and during my one-year visit at the IT University of Copenhagen, hosted by Marco Carbone. I am also thankful to all my co-authors mentioned in the introduction, from whom I learned a lot.

Currently, I am a postdoc at the University of Glasgow and supported by the UK EPSRC project *From Data Types to Session Types: A Basis for Concurrency and Distribution (ABCD)* (EP/K034413/1).

References

[1] Giovanni Bernardi, Ornela Dardha, Simon J. Gay, and Dimitrios Kouzapas. On duality relations for session types. In *TGC'14*, volume 8902 of *LNCS*, pages 51–66. Springer, 2014.

[2] Giovanni Bernardi and Matthew Hennessy. Using higher-order contracts to model session types (extended abstract). In *CONCUR'14*, volume 8704 of *LNCS*, pages 387–401. Springer, 2014.

[3] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR'08*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.

[4] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR'10*, volume 6269 of *LNCS*, pages 222–236. Springer, 2010.

[5] Luís Caires and Hugo Torres Vieira. Conversation types. *Theor. Comput. Sci.*, 411(51-52):4399–4440, 2010.

[6] Marco Carbone, Ornela Dardha, and Fabrizio Montesi. Progress as compositional lock-freedom. In *COORDINATION'14*, volume 8459 of *LNCS*, pages 49–64. Springer, 2014.

[7] Marco Carbone and Søren Debois. A graphical approach to progress for structured communication in web services. In *ICE'10*, volume 38 of *EPTCS*, pages 13–27, 2010.

[8] Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. Inference of global progress properties for dynamically interleaved multiparty sessions. In *COORDINATION'13*, volume 7890 of *LNCS*, pages 45–59. Springer, 2013.

[9] Ornela Dardha. Recursive session types revisited. In *BEAT'14*, volume 162 of *EPTCS*, pages 27–34, 2014.

[10] Ornela Dardha. *Type Systems for Distributed Programs: Components and Sessions*. Thesis, Università degli studi di Bologna, May 2014.

[11] Ornela Dardha, Elena Giachino, and Michael Lienhardt. A type system for components. In *SEFM'13*, volume 8137 of *LNCS*, pages 167–181. Springer, 2013.

[12] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *PPDP'12*, pages 139–150, New York, NY, USA, 2012. ACM.

[13] Ornela Dardha and Jorge A. Pérez. Comparing deadlock-free session typed processes. In *EXPRESS/SOS'15*, volume 190 of *EPTCS*, pages 1–15, 2015.

[14] Romain Demangeon and Kohei Honda. Full abstraction in a subtyped pi-calculus with linear types. In *CONCUR'11*, volume 6901 of *LNCS*, pages 280–296. Springer, 2011.

[15] Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro, and Nobuko Yoshida. On progress for structured communications. In *TGC'07*, volume 4912, pages 257–275. Springer, 2008.

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

[17] Simon J. Gay, Nils Gesbert, and António Ravara. Session types as generic process types. In *EXPRESS/SOS'14*, volume 160 of *EPTCS*, pages 94–110, 2014.

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

[19] Kohei Honda. Types for dyadic interaction. In *CONCUR'93*, volume 715 of *LNCS*, pages 509–523. Springer, 1993.

[20] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.

[21] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL'08*, pages 273–284. ACM, 2008.

[22] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. *Theo. Comput. Sci.*, 311(1-3):121–163, 2004.

[23] Naoki Kobayashi. A type system for lock-free processes. *Information and Computation*, 177(2):122–159, 2002.

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

[25] Naoki Kobayashi. A new type system for deadlock-free processes. In *CONCUR'06*, volume 4137 of *LNCS*, pages 233–247. Springer, 2006.

[26] Naoki Kobayashi. Type systems for concurrent programs. Extended version of [24], Tohoku University, 2007.

[27] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *POPL'96*, pages 358–371. ACM, 1996.

[28] Robin Milner. The polyadic pi-calculus: a tutorial. Technical report, Logic and Algebra of Specification, 1991.

[29] Dimitris Mostrous and Nobuko Yoshida. Two session typing systems for higher-order mobile processes. In *TLCA'07*, volume 4583 of *LNCS*, pages 321–335. Springer, 2007.

[30] Luca Padovani. From lock freedom to progress using session types. In *PLACES'13*, volume 137 of *EPTCS*, pages 3–19, 2013.

[31] Luca Padovani. Deadlock and Lock Freedom in the Linear π-Calculus. In *CSL-LICS'14*, pages 72:1–72:10. ACM, 2014.

[32] Luca Padovani. Type Reconstruction for the Linear π-Calculus with Composite and Equi-Recursive Types. In *FoSSaCS'14*, volume 8412 of *LNCS*, pages 88–102. Springer, 2014.

[33] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *LICS'93*, pages 376–385. IEEE Computer Society, 1993.

[34] Benjamin C. Pierce and Davide Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. In *POPL'97*, pages 242–255. ACM, 1997.

[35] Davide Sangiorgi. An interpretation of typed objects into typed pi-calculus. *Information and Computation*, 143(1):34–73, 1998.

[36] Davide Sangiorgi. Termination of processes. *Mathematical Structures in Computer Science*, 16(1):1–39, 2006.

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

[38] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer, 1994.

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

[40] Vasco Thudichum Vasconcelos. Fundamentals of session types. *Information and Computation*, 217:52–70, 2012.

[41] Philip Wadler. Propositions as sessions. In *ICFP'12*, pages 273–286. ACM, 2012.