EXPRESSing Session Types

Ilaria Castellani, Ornela Dardha, Luca Padovani, Davide Sangiorgi

EXPRESSing Session Types

Ilaria Castellani
INRIA, Université Côte d'Azur

Ornela Dardha
University of Glasgow

Luca Padovani
University of Camerino

Davide Sangiorgi
University of Bologna, INRIA

To celebrate the 30th edition of EXPRESS and the 20th edition of SOS we overview how session types can be expressed in a type theory for the standard π-calculus by means of a suitable encoding. The encoding allows one to reuse results about the π-calculus in the context of session-based communications, thus deepening the understanding of sessions and reducing redundancies in their theoretical foundations. Perhaps surprisingly, the encoding has practical implications as well, by enabling refined forms of deadlock analysis as well as allowing session type inference by means of a conventional type inference algorithm.

1 Origins of EXPRESS: some personal memories

This year marks an important milestone in the history of the EXPRESS/SOS workshop series. Before joining their destinies in 2012, the two workshops EXPRESS and SOS had been running on their own since 1994 and 2004, respectively. Hence, the EXPRESS/SOS'23 workshop in Antwerp will constitute the 30th edition of EXPRESS and the 20th edition of SOS.

Two of us (Ilaria Castellani and Davide Sangiorgi) were personally involved in the very first edition of EXPRESS in 1998, and indeed, they may be said to have carried the workshop to the baptismal font, together with Robert de Simone and Catuscia Palamidessi. Let us recall some facts and personal memories. The EXPRESS workshops were originally held as meetings of the European project EXPRESS, a Network of Excellence within the Human Capital and Mobility programme, dedicated to expressiveness issues in Concurrency Theory. This NoE, which lasted from January 1994 till December 1997, gathered researchers from several European countries and was particularly fruitful in supporting young researchers' mobility across different sites. The first three workshops of the NoE were held in Amsterdam (1994), Tarquinia (1995), and Dagstuhl (1996). The fourth and final workshop was held in Santa Margherita Ligure (1997). It was co-chaired by Catuscia Palamidessi and Joachim Parrow, and stood out as a distinctive event, open to external participants and organised as a conference with a call for papers. A few months after this workshop, in the first half of 1998, the co-chairs of the forthcoming CONCUR'98 conference in Nice, Robert de Simone and Davide Sangiorgi, were wondering about endowing CONCUR with a satellite event (such events were still unusual at the time) in order to enhance its attractiveness. Moreover, Davide was sharing offices with Ilaria, who had been the NoE responsible for the site of Sophia Antipolis and was also part of the organising committee of CONCUR'98. It was so, during informal discussions, that the idea of launching EXPRESS as a stand-alone event affiliated with CONCUR was conceived, in order to preserve the heritage of the NoE and give it a continuation. Thus the first edition of EXPRESS, jointly chaired by Catuscia and Ilaria, took place in Nice in 1998, as the first and unique satellite event of CONCUR. However, EXPRESS did not remain a lonely satellite for too long, as other workshops were to join the orbit of CONCUR in the following years (INFINITY, YR-CONCUR, SecCo, TRENDS, ...), including SOS in 2004. The workshop EXPRESS'98 turned out

G. Caltais and C. A. Mezzina (Eds): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2023).
EPTCS 387, 2023, pp. 8–25, doi:10.4204/EPTCS.387.2

to be successful and very well attended. Since then, EXPRESS has been treading its path as a regular satellite workshop of CONCUR, with a new pair of co-chairs every year, each co-chair serving two editions in a row. The workshop, which is traditionally held on the Monday preceding CONCUR, has always attracted good quality submissions and has maintained a faithful audience over the years.

Coincidentally, this double anniversary of EXPRESS/SOS falls in the 30th anniversary of Kohei Honda's first paper on session types [26]. For this reason, we propose an overview of a particular expressiveness issue, namely the addition of session types to process calculi for mobility such as the π-calculus.

2 Session types and their expressiveness: introduction

Expressiveness is a key topic in the design and implementation of programming languages and models. The issue is particularly relevant in the case of formalisms for parallel and distributed systems, due to the breadth and variety of constructs that have been proposed.

Most importantly, the study of expressiveness has practical applications. If the behaviours that can be programmed by means of a certain formalism L_1 can also be programmed using another formalism L_2, then methods and concepts developed for the latter language (e.g., reasoning and implementation techniques) may be transferred onto the former one that, in turn, may be more convenient to use from a programming viewpoint. An important instance is the case when L_2 is, syntactically, a subset of L_1. Indeed the quest for a "minimal" formalism is central in the work on expressiveness.

This paper is an overview of a particular expressiveness issue, namely the addition of session types onto calculi for mobility such as the π-calculus. We will review the encoding of binary session types onto the standard π-calculus [14, 15], based on an observation of Kobayashi [33]. The key idea of the encoding is to represent a sequence of communications within a session as a chain of communications on linear channels (channels that are meant to be used exactly once) through the use of explicit continuations, a technique that resembles the modelling of communication patterns in the actor model [25]. We discuss extensions of the encoding to subtyping, polymorphism and higher-order communication as well as multiparty session types. Finally, we review two applications of the encoding to the problems of deadlock analysis and of session type inference.

Session types, initially proposed in [26, 51, 27], describe *sessions*, i.e., interaction protocols in distributed systems. While originally designed for process calculi, they have later been integrated also in other paradigms, including (multi-threaded) functional programming [54, 44, 37, 40, 20, 35], component-based systems [52], object-oriented languages [18, 19, 7], languages for Web Services and Contracts [9, 38]. They have also been studied in logical-based type systems [5, 55, 6, 13, 36].

Session types allow one to describe the sequences of input and output operations that the participants of a session are supposed to follow, explicitly indicating the types of messages being transmitted. This structured *sequentiality* of operations makes session types suitable to model protocols. Central (type and term) constructs in session types are also branch and select, the former being the offering of a set of alternatives and the latter being the selection of one of the possible options at hand.

Session types were first introduced in a variant of the π-calculus to describe binary interactions. Subsequently, they have been extended to *multiparty sessions* [28], where several participants interact with each other. In the rest of this paper, we will focus on *binary session types*.

Session types guarantee privacy and communication safety within a session. Privacy means that session channels are known and used only by the participants involved in the session. Communication safety means that interaction within a session will proceed without mismatches of direction and of message type. To achieve this, a session channel is split into two endpoints, each of which is owned by one

of the participants. These endpoints are used according to dual behaviours (and thus have dual types), namely one participant sends what the other one is expecting to receive and vice versa. Indeed, *duality* is a key concept in the theory of session types.

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

If the type of the server endpoint x is

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

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

$$\overline{S} \triangleq !\texttt{Int}.!\texttt{Int}.?\texttt{Bool}.\texttt{end}$$

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

There is a precise moment at which a session between two participants is established. It is the *connection* phase, when a fresh (private) session channel is created and its endpoints are bound to each communicating process. The connection is also the moment when duality, hence mutual compliance of two session types, is verified. In order to establish a connection, primitives like `accept/request` or (νxy), are added to the syntax of terms [51, 27, 53].

When session types and session terms are added to the syntax of standard π-calculus types and terms, respectively, the syntax of types (and, as a consequence, of type environments) usually needs to be split into two separate syntactic categories, one for session types and the other for standard π-calculus types [51, 27, 56, 22]. Common typing features, like subtyping, polymorphism, recursion have then to be added to both syntactic categories. Also the syntax of processes will contain both standard π-calculus process constructs and session process constructs (for example, the constructs mentioned above to create session channels). These syntactic redundancies bring in redundancies also in the theory, and can make the proofs of properties of the language heavy. Moreover, if a new type construct is added, the corresponding properties must be checked both on standard π-types and on session types. By "standard type systems" we mean type systems originally studied in depth for sequential languages such as the λ-calculus and then transplanted onto the π-calculus as types for channel names (rather than types for terms as in the λ-calculus); they include, for instance, constructs for products, records, variants, polymorphism, linearity, capabilities, and so on.

A further motivation for investigating the expressiveness of the π-calculus with or without session types is the similarity between session constructs and standard π-calculus constructs. Consider the type $S = ?\texttt{Int}.?\texttt{Int}.!\texttt{Bool}.\texttt{end}$. This type is assigned to a session channel endpoint and it describes a structured sequence of inputs and outputs by specifying the type of messages that the channel can transmit. This way of proceeding reminds us of the *linearised* channels [34], which are channels used multiple times for communication but only in a sequential manner. Linearised types can, in turn, be encoded into linear types—i.e., channel types used *exactly once* [34]. Similarly, there are analogies between the branch and select constructs of session types and the *variant* types [45, 46] of standard π-calculus types, as well as between the duality of session types, in which the behaviour of a session channel is split into

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

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

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

Figure 1: Syntax and reduction semantics of the session π-calculus

two endpoints, and the *capability types* of the standard π-calculus, that allow one to separate the input and output usages of channels.

In this paper we follow the encoding of binary session types into linear π-types from [14, 15], then discuss some extensions and applications. The encoding was first suggested by Kobayashi [33], as a proof-of-concept without however formally studying it. Later, Demangeon and Honda [17] proposed an encoding of session types into π-types with the aim of studying the subtyping relation, and proving properties such as soundness of the encoding with respect to typing and full abstraction.

Structure of the paper. The rest of the paper is organised as follows. In Section 3 we introduce the necessary background about the session π-calculus and the linear π-calculus. In Section 4 we recall the encoding from the session π-calculus into the linear π-calculus, as well as its correctness result. In Section 5 and Section 6 we discuss respectively some extensions and some applications of the encoding.

3 Background: π-calculus and session types

In this section, we recall the syntax and semantics of our two calculi of interest: the session π-calculus and the standard typed π-calculus. We also introduce the notion of duality for session types.

$$
\begin{array}{rlll|lll}
t ::= & \ell_{\mathtt{o}}[\widetilde{t}] & \text{(linear output)} & & \sharp[\widetilde{t}] & \text{(connection)} \\
 & \ell_{\mathtt{i}}[\widetilde{t}] & \text{(linear input)} & & \langle l_i \_ t_i \rangle_{i \in I} & \text{(variant type)} \\
 & \ell_{\sharp}[\widetilde{t}] & \text{(linear connection)} & & \mathtt{Unit} & \text{(unit type)} \\
 & \emptyset[] & \text{(no capability)} & & \ldots & \text{(other types)}
\end{array}
$$

$$
\begin{array}{rlll|lll}
P,Q ::= & x!\langle \widetilde{v} \rangle.P & \text{(output)} & & \mathbf{0} & \text{(inaction)} \\
 & x?(\widetilde{y}).P & \text{(input)} & & P \mid Q & \text{(composition)} \\
 & (\nu x)P & \text{(restriction)} & & \mathbf{case}\ v\ \mathbf{of}\ \{l_i \_ (x_i) \triangleright P_i\}_{i \in I} & \text{(case)} \\
 & & & & & \\
v ::= & x & \text{(name)} & & \star & \text{(unit value)} \\
 & l \_ v & \text{(variant value)} & & &
\end{array}
$$

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

Figure 2: Syntax and reduction rules of the standard typed π-calculus

Session types and terms. The syntax for session types and session π-calculus terms is reported in Figure 1, together with the rules for the reduction semantics, in which $\equiv$ is the usual *structural congruence* relation, allowing one to rearrange parallel compositions and the scope of restrictions and to remove useless restrictions. We refer to, e.g., [53, 22] for the rules for typing. Session types range over S and types range over T; the latter include session types, standard channel types denoted by $\sharp T$, data types, such as `Unit` and any other type construct needed for mainstream programming.

Session types are: `end`, the type of a terminated channel; $?T.S$ and $!T.S$ (used in the equality test example given in the introduction) indicating, respectively, the receive and send of a value of type T, with continuation type S. Branch and select are sets of labelled session types, whose labels have indices ranging over a non-empty set I. Branch $\&\{l_i : S_i\}_{i \in I}$ indicates an external choice, namely what is offered, and it is a generalisation of the input type in which the continuation S_i *depends* on the received label l_i. Dually, select $\oplus\{l_i : S_i\}_{i \in I}$ indicates an internal choice, where only one of the available labels l_i's will be chosen, and it is a generalisation of the output type.

Session processes range over P, Q. The output process $x!\langle v \rangle.P$ sends a value v on channel endpoint x and continues as P; the input process $x?(y).P$ receives on x a value to substitute for the placeholder y in the continuation P. The selection process $x \triangleleft l_j.P$ selects label l_j on channel x and proceeds as P. The branching process $x \triangleright \{l_i : P_i\}_{i \in I}$ offers a range of labelled alternative processes on channel x. The session restriction construct $(\nu xy)P$ creates a session channel, more precisely its two endpoints x and y, and binds them in P. As usual, the term $\mathbf{0}$ denotes a terminated process and $P \mid Q$ the parallel composition of P and Q.

Duality Session type duality is a key ingredient in session types theory as it is necessary for communication safety. Two processes willing to communicate, e.g., the client and the server in the equality test, must first agree on a session protocol. Intuitively, client and server should perform dual operations: when one process sends, the other receives, when one offers, the other chooses. Hence, the dual of an input must be an output, the dual of branch must be a select, and vice versa. Formally, duality on session types is defined as the following function:

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

The static checks performed by the typing rules make sure that the peer endpoints of the same session channel have dual types. In particular, this is checked in the restriction rule (T-RES) below:

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

Standard π-calculus. The syntax and reduction semantics for the standard π-calculus are shown in Figure 2. We use t to range over standard π-types, to distinguish them from types T and session types S, given in the previous paragraph. We also use the notation $\widetilde{\cdot}$ to indicate (finite) sequences of elements. Standard π-types specify the *capabilities* of channels. The type $\emptyset[]$ is assigned to a channel without any capability, which cannot be used for any input/output action. Standard types $\ell_{\mathtt{i}}[\widetilde{t}]$ and $\ell_{\mathtt{o}}[\widetilde{t}]$ are assigned to channels used *exactly once* to receive and to send a sequence of values of type $\widetilde{t}$, respectively. The variant type $\langle l_i \_ t_i \rangle_{i\in I}$ is a labelled form of disjoint union of types t_i whose indices range over a set I.

Linear types and variant types are essential in the encoding of session types. The addition of variant types, as of any structured type, implies the addition of a constructor in the grammar for values, to produce variant values of the form l_v, and of a destructor in the grammar for processes, to consume variant values. Such a destructor is represented by the term **case** v **of** $\{l_i \_ (x_i) \triangleright P_i\}_{i\in I}$, offering different behaviours depending on which variant value l_v is received and binding v to the corresponding x_i. In the operational semantics, the reduction rule in which a variant value is consumed (Rπ-CASE) is sometimes called *case normalisation*. Unlike the session π-calculus, the standard π-calculus has just one restriction operator that acts on single names, as in $(\nu x)P$.

4 Encoding sessions

In this section we present the encoding of session π-calculus types and terms into linear π-calculus types and terms, together with the main technical results, following Dardha et al. [14, 15].

Type encoding. The encoding of session types into linear π-types is shown at the top of Figure 3. Types produced by grammar T are encoded in a homomorphic way, e.g., $[\![\sharp T]\!] \triangleq \sharp[\![T]\!]$. The encoding of `end` is a channel with no capabilities $\emptyset[]$ that cannot be used further. Type $?T.S$ is encoded as the linear input channel type carrying a pair of values whose types are the encodings of T and of S. The encoding of $!T.S$ is similar except that the type of the second component of the pair is the encoding of $\overline{S}$, since it

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

$$
\begin{array}{rcll}
[\![x]\!]_f & \triangleq & f_x & \text{(E-Name)} \\
[\![\star]\!]_f & \triangleq & \star & \text{(E-Star)}
\end{array}
$$

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

Figure 3: Encoding of types, values and processes.

describes the type of a channel as it will be used by the receiver process. The branch and the select types are encoded as linear input and linear output channels carrying variant types having labels l_i and types that are respectively the encoding of S_i and the encoding of $\overline{S_i}$ for all $i \in I$. Again, the reason for using duality of the continuation in the encoding of the select type is the same as for the output type, as select is a generalisation of output type.

Process encoding. The encoding of session processes into standard π-processes is shown at the bottom of Figure 3. The encoding of a process P is parametrised by a function f from channel names to channel names. We say that f is a *renaming function for* P if, for all the names x that occur free in P, either $f(x) = x$ or $f(x)$ is a fresh name not occurring in $\mathtt{n}(P)$, where $\mathtt{n}(P)$ is the set of all names of P, both free and bound. Also, f is the identity function on all bound names of P. Hereafter we write $\mathtt{dom}(f)$ for the domain of f and f_x as an abbreviation for $f(x)$. During the encoding of a session process, its renaming function f is progressively updated. For example, we write $f\{x \mapsto c\}$ or $f\{x,y \mapsto c\}$ for the update of f such that the names x and y are associated to c. The notion of a renaming function is extended also to values as expected. In the uses of the definition of the renaming function f for P (respectively v), process P (respectively value v) will be typed in a typing context, say Γ. It is implicitly assumed that the fresh names used by f (that is, the names y such that $y = f(x)$, for some $x \neq y$) are also fresh for Γ.

The motivation for parametrising the encoding of processes and values with a renaming function stems from the key idea of encoding a structured communication over a session channel as a chain of one-shot communications over *linear* channels. Whenever we transmit some payload on a linear channel, the payload is paired with a fresh continuation channel on which the rest of the communication takes place. Such continuation, being fresh, is different from the original channel. Thus, the renaming

function allows us to keep track of this fresh name after each communication.

We now provide some more details on the encoding of terms. Values are encoded as expected, so that a channel name x is encoded to f_x and the $\star$ unit value is encoded to itself. This encoding is trivially extended to every ground value added to the language. In the encoding of the output process, a new channel name c is created and is sent together with the encoding of the payload v along the channel f_x. The encoding of the continuation process P is parametrised by an updated f where the name x is associated to c. Similarly, the input process listens on channel f_x and receives a pair whose first element (the payload) replaces the name y and whose second element (the continuation channel c) replaces x in the continuation process by means of the updated renaming function $f\{x \mapsto c\}$. As indicated in Section 3, session restriction $(\nu xy)P$ creates two fresh names and binds them in P as the opposite endpoints of the same session channel. This is not needed in the standard π-calculus. The restriction construct $(\nu x)P$ creates and binds a unique name x in P; this name identifies both endpoints of the communicating channel. The encoding of a session restriction process $(\nu xy)P$ is a standard channel restriction process $(\nu c)[\![P]\!]_{f\{x,y\mapsto c\}}$ with the new name c used to substitute both x and y in the encoding of P. Selection $x \triangleleft l_j.P$ is encoded as the process that first creates a new channel c and then sends on f_x a variant value l_j_c, where l_j is the selected label and c is the channel created to be used for the continuation of the session. The encoding of branching receives on f_x a value, typically being a variant value, which is the guard of the **case** process. According to the transmitted label, one of the corresponding processes $[\![P_i]\!]_{f\{x\mapsto c\}}$ for $i \in I$ will be chosen. Note that the name c is bound in any process $[\![P_i]\!]_{f\{x\mapsto c\}}$. The encoding of the other process constructs, namely inaction, standard channel restriction, and parallel composition, acts as a homomorphism.

Example 4.1 (Equality test)**.** *We illustrate the encoding of session types and terms on the equality test from the introduction. Thus we also make use of boolean and integer values, and simple operations on them, whose addition to the encoding is straightforward.*

The encoding of the server's session type S is

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

while that of the client's session type $\overline{S}$ is

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

Note how the encoding of dual session types boils down to linear channel types that have the same *payload and dual* outermost *capabilities $\ell_{\mathtt{i}}[\cdot]$ and $\ell_{\mathtt{o}}[\cdot]$. This property holds in general and can be exploited to express the (complex) notion of session type duality in terms of the (simple) property of type equality, as we will see in Section 6.*

The server process, communicating on endpoint x of type S, is

$$server \triangleq x?(z_1).x?(z_2).x!\langle z_1 == z_2\rangle.\mathbf{0}$$

and the client process, communicating on endpoint y of type $\overline{S}$, is

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

Then we have

$$\begin{aligned}[\![server]\!]_{\{x\mapsto s\}} &= s?(z_1,c).[\![x?(z_2).x!\langle z_1 == z_2\rangle.\mathbf{0}]\!]_{\{x\mapsto c\}} \\ &= s?(z_1,c).c?(z_2,c').(\nu c'')c'!\langle z_1 == z_2, c''\rangle.\mathbf{0}\end{aligned}$$

Similarly,

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

The whole server-client system is thus encoded as follows, using $\emptyset$ for the identity function.

$$[\![(\nu xy)(server \mid client)]\!]_{\emptyset} = (\nu s)[\![(server \mid client)]\!]_{\{x,y\mapsto s\}} = (\nu s)\left([\![server]\!]_{\{x\mapsto s\}} \mid [\![client]\!]_{\{y\mapsto s\}}\right)$$

(The update $\{x,y \mapsto s\}$ reduces to $\{x \mapsto s\}$ on the server and to $\{y \mapsto s\}$ on the client because they do not contain occurrences of y and x respectively.)

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

First, the correctness of an encoded typing judgement on the target terms implies the correctness of the judgement on the source terms, and conversely. Similar results hold for values. The encoding is extended to session typing contexts in the expected manner.

Theorem 4.2 (Type correctness)**.** *The following properties hold:*

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

Theorem 4.2, and more precisely its proof [15, 12], shows that the encoding can be actually used to reconstruct the typing rules of session types. That is, the typing rules for an operator op of the session π-calculus can be 'read back' from the typing of the encoding of op.

Next we recall the operational correctness of the encoding. That is, the property that the encoding allows one to faithfully reconstruct the behaviour of a source term from that of the corresponding target term. We recall that $\rightarrow$ is the reduction relation of the two calculi. We write $\hookrightarrow$ for the extension of the structural congruence $\equiv$ with a *case normalisation* indicating the decomposition of a variant value (Section 3).

Theorem 4.3 (Operational correspondence)**.** *Let P be a session process, Γ a session typing context, and f a renaming function for P such that $[\![\Gamma]\!]_f \vdash [\![P]\!]_f$. Then the following statements hold.*

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

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

Statement 1 of the above theorem tells us that the reduction of an encoded process mimics faithfully the reduction of the source process, modulo structural congruence or case normalisation. Statement 2 of the theorem tells us that if the encoding of a process P reduces to the encoding of a process P' (via some intermediate process Q), then the source process P will reduce directly to P' or it might need a wrap-up under restriction. The reason for the latter is that in the session π-calculus [53], reduction only occurs under restriction and cannot occur along free names. In particular, in the theorem, f is a generic renaming function; this function could map two free names, say x and y, onto the same name; in this case, an input at x and an output at y in the source process could not produce a reduction, whereas they might in the target process.

The two theorems above allow us to derive, as a straightforward corollary, the subject reduction property for the session calculus.

Corollary 4.4 (Session Subject Reduction). *If* $\Gamma \vdash P$ *and* $P \to Q$*, then* $\Gamma \vdash Q$.

Other properties of the session π-calculus can be similarly derived from corresponding properties of the standard π-calculus. For instance, since the encoding respects structural congruence (that is, $P \equiv P'$ if and only if $[\![P]\!]_f \equiv [\![P']\!]_f$), we can derive the invariance of typing under structural congruence in the session π-calculus.

Corollary 4.5 (Session Structural Congruence). *If* $\Gamma \vdash P$ *and* $P \equiv P'$*, then also* $\Gamma \vdash P'$.

5 Extensions

In this section we discuss several extensions for the presented encoding, which have been proposed in order to accommodate the additional features of subtyping, polymorphism, recursion, higher-order communication and multiparty interactions.

Subtyping. Subtyping is a relation between types based on a notion of substitutability. If T is a subtype of T', then any channel of type T can be safely used in a context where a channel of type T' is expected. In the standard π-calculus, subtyping originates from *capability types* — the possibility of distinguishing the input and output usage of channels [43, 46]. (This is analogous to what happens in languages with references, where capabilities are represented by the read and write usages.) Precisely, the input channel capability is co-variant, whereas the output channel capability is contra-variant in the types of values transmitted (the use of capabilities is actually necessary with linear types, as reported in Figure 2). Subtyping can then be enhanced by means of the variant types, which are co-variant both in depth and in breadth. In the case of session π-calculus, subtyping must be dealt with also at the level of session types [22]; for instance, branch and select are both co-variant in depth, whereas they are co-variant and contra-variant in breadth, respectively. This duplication of effort can become heavy, particularly when types are enriched with other constructs (a good example are recursive types). The encoding of session types naturally accommodates subtyping, indeed subtyping of the standard π-calculus can be used to derive subtyping on session types. Writing $<:$ and $\leq$ for, respectively, subtyping for session types and for standard π-types, for instance we have:

Theorem 5.1 (Encoding for Subtyping). $T <: T'$ *if and only if* $[\![T]\!] \leq [\![T']\!]$.

Polymorphism and Higher-Order Communication. *Polymorphism* is a common and useful type abstraction in programming languages, as it allows operations that are generic by using an expression with several types. Parametric polymorphism has been studied in the standard π-calculus [46], and in the π-calculus with session types [4]; for bounded polymorphism in session π-calculus see Gay [21].

The *Higher-Order* π*-calculus* (HOπ) models mobility of processes that can be sent and received and thus can be run locally [46]. Higher-order communication for the session π-calculus [39] has the same benefits as for the π-calculus, in particular, it models code mobility in a distributed scenario.

Extensions of the encoding to support polymorphism and HOπ have been studied in [14, 15, 12] and used to test its robustness. The syntax of types and terms is extended to accommodate the new constructs. For polymorphism, session types and standard π-types are extended with a *type variable* X and with *polymorphic types* $\langle X;T\rangle$ and $\langle X;t\rangle$, respectively. For higher-order communication, session types and standard π-types are extended with the functional type $T \to \sigma$, assigned to a functional term that can be used without any restriction, and with the linear functional type $T \xrightarrow{1} \sigma$ that must be used exactly once. Correspondingly, the syntax of processes is extended to accommodate the *unpacking* process

(**open** v **as** $(X;x)$ **in** P) to deal with polymorphism, and with call-by-value λ-calculus primitives, namely *abstraction* ($\lambda x:T.P$) and *application* (PQ), to deal with higher-order communication.

The encoding of the new type and process constructs is a homomorphism in all cases. Consequently, the proof cases added to Theorems 4.2 and 4.3 are trivial.

Recursion. The encoding was also extended to accommodate recursive types and replicated processes by Dardha [10]. Here, the new added types are a recursive type $\mu X.T$ and a type variable X, as well as the replicated process $*P$. 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. The paper uses a new duality function, called *complement*, which is inspired by the work of Bernardi et al. [2, 1]. Some new cases for the encoding of recursive session types and processes are:

$$\begin{array}{ll} [\![X]\!] & \triangleq X \\ [\![\mu X.S]\!] & \triangleq \mu X.[\![S]\!] \\ [\![*P]\!]_f & \triangleq *[\![P]\!]_f \end{array}$$

The extended encoding is proved to be sound and complete with respect to typing and reduction (aka operational correspondence). We refer the interested reader to [10, 11].

Multiparty Session Types. Multiparty Session Types (MPSTs) [28, 29] accommodate communications between more than two participants. Since their introduction, they have become a major area of investigation within the session type community. Their meta-theory is more complex than that of the binary case, and it is beyond the scope of this paper to revise it in detail.

The core syntax of *multiparty session types* is given by the following grammar

$$\begin{array}{rll} S & ::= & \mathtt{end} \mid X \mid \mu X.S \quad \text{(termination, type variable, recursive type)} \\ & & \mathtt{p}\oplus_{i\in I}!l_i(U_i).S_i \quad \text{(select towards role p)} \\ & & \mathtt{p}\&_{i\in I}?l_i(U_i).S_i \quad \text{(branch from role p)} \end{array}$$

$$B ::= \mathtt{Unit} \mid \ldots \quad \text{(base type)} \qquad U ::= B \mid S \text{ (closed under } \mu) \quad \text{(payload type)}$$

where selection and branching types are annotated with *roles* identifying the participant of a multiparty session to which a message is sent or from which a message is expected. The message consists of a label l_i and a payload of type U_i, whereas the continuation S_i indicates how the session endpoint is meant to be used afterwards.

A multiparty session type describes the behaviour of a participant of a multiparty session with respect to all the other participants it interacts with, identified by their role in the session type. In order to obtain the behaviour of a participant with respect to another *particular* participant of the multiparty session, say q, the multiparty session type must be *projected* onto q. Hereafter, we write $S\upharpoonright\mathtt{q}$ for the *partial projection of S onto* q, referring to [47, 48] for its precise definition. Projection yields a type defined by the following syntax, which resembles that of binary session types:

$$\begin{array}{rll} H & ::= & \mathtt{end} \mid X \mid \mu X.H \quad \text{(termination, type variable, recursive type)} \\ & & \oplus_{i\in I}!l_i(U_i).H_i \quad \text{(select)} \\ & & \&_{i\in I}?l_i(U_i).H_i \quad \text{(branch)} \end{array}$$

Projection is a key feature of MPSTs as it is needed in the technical development of a sound type system. At the same time, it also provides a hook by which multiparty sessions and multiparty session

types can be encoded in the standard π-calculus through the encoding of (binary) session types that we have outlined in Section 3.

Let us briefly comment on the encoding of MPST into linear types given by Scalas et al. [47, 48]. This encoding is fully fledged as it covers the whole MPST and it preserves the theory's *distributivity*. Previous work by Caires and Pérez [3] presents an encoding of MPST into binary session types via a *medium* process, which acts as an orchestrator for the encoding, thus losing distributivity. In the encoding of Scalas et al. no orchestrator is used, hence the encoding preserves its intended choreographic nature as opposed to being orchestrated.

The encoding of a multiparty session type from Scalas et al. is formally defined as:

$$[\![S]\!] \triangleq [\mathtt{p} : [\![S \upharpoonright \mathtt{p}]\!]]_{\mathtt{p} \in S}$$

resulting in a *record* of types with an entry *for each role* $\mathtt{p}$ occurring in the multiparty session type S. The encoding of a projected type, namely $[\![S \upharpoonright \mathtt{p}]\!]$, can then be obtained by suitably adapting the function defined in Figure 3. The main cases are summarised below, and the encoding is a homomorphism for the other constructs in the syntactic category H presented above.

$$\begin{aligned} [\![\oplus_{i \in I} !l_i(U_i).H_i]\!] &\triangleq \ell_{\mathtt{o}}[\langle l_i\_([\![U_i]\!], [\![H_i]\!])\rangle_{i \in I}] \\ [\![\&_{i \in I} ?l_i(U_i).H_i]\!] &\triangleq \ell_{\mathtt{i}}[\langle l_i\_([\![U_i]\!], [\![H_i]\!])\rangle_{i \in I}] \end{aligned}$$

The encoding of processes is quite complex and beyond the scope of this paper. The interested reader may refer to Scalas et al. [47, 49] for the formal details and a Scala implementation of multiparty sessions based on this encoding. The encoding of MPST into linear types satisfies several properties, including duality and subtyping preservation, correctness of the encoding with respect to typing, operational correspondence and deadlock freedom preservation. These properties are given in Section 6 of [47].

6 Applications

The encoding from session types to linear channel types can be thought of as a way of "explaining" a high-level type language in terms of a simpler, lower-level type language. Protocols written in the lower-level type language tend to be more cumbersome and less readable than the session types they encode. For this reason, it is natural to think of the encoding as nothing more than a theoretical study. Yet, as we are about to see in this section, the very same encoding has also enabled (or at least inspired) further advancements in the theory and practice of session types.

6.1 A Type System for Deadlock Freedom

A well-typed session π–calculus process (and equivalently a well-typed standard π–calculus one) enjoys communication safety (no message with unexpected type is ever exchanged) but not deadlock freedom. For example, both the session π–calculus process

$$(\nu x_1 x_2)(\nu y_1 y_2)\,(x_1?(z).y_1!\langle z\rangle.\mathbf{0} \mid y_2?(w).x_2!\langle w\rangle.\mathbf{0}) \tag{1}$$

and the standard π–calculus process

$$(\nu x)(\nu y)\,(x?().y!\langle\rangle.\mathbf{0} \mid y?().x!\langle\rangle.\mathbf{0}) \tag{2}$$

are well-typed in the respective typing disciplines, but the behaviours they describe on the two sessions/channels are intermingled in such a way that no communication can actually occur: the input from each session/channel must be completed in order to perform the output on the other session/channel.

Several type systems that ensure deadlock freedom in addition to communication safety have been studied for session and standard typed π-calculi. In a particular line of work, Kobayashi [30, 32] has studied a typing discipline that associates *priorities* to channel types so as to express, at the type level, the relative order in which channels are used, thus enabling the detection of circular dependencies, such as the one shown above. Later on, Padovani [41] has specialised this technique for the linear π–calculus and, as an effect of the encoding illustrated in Section 3, for the session π–calculus as well. To illustrate the technique, in this section we consider a refinement of the linear input/output types in Figure 2 as follows

$$t ::= \ell_{\mathtt{o}}[\widetilde{t}]^m \mid \ell_{\mathtt{i}}[\widetilde{t}]^n \mid \cdots$$

where m and n are integers representing priorities: the smaller the number, the higher the priority with which the channel must be used. In the process (2) above, we could assign the types $\ell_{\mathtt{i}}[]^m$ and $\ell_{\mathtt{o}}[]^n$ to respectively x and y on the lhs of $|$ and the types $\ell_{\mathtt{o}}[]^m$ and $\ell_{\mathtt{i}}[]^n$ to respectively x and y on the rhs of $|$. Note that each channel is assigned two types having *dual polarities* (each channel is used in complementary ways on the two sides of $|$) and the *same priority*. Then, the type system imposes constraints on priorities to reflect the order in which channels are used: on the lhs of $|$ we have the constraint $m < n$ since the input on x (with priority m) blocks the output on y (with priority n); on the rhs of $|$ the opposite happens, resulting in the constraint $n < m$. Obviously, these two constraints are not simultaneously satisfiable, hence the process as a whole is ruled out as ill typed.

In such simple form, this technique fails to deal with most recursive processes. We illustrate the issue through the following server process that computes the factorial of a natural number, in which we use a few standard extensions (replication, conditional, numbers and their operations) to the calculus introduced earlier.

$$*\mathit{fact}?(x,y).\mathbf{if}\ x = 0\ \mathbf{then}\ y!\langle 1\rangle\ \mathbf{else}\ (\nu z)\,(\mathit{fact}!\langle x-1, z\rangle \mid z?(k).y!\langle x \times k\rangle) \tag{3}$$

The server accepts requests on a shared channel *fact*. Each request carries a natural number x and a linear channel y on which the factorial of x is sent as response. The modelling follows the standard recursive definition of the factorial function. In particular, in the recursive case a fresh linear channel z is created from which the factorial k of $x-1$ is received. At that point, the factorial $x \times k$ of x can be sent on y. Now assume, for the sake of illustration, that m and n are the priorities associated with y and z, respectively. Since z is used in the same position as y in the recursive invocation of *fact*, we expect that z and y should have the same type hence the same priority $m = n$. This clashes with the input on z that blocks the output on y, requiring $n < m$. The key observation we can make in order to come up with a more flexible handling of priorities is that a replicated process like (3) above cannot have any *free* linear channel. In fact, the only free channel *fact* is a shared one whereas y is received by the process and z is created within the process. As a consequence, the absolute value of the priorities m and n we associate with y and z does not matter (as long as they satisfy the constraint $n < m$) and they can vary from one request to another. In more technical terms, this corresponds to making *fact polymorphic* in the priority of the channel y received from it and allowing a (priority-limited) form of *polymorphic recursion* when we type outputs such as $\mathit{fact}!\langle x-1, z\rangle$.

It must be pointed out that a process such as (3) is in the scope of Kobayashi's type systems [32]. The additional expressiveness resulting from priority polymorphism enables the successful analysis of recursive processes that interleave actions on different linear channels also in cyclic network topologies. We

do not showcase these more complex scenarios in this brief survey, instead referring the interested reader to [41] for an exhaustive presentation of the technique and to [42] for a proof-of-concept implementation.

As a final remark, it is interesting to note that this technique can be retrofitted to a calculus with native sessions, but it was born in the context of the standard π–calculus, which features a more primitive communication model. The point is that, in the standard π–calculus, sequential communications are encoded in a continuation-passing style, meaning that higher-order channels are the norm rather than the exception. So, the quest for expressive type systems ensuring (dead)lock freedom in the standard π–calculus could not ignore this feature, and this necessity has been a major source of inspiration for the support of priority polymorphism. In this direction, Carbone et al. [8] study (dead)lock freedom for session π-processes using the encoding from Section 4 and the technique from [32] and show that this combined technique is more fine-grained than other ones adopted in session π-calculi. Dardha and Pérez [16] present a full account of the deadlock freedom property in session π-calculi, and compare deadlock freedom obtained by using the encoding and the work from [32] to linear logic approaches, which are used as a yardstick for deadlock freedom.

6.2 Session Type Inference

A major concern regarding all type systems is their realisability and applicability in real-world programming languages. In this respect, session type systems pose at least three peculiar challenges: (1) the fact that session endpoints must be treated as *linear resources* that cannot be duplicated or discarded; (2) the need to *update* the session type associated with a session endpoint each time the endpoint is used; (3) the need to express *session type duality* constraints in addition to the usual *type equality* constraints. The first challenge can be easily dealt with only in those (few) languages that provide native support for linear (or at least affine) types. Alternatively, it is possible to devise mechanisms that detect linearity (or affinity) violations at runtime with a modest overhead. The second challenge can be elegantly addressed by adopting a *functional* API for sessions [24], whereby each function/method using a session endpoint returns (possibly along with other results) the same endpoint with its type suitably updated. The last challenge, which is the focus of this section, is a subtle one since session type duality is a complex relation that involves the whole structure of two session types. In fact, it has taken quite some time even just to *correctly define* duality in the presence of recursive session types [2, 23].

Somewhat surprisingly, the encoding of session types into linear channel types allows us to cope with this challenge in the most straightforward way, simply by *getting rid of it*. In Example 4.1 we have shown two session types, one dual of the other, whose respective encodings are *equal* except for the outermost capabilities. This property holds in general.

Proposition 6.1. *Let $\overline{\cdot}$ be the partial involution on types such that $\overline{\mathbf{0}[]} = \mathbf{0}[]$ and $\overline{\ell_{\mathtt{i}}[\widetilde{t}]} = \ell_{\mathtt{o}}[\widetilde{t}]$ and $\overline{\ell_{\mathtt{o}}[\widetilde{t}]} = \ell_{\mathtt{i}}[\widetilde{t}]$. Then $[\![\overline{S}]\!] = \overline{[\![S]\!]}$ for every S.*

In fact, it is possible to devise a slightly different representation of capabilities so that (session) type duality can be expressed solely in terms of type equality. To this aim, let $\circ$ and $\bullet$ be any two types which we use to represent the absence and presence of a given capability, respectively. We do not need any particular property of $\circ$ and $\bullet$ except the fact that they must be different. In fact, they need not even be inhabited. Now, we can devise a slightly different syntax for linear channel types, as follows:

$$t ::= \ell_{\kappa,\kappa}[\widetilde{t}] \mid \cdots \qquad \kappa ::= \circ \mid \bullet$$

The idea is that a linear channel type carries two separate input and output capabilities (hereafter ranged over by κ and ι), each of which can be either present or absent. For example, $\ell_{\circ,\circ}[]$ would be

the same as $\emptyset[]$, $\ell_{\bullet,\circ}[\widetilde{t}]$ would be the same as $\ell_{\mathtt{i}}[\widetilde{t}]$ and $\ell_{\circ,\bullet}[\widetilde{t}]$ would be the same as $\ell_{\mathtt{o}}[\widetilde{t}]$. With this representation of linear channel types the dual of a type can be defined simply as $\overline{\ell_{\kappa,\iota}[\widetilde{t}]} = \ell_{\iota,\kappa}[\widetilde{t}]$, where the input/output capabilities are swapped. Now, suppose that we wish to express a duality constraint $S = \overline{T}$ stating that S is the dual of T and let $\ell_{\kappa,\iota}[\widetilde{s}] = [\![S]\!]$ and $\ell_{\kappa',\iota'}[\widetilde{t}] = [\![T]\!]$ be the encodings of S and T, respectively. Using Proposition 6.1 and the revised representation of linear channel types we obtain

$$S = \overline{T} \iff \kappa = \iota' \wedge \iota = \kappa' \wedge \widetilde{s} = \widetilde{t}$$

thereby turning a session type duality constraint into a conjunction of type equality constraints.

This apparently marginal consequence of using encoded (as opposed to native) session types makes it possible to rely on completely standard features of conventional type systems to express and infer complex structural relations on session types. In particular, it allows any Hindley-Milner type inference algorithm to perform *session type inference*. `FuSe` [42] is a library implementation of session types for OCaml that showcases this idea at work. The library supports higher-order sessions, recursive session types and session subtyping by piggybacking on OCaml's type system. Clearly, the inferred (encoded) session types are not as readable as the native ones. This may pose problems in the presence of type errors. To address this issue, the library is accompanied by an external tool called `Rosetta` that decodes encoded session types and pretty prints them as native ones using the inverse of the encoding function $[\![\cdot]\!]$.[1] On similar lines, Scalas and Yoshida [50] develop `lchannels`, a Scala library for session types fully based on the encoding of session types into linear types. As a result, the structure of a session type is checked statically by analysing its encoding onto channel types in Scala, while linearity is checked dynamically at run time as in `FuSe`, as Scala has no support for linearity.

References

[1] Giovanni Bernardi, Ornela Dardha, Simon J. Gay & Dimitrios Kouzapas (2014): *On Duality Relations for Session Types*. In: *TGC*, *LNCS* 8902, Springer, pp. 51–66, doi:10.1007/978-3-662-45917-1_4.

[2] Giovanni Bernardi & Matthew Hennessy (2014): *Using Higher-Order Contracts to Model Session Types (Extended Abstract)*. In: *CONCUR*, *LNCS* 8704, Springer, pp. 387–401, doi:10.1007/978-3-662-44584-6_27.

[3] Luís Caires & Jorge A. Pérez (2016): *Multiparty Session Types Within a Canonical Binary Theory, and Beyond*. In Elvira Albert & Ivan Lanese, editors: *FORTE*, *LNCS* 9688, Springer, pp. 74–95, doi:10.1007/978-3-319-39570-8_6.

[4] Luís Caires, Jorge A. Pérez, Frank Pfenning & Bernardo Toninho (2013): *Behavioral Polymorphism and Parametricity in Session-Based Communication*. In: *ESOP*, *LNCS* 7792, Springer, pp. 330–349, doi:10.1007/978-3-642-37036-6_19.

[5] Luís Caires & Frank Pfenning (2010): *Session Types as Intuitionistic Linear Propositions*. In: *Proc. of CONCUR 2010*, *Lecture Notes in Computer Science* 6269, Springer, pp. 222–236, doi:10.1007/978-3-642-15375-4_16.

[6] Luís Caires, Frank Pfenning & Bernardo Toninho (2014): *Linear Logic Propositions as Session Types*. *MSCS*, doi:10.1017/S0960129514000218.

[7] Sara Capecchi, Mario Coppo, Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou & Elena Giachino (2009): *Amalgamating sessions and methods in object-oriented languages with generics*. *Theor. Comput. Sci.* 410(2-3), pp. 142–167, doi:10.1016/j.tcs.2008.09.016.

[1] The source code of `FuSe` and `Rosetta` is publicly available at `https://github.com/boystrange/FuSe`.

[8] Marco Carbone, Ornela Dardha & Fabrizio Montesi (2014): *Progress as Compositional Lock-Freedom*. In: *COORDINATION*, *LNCS* 8459, Springer, pp. 49–64, doi:10.1007/978-3-662-43376-8_4.

[9] Marco Carbone, Kohei Honda & Nobuko Yoshida (2007): *Structured Communication-Centred Programming for Web Services*. In: *ESOP*, *LNCS* 4421, Springer, pp. 2–17, doi:10.1007/978-3-540-71316-6_2.

[10] Ornela Dardha (2014): *Recursive Session Types Revisited*. In: *BEAT*, *EPTCS* 162, pp. 27–34, doi:10.4204/EPTCS.162.4.

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

[12] Ornela Dardha (2016): *Type Systems for Distributed Programs: Components and Sessions*. *Atlantis Studies in Computing* 7, Springer / Atlantis Press, doi:10.2991/978-94-6239-204-5.

[13] Ornela Dardha & Simon J. Gay (2018): *A New Linear Logic for Deadlock-Free Session-Typed Processes*. In Christel Baier & Ugo Dal Lago, editors: *FOSSACS*, *LNCS* 10803, Springer, pp. 91–109, doi:10.1007/978-3-319-89366-2_5.

[14] 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.

[15] Ornela Dardha, Elena Giachino & Davide Sangiorgi (2017): *Session types revisited*. *Inf. Comput.* 256, pp. 253–286, doi:10.1016/j.ic.2017.06.002.

[16] Ornela Dardha & Jorge A. Pérez (2022): *Comparing type systems for deadlock freedom*. *J. Log. Algebraic Methods Program.* 124, p. 100717, doi:10.1016/j.jlamp.2021.100717.

[17] Romain Demangeon & Kohei Honda (2011): *Full Abstraction in a Subtyped pi-Calculus with Linear Types*. In: *CONCUR*, *LNCS* 6901, Springer, pp. 280–296, doi:10.1007/978-3-642-23217-6_19.

[18] Mariangiola Dezani-Ciancaglini, Elena Giachino, Sophia Drossopoulou & Nobuko Yoshida (2007): *Bounded Session Types for Object Oriented Languages*. In: *FMCO*, *LNCS* 4709, Springer, pp. 207–245, doi:10.1007/978-3-540-74792-5_10.

[19] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida & Sophia Drossopoulou (2006): *Session Types for Object-Oriented Languages*. In: *ECOOP 2006*, *LNCS* 4067, Springer, pp. 328–352, doi:10.1007/11785477_20.

[20] Simon Fowler, Wen Kokke, Ornela Dardha, Sam Lindley & J. Garrett Morris (2021): *Separating Sessions Smoothly*. In Serge Haddad & Daniele Varacca, editors: *CONCUR*, *LIPIcs* 203, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 36:1–36:18, doi:10.4230/LIPIcs.CONCUR.2021.36.

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

[22] Simon J. Gay & Malcolm Hole (2005): *Subtyping for session types in the pi calculus*. *Acta Inf.* 42(2-3), pp. 191–225, doi:10.1007/s00236-005-0177-z.

[23] Simon J. Gay, Peter Thiemann & Vasco T. Vasconcelos (2020): *Duality of Session Types: The Final Cut*. In Stephanie Balzer & Luca Padovani, editors: *PLACES@ETAPS*, *EPTCS* 314, pp. 23–33, doi:10.4204/EPTCS.314.3.

[24] Simon J. Gay & Vasco Thudichum Vasconcelos (2010): *Linear type theory for asynchronous session types*. *J. Funct. Program.* 20(1), pp. 19–50, doi:10.1017/S0956796809990268.

[25] Carl Hewitt (1977): *Viewing Control Structures as Patterns of Passing Messages*. *Artif. Intell.* 8(3), pp. 323–364, doi:10.1016/0004-3702(77)90033-9.

[26] Kohei Honda (1993): *Types for Dyadic Interaction*. In: *CONCUR*, *LNCS* 715, Springer, pp. 509–523, doi:10.1007/3-540-57208-2_35.

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

[28] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty asynchronous session types*. In: *POPL*, 43(1), ACM, pp. 273–284, doi:10.1145/1328438.1328472.

[29] Kohei Honda, Nobuko Yoshida & Marco Carbone (2016): *Multiparty asynchronous session types*. *Journal of the ACM* 63(1), p. 9, doi:10.1145/2827695.

[30] Naoki Kobayashi (2002): *A Type System for Lock-Free Processes*. *Inf. Comput.* 177(2), pp. 122–159, doi:10.1006/inco.2002.3171.

[31] Naoki Kobayashi (2002): *Type Systems for Concurrent Programs*. In: *10th Anniversary Colloquium of UNU/IIST*, pp. 439–453, doi:10.1007/978-3-540-40007-3_26.

[32] Naoki Kobayashi (2006): *A New Type System for Deadlock-Free Processes*. In: *CONCUR*, *LNCS* 4137, Springer, pp. 233–247, doi:10.1007/11817949_16.

[33] Naoki Kobayashi (2007): *Type Systems for Concurrent Programs*. Available at http://www.kb.ecei.tohoku.ac.jp/~koba/papers/tutorial-type-extended.pdf. Extended version of [31], Tohoku University.

[34] Naoki Kobayashi, Benjamin C. Pierce & David N. Turner (1999): *Linearity and the pi-calculus*. *ACM Trans. Program. Lang. Syst.* 21(5), pp. 914–947, doi:10.1145/330249.330251.

[35] Wen Kokke & Ornela Dardha (2021): *Deadlock-free session types in linear Haskell*. In Jurriaan Hage, editor: *Haskell*, ACM, pp. 1–13, doi:10.1145/3471874.3472979.

[36] Wen Kokke & Ornela Dardha (2021): *Prioritise the Best Variation*. In Kirstin Peters & Tim A. C. Willemse, editors: *FORTE*, *LNCS* 12719, Springer, pp. 100–119, doi:10.1007/978-3-030-78089-0_6.

[37] Sam Lindley & J. Garrett Morris (2016): *Embedding session types in Haskell*. In: *Proc. of Haskell*, ACM, pp. 133–145, doi:10.1145/2976002.2976018.

[38] Fabrizio Montesi & Nobuko Yoshida (2013): *Compositional Choreographies*. In: *CONCUR*, *LNCS* 8052, Springer, pp. 425–439, doi:10.1007/978-3-642-40184-8_30.

[39] Dimitris Mostrous & Nobuko Yoshida (2007): *Two Session Typing Systems for Higher-Order Mobile Processes*. In: *TLCA*, *LNCS* 4583, Springer, pp. 321–335, doi:10.1007/978-3-540-73228-0_23.

[40] Dominic Orchard & Nobuko Yoshida (2017): *Session Types with Linearity in Haskell*. *Behavioural Types: from Theory to Tools*, pp. 219–242, doi:10.13052/rp-9788793519817.

[41] Luca Padovani (2014): *Deadlock and lock freedom in the linear π-calculus*. In Thomas A. Henzinger & Dale Miller, editors: *CSL-LICS*, ACM, pp. 72:1–72:10, doi:10.1145/2603088.2603116.

[42] Luca Padovani (2017): *Type-Based Analysis of Linear Communications*. In Simon Gay & António Ravara, editors: *Behavioural Types: from Theory to Tools*, River Publishers, pp. 193–217, doi:10.13052/rp-9788793519817.

[43] Benjamin C. Pierce & Davide Sangiorgi (1993): *Typing and Subtyping for Mobile Processes*. In: *LICS*, IEEE Computer Society, pp. 376–385, doi:10.1109/LICS.1993.287570.

[44] Riccardo Pucella & Jesse A. Tov (2008): *Haskell session types with (almost) no class*. In: *Proc. of Haskell*, ACM, doi:10.1145/1411286.1411290.

[45] Davide Sangiorgi (1998): *An Interpretation of Typed Objects into Typed pi-Calculus*. *Inf. Comput.* 143(1), pp. 34–73, doi:10.1006/inco.1998.2711.

[46] Davide Sangiorgi & David Walker (2001): *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press.

[47] Alceste Scalas, Ornela Dardha, Raymond Hu & Nobuko Yoshida (2017): *A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming*. In Peter Müller, editor: *ECOOP*, *LIPIcs* 74, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 24:1–24:31, doi:10.4230/LIPIcs.ECOOP.2017.24.

[48] Alceste Scalas, Ornela Dardha, Raymond Hu & Nobuko Yoshida (2017): *A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming*. Technical Report 2, Imperial College London. Available at https://www.doc.ic.ac.uk/research/technicalreports/2017/#2.

[49] Alceste Scalas, Ornela Dardha, Raymond Hu & Nobuko Yoshida (2017): *A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming (Artifact)*. Dagstuhl Artifacts Ser. 3(2), pp. 03:1–03:2, doi:10.4230/DARTS.3.2.3.

[50] Alceste Scalas & Nobuko Yoshida (2016): *Lightweight Session Programming in Scala*. In Shriram Krishnamurthi & Benjamin S. Lerner, editors: *ECOOP*, *LIPIcs* 56, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 21:1–21:28, doi:10.4230/LIPIcs.ECOOP.2016.21.

[51] Kaku Takeuchi, Kohei Honda & Makoto Kubo (1994): *An Interaction-based Language and its Typing System*. In: *PARLE*, *LNCS* 817, Springer, pp. 398–413, doi:10.1007/3-540-58184-7_118.

[52] Antonio Vallecillo, Vasco Thudichum Vasconcelos & António Ravara (2006): *Typing the Behavior of Software Components using Session Types*. Fundam. Inform. 73(4), pp. 583–598. Available at https://content.iospress.com/articles/fundamenta-informaticae/fi73-4-07.

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

[54] Vasco Thudichum Vasconcelos, Simon J. Gay & António Ravara (2006): *Type checking a multithreaded functional language with session types*. Theor. Comput. Sci. 368(1-2), pp. 64–87, doi:10.1016/j.tcs.2006.06.028.

[55] Philip Wadler (2012): *Propositions as sessions*. In: *ICFP*, ACM, pp. 273–286, doi:10.1145/2364527.2364568.

[56] Nobuko Yoshida & Vasco Thudichum Vasconcelos (2007): *Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited: Two Systems for Higher-Order Session Communication*. Electr. Notes Theor. Comput. Sci. 171(4), pp. 73–93, doi:10.1016/j.entcs.2007.02.056.