The Concurrent Calculi Formalisation Benchmark

Marco Carbone[1], David Castro-Perez[2], Francisco Ferreira[3], Lorenzo Gheri[4], Frederik Krogsdal Jacobsen[5(✉)], Alberto Momigliano[6], Luca Padovani[7], Alceste Scalas[5], Dawit Tirore[1], Martin Vassor[8], Nobuko Yoshida[8], and Daniel Zackon[9]

[1] IT University of Copenhagen, Copenhagen, Denmark
{maca,dati}@itu.dk
[2] University of Kent, Canterbury, UK
d.castro-perez@kent.ac.uk
[3] Royal Holloway, University of London, Egham, UK
francisco.ferreiraruiz@rhul.ac.uk
[4] University of Liverpool, Liverpool, UK
lorenzo.gheri@liverpool.ac.uk
[5] Technical University of Denmark, Kgs. Lyngby, Denmark
{fkjac,alcsc}@dtu.dk
[6] Università degli Studi di Milano, Milan, Italy
momigliano@di.unimi.it
[7] Università di Camerino, Camerino, Italy
luca.padovani@unicam.it
[8] University of Oxford, Oxford, UK
{martin.vassor,nobuko.yoshida}@cs.ox.ac.uk
[9] McGill University, Montreal, Canada
daniel.zackon@mail.mcgill.ca

Abstract. POPLMark and POPLMark Reloaded sparked a flurry of work on machine-checked proofs, and fostered the adoption of proof mechanisation in programming language research. Both challenges were purposely limited in scope, and they do not address concurrency-related issues. We propose a new collection of benchmark challenges focused on the difficulties that typically arise when mechanising formal models of concurrent and distributed programming languages, such as process calculi. Our benchmark challenges address three key topics: linearity, scope extrusion, and coinductive reasoning. The goal of this new benchmark is to clarify, compare, and advance the state of the art, fostering the adoption of proof mechanisation in future research on concurrency.

Keywords: Mechanisation · Process calculi · Benchmark · Linearity · Scope extrusion · Coinduction

1 Introduction

The POPLMark challenge [4] played a pivotal role in advancing the field of proof assistants, libraries, and best practices for the mechanisation of programming

Published by Springer Nature Switzerland AG 2024
I. Castellani and F. Tiezzi (Eds.): COORDINATION 2024, LNCS 14676, pp. 149–158, 2024.
https://doi.org/10.1007/978-3-031-62697-5_9

language research. By providing a shared framework for systematically evaluating mechanisation techniques, it catalysed a significant shift towards publications that include mechanised proofs within the programming language research community. POPLMark Reloaded [1] introduced a similar programme for proofs using logical relations. These initiatives had a narrow focus, and their authors recognised the importance of addressing topics such as coinduction and linearity in the future.

In this spirit, we introduce a new collection of benchmarks crafted to tackle common challenges encountered during the mechanisation of formal models of concurrent and distributed programming languages. We focus on process calculi, as they provide a simple but realistic showcase of these challenges. Concurrent calculi are notably subtle: for instance, it took some years before an incorrect subject reduction proof in the original paper on session subtyping [25] was discovered and then rectified in the extended journal version [26] with the use of polarities. Similarly, other key results in papers on session types have subsequently been proven incorrect [27,45], demonstrating the need for machine-checked proofs.

While results about concurrent formalisms have already been mechanised (as we will discuss further below), our experience is that choosing appropriate mechanisation techniques and tools remains a significant challenge and that their trade-offs are not well understood. This often leads researchers toward a trial-and-error approach, resulting in sub-optimal solutions, wasted mechanisation efforts, and techniques and results that are hard to reuse. For example, Cruz-Filipe et al. [17] note that the high-level parts of mechanised proofs closely resemble the informal ones, while the main challenge lies in getting the infrastructure right.

Our benchmark challenges (detailed on our website linked below) consider *in isolation* three key aspects that frequently pose difficulties when mechanising concurrency theory: *linearity*, *scope extrusion*, and *coinductive reasoning*, as we will discuss in more detail in the next section. Mechanisations must often address several of these aspects at the same time; however, we see the combination of techniques as a next step, as argued in Sect. 3.

We have begun collecting solutions to our challenges on our website:

https://concurrentbenchmark.github.io/

We intend to use the website to promote best practices and tutorials derived from solutions to our challenges. We encourage readers to try the challenges using their favourite techniques, and to send us their solutions and experience reports.

2 Overview and Design Considerations

In this section, we outline the factors considered when designing the benchmark challenges. We begin with some general remarks, then describe the individual design considerations for each challenge, and the criteria for evaluating solutions.

Similarly to the authors of POPLMark, we seek to answer several questions about the mechanisation of the meta-theory of process calculi:

(Q1) What is the current state of the art?
(Q2) Which techniques and best practices can be recommended?
(Q3) What improvements are needed to make mechanisation tools more user-friendly?

To provide a framework in which to answer these questions, our benchmark is designed to satisfy three main design goals:

(G1) To enable the comparison of proof mechanisation approaches by making the challenges accessible to mechanisation experts who may be unfamiliar with concurrency theory;
(G2) To encourage the development of guidelines and tutorials demonstrating and comparing existing proof mechanisation techniques, libraries, and proof assistant features; and
(G3) To prioritise the exploration of reusable mechanisation techniques.

We also aim at strengthening the culture of mechanisation, by rallying the community to collaborate on exploring and developing new tools and techniques.

To achieve design goal (G1), our challenges explore the three aspects (linearity, scope extrusion, coinduction) independently, so that they may be solved individually and in any order; each challenge is small and easily understandable with basic knowledge of textbook concurrency theory, process calculi, and type theory. For mechanisation experts, our challenges should thus be accessible even without any prior understanding of process calculi. The process calculi used in the challenges focus on the features that we want to emphasise, and omit all constructs that would complicate the mechanisation without bringing tangible insights. For concurrency experts venturing into mechanisation, our challenges thus serve as good first steps. The minimality and uniformity of the calculi also allows us to target design goal (G2). For experts in both mechanisation and concurrency, our challenges serve as a framework in which to consider and share best practices and tutorials. Aligned with design goal (G3), our challenges concern the fundamental meta-theory of process calculi. Our challenges centre around well-established results, showcasing proof techniques that can be leveraged in many applications (as we will further discuss in Sect. 3).

2.1 Linearity

Linear typing systems enable the tracking of resource usage in a program. In the case of typed (in particular, session-typed) process calculi, linearity is widely used for checking if and how a channel is used to send or receive values. This substructurality [39, Ch. 1] gives rise to mechanisation difficulties: *e.g.* deciding how to *split the typing context* in a parallel composition.

The goal of our challenge on linear reasoning is to prove a type safety theorem for a process calculus with session types, by combining subject reduction with the absence of errors. For simplicity we model only linear (as opposed to shared) channels. Inspired by Vasconcelos [49], we define a syntax where a restriction ($\boldsymbol{\nu} ab$) binds two dual names a and b as opposite endpoints of the same channel;

their duality is reflected in the type system. We model a simple notion of error: well-typed processes must never use dual channel endpoints in a non-dual way (*e.g.* by performing concurrent send/receive operations on the same endpoint, or two concurrent send operations on dual endpoints). The operational semantics is a standard reduction relation. Proving subject reduction thus requires proving type preservation for structural congruence.

We designed this challenge to focus on linear reasoning while minimising definitions and other concerns. We therefore forgo name passing: send/receive operations only support values that do not include channel names, so the topology of the communication network described by a process cannot change. We do not allow recursion or replication, hence infinite behaviours cannot be expressed. We also forgo more sophisticated notions of error-freedom (*e.g.* deadlock freedom) as proving them would distract from the core linear aspects of the challenge.

In mechanised meta-theory, addressing linearity means choosing an appropriate representation of a linear context. While the latter is perhaps best seen as a multiset, most proof assistants have better support for lists. This representation is intuitive, but may require establishing a large number of technical lemmata that are orthogonal to the problem under study (in our case, proving type safety for session types). Several designs are possible: one can label occurrences of resources to constrain their usage (*e.g.* [16]), or impose a multiset structure over lists (*e.g.* [15,19]). Alternatively, contexts can be implemented as finite maps (as in [12]), whose operations are sensitive to a linear discipline. In all these cases, the effort required to develop the infrastructure is significant. One alternative strategy is to bypass the problem of context splitting by adopting ideas from algorithmic linear type checking. One such approach, known as "typing with leftovers," is exemplified in [51]. Similarly, context splitting can be eliminated by delegating linearity checks to a *linear predicate* defined on the process structure (*e.g.* [44]). These checks serve as additional conditions within the typing rules. Whatever the choice, list-based encodings can be refined to be intrinsically-typed if the proof assistant supports dependent types (see [16,42,47]).

A radically different approach is to adopt a *substructural* meta-logical framework, which handles resource distribution implicitly, including splitting and substitution: users need only map their linear operations to the ones offered by the framework. The only such framework is *Celf* [46] (see the encoding of session types in [8]); unfortunately, *Celf* does not yet fully support the verification of meta-theoretic properties. A compromise is the *two-level* approach, *i.e.* encoding a substructural specification logic in a mainstream proof assistant and then using that logic to state and prove linear properties (for a recent example, see [23]).

2.2 Scope Extrusion

This challenge revolves around the mechanisation of scope extrusion, by which a process can send restricted names to another process, as long as the restriction can safely be extruded to include the receiving process. The setting for this challenge is a "classic" untyped π-calculus, where (unlike the calculi in the other

challenges) names can be sent and received, and bound by input constructs. We define two different semantics for our system:

1. A reduction system: this avoids explicit reasoning about scope extrusion by using structural congruence, allowing implementers to explore different ways to encode the latter (*e.g.* via process contexts or compatible refinement);
2. An (early) labelled transition system.

The goal of our challenge on scope extrusion is to prove that the two semantics are equivalent up to structural congruence.

This is the challenge most closely related to POPLMark, as it concerns the properties of binders, whose encoding has been extensively studied with respect to λ-calculi. Process calculi present additional challenges, typically including several different binding constructs: inputs bind a received name or value, recursive processes bind recursion variables, and restrictions bind names. The first two act similarly to the binders in λ-calculi, but restrictions may be more challenging due to scope extrusion. Scope extrusion requires reasoning about free variables, so approaches that identify α-equivalent processes cannot be directly applied.

Given those peculiarities, the syntax and semantics of π-calculi have been mechanised from an early age (see [37]) with many proof assistants and in many encoding styles. Despite this, almost all of these mechanisations rely on ad-hoc solutions to encode scope extrusion. They range from concrete encodings based on named syntax [37] to basic de Bruijn [30,38] and locally-nameless representation [12]. Nominal approaches are also common (see [6]), but they may be problematic in proof assistants based on constructive type theories. An overall comparison is still lacking, but the case study [3] explores four approaches to encoding binders in Coq in the context of higher-order process calculi. The authors report that working directly with de Bruijn indices was easiest since the approaches developed for λ-calculus binders worked poorly with scope extrusion.

Higher-order abstract syntax (HOAS) has seen extensive use in formal reasoning in this area [13,14,22,32,48]. Its weak form aligns reasonably well with mainstream inductive proof assistants, significantly simplifying the encoding of typing systems and operational semantics. However, when addressing more intricate concepts like bisimulation, extensions to HOAS are needed. These extensions may take the form of additional axioms [32] or require niche proof assistants such as Abella, which features a special quantifier for handling properties related to names [24].

2.3 Coinduction

Process calculi typically include constructs that allow processes to adopt infinite behaviours. Coinduction serves as a fundamental method for the definition and analysis of infinite objects, enabling the examination of their behaviours.

The goal of our challenge on coinductive reasoning is to prove that *strong barbed bisimilarity* can be turned into a congruence by making it sensitive to substitution and parallel composition. The crux of our challenge is the effective

use of coinductive up-to techniques. The intention is that the result should be relatively easy to achieve once the main properties of bisimilarity are established.

The setting for our challenge is an untyped π-calculus augmented with process replication in order to enable infinite behaviours. We do not include name passing since it is orthogonal to our aim of exploring coinductive proof techniques. We base our definition of bisimilarity on a labelled transition system semantics and an observability predicate describing the communication steps available to a process. The description of strong barbed bisimulation is one of the first steps when studying the behaviour of process calculi, both in textbooks (*e.g.* [43]) and in existing mechanisations. Though weak barbed congruence is a more common behavioural equivalence, we prefer strong equivalences to simplify the theory by avoiding the need to abstract over the number of internal transitions in a trace.

While many proof assistants support coinductive techniques, they do so through different formalisms. Some systems even offer multiple abstractions for utilising coinduction. For instance, Agda offers musical notation, co-patterns and productivity checking via sized types [2]; Coq features guarded recursion and refined fixed point approaches via libraries for *e.g.* parameterised coinduction [34], coinduction up-to [41] and interaction trees [50].

When reasoning over bisimilarity many authors rely on the native coinduction offered by the chosen proof assistant [7,27,35,47], while others prefer a more "set-theoretic" approach [6,30,36,40]. Some use both and establish an internal adequacy [32]. Few extend the proof assistant foundations to allow, *e.g.*, reasoning about bisimilarity up-to [14].

2.4 Evaluation Criteria

The motivation behind our benchmark is to obtain evidence towards answering questions (Q1) to (Q3). We are therefore interested not only in the solutions, but also in the experience of solving the challenges with the chosen approach. Solutions to our challenges should be compared on three axes:

1. Mechanisation overhead: the amount of manually-written infrastructure and setup needed to express the definitions in the mechanisation;
2. Adequacy of the formal statements in the mechanisation: whether the proven theorems are easily recognisable as the theorems from the challenge; and
3. Cost of entry for the tools and techniques employed: the difficulty of learning to use the techniques.

Solutions to our challenges need not strictly follow the definitions and lemmata set out in the challenge text, but solutions which deviate from the original challenges should present more elaborate argumentation for their adequacy.

3 Future Work and Conclusions

Our benchmark challenges do not cover all issues in the field, but focus on the fundamental aspects of linearity, scope extrusion, and coinduction. Many mech-

anisations need to combine techniques to handle several of these aspects, and some may also need to handle aspects that are not covered by our benchmark.

Combining techniques for mechanising the fundamental aspects covered in our benchmark is non-trivial. While we focus on the aspects individually to simplify the challenges, we are also interested in exploring how techniques interact.

Much current research on concurrent calculi includes aspects that are not covered by our benchmark challenges, for example constructs such as choice and recursion. Some interesting research topics that build on the fundamental aspects in our benchmark include multiparty session types [31], choreographies [11], higher-order calculi [29], conversation types [10], psi-calculi [5], and encodings between different calculi [20,28]. The meta-theory of these topics includes aspects—*e.g.* well-formedness conditions on global types, partiality of end-point projection functions, *etc.*—that we do not address.

Our coinduction challenge only treats two notions of process equivalence, but many more exist in the literature. Coinduction may also play a role in recursive processes and session types: recursive session types can be expressed in *infinitary form* by interpreting their typing rules coinductively [21,33].

Unlike POPLMark, we consider *animation* of calculi (as in [13]) out of scope for our benchmark. Finally, our challenges encourage, but do not require, exploring proof automation, as offered by *e.g.* the *Hammer* protocol [9,18].

Ultimately, the fundamental aspects covered by our benchmark serve as the building blocks for most current research on concurrent calculi. It is our hope and aim that exploring and comparing solutions to our challenges will move the community closer to a future where the key basic proof techniques for concurrent calculi are as easy to mechanise as they are to write on paper.

Acknowledgments. The work is partially supported by the UK Engineering and Physical Sciences Research Council (EPSRC) grants EP/T006544/2, EP/V000462/1 and EP/X015955/1; Independent Research Fund Denmark RP-1 grant "Hyben"; and Independent Research Fund Denmark RP-1 grant "MECHANIST".

Disclosure of Interests. Alberto Momigliano is a member of the Gruppo Nazionale Calcolo Scientifico-Istituto Nazionale di Alta Matematica (GNCS-INdAM).

References

1. Abel, A., et al.: POPLMark reloaded: mechanizing proofs by logical relations. J. Funct. Program. **29**, 19 (2019). https://doi.org/10.1017/S0956796819000170
2. Abel, A., Pientka, B., Thibodeau, D., Setzer, A.: Copatterns: programming infinite structures by observations. In: POPL 2013: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 27–38. ACM, New York (2013). https://doi.org/10.1145/2429069.2429075
3. Ambal, G., Lenglet, S., Schmitt, A.: HOπ in Coq. J. Autom. Reason. **65**(1), 75–124 (2021). https://doi.org/10.1007/S10817-020-09553-0
4. Aydemir, B.E., et al.: Mechanized metatheory for the masses: the PoplMark challenge. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 50–65. Springer, Heidelberg (2005). https://doi.org/10.1007/11541868_4

5. Bengtson, J., Johansson, M., Parrow, J., Victor, B.: Psi-calculi: a framework for mobile processes with nominal data and logic. Log. Methods Comput. Sci. **7** (2011). https://doi.org/10.2168/LMCS-7(1:11)2011
6. Bengtson, J., Parrow, J.: Formalising the pi-calculus using nominal logic. Log. Methods Comput. Sci. **5** (2009). https://doi.org/10.2168/LMCS-5(2:16)2009
7. Bengtson, J., Parrow, J., Weber, T.: Psi-calculi in Isabelle. J. Autom. Reason. **56**, 1–47 (2016). https://doi.org/10.1007/s10817-015-9336-2
8. Bock, P., Murawska, A., Bruni, A., Schürmann, C.: Representing session types (2016). https://pure.itu.dk/en/publications/representing-session-types, in Dale Miller's Festschrift
9. Böhme, S., Nipkow, T.: Sledgehammer: judgement day. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS (LNAI), vol. 6173, pp. 107–121. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14203-1_9
10. Caires, L., Vieira, H.T.: Conversation types. Theor. Comput. Sci. **411**(51–52), 4399–4440 (2010). https://doi.org/10.1016/j.tcs.2010.09.010
11. Carbone, M., Montesi, F.: Deadlock-freedom-by-design: multiparty asynchronous global programming. In: POPL 2013: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 263–274. ACM, New York (2013). https://doi.org/10.1145/2429069.2429101
12. Castro, D., Ferreira, F., Yoshida, N.: EMTST: engineering the meta-theory of session types. In: Biere, A., Parker, D. (eds.) TACAS 2020. LNCS, vol. 12079, pp. 278–285. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45237-7_17
13. Castro-Perez, D., Ferreira, F., Gheri, L., Yoshida, N.: Zooid: a DSL for certified multiparty computation: from mechanised metatheory to certified multiparty processes. In: PLDI 2021: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, pp. 237–251. ACM, New York (2021). https://doi.org/10.1145/3453483.3454041
14. Chaudhuri, K., Cimini, M., Miller, D.: A lightweight formalization of the metatheory of bisimulation-up-to. In: Leroy, X., Tiu, A. (eds.) CPP 2015: Proceedings of the 4th ACM SIGPLAN Conference on Certified Programs and Proofs, pp. 157–166. ACM (2015). https://doi.org/10.1145/2676724.2693170
15. Chaudhuri, K., Lima, L., Reis, G.: Formalized meta-theory of sequent calculi for linear logics. Theor. Comput. Sci. **781**, 24–38 (2019). https://doi.org/10.1016/j.tcs.2019.02.023
16. Ciccone, L., Padovani, L.: A dependently typed linear π-calculus in Agda. In: PPDP 2020: 22nd International Symposium on Principles and Practice of Declarative Programming, pp. 8:1–8:14. ACM (2020). https://doi.org/10.1145/3414080.3414109
17. Cruz-Filipe, L., Montesi, F., Peressotti, M.: Formalising a turing-complete choreographic language in Coq. In: Cohen, L., Kaliszyk, C. (eds.) ITP 2021: Proceedings of the 12th International Conference on Interactive Theorem Proving. Leibniz International Proceedings in Informatics, Dagstuhl, Germany, vol. 193, pp. 15:1–15:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.ITP.2021.15
18. Czajka, L., Kaliszyk, C.: Hammer for Coq: automation for dependent type theory. J. Autom. Reason. **61**(1–4), 423–453 (2018). https://doi.org/10.1007/S10817-018-9458-4
19. Danielsson, N.A.: Bag equivalence via a proof-relevant membership relation. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 149–165. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32347-8_11

20. Dardha, O., Giachino, E., Sangiorgi, D.: Session types revisited. Inf. Comput. **256**, 253–286 (2017). https://doi.org/10.1016/j.ic.2017.06.002
21. Derakhshan, F., Pfenning, F.: Circular proofs as session-typed processes: a local validity condition. Log. Methods Comput. Sci. **18**(2) (2022). https://doi.org/10.46298/LMCS-18(2:8)2022
22. Despeyroux, J.: A higher-order specification of the π-calculus. In: van Leeuwen, J., Watanabe, O., Hagiya, M., Mosses, P.D., Ito, T. (eds.) TCS 2000. LNCS, vol. 1872, pp. 425–439. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44929-9_30
23. Felty, A.P., Olarte, C., Xavier, B.: A focused linear logical framework and its application to metatheory of object logics. Math. Struct. Comput. Sci. **31**(3), 312–340 (2021). https://doi.org/10.1017/S0960129521000323
24. Gacek, A., Miller, D., Nadathur, G.: Nominal abstraction. Inf. Comput. **209**(1), 48–73 (2011). https://doi.org/10.1016/J.IC.2010.09.004
25. Gay, S., Hole, M.: Types and subtypes for client-server interactions. In: Swierstra, S.D. (ed.) ESOP 1999. LNCS, vol. 1576, pp. 74–90. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49099-X_6
26. Gay, S.J., Hole, M.: Subtyping for session types in the pi calculus. Acta Informatica **42**(2–3), 191–225 (2005). https://doi.org/10.1007/S00236-005-0177-Z
27. Gay, S.J., Thiemann, P., Vasconcelos, V.T.: Duality of session types: the final cut. In: Proceedings of the PLACES 2020. Electronic Proceedings in Theoretical Computer Science, vol. 314, pp. 23–33. Open Publishing Association (2020). https://doi.org/10.4204/eptcs.314.3
28. Gorla, D.: Towards a unified approach to encodability and separation results for process calculi. Inf. Comput. **208**(9), 1031–1053 (2010). https://doi.org/10.1016/j.ic.2010.05.002
29. Hirsch, A.K., Garg, D.: Pirouette: Higher-order typed functional choreographies. Proc. ACM Program. Lang. **6** (2022). https://doi.org/10.1145/3498684
30. Hirschkoff, D.: A full formalisation of π-calculus theory in the calculus of constructions. In: Gunter, E.L., Felty, A. (eds.) TPHOLs 1997. LNCS, vol. 1275, pp. 153–169. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0028392
31. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM **63**(1) (2016). https://doi.org/10.1145/2827695
32. Honsell, F., Miculan, M., Scagnetto, I.: π-calculus in (co)inductive-type theory. Theor. Comput. Sci. **253**(2), 239–285 (2001). https://doi.org/10.1016/S0304-3975(00)00095-5
33. Horne, R., Padovani, L.: A logical account of subtyping for session types. In: Castellani, I., Scalas, A. (eds.) Proceedings of the 14th Workshop on Programming Language Approaches to Concurrency and Communication-Centric Software. EPTCS, vol. 378, pp. 26–37. Open Publishing Association (2023). https://doi.org/10.4204/EPTCS.378.3
34. Hur, C.K., Neis, G., Dreyer, D., Vafeiadis, V.: The power of parameterization in coinductive proof. In: POPL 2013: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 193–206. ACM, New York (2013). https://doi.org/10.1145/2429069.2429093
35. Kahsai, T., Miculan, M.: Implementing Spi calculus using nominal techniques. In: Beckmann, A., Dimitracopoulos, C., Löwe, B. (eds.) CiE 2008. LNCS, vol. 5028, pp. 294–305. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69407-6_33

36. Maksimović, P., Schmitt, A.: HOCore in Coq. In: Urban, C., Zhang, X. (eds.) ITP 2015. LNCS, vol. 9236, pp. 278–293. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22102-1_19
37. Melham, T.F.: A mechanized theory of the π-calculus in HOL. Nordic J. Comput. **1**(1), 50–76 (1994)
38. Perera, R., Cheney, J.: Proof-relevant π-calculus: a constructive account of concurrency and causality. Math. Struct. Comput. Sci. **28**(9), 1541–1577 (2018). https://doi.org/10.1017/S096012951700010X
39. Pierce, B.C. (ed.): Advanced Topics in Types and Programming Languages. MIT Press, London (2004)
40. Pohjola, J., Gómez-Londoño, A., Shaker, J., Norrish, M.: Kalas: a verified, end-to-end compiler for a choreographic language. In: Andronick, J., de Moura, L. (eds.) ITP 2022: Proceedings of the 13th International Conference on Interactive Theorem Proving. Leibniz International Proceedings in Informatics, Dagstuhl, Germany, vol. 237, pp. 27:1–27:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.ITP.2022.27
41. Pous, D.: Coinduction all the way up. In: Grohe, M., Koskinen, E., Shankar, N. (eds.) Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2016, New York, NY, USA, 5–8 July 2016, pp. 307–316. ACM (2016). https://doi.org/10.1145/2933575.2934564
42. Rouvoet, A., Poulsen, C.B., Krebbers, R., Visser, E.: Intrinsically-typed definitional interpreters for linear, session-typed languages. In: Proceedings of the CPP 2020, pp. 284–298. ACM (2020). https://doi.org/10.1145/3372885.3373818
43. Sangiorgi, D., Walker, D.: The π-calculus: A Theory of Mobile Processes. Cambridge University Press, Cambridge (2001)
44. Sano, C., Kavanagh, R., Pientka, B.: Mechanizing session-types using a structural view: enforcing linearity without linearity. Proc. ACM Program. Lang. **7**(OOPSLA), 235:374–235:399 (2023). https://doi.org/10.1145/3622810
45. Scalas, A., Yoshida, N.: Less is more: Multiparty session types revisited. Proc. ACM Program. Lang. **3** (2019). https://doi.org/10.1145/3290343
46. Schack-Nielsen, A., Schürmann, C.: Celf – a logical framework for deductive and concurrent systems (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 320–326. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71070-7_28
47. Thiemann, P.: Intrinsically-typed mechanized semantics for session types. In: Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming, PPDP 2019. ACM, New York (2019). https://doi.org/10.1145/3354166.3354184
48. Tiu, A., Miller, D.: Proof search specifications of bisimulation and modal logics for the π-calculus. ACM Trans. Comput. Logic **11**(2) (2010). https://doi.org/10.1145/1656242.1656248
49. Vasconcelos, V.T.: Fundamentals of session types. Inf. Comput. **217**, 52–70 (2012). https://doi.org/10.1016/j.ic.2012.05.002
50. Xia, L.Y., et al.: Interaction trees: representing recursive and impure programs in Coq. Proc. ACM Program. Lang. **4** (2019). https://doi.org/10.1145/3371119
51. Zalakain, U., Dardha, O.: π with leftovers: a mechanisation in Agda. In: Peters, K., Willemse, T.A.C. (eds.) FORTE 2021. LNCS, vol. 12719, pp. 157–174. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78089-0_9