Session Types Revisited: A Decade Later

Ornella Dardha  
University of Glasgow  
Glasgow, United Kingdom  
ornella.dardha@glasgow.ac.uk

Elena Giachino  
University of Bologna  
Bologna, Italy  
elena.giachino@unibo.it

Davide Sangiorgi  
University of Bologna  
Bologna, Italy  
davide.sangiorgi@cs.unibo.it

CCS CONCEPTS
- Theory of computation → Process calculi; Logic and verification; • Software and its engineering → Software verification and validation.

KEYWORDS
session types, linear types, process calculus, encoding

This short paper is a retrospection of our original work titled Session Types Revisited [11] published at PPDP’12. We thank the PPDP Steering Committee for awarding us the “PPDP 10 Year Most Influential Paper Award”. In the following, we recall our main contributions, highlight the impact in the literature and we conclude by presenting some ideas for future developments.

1 INTRODUCTION
Session types [21, 22, 42] are a formalism used in communication-based programming and were originally designed for process calculi. In the last nearly 30 years, session types have flourished into an active research area. In addition to concurrency, session types have been studied for other programming paradigms, including OOP, functional, or web services [5, 7, 15, 16, 43, 44]. A key concept in session types theory is duality, which describes complementary behaviours and is a distinctive feature of session types with no clear analogue for e.g., in λ- or π-calculus standard type systems. However, duality as previously defined in the literature [18, 42] can lead to communication type mismatch, in particular in the presence of (non-tail) recursive types. Consequently, several new definitions for duality have been published to fix this issue [2, 3, 19].

Example 1 (Equality Test: Session Types). Let us now illustrate session types and the use of duality with a simple example: the equality test. Consider a server and a client, which communicate over a session channel with endpoints x and y, owned by the server and the client, respectively and exclusively. (We will look at processes in the next section, once we have introduced their syntax). These endpoints must have dual types to guarantee communication safety. If the type of x is

\[ S \triangleq \text{Int}?.\text{Int}!.\text{Bool}.\text{end} \]

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

\[ \overline{S} = \text{!Int}!\text{Int}?.\text{Bool}.\text{end} \]

—the client listening on channel endpoint y sends an integer, followed by another integer, and then receives a boolean—namely, it is exactly the dual type of S, formally denoted by \( \overline{S} \).

Session types and terms are added to the syntax of standard \( \pi \)-calculus types and terms, leading to a split into two separate syntactic categories, one for sessions and the other for standard \( \pi \)-calculus [18, 22, 42, 46]. Typing features, such as subtyping, polymorphism, recursion are then added to both syntactic categories. Additionally, the syntax of processes contains both standard \( \pi \)-calculus processes and session processes, like branching and selection (given in §2). This introduces redundancy not only in the syntax, but also in the theory, and can make the proofs of properties heavy: if a new type construct is added, a related property must be checked for both standard \( \pi \)-types and session types. Note, by “standard” type systems we mean type systems originally studied in depth in sequential languages such as the \( \lambda \)-calculus and then transplanted onto the \( \pi \)-calculus as types for channel names (rather than types for terms as in the \( \lambda \)-calculus). Such type systems may include constructs for products, records, variants, polymorphism, linearity, and so on.

In Session Types Revisited [11] our aim was to understand the extent to which the above redundancy and the duality in session types were necessary. We explored linear channel types [28]—to type channels that are used exactly once; and variant types [37, 38] to type branching and selection processes. By following Kobayashi [27], we defined a continuation-passing style encoding of binary session types into linear and variant types. We proved our encoding is sound and complete with respect to typing and operational semantics. By exploiting this encoding, session types and their theory can be derived from the theory of the standard typed \( \pi \)-calculus. For instance, properties such as subject reduction and type safety become straightforward corollaries. To test the robustness of our encoding, we extended it to accommodate subtyping [18], polymorphism [17, 38] and higher-order communication [33]. We also presented an optimisation of linear channels enabling the reuse of the same channel for the continuation of the communication. Finally, in our encoding, dual session types are mapped onto linear types that are identical, except for the outermost input or output capability, thus completely bypassing the issues with duality that we encounter in the session types literature.

2 SESSION TYPES REVISITED

Session Types and Terms. Session types are given on the left of the encoding of types in Figure 1—they are the domain of the encoding. \( S \) ranges over session types and \( T \) over types, which include session types, standard channel types denoted \( \otimes T \), data types 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 Example 1) indicating, respectively receive and send a value of type \( T \) and continuation \( S \). Branch and select are sets of labelled session types, with labels ranging over a set of indices \( I \). 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. Dually, select
The encoding of terms is given on the right of Figure 1. The encoding of terms is parameterised by a function \( f \) from names to names. Since we are using linear channels, once a channel is used, it cannot be used again for communication. To enable structured communications like session types do, the channel must be renamed: a new channel is created and is sent to the communicating partner in order to use it to continue the rest of the session. This procedure is repeated at every step of communication and the function \( f \) is updated to the new name created. This is the continuation-passing principle. In the encoding of the output process, a new channel \( c \) is created and is sent together with the encoding of the payload \( o \) on the renamed channel \( fc \); the encoding of the continuation process \( P \) is parametrised in \( f \), where name \( x \) is updated to \( c \). Similarly, the input process listens on channel \( f(c) \) and receives a value to substitute \( y \) as well as a fresh channel \( c \) to substitute \( x \) in the encoded continuation process where \( f(x) = c \). Selection process \( x.\ell_1 .P \) is encoded as the process that first creates a new channel \( c \) and then sends on \( f(c) \) a variant label \( l_1 \), where \( l_1 \) is the original selected label and \( c \) is the new channel to be used for the continuation of the session. The encoding of branching is the input process receiving on \( f(c) \) a value, typically being a variant value, which is the guard of the case process. According to the chosen label, one of the corresponding processes \( P_i \) for \( i \in I \), will be chosen. Note that the name \( c \) is bound in any continuation process \( P_i \). The encoding of a session restriction process \((\nu x)P\) is a process \((\nu c)[P]_{f,\{x\mapsto c\}}\) with the new name \( c \) used to substitute both \( x \) and \( y \) in the encoding of \( P \). The encoding of other process constructs, like inaction and parallel composition, is a homomorphism, for e.g., \([P]_{f,\{x\mapsto c\}} \preceq [P]_{f,\{x\mapsto c\}}\).

**Example 2 (Equality Test: Encoding).** Let us now illustrate the encoding of session types and terms on the equality test from Example 1. The encoding of the server’s channel endpoint \( x \) of type \( S \) is

\[
[S] = \ell_1[\text{Int}, \ell_1[\text{Int}, \ell_0[\text{Bool}, \ell_0[][[], []]]]]
\]

Dually, the encoding of the client’s channel endpoint \( y \) of type \( S \) is

\[
[S] = \ell_1[\text{Int}, \ell_1[\text{Int}, \ell_0[\text{Bool}, \ell_0[][[], []]]]]
\]

Here we notice, how the encoding of dual types boils down to dual capabilities \( \ell_1[\cdot] \) and \( \ell_0[\cdot] \) only at the outermost level, while the payloads are identical.

Since we introduced the syntax of processes, we are now in the position to design the server and a client processes.

Let \( \text{server} = [x?\langle v_1, c \rangle . x?\langle v_2 \rangle \cdot x \approx v_2 . 0] \cdot 0 \) be the server process communicating on endpoint \( x \) of type \( S \). Then,

\[
[\text{server}]_{\{x\mapsto v\}} = x?\langle v_1, c \rangle . x?\langle v_2, c' \rangle . ((qc')c'' \cdot (q_1 \mapsto v_2, c'')) . 0
\]
Let \( \text{client} \vdash \forall y . (3, y ? (eq) . 0 \ ) \) be the client process communicating on endpoint \( y \) of type \( \Sigma \). Then,
\[
\| \text{client} \|_{(y \rightarrow)} = (\nu c) v ! (3, c) . (\mu c') c ! (5, c') . c' ? (eq, c'') . 0
\]
Lastly, the whole server-client closed system is encoded as
\[
\| (v x y) (\text{server} \upharpoonright \text{client}) \|_0 = (v s) (s (\text{server} \upharpoonright \text{client}) \|_{(x, y \rightarrow)}) = (v s) (s (\text{server} \|_{(x \rightarrow)} \upharpoonright s \text{client} \|_{y \rightarrow}))
\]
where the initial renaming function is the identity function simply denoted as \( 0 \) above.

## 3 IMPACT

The publication of our work \([11]\) opened the pathway to new research, both in theory and practice of session types. The main applicability of the encoding is to transport the properties of the standard typed \( \pi \)-calculus into session typed languages. This is possible due to the soundness and completeness of the encoding with respect to both typing and semantics, aka operational correspondence. We have already seen how the encoding bypasses the issues with duality, so we now discuss other areas of impact, without pretensions to being exhaustive.

### New Session Types Theory

Building upon the original work, the authors extended it to include recursive types, together with detailed examples and proofs \([8, 9, 12]\). Following the original work, several variations of encodings have emerged, most notably minimal session types \([1]\) and the encoding of multiparty session types (MPST) into linear types \([39]\). Padovani \([35]\) extends the linear \( \pi \)-calculus with composite types and uses our encoding to develop a type inference algorithm for session types with composite types. The resulting type inference algorithm is simple thanks to our encoding and its impact on the duality relation. Graversen et al. \([20]\) also use the encoding to generate type inference for session types based on constraint generation and solving.

#### (Dead) Lock Freedom

The encoding can be used to verify liveliness properties such as (dead)lock freedom and progress. Carbone et al. \([6]\) show that progress is a compositional form of deadlock freedom and by using our encoding and a very expressive standard \( \pi \)-calculus type system \([26]\) they manage to type more session processes than the literature. Padovani \([34]\) defines a new technique building on Kobayashi’s work \([26]\) to verify (dead)lock freedom for the linear \( \pi \)-calculus with cyclic networks. He then uses the encoding to bring this technique onto the session typed \( \pi \)-calculus. Dardha and Pérez \([13, 14]\) give a detailed account of deadlock freedom in the \( \pi \)-calculus by comparing session types, linear logic and standard \( \pi \)-types via our encoding.

### Session Types Implementation

The encoding has been used as a technique for implementing session types into mainstream programming languages with subtyping, recursive types and type inference. Padovani \([36]\) implemented FuSe, a library for session types in OCaml. Scalars and Yoshida \([41]\) implemented lchannels, a library for session types in Scala. Building on the encoding of MPST into linear types, Scala et al. \([39]\) use it to implement MPST for Akka/Scala \([40]\). These works use the continuation-passing encoding by exploring the native channel types in the corresponding languages, but perform runtime checks for linearity. Imai et al. \([23]\) implemented another session library for OCaml, but with static checks. Kokke and Dardha \([29]\) use the encoding to implement deadlock-free session types in Linear Haskell. Finally, our encoding facilitated the implementation of session types in Go \([31]\).

## 4 FUTURE DEVELOPMENTS

We speculate that the encoding of MPST will also take off in the same way as that of binary session types, in particular it will be used for implementations in mainstream programming languages. In addition to new implementations leveraging the binary/MPST encodings, we will also see mechanisations of session types using proofs assistants such as Agda or Coq. Based on Padovani’s work \([35]\) and the mechanisation of the linear \( \pi \)-calculus and its type inference in Agda \([47, 48]\), Dardha and collaborators are currently working on the mechanisation of type inference for session types in Agda. We think that connections between the encodings and linear logic will start to emerge, possibly building on the Curry-Howard isomorphisms between session types and linear logic \([4, 10, 45]\), which has gained momentum in the last decade. Lastly, we think connections between the encoding and typestates—a formalism similar to type specification types developed in the context of object-oriented programming \([24, 30]\)—may emerge, possibly leveraging connections between the \( \pi \)-calculus and objects \([37]\).

### REFERENCES
