Session Types Revisited: A Decade Later

Ornela Dardha
University of Glasgow
Glasgow, United Kingdom
ornela.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, 32, 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

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

—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

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

—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 π-calculus types and terms, leading to a split into two separate syntactic categories, one for sessions and the other for standard π-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 π-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 π-types and session types. Note, by "standard" type systems we mean type systems originally studied in depth in 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). 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 π-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 $\sharp T$, data types and any other type construct needed for mainstream programming. Session types are: $\texttt{end}$, 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

$$\begin{array}{rcl@{\qquad}rcl}
\llbracket \mathsf{end} \rrbracket & \triangleq & \emptyset[\,] & \llbracket x!\langle v\rangle.P \rrbracket_f & \triangleq & (\nu c)f_x!\langle \llbracket v \rrbracket_f, c\rangle.\llbracket P \rrbracket_{f,\{x\mapsto c\}} \\
\llbracket ?T.S \rrbracket & \triangleq & \ell_{\mathrm{i}}[\llbracket T \rrbracket, \llbracket S \rrbracket] & \llbracket x?(y).P \rrbracket_f & \triangleq & f_x?(y,c).\llbracket P \rrbracket_{f,\{x\mapsto c\}} \\
\llbracket !T.S \rrbracket & \triangleq & \ell_{\mathrm{o}}[\llbracket T \rrbracket, \llbracket \overline{S} \rrbracket] & \llbracket x \triangleleft l_j.P \rrbracket_f & \triangleq & (\nu c)f_x!\langle l_j\_c\rangle.\llbracket P \rrbracket_{f,\{x\mapsto c\}} \\
\llbracket \&\{l_i : S_i\}_{i\in I} \rrbracket & \triangleq & \ell_{\mathrm{i}}[\langle l_i\_\llbracket S_i \rrbracket\rangle_{i\in I}] & \llbracket x \triangleright \{l_i : P_i\}_{i\in I} \rrbracket_f & \triangleq & f_x?(y).\ \mathbf{case}\ y\ \mathbf{of}\ \{l_i\_(c) \triangleright \llbracket P_i \rrbracket_{f,\{x\mapsto c\}}\}_{i\in I} \\
\llbracket \oplus\{l_i : S_i\}_{i\in I} \rrbracket & \triangleq & \ell_{\mathrm{o}}[\langle l_i\_\llbracket \overline{S_i} \rrbracket\rangle_{i\in I}] & \llbracket (\nu xy)P \rrbracket_f & \triangleq & (\nu c)\llbracket P \rrbracket_{f,\{x,y\mapsto c\}}
\end{array}$$

Figure 1: Encoding of session types (left) and session terms (right)

$\oplus\{l_i : S_i\}_{i\in I}$ indicates an internal choice, only one of the labels in I will be chosen, and it is a generalisation of the output type.

Session processes ranging over P, Q are given on the left of the encoding of processes in Figure 1—they are the domain of the encoding. The output process $x!\langle v\rangle.P$ sends a value v on channel endpoint x and proceeds as process P; the input process $x?(y).P$ receives on x a value to substitute the placeholder y in the continuation process P. The selection process $x \triangleleft l_j.P$ on x selects label l_j and proceeds as process P. The branching process $x \triangleright \{l_i : P_i\}_{i\in I}$ on x offers a range of labelled alternative processes. $(\nu xy)P$ is the session restriction construct; it creates a session channel, more precisely its two endpoints x and y and binds them in P. Inaction $\mathbf{0}$ and parallel composition $P \mid Q$ are standard and we omit them here.

Standard π-Types and Terms. Standard π-types are given on the right of the encoding of types in Figure 1—they are image of the encoding. Let τ indicate standard π-types, to differentiate from T, S presented above. We will use a tilde $\tilde{\cdot}$ to indicate a sequence of elements. The type $\emptyset[\,]$ is assigned to a channel without any capability. Linear types $\ell_{\mathrm{i}}[\widetilde{\tau}]$ and $\ell_{\mathrm{o}}[\widetilde{\tau}]$ are assigned to channels used *exactly once* to receive and to send values of type $\widetilde{\tau}$, respectively. The variant type $\langle l_i\_\tau_i\rangle_{i\in I}$ is a labelled form of disjoint union of standard types τ_i ranging in a set of indices I.

Standard π-processes ranging over P, Q are given on the right of the encoding of processes in Figure 1—they are the image of the encoding. The output process $x!\langle\tilde{v}\rangle.P$ and input process $x?(\tilde{y}).P$ are similar to the session counterpart, with the only difference that here we need a tuple of values $\tilde{v}$ or placeholders $\tilde{y}$; Process **case** v **of** $\{l_i\_(x_i) \triangleright P_i\}_{i\in I}$ offers different behaviours depending on which labelled value, called variant value l_v it receives. Restriction process $(\nu x)P$ creates a new name x and binds it with scope P. Differently from session π-calculus, here we have only one name x. Inaction and parallel composition are standard.

Encoding of Types and Terms. The encoding of types is given on the left of Figure 1. The encoding of end is a channel with no capabilities $\emptyset[\,]$, meaning that it 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. However, notice that the dual of the continuation $\overline{S}$ is sent, since it is the type of a channel as 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 respectively, the encoding of S_i and the encoding of $\overline{S_i}$ for all $i \in I$. Again, the reason for duality of the continuation is the same as for the output type. All the other types are encoded in a homomorphic way, for e.g., $\llbracket \sharp T \rrbracket \triangleq \sharp\llbracket T \rrbracket$.

The encoding of terms is given on the right of Figure 1. The encoding of terms is parametrised 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 v on the renamed channel f_x; 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_x and receives a value to substitute y as well as a fresh channel c to substitute x in the encoded continuation process where f is updated with $\{x \mapsto c\}$. Selection process $x \triangleleft l_j.P$ 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 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_x 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 $\llbracket P_i \rrbracket_{f,\{x\mapsto c\}}$ for $i \in I$, will be chosen. Note that the name c is bound in any continuation process $\llbracket P_i \rrbracket_{f,\{x\to c\}}$. The encoding of a session restriction process $(\nu xy)P$ is a process $(\nu c)\llbracket P \rrbracket_{f,\{x,y\mapsto c\}}$ 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., $\llbracket P \mid Q \rrbracket_f \triangleq \llbracket P \rrbracket_f \mid \llbracket Q \rrbracket_f$.

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

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

Dually, the encoding of the client's channel endpoint y of type $\overline{S}$ is

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

Here we notice, how the encoding of dual types boils down to dual capabilities $\ell_{\mathrm{i}}[\cdot]$ and $\ell_{\mathrm{o}}[\cdot]$ *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 $\mathit{server} \triangleq x?(v_1).x?(v_2).x!\langle v_1 == v_2\rangle.\mathbf{0}$ be the server process communicating on endpoint x of type S. Then,

$$\llbracket \mathit{server} \rrbracket_{\{x\mapsto s\}} = s?(v_1, c).c?(v_2, c').(\nu c'')c'!\langle v_1 == v_2, c''\rangle.\mathbf{0}$$

Let $client \triangleq y!\langle 3\rangle.y!\langle 5\rangle.y?(eq).\mathbf{0}$ be the client process communicating on endpoint y of type $\overline{S}$. Then,

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

Lastly, the whole server-client closed system is encoded as

$$\begin{aligned}[\![(\nu xy)(server \mid client)]\!]_{\emptyset} \\ &= (\nu s)[\![(server \mid client)]\!]_{\{x,y\mapsto s\}} \\ &= (\nu s)\big([\![server]\!]_{\{x\mapsto s\}} \mid [\![client]\!]_{\{y\mapsto s\}}\big)\end{aligned}$$

where the initial renaming function is the identity function simply denoted as $\emptyset$ 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 π-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 π-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 liveness 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 π-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 π-calculus with cyclic networks. He then uses the encoding to bring this technique onto the session typed π-calculus. Dardha and Pérez [13, 14] give a detailed account of deadlock freedom in the π-calculus by comparing session types, linear logic and standard π-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. Scalas and Yoshida [41] implemented lchannels, a library for session types in Scala. Building on the encoding of MPST into linear types, Scalas 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 π-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 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 session types developed in the context of object-oriented programming [24, 30]—may emerge, possibly leveraging connections between the π-calculus and objects [37].

REFERENCES

[1] Alen Arslanagic, Anda-Amelia Palamariuc, and Jorge A. Pérez. 2021. Minimal Session Types for the π-calculus. In *PPDP*. ACM, 12:1–12:15. https://doi.org/10.1145/3479394.3479407

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

[3] Giovanni Bernardi and Matthew Hennessy. 2014. Using Higher-Order Contracts to Model Session Types (Extended Abstract). In *CONCUR (Lecture Notes in Computer Science, Vol. 8704)*. Springer, 387–401. https://doi.org/10.1007/978-3-662-44584-6_27

[4] Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *CONCUR (LNCS, Vol. 6269)*. Springer, 222–236. https://doi.org/10.1007/978-3-642-15375-4_16

[5] Sara Capecchi, Mario Coppo, Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, and Elena Giachino. 2009. Amalgamating sessions and methods in object-oriented languages with generics. *Theor. Comput. Sci.* 410, 2-3 (2009), 142–167.

[6] Marco Carbone, Ornela Dardha, and Fabrizio Montesi. 2014. Progress as Compositional Lock-Freedom. In *COORDINATION (LNCS, Vol. 8459)*. Springer, 49–64.

[7] Marco Carbone, Kohei Honda, and Nobuko Yoshida. 2007. Structured Communication-Centred Programming for Web Services. In *ESOP (LNCS, Vol. 4421)*. Springer, 2–17.

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

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

[10] Ornela Dardha and Simon J. Gay. 2018. A New Linear Logic for Deadlock-Free Session-Typed Processes. In *FOSSACS (LNCS, Vol. 10803)*. Springer, 91–109. https://doi.org/10.1007/978-3-319-89366-2_5

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

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

[13] Ornela Dardha and Jorge A. Pérez. 2015. Comparing Deadlock-Free Session Typed Processes. In *EXPRESS/SOS (EPTCS, Vol. 190)*. 1–15. https://doi.org/10.4204/EPTCS.190.1

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

[15] Mariangiola Dezani-Ciancaglini, Elena Giachino, Sophia Drossopoulou, and Nobuko Yoshida. 2007. Bounded Session Types for Object Oriented Languages.

In *FMCO (LNCS, Vol. 4709)*. Springer, 207–245. https://doi.org/10.1007/978-3-540-74792-5_10

[16] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. 2006. Session Types for Object-Oriented Languages. In *ECOOP (LNCS, Vol. 4067)*. Springer, 328–352.

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

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

[19] Simon J. Gay, Peter Thiemann, and Vasco T. Vasconcelos. 2020. Duality of Session Types: The Final Cut. In *PLACES (EPTCS, Vol. 314)*. 23–33. https://doi.org/10.4204/EPTCS.314.3

[20] Eva Fajstrup Graversen, Jacob Buchreitz Harbo, Hans Hüttel, Mathias Ormstrup Bjerregaard, Niels Sonnich Poulsen, and Sebastian Wahl. 2014. Type Inference for Session Types in the π-calculus. In *Web Services, Formal Methods, and Behavioral Types*. Springer, 103–121.

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

[22] Kohei Honda, Vasco Vasconcelos, and Makoto Kubo. 1998. Language primitives and type disciplines for structured communication-based programming. In *ESOP (LNCS, Vol. 1381)*. Springer, 22–138.

[23] Keigo Imai, Nobuko Yoshida, and Shoji Yuen. 2019. Session-ocaml: A session-based library with polarities and lenses. *Sci. Comput. Program.* 172 (2019), 135–159. https://doi.org/10.1016/j.scico.2018.08.005

[24] Mathias Jakobsen, Alice Ravier, and Ornela Dardha. 2021. Papaya: Global Typestate Analysis of Aliased Objects. In *PPDP*. ACM. https://doi.org/10.1145/3479394.3479414

[25] Naoki Kobayashi. 2002. Type Systems for Concurrent Programs. In *10th Anniversary Colloquium of UNU/IIST*. 439–453.

[26] Naoki Kobayashi. 2006. A New Type System for Deadlock-Free Processes. In *CONCUR (LNCS, Vol. 4137)*. Springer, 233–247.

[27] Naoki Kobayashi. 2007. Type Systems for Concurrent Programs. (2007). Extended version of [25], Tohoku University.

[28] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. 1999. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.* 21, 5 (1999), 914–947.

[29] Wen Kokke and Ornela Dardha. 2021. Deadlock-free session types in linear Haskell. In *Haskell*. ACM, 1–13. https://doi.org/10.1145/3471874.3472979

[30] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. 2018. Typechecking protocols with Mungo and StMungo: A session type toolchain for Java. *Sci. Comput. Program.* 155 (2018), 52–75. https://doi.org/10.1016/j.scico.2017.10.006

[31] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. 2018. A static verification framework for message passing in Go using behavioural types. In *ICSE*. ACM, 1137–1148. https://doi.org/10.1145/3180155.3180157

[32] Fabrizio Montesi and Nobuko Yoshida. 2013. Compositional Choreographies. In *CONCUR (LNCS, Vol. 8052)*. Springer, 425–439.

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

[34] Luca Padovani. 2014. Deadlock and Lock Freedom in the Linear π-Calculus. In *CSL-LICS*. ACM, New York, NY, USA. https://doi.org/10.1145/2603088.2603116

[35] Luca Padovani. 2015. Type Reconstruction for the Linear π-Calculus with Composite Regular Types. *Logical Methods in Computer Science* Volume 11, Issue 4 (2015). https://doi.org/10.2168/LMCS-11(4:13)2015

[36] Luca Padovani. 2017. A simple library implementation of binary sessions. *Journal of Functional Programming* 27 (2017). https://doi.org/10.1017/S0956796816000289

[37] Davide Sangiorgi. 1998. An Interpretation of Typed Objects into Typed pi-Calculus. *Inf. Comput.* 143, 1 (1998), 34–73.

[38] Davide Sangiorgi and David Walker. 2001. *The Pi-Calculus - a theory of mobile processes.* Cambridge University Press. I–XII, 1–580 pages.

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

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

[41] Alceste Scalas and Nobuko Yoshida. 2016. Lightweight Session Programming in Scala. In *ECOOP (LIPIcs, Vol. 56)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 21:1–21:28. https://doi.org/10.4230/LIPIcs.ECOOP.2016.21

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

[43] Antonio Vallecillo, Vasco Thudichum Vasconcelos, and António Ravara. 2006. Typing the Behavior of Software Components using Session Types. *Fundam. Inform.* 73, 4 (2006), 583–598.

[44] Vasco Thudichum Vasconcelos, Simon J. Gay, and António Ravara. 2006. Type checking a multithreaded functional language with session types. *Theor. Comput. Sci.* 368, 1-2 (2006), 64–87.

[45] Philip Wadler. 2012. Propositions as sessions. In *ICFP*. ACM, 273–286. https://doi.org/10.1145/2364527.2364568

[46] Nobuko Yoshida and 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 (2007), 73–93.

[47] Uma Zalakain and Ornela Dardha. 2021. π with Leftovers: A Mechanisation in Agda. In *FORTE (LNCS, Vol. 12719)*. Springer, 157–174. https://doi.org/10.1007/978-3-030-78089-0_9

[48] Uma Zalakain and Ornela Dardha. 2021. Co-Contextual Typing Inference for the Linear π-Calculus in Agda (Extended Abstract). In *TyDe*. http://www.dcs.gla.ac.uk/~ornela/publications/ZDb21.pdf