Recursive Session Types Revisited

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

Session types model structured communication-based programming. In particular, binary session types for the π-calculus describe communication between exactly two participants in a distributed scenario. Adding sessions to the π-calculus means augmenting it with type and term constructs. In a previous paper, we tried to understand to which extent the session constructs are more complex and expressive than the standard π-calculus constructs. Thus, we presented an encoding of binary session π-calculus to the standard typed π-calculus by adopting *linear* and *variant* types and the *continuation-passing* principle. In the present paper, we focus on recursive session types and we present an encoding into recursive linear π-types. This encoding is a conservative extension of the former in that it preserves the results therein obtained. Most importantly, it adopts a new treatment of the duality relation, which in the presence of recursive types has been proven to be quite challenging.

1 Introduction

Session types are a type formalism used to model structured communication-based programming for distributed systems. In particular, binary session types for the π-calculus describe communication between exactly two participants in such scenario [6, 8, 11, 12]. When sessions are added to the standard typed π-calculus, the syntax of types and terms is augmented with ad-hoc constructs, added on top of the already existing ones. This yields a duplication of type and term constructs, e.g. restriction of session channels and restriction of standard π-channels or, recursive session types and recursive standard π-types [6]. Most importantly, this redundancy is also propagated in the theory of session types: various properties are proven for session types as well as for standard π-types. In a previous work [5], we focused on a subset of binary session types, namely the finite ones, and posed the following question:

To which extent session constructs are more complex and more expressive than the standard π-calculus constructs?

We answered this question by showing an encoding of finite binary session types into finite *linear* π-types and of finite session processes into finite standard π-processes. In the present paper, we extend the encoding to an infinite setting, namely to recursive session types and replicated processes, and pose the same question. We encode recursive session types into recursive linear π-types and replicated session processes into replicated standard π-processes. We show that the current encoding *i*) is sound and complete with respect to typing derivations, intuitively meaning: "a session process is well-typed if and only if its encoding is well-typed"; and *ii*) satisfies the operational correspondence property, intuitively meaning: "a session process and its encoding reduce to processes still related by the encoding".

The interest and benefits of this encoding are mainly in *expressivity* and *reusability* for a larger setting than the one adopted in [5]. The encoding is an expressivity result for recursive types. Its faithfulness, proved by *i*) and *ii*), permits reusability of already existing theory for standard typed π-calculus: e.g.,

*The author is supported by the UK EPSRC project *From Data Types to Session Types: A Basis for Concurrency and Distribution* (ABCD) (EP/K034413/1).

Submitted to:
BEAT 2014

subject reduction or type safety for session π-calculus can be obtained as corollaries from the encoding and the corresponding properties in the standard typed π-calculus.

The present encoding is not just an extension of the former, it presents novelties and differences with respect to [5], as listed below. *Duality* is a fundamental notion of session types, as it describes compatible behaviours between communicating parties. The most used duality is the *inductive duality function* $\overline{\cdot}$ [8, 12]. Recent work [2] has shown the inadequacy of $\overline{\cdot}$ in the presence of recursive types, because it does not commute with unfolding. As a consequence, using relations like subtyping or type equivalence becomes challenging, because these relations explicitly use unfolding of recursive types. In the light of such discovery, the present encoding adopts the *complement function* $\mathsf{cplt}()$ defined in [2], which is shown to be adequate, instead of $\overline{\cdot}$, adopted in the former encoding. Since $\mathsf{cplt}()$ and $\overline{\cdot}$ coincide for finite session types, the encoding in [5] remains sound and the present encoding is a conservative extension of the former, in that it preserves all the properties that the former encoding satisfies. For completeness, the present one is extended to standard variables and hence non session π-processes: in this case the encoding is an homomorphism and no linearity is required. On top of $\mathsf{cplt}()$, we present the *co-inductive duality relation*, which is shown to contain the complement [1], and is used in the type system for session π-calculus [3, 6]. This permits us to give a definition of complement and co-inductive duality for linear π-calculus types, which is another contribution of the present paper.

Structure of the paper. In § 2 we present the syntax of types and terms for both the session π-calculus and the standard types π-calculus. In § 3 we present the encoding of recursive session types and session processes and we state the main results for the encoding. In § 4 we give a detailed example of the encoding of a well-typed replicated process which uses recursive session types. We conclude in § 5. The proofs of the result herein presented can be found in the online version of the paper [3].

2 The Model

2.1 Background on π-calculus with sessions

Syntax. The syntax of the π-calculus with session types [6, 12] is given in Figure 1.

$$
\begin{array}{rclllllllll}
P,Q & ::= & x!\langle v\rangle.P & \text{(output)} & \mid & x?(y).P & \text{(input)} & \mid & x \triangleleft l_j.P & \text{(selection)} \\
 & & x \triangleright \{l_i : P_i\}_{i\in I} & \text{(branching)} & \mid & P \mid Q & \text{(parallel)} & \mid & (\boldsymbol{\nu} xy)P & \text{(session res.)} \\
 & & *P & \text{(replication)} & \mid & (\boldsymbol{\nu} x)P & \text{(channel res.)} & \mid & \mathbf{0} & \text{(inaction)} \\
v & ::= & x & \text{(variable)} & \mid & \mathbf{1} & \text{(unit value)} & & &
\end{array}
$$

Figure 1: π-calculus with sessions, syntax.

P,Q range over processes, x,y over variables, v over values and l over labels. A value is a variable or $\mathbf{1}$. A process is an output $x!\langle v\rangle.P$ which sends v on x with continuation P; an input $x?(y).P$ which receives a value on x and proceeds as P; a selection $x \triangleleft l_j.P$ which selects l_j on x and proceeds as P; a branching $x \triangleright \{l_i : P_i\}_{i\in I}$ which offers a set of labelled processes on x, with labels being all different; a parallel composition $P \mid Q$ of P,Q; replicated $*P$ which spawns copies of P; a session restriction $(\boldsymbol{\nu} xy)P$ or a standard channel restriction $(\boldsymbol{\nu} x)P$; or $\mathbf{0}$, the terminated process. Session restriction differs from the standard one: $(\boldsymbol{\nu} xy)$ states that x and y, called *co-variables*, are the opposite endpoints of a session channel and are bound in P. It models session creation and the connection phase [8, 11].

Session types. The syntax of types for the π-calculus with sessions [6] is given in Figure 2.

$$
\begin{array}{lllllllllll}
S & ::= & !T.S & \text{(send)} & \mid & ?T.S & \text{(receive)} & \mid & \oplus\{l_i : S_i\}_{i\in I} & \text{(select)} \\
 & & \&\{l_i : S_i\}_{i\in I} & \text{(branch)} & \mid & X & \text{(type var.)} & \mid & \overline{X} & \text{(dual type var.)} \\
 & & \mathsf{end} & \text{(termination)} & \mid & \mu X.S & \text{(rec. session type)} & & & \\
T & ::= & S & \text{(session type)} & \mid & \sharp T & \text{(channel type)} & \mid & X & \text{(type variable)} \\
 & & \mu X.T & \text{(recursive type)} & \mid & \mathtt{Unit} & \text{(unit type)} & & &
\end{array}
$$

Figure 2: π-calculus with sessions, types.

S ranges over session types and T over types. A session type can be $!T.S$ or $?T.S$ which respectively, sends or receives a value of type T and continuation S; select $\oplus\{l_i : S_i\}_{i\in I}$ or branch $\&\{l_i : S_i\}_{i\in I}$ which are sets of labelled session types indicating respectively, internal and external choice, with labels being all different; a (dualised) type variable $X, \overline{X}$, or recursive session type $\mu X.S$ or the terminated type end. A type can be a session type S; a standard channel type $\sharp T$; a type variable X or recursive type $\mu X.T$ or a unit type $\mathtt{Unit}$. Recursive (session) types are required to be *guarded*, meaning that in $\mu X.T$, variable X may occur free in T only under at least one of the other type constructs. To work with recursive types we need the *unfolding* function (UNF) which unfolds a recursive type until the first type constructor different from μX is reached (see [3]). Finally, we use $\mathtt{SType}$ to denote the set of *closed* (no free type variables) and *guarded* session types.

On duality for session types. Below we give an adaptation of the complement function [2] to $X, \overline{X}$.

Definition 2.1 (Complement function for session types)**.** *The complement function is defined as:*

$$
\begin{array}{lcllcl}
\mathsf{cplt}(?T.S) & = & !T.\mathsf{cplt}(S) & \mathsf{cplt}(X) & = & \overline{X} \\
\mathsf{cplt}(!T.S) & = & ?T.\mathsf{cplt}(S) & \mathsf{cplt}(\overline{X}) & = & X \\
\mathsf{cplt}(\&\{l_i : S_i\}_{i\in I}) & = & \oplus\{l_i : \mathsf{cplt}(S)_i\}_{i\in I} & \mathsf{cplt}(\mu X.S) & = & \mu X.\mathsf{cplt}(S\lfloor^{\mu X.S}/_X\rfloor) \\
\mathsf{cplt}(\oplus\{l_i : S_i\}_{i\in I}) & = & \&\{l_i : \mathsf{cplt}(S)_i\}_{i\in I} & \mathsf{cplt}(\mathsf{end}) & = & \mathsf{end}
\end{array}
$$

It uses a syntactic substitution $\lfloor^{-}/_{-}\rfloor$, which acts only on carried types and is formally defined in [1–3]. Below we give the definition of standard type substitution for (dualised) type variables [7].

$$
\begin{array}{lll}
X\{S/X\} = S & Y\{S/X\} = Y & \text{If } X \neq Y \\
\overline{X}\{S/X\} = \mathsf{cplt}(S) & \overline{Y}\{S/X\} = \overline{Y} & \text{If } X \neq Y
\end{array}
$$

However, when describing opposite behaviours between communicating parties, in this paper we adopt the *co-inductive duality relation* $\perp_{\mathsf{S}}$ by following [6]. The benefits of this approach are: *i*) $\perp_{\mathsf{S}}$ commutes with unfolding [2] and hence it is adequate; *ii*) as stated in [6], since it is a relation it captures dual behaviours that $\overline{\cdot}$, $\mathsf{cplt}()$ do not capture, like $\mu X.?\mathtt{Unit}.X$ and $!\mathtt{Unit}.\mu X.!\mathtt{Unit}.X$. *iii*) as stated in [1], it contains $\mathsf{cplt}()$. Before defining $\perp_{\mathsf{S}}$, we need the notion of type equivalence and hence subtyping. For simplicity, we omit subtyping on base types and on standard channel types, which are given in [2, 6].

Definition 2.2 (Subtyping and type equivalence for session types [6])**.** *A relation* $\mathcal{R} \subseteq \mathtt{SType} \times \mathtt{SType}$ *is a* type simulation *if* $(T, S) \in \mathcal{R}$ *implies the following:*

i) *if* $\text{UNF}(T) = \mathsf{end}$ *then* $\text{UNF}(S) = \mathsf{end}$

ii) *if* $\text{UNF}(T) = ?T_m.T'$ *then* $\text{UNF}(S) = ?S_m.S'$ *and* $T_m \;\mathcal{R}\; S_m$ *and* $T' \;\mathcal{R}\; S'$

iii) *if* $\text{UNF}(T) = !T_m.T'$ *then* $\text{UNF}(S) = !S_m.S'$ *and* $S_m \;\mathcal{R}\; T_m$ *and* $T' \;\mathcal{R}\; S'$

iv) *if* $\text{UNF}(T) = \&\{l_i : T_i\}_{i\in I}$ *then* $\text{UNF}(S) = \&\{l_j : S_j\}_{j\in J}$, $I \subseteq J$, $T_i \;\mathcal{R}\; S_i$, $\forall i \in I$

v) *if* $\text{UNF}(T) = \oplus\{l_i : T_i\}_{i\in I}$ *then* $\text{UNF}(S) = \oplus\{l_j : S_j\}_{j\in J}$, $J \subseteq I$, $T_j \;\mathcal{R}\; S_j$, $\forall j \in J$

The subtyping relation $\leq_{\mathsf{s}}$ *is defined by* $T \leq_{\mathsf{s}} S$ *if and only if there exists a type simulation* $\mathcal{R}$ *such that* $(T,S) \in \mathcal{R}$. *The* type equivalence relation $=_{\mathsf{s}}$ *is defined by* $T =_{\mathsf{s}} S$ *if and only if* $T \leq_{\mathsf{s}} S$ *and* $S \leq_{\mathsf{s}} T$.

Definition 2.3 (Co-inductive duality for session types [6])**.** *A relation* $\mathcal{R} \in \mathtt{SType} \times \mathtt{SType}$ *is a* duality relation *if* $(T,S) \in \mathcal{R}$ *implies the following conditions:*
i) If $\textsc{unf}(T) = \mathsf{end}$ *then* $\textsc{unf}(S) = \mathsf{end}$
ii) If $\textsc{unf}(T) = ?T_m.T'$ *then* $\textsc{unf}(S) = !S_m.S'$ *and* $T' \ \mathcal{R} \ S'$ *and* $T_m =_{\mathsf{s}} S_m$
iii) If $\textsc{unf}(T) = !T_m.T'$ *then* $\textsc{unf}(S) = ?S_m.S'$ *and* $T' \ \mathcal{R} \ S'$ *and* $T_m =_{\mathsf{s}} S_m$
iv) If $\textsc{unf}(T) = \&\{l_i : T_i\}_{i\in I}$ *then* $\textsc{unf}(S) = \oplus\{l_i : S_i\}_{i\in I}$ *and* $\forall i \in I,\ T_i \ \mathcal{R} \ S_i$
v) If $\textsc{unf}(T) = \oplus\{l_i : T_i\}_{i\in I}$ *then* $\textsc{unf}(S) = \&\{l_i : S_i\}_{i\in I}$ *and* $\forall i \in I,\ T_i \ \mathcal{R} \ S_i$
The co-inductive duality relation $\perp_{\mathsf{s}}$ *is defined by* $T \perp_{\mathsf{s}} S$ *iff* $\exists \ \mathcal{R}$*, a duality relation such that* $(T,S) \in \mathcal{R}$.

Proposition 2.4. *Let* $T,S \in \mathtt{SType}$, $\mathsf{cplt}(T) = S \implies T \perp_{\mathsf{s}} S$.

Proposition 2.5 (Idempotence)**.** *Let* $T,S,U \in \mathtt{SType}$. *If* $T \perp_{\mathsf{s}} S$ *and* $S \perp_{\mathsf{s}} U$ *then* $T =_{\mathsf{s}} U$.

By Proposition 2.4 and Proposition 2.5 we have the following.

Proposition 2.6. *Let* $T,S,U \in \mathtt{SType}$. *If* $\mathsf{cplt}(T) = S$ *and* $\mathsf{cplt}(S) = U$ *then* $T =_{\mathsf{s}} U$.

2.2 Background on standard π-calculus

Syntax. The syntax of the polyadic π-calculus [10] is given in Figure 3.

P,Q	$::=$	$x!\langle\tilde{v}\rangle.P$	(output)	$\mid$	$x?(\tilde{y}).P$	(input)	$\mid$	$P \mid Q$		(parallel)
		$(\nu x)P$	(channel res.)	$\mid$	$*P$	(repl.)	$\mid$	$\mathbf{case}\ v\ \mathbf{of}\ \{l_i\_x_i \triangleright P_i\}_{i\in I}$		(case)
		$\mathbf{0}$	(inaction)							
v	$::=$	x	(variable)	$\mid$	$\mathbf{1}$	(unit val.)	$\mid$	l_v		(variant val.)

Figure 3: Standard π-calculus, syntax.

P,Q range over processes, x,y over variables, l over labels and v over values, i.e., variables, $\mathbf{1}$, or variant values. A process can be an output $x!\langle\tilde{v}\rangle.P$ which sends $\tilde{v}$ on x and proceeds as P; an input $x?(\tilde{y}).P$ which receives a sequence of values on x, substitutes them for $\tilde{y}$ in P; a parallel composition $P \mid Q$ of P,Q; replicated $*P$; a restriction $(\nu x)P$ which creates a new channel x and binds it in P; a $\mathbf{case}\ v\ \mathbf{of}\ \{l_i\_x_i \triangleright P_i\}_{i\in I}$ which offers a set of labelled processes, with labels being all different; or inaction $\mathbf{0}$.

Standard π-types. The syntax of π-types [9, 10] is defined in Figure 4.

τ	$::=$	$\emptyset[]$	(no capability)	$\mid$	$\sharp[\widetilde{T}]$	(connection)			
		$\ell_{\mathsf{i}}\,[\widetilde{T}]$	(linear input)	$\mid$	$\ell_{\mathsf{o}}\,[\widetilde{T}]$	(linear output)	$\mid$	$\ell_{\sharp}\,[\widetilde{T}]$	(linear connection)
T	$::=$	τ	(channel type)	$\mid$	$\langle l_i : T_i\rangle_{i\in I}$	(variant type)	$\mid$	X	(type var.)
		$\overline{X}$	(dual type var.)	$\mid$	$\mathtt{Unit}$	(unit type)	$\mid$	$\mu X.T$	(recursive type)

Figure 4: Standard π-calculus, types.

τ ranges over channel types and T over types. A channel type is a type with no capability $\emptyset[]$, meaning it cannot be used further; a connection $\sharp[\widetilde{T}]$, indefinitely used; a linear input $\ell_{\mathsf{i}}\,[\widetilde{T}]$, a linear output $\ell_{\mathsf{o}}\,[\widetilde{T}]$ or the combination of both, i.e., a linear connection $\ell_{\sharp}\,[\widetilde{T}]$ used *exactly once* [9] according to its capability.

A type can be a channel type τ; a variant $\langle l_i : T_i \rangle_{i \in I}$ being a set of labelled types, with labels being all different; a (dualised) type variable $X, \overline{X}$ or recursive type $\mu X.T$ or `Unit`. Again, we require recursive types to be *guarded* and use `PType` to denote the set of *closed* (no free type variables) and *guarded* standard π-types. To conclude, the definition of *unfolding* is the same as in the previous section.

On duality for linear types. Inspired by duality on session types, below we give the definition of complement function $\mathsf{picplt}()$ and co-inductive duality relation $\perp_{\mathsf{p}}$ for linear π-types.

Definition 2.7 (Complement function for linear π-types)**.** *The complement function is defined as:*

$$\begin{array}{lcllcl}
\mathsf{picplt}(\ell_{\mathsf{i}}\,[\widetilde{T}]) & = & \ell_{\mathsf{o}}\,[\widetilde{T}] & \mathsf{picplt}(X) & = & \overline{X} \\
\mathsf{picplt}(\ell_{\mathsf{o}}\,[\widetilde{T}]) & = & \ell_{\mathsf{i}}\,[\widetilde{T}] & \mathsf{picplt}(\overline{X}) & = & X \\
\mathsf{picplt}(\mu X.T) & = & \mu X.\mathsf{picplt}(T\lfloor^{\mu X.T}/_{X}\rfloor) & \mathsf{picplt}(\emptyset[]) & = & \emptyset[]
\end{array}$$

The definition of type substitution for linear types is the same as in the previous section, where $\mathsf{cplt}()$ is replaced by $\mathsf{picplt}()$. Before defining $\perp_{\mathsf{p}}$, we give a co-inductive definition of subtyping and type equivalence for linear π-types. For simplicity, we omit the subtyping on base types, standard channel types, or variant types which can be found in the literature [10].

Definition 2.8 (Subtyping and type equivalence for linear π-types)**.** *A relation* $\mathcal{R} \subseteq \mathtt{PType} \times \mathtt{PType}$ *is a* type simulation *if* $(T, S) \in \mathcal{R}$ *implies the following:*
i) if $\textsc{unf}(T) = \emptyset[]$ *then* $\textsc{unf}(S) = \emptyset[]$
ii) if $\textsc{unf}(T) = \ell_{\mathsf{i}}\,[\widetilde{T}]$ *then* $\textsc{unf}(S) = \ell_{\mathsf{i}}\,[\widetilde{S}]$ *and* $\widetilde{T}\ \mathcal{R}\ \widetilde{S}$
iii) if $\textsc{unf}(T) = \ell_{\mathsf{o}}\,[\widetilde{T}]$ *then* $\textsc{unf}(S) = \ell_{\mathsf{o}}\,[\widetilde{S}]$ *and* $\widetilde{T}\ \mathcal{R}\ \widetilde{S}$
The subtyping relation $\leq_{\mathsf{s}}$ *is defined by* $T \leq_{\mathsf{p}} S$ *if and only if there exists a type simulation* $\mathcal{R}$ *such that* $(T, S) \in \mathcal{R}$. *The* type equivalence relation $=_{\mathsf{p}}$ *is defined by* $T =_{\mathsf{p}} S$ *if and only if* $T \leq_{\mathsf{p}} S$ *and* $S \leq_{\mathsf{p}} T$.

Definition 2.9 (Co-inductive duality for linear π- types)**.** *A relation* $\mathcal{R} \in \mathtt{PType} \times \mathtt{PType}$ *is a* duality relation *if* $(T, S) \in \mathcal{R}$ *implies the following conditions:*
i) If $\textsc{unf}(T) = \emptyset[]$ *then* $\textsc{unf}(S) = \emptyset[]$
ii) If $\textsc{unf}(T) = \ell_{\mathsf{i}}\,[\widetilde{T}]$ *then* $\textsc{unf}(S) = \ell_{\mathsf{o}}\,[\widetilde{S}]$ *and* $\widetilde{T} =_{\mathsf{p}} \widetilde{S}$
iii) If $\textsc{unf}(T) = \ell_{\mathsf{o}}\,[\widetilde{T}]$ *then* $\textsc{unf}(S) = \ell_{\mathsf{i}}\,[\widetilde{T}]$*and* $\widetilde{T} =_{\mathsf{p}} \widetilde{S}$
The co-inductive duality relation $\perp_{\mathsf{p}}$ *is defined by* $T \perp_{\mathsf{p}} S$ *iff* $\exists\ \mathcal{R}$*, a duality relation such that* $(T, S) \in \mathcal{R}$.

Proposition 2.10. *Let* $T, S \in \mathtt{PType}$, $\mathsf{picplt}(T) = S \implies T \perp_{\mathsf{p}} S$.

Proposition 2.11 (Idempotence)**.** *Let* $T, S, U \in \mathtt{PType}$. *If* $T \perp_{\mathsf{p}} S$ *and* $S \perp_{\mathsf{p}} U$ *then* $T =_{\mathsf{p}} U$.

Proposition 2.12. *Let* $T, S, U \in \mathtt{PType}$. *If* $\mathsf{picplt}(T) = S$ *and* $\mathsf{picplt}(S) = U$ *then* $T =_{\mathsf{p}} U$.

3 Encoding recursive session types

Below we give the encoding of recursive session types into recursive linear π-types and of session π-processes into standard π-processes. It is based on the notions of: *linearity*, *variant types* and *continuation-passing* principle. To preserve communication safety and privacy of session types, we use linear channels. To encode internal and external choice, we adopt variant types and the **case** process. To preserve the sequentiality of session types and hence session fidelity, we adopt the continuation-passing principle.

Types encoding. The encoding of session types is presented in Figure 5. Type end is encoded as the channel type with no capability $\emptyset[]$; output $!T.S$ and input $?T.S$ session types are encoded as linear output $\ell_{\mathsf{o}}\,[[\![T]\!], [\![\mathsf{cplt}(S)]\!]]$ and linear input $\ell_{\mathsf{i}}\,[[\![T]\!], [\![S]\!]]$ channel types carrying the encoding of type T and of

$$
\begin{array}{lcl}
[\![\mathsf{end}]\!] & \stackrel{\text{def}}{=} & \emptyset[\,] \\
[\![!T.S]\!] & \stackrel{\text{def}}{=} & \ell_{\mathsf{o}}\ [[\![T]\!], [\![\mathsf{cplt}(S)]\!]] \\
[\![?T.S]\!] & \stackrel{\text{def}}{=} & \ell_{\mathsf{i}}\ [[\![T]\!], [\![S]\!]] \\
[\![\oplus\{l_i : S_i\}_{i\in I}]\!] & \stackrel{\text{def}}{=} & \ell_{\mathsf{o}}\ [\langle l_i : [\![\mathsf{cplt}(S_i)]\!]\rangle_{i\in I}] \\
[\![\&\{l_i : S_i\}_{i\in I}]\!] & \stackrel{\text{def}}{=} & \ell_{\mathsf{i}}\ [\langle l_i : [\![S_i]\!]\rangle_{i\in I}] \\
[\![X]\!] & \stackrel{\text{def}}{=} & X \\
[\![\overline{X}]\!] & \stackrel{\text{def}}{=} & \overline{X} \\
[\![\mu X.S]\!] & \stackrel{\text{def}}{=} & \mu X.[\![S]\!]
\end{array}
\qquad
\begin{array}{lcl}
[\![x]\!]_f & \stackrel{\text{def}}{=} & f_x \\
[\![x!\langle v\rangle.P]\!]_f & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} c)f_x!\langle v, c\rangle.[\![P]\!]_{f,\{x\mapsto c\}} \\
[\![x?(y).P]\!]_f & \stackrel{\text{def}}{=} & f_x?(y, c).[\![P]\!]_{f,\{x\mapsto c\}} \\
[\![x \triangleleft l_j.P]\!]_f & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} c)f_x!\langle l_j\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}} \\
[\![x \triangleright \{l_i : P_i\}_{i\in I}]\!]_f & \stackrel{\text{def}}{=} & f_x?(y).\ \mathbf{case}\, y\, \mathbf{of}\, \{l_i\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I} \\
[\![(\boldsymbol{\nu} xy)P]\!]_f & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} c)[\![P]\!]_{f,\{x,y\mapsto c\}} \\
[\![P \mid Q]\!]_f & \stackrel{\text{def}}{=} & [\![P]\!]_f \mid [\![Q]\!]_f \\
[\![(\boldsymbol{\nu} x)P]\!]_f & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} x)[\![P]\!]_f \\
[\![*P]\!]_f & \stackrel{\text{def}}{=} & *[\![P]\!]_f \\
[\![\mathbf{0}]\!]_f & \stackrel{\text{def}}{=} & \mathbf{0}
\end{array}
$$

Figure 5: Encoding of session types and terms

continuation type $\mathsf{cplt}(S)$ and S, respectively. $\mathsf{cplt}(S)$ is adopted in the output since it is the type of a channel as seen by the receiver, namely the communicating counterpart. Select $\oplus\{l_i : S_i\}_{i\in I}$ and branch $\&\{l_i : S_i\}_{i\in I}$ are encoded as linear output $\ell_{\mathsf{o}}\ [\langle l_i : [\![\mathsf{cplt}(S_i)]\!]\rangle_{i\in I}]$ and linear input $\ell_{\mathsf{i}}\ [\langle l_i : [\![S_i]\!]\rangle_{i\in I}]$ types carrying a variant type with the encoded continuation types; the reason for $\mathsf{cplt}(S_i)$ is the same as before. The encoding of a (dualised) type variable and a recursive session type is an homomorphism.

Terms encoding. The encoding of session π-terms, presented in Figure 5, uses a partial function f from variables to variables which performs a renaming of linear variables into new linear variables to respect their nature of being used exactly once and it is the identity function over standard variables. The formal definition can be found in [3]. A variable x is encoded as f_x; an output $x!\langle v\rangle.P$ is encoded as an output on f_x of v and the freshly created channel c which replaces x in the encoding of P. The encoding of an input $x?(y).P$ is an input on f_x with placeholders y and the continuation channel c used in $[\![P]\!]_{f,\{x\mapsto c\}}$. The encodings of selection $x \triangleleft l_j.P$ and branching $x \triangleright \{l_i : P_i\}_{i\in I}$ are the output and input processes on f_x, respectively. The output carries a variant value l_j_c where l_j is the selected label and c the new channel to be used in the continuation. The input has a continuation of a **case**, offering the encoded processes of the branching. The session restriction process $(\boldsymbol{\nu} xy)P$ is encoded as the restriction on c which replaces both of the endpoint x, y in the encoding of P. The rest of the equations states that the encoding of parallel composition, standard channel restriction and replication is an homomorphism and the encoding of the inaction process is the identity function.

Results of the encoding. The proofs of the following results can be found in [3]. The following two lemmas relate the encoding of equal and dual session types to equal and dual linear π-types.

Lemma 3.1 (Encoding $=_{\mathsf{s}}$)**.** *$T, S \in \mathsf{SType}$ and $T =_{\mathsf{s}} S$. If $[\![T]\!] = \tau$, then $[\![S]\!] = \sigma$ and $\tau =_{\mathsf{p}} \sigma$.*

Lemma 3.2 (Encoding $\perp_{\mathsf{s}}$)**.** *$T, S \in \mathsf{SType}$ and $T \perp_{\mathsf{s}} S$. If $[\![\textsc{unf}(T)]\!] = \tau$ then, $[\![\textsc{unf}(S)]\!] = \sigma$ and $\tau \perp_{\mathsf{p}} \sigma$.*

Lemma 3.3 (Value Typing)**.** *$\Gamma \vdash v : T$ if and only if $[\![\Gamma]\!]_f \vdash [\![v]\!]_f : [\![T]\!]$.* [1]

Theorem 3.4 (Process Typing)**.** *$\Gamma \vdash P$ if and only if $[\![\Gamma]\!]_f \vdash [\![P]\!]_f$.*

Theorem 3.5 (Operational Correspondence)**.** *Let P be a session process. The following hold.*

1. *If $P \to P'$ then $[\![P]\!]_f \to\hookrightarrow [\![P']\!]_f$,*

[1] The encoding is extended to typing environments Γ and the details can be checked in [3].

2. *If* $[\![P]\!]_f \to Q$ *then,* $\exists\, P', \mathcal{E}[\cdot]$ *such that* $\mathcal{E}[P] \to \mathcal{E}[P']$ *and* $Q \hookrightarrow [\![P']\!]_{f'}$*, where* f' *is the updated* f *after reduction and* $f_x = f_y$ *for all* $(\nu xy) \in \mathcal{E}[\cdot]$.

$\hookrightarrow$ denotes $\equiv$ possibly extended with a **case** reduction; $\mathcal{E}[\cdot]$ is an *evaluation context*.

4 Example of encoding

We present an error-free process which requires recursive session types. We let a, b range over standard channels and x, y, z, v, w range over session channels; we associate a type to a variable in an object position in input or restriction as in [6]. The typing rules and the operational semantics can be found in [3].

Let $P = *(a?(x : T).x \triangleleft l.a!\langle x\rangle.\mathbf{0})$ be a replicated process that on a standard channel a receives a session channel x on which selects l and proceeds as $a!\langle x\rangle$. We have the following typing derivation for P:

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{a : \sharp T \vdash \mathbf{0}}\ \text{T-Nil} \qquad S \le_{\mathsf{s}} T}{a : \sharp T, x : S \vdash a!\langle x\rangle.\mathbf{0}}\ \text{T-Out}}{a : \sharp T, x : \oplus\{l : S\} \vdash x \triangleleft l.a!\langle x\rangle.\mathbf{0} \qquad T \le_{\mathsf{s}} \oplus\{l : S\}}\ \text{T-Select}}{\dfrac{a : \sharp T \vdash a?(x : T).x \triangleleft l.a!\langle x\rangle.\mathbf{0} \qquad \mathsf{un}(a : \sharp T)}{a : \sharp T \vdash *(a?(x : T).x \triangleleft l.a!\langle x\rangle.\mathbf{0})}\ \text{T-Rep}}\ \text{T-In}
$$

For this derivation to hold, T and S need to be such that $T \le_{\mathsf{s}} \oplus\{l : S\}$ and $S \le_{\mathsf{s}} T$. The simplest way to solve this system of subtyping in-equations is to have $S = T$, which requires $T = \mu X.\oplus\{l : X\}$.

Let $Q = *(b?(x : U).x \triangleright \{l : b!\langle x\rangle.\mathbf{0}\})$; it has dual behaviour to P and the typing derivation is similar to above, with T-Select replaced by T-Branch. We now have the in-equations $U \le_{\mathsf{s}} \&\{l : S'\}$ and $S' \le_{\mathsf{s}} U$. We let $U = \mu X.\&\{l : X\}$. By Definition 2.3, we have $U \perp_{\mathsf{s}} T$. We now close P and Q with two auxiliary output processes, $a!\langle v\rangle.\mathbf{0}$ and $b!\langle w\rangle.\mathbf{0}$, where v, w are to be co-variables. Then, we have:

$$
\begin{aligned}
\emptyset \vdash Sys =& (\nu a : \sharp T)(\nu b : \sharp U)(\nu vw : T)(a!\langle v\rangle.\mathbf{0} \mid b!\langle w\rangle.\mathbf{0} \mid P \mid Q)\\
\equiv\to& (\nu a : \sharp T)(\nu b : \sharp U)(\nu vw : T)(v \triangleleft l.a!\langle v\rangle.\mathbf{0} \mid w \triangleright \{l : b!\langle w\rangle.\mathbf{0}\} \mid P \mid Q)\\
\to& (\nu a : \sharp T)(\nu b : \sharp U)(\nu vw : T)(a!\langle v\rangle.\mathbf{0} \mid b!\langle w\rangle.\mathbf{0} \mid P \mid Q) = Sys \to^*
\end{aligned}
$$

The encoding of types is as follows. Since $U \perp_{\mathsf{s}} T$ by Lemma 3.2 we have $\upsilon \perp_{\mathsf{p}} \tau$.

$$
\begin{aligned}
[\![U]\!] &= [\![\mu X.\&\{l : X\}]\!] = \mu X.[\![\&\{l : X\}]\!] = \mu X.\ell_{\mathsf{i}}\,[\langle l : [\![X]\!]\rangle] = \mu X.\ell_{\mathsf{i}}\,[\langle l : X\rangle] = \upsilon\\
[\![T]\!] &= [\![\mu X.\oplus\{l : X\}]\!] = \mu X.[\![\oplus\{l : X\}]\!] = \mu X.\ell_{\mathsf{o}}\,[\langle l : [\![\mathsf{cplt}(X)]\!]\rangle] = \mu X.\ell_{\mathsf{o}}\,[\langle l : \overline{X}\rangle] = \tau
\end{aligned}
$$

Duality of session types boils down to opposite capabilities in the outermost level $(\ell_{\mathsf{i}}, \ell_{\mathsf{o}})$ and the *same* carried type, where in [5] same means *syntactic identity* and in the present means *type equivalence*. Unfolding is performed in order to test linear type duality and the type equivalence of the carried type.

$$
\begin{aligned}
[\![P]\!]_f &= [\![*(a?(x).x \triangleleft l.a!\langle x\rangle.\mathbf{0})]\!]_f = *[\![(a?(x).x \triangleleft l.a!\langle x\rangle.\mathbf{0})]\!]_f = *(a?(x).[\![x \triangleleft l.a!\langle x\rangle.\mathbf{0}]\!]_f)\\
&= *(a?(x).(\nu c)x!\langle l\_c\rangle.[\![a!\langle x\rangle.\mathbf{0}]\!]_{f,\{x\mapsto c\}}) = *(a?(x).(\nu c)x!\langle l\_c\rangle.a!\langle c\rangle.\mathbf{0})\\
[\![Q]\!]_f &= [\![*(b?(x).x \triangleright \{l : b!\langle x\rangle.\mathbf{0}\})]\!]_f = *[\![(b?(x).x \triangleright \{l : b!\langle x\rangle.\mathbf{0}\})]\!]_f = *(b?(x).[\![x \triangleright \{l : b!\langle x\rangle.\mathbf{0}\})]\!]_f\\
&= *(b?(x).x?(y).\mathbf{case}\, y\, \mathbf{of}\, \{l\_c \triangleright [\![b!\langle x\rangle.\mathbf{0}]\!]_{f,\{x\mapsto c\}}\}) = *(b?(x).x?(y).\mathbf{case}\, y\, \mathbf{of}\, \{l\_c \triangleright b!\langle c\rangle.\mathbf{0}\}\})
\end{aligned}
$$

$$
\begin{aligned}
\emptyset \vdash [\![Sys]\!]_f =& (\nu a)(\nu b)(\nu z)[\![(a!\langle v\rangle.\mathbf{0} \mid b!\langle w\rangle.\mathbf{0} \mid P \mid Q)]\!]_{f,\{v,w\mapsto z\}} = (\nu a)(\nu b)(\nu z)(a!\langle z^+\rangle.\mathbf{0} \mid b!\langle z^-\rangle.\mathbf{0} \mid [\![P]\!]_f \mid [\![Q]\!]_f)\\
\equiv& (\nu a)(\nu b)(\nu z)(a!\langle z^+\rangle.\mathbf{0} \mid b!\langle z^-\rangle.\mathbf{0} \mid a?(x).(\nu c)x!\langle l\_c\rangle.a!\langle c\rangle.\mathbf{0} \mid b?(x).x?(y).\mathbf{case}\, y\, \mathbf{of}\, \{l\_c \triangleright b!\langle c\rangle.\mathbf{0}\}\} \mid\\
& *(a?(x).(\nu c)x!\langle l\_c\rangle.a!\langle c\rangle.\mathbf{0}) \mid *(b?(x).x?(y).\mathbf{case}\, y\, \mathbf{of}\, \{l\_c \triangleright b!\langle c\rangle.\mathbf{0}\}\}))
\end{aligned}
$$

$$\rightarrow (\nu a)(\nu b)(\nu z)((\nu c)z^+!\langle l\_c^-\rangle.a!\langle c^+\rangle.\mathbf{0} \mid z^-?(y).\mathbf{case}\, y\, \mathbf{of}\, \{l\_c^- \triangleright b!\langle c^-\rangle.\mathbf{0}\}\} \mid [\![P]\!]_f \mid [\![Q]\!]_f)$$
$$\rightarrow (\nu a)(\nu b)(\nu z)((\nu c).a!\langle c^+\rangle.\mathbf{0} \mid \mathbf{case}\, l\_c^-\, \mathbf{of}\, \{l\_c^- \triangleright b!\langle c^-\rangle.\mathbf{0}\}\} \mid [\![P]\!]_f \mid [\![Q]\!]_f)$$
$$\hookrightarrow (\nu a)(\nu b)((\nu c).a!\langle c^+\rangle.\mathbf{0} \mid b!\langle c^-\rangle.\mathbf{0}\}\} \mid [\![P]\!]_f \mid [\![Q]\!]_f) \equiv [\![Sys]\!]_f \rightarrow^*$$

5 Conclusions and Future Work

In this paper we present an encoding of recursive session types into recursive linear types and session processes into corresponding π- processes. The encoding is a conservative extension of the one given in [5]. It uses $\mathsf{cplt}()$ instead of $\bar{\cdot}$, because the latter is inadequate in the presence of recursive types [1,2]. Since these two functions coincide for finite session types, the encoding in [5] remains sound. We prove the faithfulness of the present encoding with respect to typing derivations and operational semantics, following the same line of [4,5]. As long as future work is concerned, we would like to test our encoding under different dualities for sessions presented in [1]. Moreover, as in [5] we would like to extend the present encoding to advanced features like polymorphism or higher-order, or multiparty session types.

Acknowledgements. The author would like to thank Simon J. Gay, Elena Giachino and Davide Sangiorgi for the inspiring and very useful discussions.

References

[1] Giovanni Bernardi, Ornela Dardha, Simon J. Gay & Dimitrios Kouzapas (2014): *On duality relations for session types*. To appear in Proc. of TGC.

[2] Giovanni Bernardi & Matthew Hennessy (2013): *Using higher-order contracts to model session types*. CoRR abs/1310.6176. Available at http://arxiv.org/abs/1310.6176.

[3] Ornela Dardha (2014): *Recursive Session Types Revisited*. http://www.dcs.gla.ac.uk/~ornela/my_papers/D14-Extended.pdf.

[4] Ornela Dardha (2014): *Type Systems for Distributed Programs: Components and Sessions*. Ph.D. thesis, University of Bologna. http://www.dcs.gla.ac.uk/~ornela/my_papers/DardhaPhDThesis.pdf.

[5] Ornela Dardha, Elena Giachino & Davide Sangiorgi (2012): *Session types revisited*. In: *PPDP*, ACM, New York, NY, USA, pp. 139–150, doi:10.1145/2370776.2370794.

[6] Simon Gay & Malcolm Hole (2005): *Subtyping for Session Types in the Pi Calculus*. Acta Informatica 42(2-3), pp. 191–225, doi:10.1007/s00236-005-0177-z.

[7] Simon J. Gay (2008): *Bounded polymorphism in session types*. Mathematical Structures in Computer Science 18(5), pp. 895–930, doi:10.1017/S0960129508006944.

[8] Kohei Honda, Vasco Vasconcelos & Makoto Kubo (1998): *Language primitives and type disciplines for structured communication-based programming*. In: *ESOP'98*, *LNCS* 1381, springer, Heidelberg, Germany, pp. 22–138, doi:10.1007/BFb0053567.

[9] Naoki Kobayashi, Benjamin Pierce & David Turner (1996): *Linear Types and π-calculus*. In: *POPL*, 21(5), ACM Press, New York, NY, USA, pp. 358–371, doi:10.1145/330249.330251.

[10] Davide Sangiorgi & David Walker (2001): *The π-calculus - a theory of mobile processes*. Cambridge University Press.

[11] Kaku Takeuchi, Kohei Honda & Makoto Kubo (1994): *An Interaction-based Language and its Typing System*. In: *PARLE'94*, pp. 398–413, doi:10.1007/3-540-58184-7_118.

[12] Vasco T. Vasconcelos (2012): *Fundamentals of session types*. Information Computation 217, pp. 52–70, doi:10.1016/j.ic.2012.05.002.