The Concurrent Calculi Formalisation Benchmark

Marco Carbone[1][0000−0001−9479−2632], David Castro-Perez[2][0000−0002−6939−4189], Francisco Ferreira[3][0000−0001−8494−7696], Lorenzo Gheri[4][0000−0002−3191−7722], Frederik Krogsdal Jacobsen[5][0000−0003−3651−8314], Alberto Momigliano[6][0000−0003−0942−4777], Luca Padovani[7][0000−0001−9097−1297], Alceste Scalas[5][0000−0002−1153−6164], Dawit Tirore[1][0000−0002−1997−5161], Martin Vassor[8][0000−0002−2057−0495], Nobuko Yoshida[8][0000−0002−3925−8557], and Daniel Zackon[9][0009−0008−6153−2955]

[1] IT University of Copenhagen, Copenhagen, Denmark maca@itu.dk, dati@itu.dk
[2] University of Kent, Canterbury, United Kingdom d.castro-perez@kent.ac.uk
[3] Royal Holloway, University of London, Egham, United Kingdom francisco.ferreiraruiz@rhul.ac.uk
[4] University of Liverpool, Liverpool, United Kingdom lorenzo.gheri@liverpool.ac.uk
[5] Technical University of Denmark, Kgs. Lyngby, Denmark fkjac@dtu.dk, 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, United Kingdom martin.vassor@cs.ox.ac.uk, nobuko.yoshida@cs.ox.ac.uk
[9] McGill University, Montreal, Canada daniel.zackon@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 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 [27] was discovered and then rectified in the extended journal version [28] with the use of polarities. Similarly, other key results in papers on session types have subsequently been proven incorrect [29,47], 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. [19] 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 in appendix A and on our website) consider *in isolation* three key aspects that frequently pose difficulties when mechanising concurrency theory, which we will discuss in more detail in the next section: *linearity*, *scope extrusion*, and *coinductive reasoning*. Mechanisations must often address several of these aspects at the same time; however, we see the combination of techniques as a next step, as discussed in section 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:

(Q1) What is the current state of the art in the mechanisation of the meta-theory of process calculi?

(Q2) Which techniques and best practices can be recommended when starting new mechanisation projects involving process calculi?

(Q3) What improvements are needed to make mechanisation tools more user-friendly with regard to the issues faced when mechanising process calculi?

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 mechanisation techniques that are reusable for future research.

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. 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. The minimality and uniformity of the calculi also allows us to target design goal **(G2)**. 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 section 3).

2.1 Linearity

Linear typing systems allow for 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 [42, 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 [51], 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.* [18]), or impose a multiset structure over lists (*e.g.* [17, 21]). Alternatively, contexts can be implemented as finite maps (as in [14]), 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 [53]. Similarly, context splitting can be eliminated by delegating linearity checks to a *linear predicate* defined on the process structure (*e.g.* [46]). 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 [18, 44, 49]).

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* [48] (see the encoding of session types in [10]); 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 [25]).

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 implementors 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 [39]) 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 [39] to basic de Bruijn [32, 41] and locally-nameless representation [14]. Nominal approaches are also common (see [8]), 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 [15, 16, 24, 34, 50]. Its weak form aligns 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 [34] or require niche proof assistants such as Abella, which features a special quantifier for handling properties related to names [26].

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.* [45]) 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 sized types [2]; Coq features guarded recursion and refined fixed point approaches via the parameterised coinduction [36] and interaction tree [52] libraries.

When reasoning over bisimilarity many authors rely on the native coinduction offered by the chosen proof assistant [9, 29, 37, 49], while others prefer a more "set-theoretic" approach [8, 32, 38, 43]. Some use both and establish an internal adequacy [34]. Few extend the proof assistant foundations to allow, *e.g.*, reasoning about bisimilarity up-to [16].

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 mechanisations 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 constructs and recursion. Some interesting research topics that build on the fundamental aspects in our benchmark include multiparty session types [33], choreographies [13], higher-order calculi [31], conversation types [12], psi-calculi [7],

and encodings between different calculi [22, 30]. 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 addresses 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 [23, 35].

Unlike POPLMark, we consider *animation* of calculi (as in [15]) 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 [11, 20].

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.

References

1. Abel, A., Allais, G., Hameer, A., Pientka, B., Momigliano, A., Schäfer, S., Stark, K.: 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 '13: Proc. 40th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. p. 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., Bohannon, A., Fairbairn, M., Foster, J.N., Pierce, B.C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., Zdancewic, S.: Mechanized metatheory for the masses: The POPLMark challenge. In: Hurd, J., Melham, T. (eds.) Theorem Proving in Higher Order Logics. pp. 50–65. Springer, Berlin & Heidelberg (2005). https://doi.org/10.1007/11541868_4
5. Barber, A.: Dual intuitionistic linear logic. Tech. Rep. ECS-LFCS-96-347, University of Edinburgh (1996), https://www.lfcs.inf.ed.ac.uk/reports/96/ECS-LFCS-96-347/
6. Barendregt, H.P.: The Lambda Calculus: Its Syntax and Semantics, Studies in Logic and the Foundation of Mathematics, vol. 103. North-Holland, 2 edn. (1984)
7. 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** (Mar 2011). https://doi.org/10.2168/LMCS-7(1:11)2011
8. Bengtson, J., Parrow, J.: Formalising the pi-calculus using nominal logic. Log. Methods Comput. Sci. **5** (Jun 2009). https://doi.org/10.2168/LMCS-5(2:16)2009
9. 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
10. 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

11. Böhme, S., Nipkow, T.: Sledgehammer: Judgement day. In: Giesl, J., Hähnle, R. (eds.) IJCAR '10: Proc. 5th Int. Joint Conf. on Automated Reasoning. Lect. Notes Comput. Sci., vol. 6173, pp. 107–121. Springer (2010). https://doi.org/10.1007/978-3-642-14203-1_9
12. 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
13. Carbone, M., Montesi, F.: Deadlock-freedom-by-design: Multiparty asynchronous global programming. In: POPL '13: Proc. 40th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. p. 263–274. ACM, New York (2013). https://doi.org/10.1145/2429069.2429101
14. Castro, D., Ferreira, F., Yoshida, N.: EMTST: Engineering the meta-theory of session types. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. Lect. Notes Comput. Sci., vol. 12079, pp. 278–285. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45237-7_17
15. 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 '21: Proc. 42nd ACM SIGPLAN Int. Conf. on Programming Language Design and Implementation. p. 237–251. ACM, New York (2021). https://doi.org/10.1145/3453483.3454041
16. Chaudhuri, K., Cimini, M., Miller, D.: A lightweight formalization of the metatheory of bisimulation-up-to. In: Leroy, X., Tiu, A. (eds.) CPP '15: Proc. 4th ACM SIGPLAN Conf. on Certified Programs and Proofs. pp. 157–166. ACM (2015). https://doi.org/10.1145/2676724.2693170
17. 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
18. Ciccone, L., Padovani, L.: A dependently typed linear π-calculus in Agda. In: PPDP '20: 22nd Int. Symp. on Principles and Practice of Declarative Programming. pp. 8:1–8:14. ACM (2020). https://doi.org/10.1145/3414080.3414109
19. Cruz-Filipe, L., Montesi, F., Peressotti, M.: Formalising a Turing-complete choreographic language in Coq. In: Cohen, L., Kaliszyk, C. (eds.) ITP '21: Proc. 12th Int. Conf. on Interactive Theorem Proving. Leibniz Int. Proc. Inform., vol. 193, pp. 15:1–15:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021). https://doi.org/10.4230/LIPIcs.ITP.2021.15
20. 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
21. Danielsson, N.A.: Bag equivalence via a proof-relevant membership relation. In: ITP '12: Proc. 3rd Int. Conf. on Interactive Theorem Proving. Lect. Notes Comput. Sci., vol. 7406, pp. 149–165. Springer (2012). https://doi.org/10.1007/978-3-642-32347-8_11
22. 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
23. 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
24. Despeyroux, J.: A higher-order specification of the π-calculus. In: van Leeuwen, J., Watanabe, O., Hagiya, M., Mosses, P.D., Ito, T. (eds.) Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics. Lect. Notes Comput. Sci., vol. 1872, pp. 425–439. Springer, Berlin & Heidelberg (2000). https://doi.org/10.1007/3-540-44929-9_30

25. 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
26. 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
27. Gay, S.J., Hole, M.: Types and subtypes for client-server interactions. In: Swierstra, S.D. (ed.) ESOP '99: Proc. 8th European Symp. on Programming. Lect. Notes Comput. Sci., vol. 1576, pp. 74–90. Springer (1999). https://doi.org/10.1007/3-540-49099-X_6
28. 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
29. Gay, S.J., Thiemann, P., Vasconcelos, V.T.: Duality of session types: The final cut. In: Proc. PLACES 2020. Electronic Proc. in Theoretical Computer Science, vol. 314, p. 23–33. Open Publishing Association (Apr 2020). https://doi.org/10.4204/eptcs.314.3
30. 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
31. Hirsch, A.K., Garg, D.: Pirouette: Higher-order typed functional choreographies. Proc. ACM Program. Lang. **6** (Jan 2022). https://doi.org/10.1145/3498684
32. Hirschkoff, D.: A full formalisation of π-calculus theory in the calculus of constructions. In: Gunter, E.L., Felty, A. (eds.) Theorem Proving in Higher Order Logics. Lect. Notes Comput. Sci., vol. 1275, pp. 153–169. Springer, Berlin & Heidelberg (1997). https://doi.org/10.1007/BFb0028392
33. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM **63**(1) (Mar 2016). https://doi.org/10.1145/2827695
34. Honsell, F., Miculan, M., Scagnetto, I.: π-calculus in (co)inductive-type theory. Theoretical Computer Science **253**(2), 239–285 (2001). https://doi.org/10.1016/S0304-3975(00)00095-5
35. Horne, R., Padovani, L.: A logical account of subtyping for session types. In: Castellani, I., Scalas, A. (eds.) Proc. 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
36. Hur, C.K., Neis, G., Dreyer, D., Vafeiadis, V.: The power of parameterization in coinductive proof. In: POPL '13: Proc. 40th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. p. 193–206. ACM, New York (2013). https://doi.org/10.1145/2429069.2429093
37. Kahsai, T., Miculan, M.: Implementing spi calculus using nominal techniques. In: Beckmann, A., Dimitracopoulos, C., Löwe, B. (eds.) Logic and Theory of Algorithms. Lect. Notes Comput. Sci., vol. 5028, pp. 294–305. Springer, Berlin & Heidelberg (2008). https://doi.org/10.1007/978-3-540-69407-6_33
38. Maksimović, P., Schmitt, A.: HOCore in Coq. In: Urban, C., Zhang, X. (eds.) Interactive Theorem Proving. Lect. Notes Comput. Sci., vol. 9236, pp. 278–293. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22102-1_19
39. Melham, T.F.: A mechanized theory of the π-calculus in HOL. Nordic J. of Computing **1**(1), 50–76 (Mar 1994)
40. Milner, R.: Communication and Concurrency. Prentice-Hall, USA (1989)
41. 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

42. Pierce, B.C. (ed.): Advanced Topics in Types and Programming Languages. MIT Press, London (Dec 2004)
43. Pohjola, J.r., 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 '22: Proc. 13th Int. Conf. on Interactive Theorem Proving. Leibniz Int. Proc. Inform., vol. 237, pp. 27:1–27:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2022). https://doi.org/10.4230/LIPIcs.ITP.2022.27
44. Rouvoet, A., Poulsen, C.B., Krebbers, R., Visser, E.: Intrinsically-typed definitional interpreters for linear, session-typed languages. In: Proc. CPP '20. pp. 284–298. ACM (2020). https://doi.org/10.1145/3372885.3373818
45. Sangiorgi, D., Walker, D.: The π-calculus: A Theory of Mobile Processes. Cambridge University Press, USA (2001)
46. 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 (Oct 2023). https://doi.org/10.1145/3622810
47. Scalas, A., Yoshida, N.: Less is more: Multiparty session types revisited. Proc. ACM Program. Lang. **3** (Jan 2019). https://doi.org/10.1145/3290343
48. Schack-Nielsen, A., Schürmann, C.: Celf - A logical framework for deductive and concurrent systems (system description). In: IJCAR. Lect. Notes Comput. Sci., vol. 5195, pp. 320–326. Springer (2008). https://doi.org/10.1007/978-3-540-71070-7_28
49. Thiemann, P.: Intrinsically-typed mechanized semantics for session types. In: Proc. 21st Int. Symp. on Principles and Practice of Declarative Programming. PPDP '19, ACM, New York (2019). https://doi.org/10.1145/3354166.3354184
50. Tiu, A., Miller, D.: Proof search specifications of bisimulation and modal logics for the π-calculus. ACM Trans. Comput. Logic **11**(2) (Jan 2010). https://doi.org/10.1145/1656242.1656248
51. Vasconcelos, V.T.: Fundamentals of session types. Inf. Comput. **217**, 52–70 (2012). https://doi.org/10.1016/j.ic.2012.05.002
52. Xia, L.Y., Zakowski, Y., He, P., Hur, C.K., Malecha, G., Pierce, B.C., Zdancewic, S.: Interaction trees: representing recursive and impure programs in Coq. Proc. ACM Program. Lang. **4** (dec 2019). https://doi.org/10.1145/3371119
53. Zalakain, U., Dardha, O.: π with leftovers: A mechanisation in Agda. In: Peters, K., Willemse, T.A.C. (eds.) FORTE '21: Proc. Int. Conf. on Formal Techniques for Distributed Objects, Components, and Systems. Lect. Notes Comput. Sci., vol. 12719, pp. 157–174. Springer (2021). https://doi.org/10.1007/978-3-030-78089-0_9

A Challenges

A.1 Preliminaries

First, we list some common notions and conventions that we use in the challenges. Since the calculi under study are somewhat different, each section lists the changes that apply.

We assume the existence of some set of *base values*, represented by the symbols $a, b, \ldots$, the existence of some set of *variables*, represented by the symbols $l, m, \ldots$, and the existence of some set of *names*, represented by the symbols $x, y, \ldots$.[10] We assume that all of these sets are infinite and that their elements can be compared for equality. We assume that all of these sets are infinite and that their elements can be compared for equality.

The syntax of processes includes: the process $\mathbf{0}$ or *inaction*, a process which can do nothing. The process $P \mid Q$ is the *parallel composition* of process P and process Q. The two components can proceed independently of each other, or they can interact via shared names.

For communication, processes include *input* and *output*, whose signature signature depends on the calculus being value-passing or name-passing. We use here the metavariables c, k to abstract over this choice — i.e. c may be either a value or a variable or a name, whereas k may be a variable or a name. The process $x!c.P$ is an *output*, which can send c via x, then continue as P. The process $x?(k).P$ is an *input*, which can receive a c via x, then continue as P with the received element substituted for k. The input operator thus binds k in P.

The process $(\boldsymbol{\nu} x)\ P$ is the *restriction* of the name x to P, binding x in P.

The process $!P$ is the *replication* of the process P. It can be thought of as the infinite composition $P \mid P \mid \cdots$. Replication makes it possible to express infinite behaviours.

We use the notation $\mathrm{fn}(P)$ to denote the set of names that occur free, $\mathrm{bn}(P)$ to denote the set of names that occur bound in P and $\mathrm{fv}(P)$ to denote the set of variables that occur free in P. We use the notation $\mathrm{bv}(P)$ for the set of variables that occur bound in P. We use the notation $P\{a/l\}$ to denote the process P with base value a substituted for variable l. Similarly, $P\{x/y\}$ denotes the process P with name x substituted for name y. We use the notation $P\sigma$ to denote the process P with a finite number of arbitrary substitutions applied to it.

Two processes P and Q are α-convertible, written $P =_\alpha Q$, if Q can be obtained from P by a finite number of substitutions of bound variables. As a convention, we identify α-convertible processes and we assume that bound names and bound variables of any processes are chosen to be different from the names and variables that occur free in any other entities under consideration,

[10] Unlike the standard π-calculus, we distinguish variables from names to better control the expressiveness of the calculi under study, and the scope of the corresponding challenges: the key distinction is that names are used as communication channels (and can be sent and received in the scope extrusion challenge), whereas variables are only bound by inputs and cannot be restricted, sent nor received, in the style of value-passing CCS [40].

such as processes, substitutions, and sets of names or variables. This is justified because any overlapping names and variables may be α-converted such that the assumption is satisfied.

A *context* is obtained by taking a process and replacing a single occurrence of $\mathbf{0}$ in it with the special *hole* symbol $[\cdot]$. As a convention, we do *not* identify α-convertible contexts. A context acts as a function between processes: a context C can be *applied* to a process P, written $C[P]$, by replacing the hole in C by P, thus obtaining another process. The replacement is literal, so names and variables that are free in P can become bound in $C[P]$.

We say that an equivalence relation $\mathcal{S}$ is a *congruence* if $(P, Q) \in \mathcal{S}$ implies that for any context C, $(C[P], C[Q]) \in \mathcal{S}$.

A.2 Challenge: Linearity and Behavioural Type Systems

This challenge formalises a proof that requires reasoning about the linearity of channels. Linearity is the notion that a channel must be used exactly once by a process. This is necessary to prove properties about session type systems, and the key issue of this challenge is reasoning about the linearity of context splitting operations. Linear reasoning is also necessary to formalise, *e.g.*, linear and affine types for the π-calculus and cut elimination in linear logics.

The setting for this challenge is a small calculus with a session type system, the syntax and semantics of which are given below. The calculus is a fragment of the one presented in [51], formulated in the dual style of [5].

The main objective of this challenge is to prove type preservation (also known as subject reduction), *i.e.*, that well-typed processes can only transition to processes which are also well-typed in the same context. The second objective is to prove type safety, *i.e.*, that well-typed processes are also well-formed in the sense that they do not use endpoints in a non-dual way.

Syntax. The syntax is given by the grammar

$$\begin{array}{lcl} v, w & ::= & a \mid l \\ P, Q & ::= & \mathbf{0} \mid x!v.P \mid x?(l).P \mid (P \mid Q) \mid (\boldsymbol{\nu} xy)\ P \end{array}$$

where a *value* $v, w, \ldots$ is either a base value a or a variable l.

The output process $x!v.P$ sends the value v via x and then continues as P. The intention is that the value v must be a base value when it is actually sent, and this is enforced in the semantics later on. The input process $x?(l).P$ waits for a base value from x and then continues as P with the received value substituted for the variable l. The process $(\boldsymbol{\nu} xy)\ P$ represents a *session* with endpoints named x and y which are bound in P. In P, the names x and y can be used to exchange messages over the session (sending on x and receiving on y or vice versa). Note that in this calculus channels cannot be sent in messages, therefore the topology of the communication network described by a process cannot change. Also, there is no recursion or replication in the syntax, hence no infinite behaviours can be expressed. In particular, we only model linear (as opposed to shared) channels.

Semantics. We describe the actions that the system can perform through a small step operational semantics. As usual, we use a *structural congruence* relation that equates processes that we deem to be indistinguishable. Structural congruence is the smallest congruence relation that satisfies the following axioms:

$$\textsc{Sc-Par-Comm}\ \frac{}{P \mid Q \equiv Q \mid P} \qquad \textsc{Sc-Par-Assoc}\ \frac{}{(P \mid Q) \mid R \equiv P \mid (Q \mid R)} \qquad \textsc{Sc-Par-Inact}\ \frac{}{P \mid \mathbf{0} \equiv P}$$

$$\textsc{Sc-Res-Par}\ \frac{\{x, y\} \cap \mathrm{fn}(Q) = \emptyset}{(\boldsymbol{\nu} xy)\ P \mid Q \equiv (\boldsymbol{\nu} xy)\ (P \mid Q)} \qquad \textsc{Sc-Res-Inact}\ \frac{}{(\boldsymbol{\nu} xy)\ \mathbf{0} \equiv \mathbf{0}}$$

$$\textsc{Sc-Res}\ \frac{}{(\boldsymbol{\nu} x_1 y_1)\ (\boldsymbol{\nu} x_2 y_2)\ P \equiv (\boldsymbol{\nu} x_2 y_2)\ (\boldsymbol{\nu} x_1 y_1)\ P}$$

The operational semantics are defined as the following relation on processes:

$$\textsc{R-Com}\ \frac{}{(\boldsymbol{\nu} xy)\ (x!a.P \mid y?(l).Q \mid R) \to (\boldsymbol{\nu} xy)\ (P \mid Q\{a/l\} \mid R)} \qquad \textsc{R-Res}\ \frac{P \to Q}{(\boldsymbol{\nu} xy)\ P \to (\boldsymbol{\nu} xy)\ Q}$$

$$\textsc{R-Par}\ \frac{P \to Q}{P \mid R \to Q \mid R} \qquad \textsc{R-Struct}\ \frac{P \equiv P' \qquad P' \to Q' \qquad Q \equiv Q'}{P \to Q}$$

Note that reductions are allowed only for restricted pairs of session endpoints. This makes it possible to formulate subject reduction so that the typing context is exactly the same before and after each reduction. Note also that due to rule R-Com, the process $y?(l).P$ can receive *any* base value. Since the rule R-Com only applies to sending base values, there is no way to send a variable or a name.

Session Types. Our process syntax allows us to write processes that are ill formed in the sense that they either use the endpoints bound by a restriction to communicate in a way that does not follow the intended duality, or attempt to send something which is not a base value. As an example, the process $(\boldsymbol{\nu} xy)\ (x!a.\mathbf{0} \mid y!a.\mathbf{0})$ attempts to send a base value on both x and y, whereas one of the names should be used for receiving in order to guarantee progress. Another example is the process $(\boldsymbol{\nu} xy)\ (x!l.\mathbf{0} \mid y?(l).\mathbf{0})$, which attempts to send a variable that is not instantiated at the time of sending. To prevent these issues, we introduce a *session type system* which rules out ill-formed processes.

Syntax. Our type system does not type processes directly, but instead focuses on the channels used in the process. The syntax of *session types* S, T, unrestricted typing contexts Γ and linear typing contexts Δ is as follows:

$$\begin{array}{rcl} S, T & ::= & \mathbf{end} \mid \mathbf{base} \mid ?.S \mid !.S \\ \Gamma & ::= & \cdot \mid \Gamma, l \\ \Delta & ::= & \cdot \mid \Delta, x : S \end{array}$$

The *end type* **end** describes an endpoint over which no further interaction is possible. The *base type* **base** describes base values. The *input type* $?.S$ describes endpoints used for receiving a value and then according to S. The *output type* $!.S$ describes endpoints used for sending a value and then according to S.

Typing contexts gather type information about names and variables. *Unrestricted* contexts are simply sets of names since we only have one base type. *Linear contexts* associate a type to endpoints. We use the comma as split/union, overloaded to singletons, and $\cdot$ as the empty context. We extend the Barendregt convention [6] to contexts, so that all entries are distinct. Note that the order in which information is added to a type context does not matter.

Since we need to determine whether endpoints are used in complementary ways to determine whether processes are well formed, we need to formally define the dual of a type as follows:

$$\overline{?.S} = !.\overline{S} \qquad \overline{!.S} = ?.\overline{S} \qquad \overline{\mathbf{end}} = \mathbf{end}$$

Note that the dual function is partial since it is undefined for the base type.

Typing Rules. Our type system is aimed at maintaining two invariants:

1. No endpoint is used simultaneously by parallel processes;
2. The two endpoints of the same session have dual types.

The first invariant is maintained by linearly splitting type contexts when typing compositions of processes, the second by requiring duality when typing restrictions.

We have two typing judgments: one for values, and one for processes. The typing rules for values are:

$$\text{T-Base}\ \frac{}{\Gamma \vdash_v a : \mathbf{base}} \qquad \text{T-Var}\ \frac{}{\Gamma, l \vdash_v l : \mathbf{base}}$$

The typing rules for processes are as follows:

$$\text{T-Inact}\ \frac{\mathbf{end}(\Delta)}{\Gamma; \Delta \vdash \mathbf{0}} \qquad \text{T-Par}\ \frac{\Gamma; \Delta_1 \vdash P \qquad \Gamma; \Delta_2 \vdash Q}{\Gamma; \Delta_1, \Delta_2 \vdash P \mid Q} \qquad \text{T-Res}\ \frac{\Gamma; (\Delta, x : T, y : \overline{T}) \vdash P}{\Gamma; \Delta \vdash (\boldsymbol{\nu} xy)\ P}$$

$$\text{T-Out}\ \frac{\Gamma \vdash_v v : \mathbf{base} \qquad \Gamma; \Delta, x : T \vdash P}{\Gamma; (\Delta, x : !.T) \vdash x!v.P} \qquad \text{T-IN}\ \frac{(\Gamma, l); (\Delta, x : T) \vdash P}{\Gamma; (\Delta, x : ?.T) \vdash x?(l).P}$$

Note that we do not need a judgment for typing channels, since it is already folded into the T-In and T-Out rules.

Challenge. The objective of this challenge is to prove subject reduction and type safety for our calculus with session types. We start with some lemmata:

Lemma 1 (Weakening).

1. *If* $\Gamma; \Delta \vdash P$*, then* $(\Gamma, l); \Delta \vdash P$*.*
2. *If* $\Gamma; \Delta \vdash P$*, then* $\Gamma; (\Delta, x : \mathbf{end}) \vdash P$*.*

Proof. By induction on the given derivations.

Lemma 2 (Strengthening). *If* $\Gamma; (\Delta, x : T) \vdash P$ *and* $x \notin fn(P)$*, then* $\Gamma; \Delta \vdash P$*.*

Proof. By induction on the derivation of $\Gamma; (\Delta, x : T) \vdash P$.

T-End Since $\mathbf{end}(\Delta)$, we can just reapply the rule without $x : T$.
T-Par In this case, we have that $\Delta, x : T = \Delta_0, \Delta_1$. By cases on which context x is in, we just apply the induction hypothesis on that context.
T-Res Without loss of generality, we assume that $x \notin \{y, z\}$, for $P = (\boldsymbol{\nu} yz)\ P'$. Since $\Gamma; (\Delta, y : T_0, z : \overline{T_0}, x : T) \vdash P'$, by induction hypothesis, we have $\Gamma; (\Delta, y : T_0, z : \overline{T_0}) \vdash P'$. Applying again T-Res, we have $\Gamma; \Delta \vdash (\boldsymbol{\nu} yz)\ P'$.

The remaining cases are analogous.

Lemma 3 (Substitution). *If* $(\Gamma, l); \Delta \vdash P$ *and* $\Gamma \vdash_v a : \mathbf{base}$ *then* $\Gamma; \Delta \vdash P\{a/l\}$*.*

Proof. By induction on the derivation of $(\Gamma, l); \Delta \vdash P$.

T-End Immediate since $\mathbf{end}(\Delta)$.
T-Par For $P = P_0 \mid P_1$, we apply the induction hypothesis on the derivations for P_0 and P_1.
T-Res Immediate on the premise of the rule.
T-Out Let $P = x!v.P'$. We have that $(\Gamma, l) \vdash_v v : \mathbf{base}$. We must show $\Gamma \vdash_v v : \mathbf{base}$ to build the conclusion with the induction hypothesis and T-Out. We proceed by cases on the structure of $(\Gamma, l) \vdash_v v : \mathbf{base}$.

The remaining cases are analogous.

To prove that congruence preserves typing we need to spell out in more detail the former. First, we denote with $\cdot \overset{a}{\equiv} \cdot$ the relation induced by the six axioms.

Lemma 4 (Preservation for $\overset{a}{\equiv}$). *If* $P \overset{a}{\equiv} Q$*, then* $\Gamma; \Delta \vdash P$ *iff* $\Gamma; \Delta \vdash Q$*.*

Proof. By case analysis on the Sc rule applied:

Par-Comm/Assoc By rearranging sub-derivations noting that order does not matter for linear contexts.
Par-Inact Right-to-left by Lemma 2. Vice-versa, by picking T to be $\mathbf{end}$ and applying Lemma 1, part 2.

Res-Par By case analysis on $x : T$ being linear or **end** and applying weakening and strengthening accordingly.
Res-Inact: By Lemma 1.
Res Noting that order does not matter.

Now, following Sangiorgi and Walker [45], we give rules for the compatible equivalence relation induced by $\cdot \stackrel{a}{\equiv} \cdot$, which we still write as $\cdot \equiv \cdot$: namely, add to reflexivity, symmetry and transitivity the following condition:

$$\textsc{Cong}\quad \frac{P \stackrel{a}{\equiv} Q}{C[P] \equiv C[Q]}$$

Lemma 5 (Preservation for $\equiv$). *If $P \equiv Q$, then $\Gamma; \Delta \vdash P$ iff $\Gamma; \Delta \vdash Q$.*

Proof. By induction on the structure of the derivation of $P \equiv Q$, with an inner induction of the structure of a process context.

Refl Immediate
Sym By IH.
Trans By two appeals to the IH.
Cong By induction on the structure of C. If the context is a hole, apply Lemma 4. In the step case, apply the IH: for example if C has the form $C' \mid R$, noting that $(C' \mid R)[P]$ is equal to $C'[P] \mid R$ we have $\Gamma; \Delta \vdash (C' \mid R)[P]$ iff $\Gamma; \Delta \vdash (C' \mid R)[Q]$ by rule T-Par and the IH.

Theorem 1 (Subject reduction). *If $\Gamma; \Delta \vdash P$ and $P \to Q$, then $\Gamma; \Delta \vdash Q$.*

Proof. By induction on the derivation of $P \to Q$. The cases R-Par and R-Res follow immediately by IH. Case R-Struct appeals twice to preservation of $\equiv$ (Lemma 5) and to the IH. For R-Com, suppose that T-Res introduces in Δ the assumptions $x : !U, y : ?\overline{U}$. Building the only derivation for the hypothesis, we know that $\Delta = \Delta_1, \Delta_2, \Delta_3$ where $\Gamma; \Delta_3 \vdash R$. We also have $\Gamma; (\Delta_1, x : U) \vdash P$, $\mathcal{D}_2$ a proof of $\Gamma, l; (\Delta_2, y : \overline{U}) \vdash Q$ and $\mathcal{V}$ a proof of $\Gamma \vdash_v a : \mathbf{base}$. From $\mathcal{D}_2$ and $\mathcal{V}$ we use the substitution lemma 3 to obtain $\Gamma; \Delta_2, y : \overline{U} \vdash Q\{a/l\}$. We then conclude the proof with rules T-Par (twice) and T-Res.

To formulate safety, we need to formally define what we mean by well-formed process. We say that a process P is *prefixed at variable* x if $P \equiv x!v.P'$ or $P \equiv x?(l).P'$ for some P'. A process P is then *well formed* if, for every P_1, P_2, and R such that $P \equiv (\boldsymbol{\nu} x_1 y_1)\ \ldots (\boldsymbol{\nu} x_n y_n)\ (P_1 \mid P_2 \mid R)$, with $n \geq 0$, it holds that, if P_1 is prefixed at x_1 and P_2 is prefixed at y_1 (or vice versa), then $P_1 \mid P_2 \equiv x_1!a.P_1' \mid y_1?(l).P_2'$, for some P_1' and P_2'.

Note that well-formed processes do not necessarily reduce. For example, the process

$$(\boldsymbol{\nu} x_1 y_1)\ (\boldsymbol{\nu} x_2 y_2)\ (x_1!a.y_2?(l).\mathbf{0} \mid y_2!x_2.y_1?(l).\mathbf{0})$$

is well formed but also irreducible.

Theorem 2 (Type safety). *If $\Gamma; \cdot \vdash P$, then P is well-formed.*

Proof. In order to prove that P is well-formed, let us consider any process of the form $(\boldsymbol{\nu} x_1 y_1)\ \ldots (\boldsymbol{\nu} x_n y_n)\ (P_1 \mid P_2 \mid R)$ that is structurally congruent to P. Clearly, by Lemma 5, well-typedness must be preserved by structural congruence. Moreover, assume that P_1 is prefixed at x_1 and P_2 is prefixed at y_1 such that $P_1 \equiv x_1!v.P_1'$ (the opposite case proceeds similarly). We need to show that $P_2 \equiv y_1?(l).P_2'$. This can be easily done by contradiction. In fact, if $P_2 \equiv y_1!v.P_2'$ then the typing rule for restriction would be violated since the type of x_1 and y_1 cannot be dual.

Corollary 1. *If $\Gamma; \cdot \vdash P$ and $P \to Q$, then Q is well formed.*

A.3 Challenge: Name Passing and Scope Extrusion

This challenge formalises a proof that requires explicit scope extrusion. Scope extrusion is the notion that a process can send restricted names to another process, as long as the restriction can safely be "extruded" (*i.e.*, expanded) to include the receiving process. This, for instance, allows a process to set up a private connection by sending a restricted name to another process, then using this name for further communication. The key issue of this challenge is reasoning about names that are "in the process" of being scope-extruded, which often presents difficulties for the mechanisation of binders.

Reasoning about scope extrusion explicitly can sometimes be avoided by introducing a structural congruence rule into the semantics, but doing this means we lose information about the scope when reasoning about the semantics. Explicitly reasoning about scope extrusion is necessary to describe, *e.g.*, runtime monitors and compositions of systems.

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 (similarly to variables in the other calculi). We define two different semantics for our system: one that avoids explicit reasoning about scope extrusion, and one that does not. The objective of this challenge is to prove that the two semantics are equivalent up to structural congruence.

Syntax. The syntax of processes is given by:

$$P, Q \quad ::= \quad \mathbf{0} \quad \mid \quad (P \mid Q) \quad \mid \quad x!y.P \quad \mid \quad x?(y).P \quad \mid \quad (\boldsymbol{\nu} x)\ P$$

The process $x!y.P$ is an *output*, which can send the name y via x, then continue as P. The process $x?(y).P$ is an *input*, which can receive a name via x, then continue as P with the received name substituted for y. The input operator thus binds the name y in P. Note that the scope of a restriction may change when processes interact. Namely, a restricted name may be sent *outside* of its scope. Note that there is no recursion or replication in the syntax, and thus no infinite behaviours can be expressed. This simplifies the theory and is orthogonal to the concept of scope extrusion.

Reduction Semantics. The first semantics is an operational reduction semantics, which avoids reasoning explicitly about scope extrusion by way of a structural congruence rule. *Structural congruence* is the smallest congruence relation that satisfies the following axioms:

$$\textsc{Sc-Par-Assoc}\ \frac{}{P \mid (Q \mid R) \equiv (P \mid Q) \mid R} \qquad \textsc{Sc-Par-Comm}\ \frac{}{P \mid Q \equiv Q \mid P} \qquad \textsc{Sc-Par-Inact}\ \frac{}{P \mid \mathbf{0} \equiv P}$$

$$\textsc{Sc-Res-Par}\ \frac{x \notin \mathrm{fn}(Q)}{(\boldsymbol{\nu} x)\ P \mid Q \equiv (\boldsymbol{\nu} x)\ (P \mid Q)} \qquad \textsc{Sc-Res-Inact}\ \frac{}{(\boldsymbol{\nu} x)\ \mathbf{0} \equiv \mathbf{0}} \qquad \textsc{Sc-Res}\ \frac{}{(\boldsymbol{\nu} x)\ (\boldsymbol{\nu} y)\ P \equiv (\boldsymbol{\nu} y)\ (\boldsymbol{\nu} x)\ P}$$

The operational semantics is defined as the following relation on processes:

$$\textsc{R-Com}\ \frac{}{x!y.P \mid x?(z).Q \to P \mid Q\{y/z\}} \qquad \textsc{R-Res}\ \frac{P \to Q}{(\boldsymbol{\nu} x)\ P \to (\boldsymbol{\nu} x)\ Q} \qquad \textsc{R-Par}\ \frac{P \to Q}{P \mid R \to Q \mid R}$$

$$\textsc{R-Struct}\ \frac{P \equiv P' \qquad P' \to Q' \qquad Q \equiv Q'}{P \to Q}$$

Note that there is no rule for inferring an action of an input or output process except those that match the input/output capability. Note also that due to rule R-Com, the process $x?(z).P$ can receive *any* name. Finally, note that rule R-Struct allows for applying the structural congruence both before and after the reduction: this makes the reduction relation closed under structural congruence.

Transition System Semantics. The second semantics of the system describe the actions that the system can perform by defining a labelled transition relation on processes. The transitions are labelled by *actions*, the syntax of which is:

$$\alpha \quad ::= \quad x!y \quad \mid \quad x?y \quad \mid \quad x!(y) \quad \mid \quad \tau$$

The *free output action* $x!y$ is sending the name y via x. The *input action* $x?y$ is receiving the name y via x. The *bound output action* $x!(y)$ is sending a fresh name y via x. The *internal action* τ is performing internal communication.

We extend the notion of free and bound occurrences with $\mathrm{fn}(\alpha)$ to denote the set of names that occur free in the action α and $\mathrm{bn}(\alpha)$ to denote the set of names that occur bound in the action α. In the free output action $x!y$ and the input action $x?y$, both x and y are free names. In the bound output action $x!(y)$, x is a free name, while y is a bound name. We also use the notation $\mathrm{n}(\alpha)$ to denote the union of $\mathrm{fn}(\alpha)$ and $\mathrm{bn}(\alpha)$, i.e. the set of all names that occur in the action α.

The transition relation is then defined by the following rules:

$$\textsc{Out}\ \frac{}{x!y.P \xrightarrow{x!y} P} \qquad \textsc{In}\ \frac{}{x?(z).P \xrightarrow{x?y} P\{y/z\}}$$

$$\textsc{Par-L}\ \frac{P \xrightarrow{\alpha} P' \qquad \mathrm{bn}(\alpha) \cap \mathrm{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$$

$$\textsc{Par-R}\ \frac{Q \xrightarrow{\alpha} Q' \qquad \mathrm{bn}(\alpha) \cap \mathrm{fn}(P) = \emptyset}{P \mid Q \xrightarrow{\alpha} P \mid Q'}$$

$$\textsc{Comm-L}\ \frac{P \xrightarrow{x!y} P' \qquad Q \xrightarrow{x?y} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\textsc{Comm-R}\ \frac{P \xrightarrow{x?y} P' \qquad Q \xrightarrow{x!y} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\textsc{Close-L}\ \frac{P \xrightarrow{x!(z)} P' \qquad Q \xrightarrow{x?z} Q' \qquad z \notin \mathrm{fn}(Q)}{P \mid Q \xrightarrow{\tau} (\boldsymbol{\nu} z)\ P' \mid Q'}$$

$$\textsc{Open}\ \frac{P \xrightarrow{x!z} P' \qquad z \neq x}{(\boldsymbol{\nu} z)\ P \xrightarrow{x!(z)} P'}$$

$$\textsc{Close-R}\ \frac{P \xrightarrow{x?z} P' \qquad Q \xrightarrow{x!(z)} Q' \qquad z \notin \mathrm{fn}(P)}{P \mid Q \xrightarrow{\tau} (\boldsymbol{\nu} z)\ P' \mid Q'}$$

$$\textsc{Res}\ \frac{P \xrightarrow{\alpha} P' \qquad z \notin \mathrm{n}(\alpha)}{(\boldsymbol{\nu} z)\ P \xrightarrow{\alpha} (\boldsymbol{\nu} z)\ P'}$$

Note that there is no rule for inferring transitions from $\mathbf{0}$, and that there is no rule for inferring an action of an input or output process except those that match the input/output capability. Note also that due to rule In, the process $x?(z).P$ can receive *any* name.

We keep the convention that bound names of any processes or actions are chosen to be different from the names that occur free in any other entities under consideration, such as processes, actions, substitutions, and sets of names. The convention has one exception, namely that in the transition $P \xrightarrow{x!(z)} Q$, the name z (which occurs bound in P and the action $x!(z)$) may occur free in Q. Without this exception it would be impossible to express scope extrusion.

Challenge.

Lemma 6. *If $P \equiv Q$ and $P \xrightarrow{\alpha} P'$, then for some Q' we have $Q \xrightarrow{\alpha} Q'$ and $P' \equiv Q'$.*

Proof (Sketch). First, show the special case when P can be rewritten to Q with a single application of an axiom of the structural congruence to some subterm of P. The proof is then by induction on the number of such steps.

For the proof of the challenge theorem, we introduce the notion of a *normalized derivation* of a reduction $P \to Q$, which is of the following form. The first rule applied is R-Com. The derivation continues with an application of R-Par,

followed by zero or more applications of R-Res. The last rule is an application of R-Struct.

Lemma 7. *Every reduction has a normalized derivation.*

Proof (Sketch). To obtain a normalized derivation from an arbitrary derivation we will need to check that rules R-Com, R-Par and R-Res commute with R-Struct, and that two applications of R-Struct can be combined into one.

Lemma 8. *If $P \to Q$, then there are $x, y, z, z_1, \ldots, z_n, R_1, R_2$, and S such that*

$$P \equiv (\boldsymbol{\nu} z_1) \ldots (\boldsymbol{\nu} z_n)\ ((x!y.R_1 \mid x?(z).R_2) \mid S)$$
$$Q \equiv (\boldsymbol{\nu} z_1) \ldots (\boldsymbol{\nu} z_n)\ ((R_1 \mid R_2\{y/z\}) \mid S)$$

Proof. Follows immediately from lemma 7 and the shape of a normalized derivation.

The objective of this challenge is to prove the following theorems, which together show the equivalence between the reduction semantics and the transition system semantics up to structural congruence. The first of the theorems involves reasoning about scope extrusion more directly than the other, and if time does not permit proving both of the theorems, theorem 3 should be proven first.

Theorem 3. *$P \xrightarrow{\tau} Q$ implies $P \to Q$.*

Proof (Sketch). The proof is by induction on the inference of $P \xrightarrow{\tau} Q$ using the following lemmata:

1. if $Q \xrightarrow{x!y} Q'$ then $Q \equiv (\boldsymbol{\nu} w_1) \ldots (\boldsymbol{\nu} w_n)\ (x!y.R \mid S)$ and $Q' \equiv (\boldsymbol{\nu} w_1) \ldots (\boldsymbol{\nu} w_n)\ (R \mid S)$ where $x, y \notin \{w_1, \ldots, w_n\}$.
2. if $Q \xrightarrow{x!(z)} Q'$ then $Q \equiv (\boldsymbol{\nu} z)\ (\boldsymbol{\nu} w_1) \ldots (\boldsymbol{\nu} w_n)\ (x!z.R \mid S)$ and $Q' \equiv (\boldsymbol{\nu} w_1) \ldots (\boldsymbol{\nu} w_n)\ (R \mid S)$ where $x \notin \{z, w_1, \ldots, w_n\}$.
3. if $Q \xrightarrow{x?y} Q'$ then $Q \equiv (\boldsymbol{\nu} w_1) \ldots (\boldsymbol{\nu} w_n)\ (x?(z).R \mid S)$ and $Q' \equiv (\boldsymbol{\nu} w_1) \ldots (\boldsymbol{\nu} w_n)\ (R\{y/z\} \mid S)$ where $x \notin \{w_1, \ldots, w_n\}$.

Theorem 4. *$P \to Q$ implies the existence of a Q' such that $P \xrightarrow{\tau} Q'$ and $Q \equiv Q'$.*

Proof. If $P \to Q$, then by lemma 8, $P \equiv P'$ with

$$P' = (\boldsymbol{\nu} z_1) \ldots (\boldsymbol{\nu} z_n)\ ((x!y.R_1 \mid x?(z).R_2) \mid S)$$

and $Q \equiv Q'$ with

$$Q' = (\boldsymbol{\nu} z_1) \ldots (\boldsymbol{\nu} z_n)\ ((R_1 \mid R_2\{y/z\}) \mid S)\ .$$

We can easily check that $P' \xrightarrow{\tau} Q'$ and so by lemma 6, $P \xrightarrow{\tau} Q'$.

A.4 Challenge: Coinduction and Infinite Processes

This challenge is about the mechanisation of proofs concerning processes with infinite behaviours. This is usually connected to *coinductive* definitions where an infinite structure is defined as the greatest fixed point of a recursive definition. Coinduction is a technique for defining and proving properties of such infinite structures.

For this challenge, we adopt a fragment of the untyped π-calculus that includes process replication. The objective of this challenge is to draw a formal connection between strong barbed congruence and strong barbed bisimilarity. The result establishes that two processes are strong barbed congruent if the processes obtained by applying a finite number of substitutions to them and composing them in parallel with an arbitrary process are strongly barbed bisimilar. The key issue of this challenge is the coinductive reasoning about the infinite behaviours of the replication operator.

Syntax. The syntax of values and processes is given by:

$$\begin{array}{lcl} v, w & ::= & a \mid l \\ P, Q & ::= & \mathbf{0} \mid x!v.P \mid x?(l).P \mid (P \mid Q) \mid (\boldsymbol{\nu} x)\ P \mid\ !P \end{array}$$

The output process $x!v.P$ sends the value v on channel x and continues as P. The intention is that v must be a base value when it is actually sent, and this is enforced in the semantics later on. The input process $x?(l).P$ waits for a base value from channel x and then continues as P with the received value substituted for the variable l. Since replication allows for infinite copies of the process P, processes can dynamically create an infinite number of names during execution.

Semantics. We choose to give a labelled transition system semantics for this challenge.

The transitions are labelled by *actions*, the syntax of which is as follows:

$$\alpha \ ::= \ x!a \mid x?a \mid \tau$$

The *output action* $x!y$ is sending the base value a via x. The *input action* $x?y$ is receiving the base value y via x. The *internal action* τ is performing internal communication. We use the notation $\mathrm{n}(\alpha)$ to denote the set of names that occur in the action α.

The transition relation is defined by the following rules:

$$\textsc{Out}\ \frac{}{x!a.P \xrightarrow{x!a} P} \qquad \textsc{In}\ \frac{}{x?(l).P \xrightarrow{x?a} P\{a/l\}}$$

$$\textsc{Par-L}\ \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \textsc{Par-R}\ \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'}$$

$$\textsc{Comm-L}\ \frac{P \xrightarrow{x!a} P' \qquad Q \xrightarrow{x?a} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \qquad \textsc{Comm-R}\ \frac{P \xrightarrow{x?a} P' \qquad Q \xrightarrow{x!a} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\textsc{Res}\ \frac{P \xrightarrow{\alpha} P' \qquad x \notin \mathrm{n}(\alpha)}{(\boldsymbol{\nu} x)\ P \xrightarrow{\alpha} (\boldsymbol{\nu} x)\ P'} \qquad \textsc{Rep}\ \frac{P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P' \mid !P}$$

Note that there is no rule for inferring transitions from $\mathbf{0}$, and that there is no rule for inferring an action of an input or output process except those that match the input/output capability. Note also that due to rule In, the process $x?(l).P$ can receive *any* base value. Since the rule Out only applies to base values, there is no way to send a variable.

Strong Barbed Bisimilarity. Bisimilarity is a notion of equivalence for processes and builds on a notion of *observables*, i.e., what we can externally observe from the semantics of a process. If we allowed ourselves only to observe internal transitions (i.e., observe that a process is internally performing a step of computation) we would relate either too few processes (in the strong case where we relate only processes with exactly the same number of internal transitions) or every process (in the weak case where we relate processes with any amount of internal transitions). As a result, we must allow ourselves to observe more than just internal transitions, and we choose to describe a process's observables as the names it might use for sending and receiving.

To this end, we define the *observability predicate* $P \downarrow_\mu$ as follows:

$$\begin{aligned} P \downarrow_{x?} &\quad \text{if } P \text{ can perform an input action via } x. \\ P \downarrow_{x!} &\quad \text{if } P \text{ can perform an output action via } x. \end{aligned}$$

A symmetric relation $\mathcal{R}$ is a *strong barbed bisimulation* if $(P, Q) \in \mathcal{R}$ implies

$$P \downarrow_\mu \ \text{implies}\ Q \downarrow_\mu \tag{1}$$

$$P \xrightarrow{\tau} P' \ \text{implies}\ Q \xrightarrow{\tau} Q' \ \text{and}\ (P', Q') \in \mathcal{R} \tag{2}$$

Two processes are said to be *strong barbed bisimilar*, written $P \mathrel{\dot{\sim}} Q$, if there exists a strong barbed bisimulation $\mathcal{R}$ such that $(P, Q) \in \mathcal{R}$. Note that *strong barbed bisimilarity* $\dot{\sim}$ is the largest strong barbed bisimulation. Also, since our processes have potentially infinite behaviours, bisimilarity cannot be defined inductively since it is the largest strong barbed bisimulation.

Theorem 5. $\dot{\sim}$ *is an equivalence relation.*

Proof. We prove the three properties separately:

- Reflexivity is straightforward: for any P, we need to show that $P \dot{\sim} P$. In order to do so, we choose the identity relation and prove that it is a strong barbed bisimulation. Condition 1 follows trivially by definition. Condition 2 follows coinductively since we must always reach identical pairs.
- Symmetry follows immediately by definition.
- For transitivity, we need to prove that if $P \dot{\sim} Q$ and $Q \dot{\sim} R$ then $P \dot{\sim} R$. In order to do so, we prove that the relation $\mathcal{R} = \{(P, R) \mid \exists Q \text{ such that } P \dot{\sim} Q \wedge Q \dot{\sim} R\}$ is a strong barbed bisimulation. Let us assume that $(P, R) \in \mathcal{R}$. Hence, there exists a Q such that $P \dot{\sim} Q$ and $Q \dot{\sim} R$. Clearly, if $P \downarrow_\mu$ then, by $P \dot{\sim} Q$, $Q \downarrow_\mu$. And, by $Q \dot{\sim} R$, $R \downarrow_\mu$. Moreover, if $P \xrightarrow{\tau} P'$ there exists Q' such that $Q \xrightarrow{\tau} Q'$ and $P' \dot{\sim} Q'$. And also, $R \xrightarrow{\tau} R'$ with $Q' \dot{\sim} R'$. Finally, by definition of $\mathcal{R}$, $(P', R') \in \mathcal{R}$.

Unfortunately, strong barbed bisimilarity is not a good process equivalence since it is not a congruence, hence it does not allow for substituting a process with an equivalent one in any context. For instance, the processes $x!a.y!b.\mathbf{0}$ and $x!a.\mathbf{0}$ are strong barbed bisimilar, i.e., $x!a.y!b.\mathbf{0} \dot{\sim} x!a.\mathbf{0}$. This is because $x!$ is the only observable in both processes and they cannot perform a τ-action. However, in the context $C = [\cdot] \mid x?(l).\mathbf{0}$, the relation no longer holds: in fact, $x!a.y!b.\mathbf{0} \mid x?(l).\mathbf{0} \not\dot{\sim} x!a.\mathbf{0} \mid x?(l).\mathbf{0}$ because the left process can perform a τ-action such that $y!$ becomes observable, whereas the right process cannot.

Strong Barbed Congruence. In order to detect cases like the one above, we need to restrict strong barbed bisimilarity so that it becomes a congruence, i.e., we have to consider the environment in which processes may be placed.

We say that two processes P and Q are *strong barbed congruent*, written $P \simeq^c Q$, if $C[P] \dot{\sim} C[Q]$ for every context C.

Lemma 9. $\simeq^c$ *is the largest congruence included in* $\dot{\sim}$.

Proof. We first prove that $\simeq^c$ is indeed a congruence, i.e. it is an equivalence relation that is preserved by all contexts. Proving that $\simeq^c$ is an equivalence is easy; to prove that $\simeq^c$ is preserved by all contexts, we show that $\forall C : P \simeq^c Q$ implies $C[P] \simeq^c C[Q]$, by structural induction on the context C.

To prove that $\simeq^c$ is the largest congruence included in $\dot{\sim}$, we show that for any congruence $\mathcal{S} \subseteq \dot{\sim}$ we have $\mathcal{S} \subseteq \simeq^c$. Take any P, Q such that $P \mathcal{S} Q$ (hence, $P \dot{\sim} Q$): since S is a congruence by hypothesis, this implies $\forall C : C[P] \mathcal{S} C[Q]$ (hence, $C[P] \dot{\sim} C[Q]$). Therefore, by the definition of $\simeq^c$, we have $P \simeq^c Q$, from which we conclude $\mathcal{S} \subseteq \simeq^c$.

Challenge. The objective of this challenge is to prove a theorem that shows that making strong barbed bisimilarity sensitive to substitution and parallel composition is enough to show strong barbed congruence. To prove the theorem, we will use an *up-to technique*, utilizing the following definition and lemma. A relation $\mathcal{S}$ is called a *strong barbed bisimulation up to* $\dot{\sim}$ if, whenever $(P, Q) \in \mathcal{S}$, the following conditions hold:

1. $P \downarrow_\mu$ if and only if $Q \downarrow_\mu$.
2. if $P \xrightarrow{\tau} P'$ then $Q \xrightarrow{\tau} Q'$ for some Q' with $P' \dot{\sim}\mathcal{S}\dot{\sim} Q'$.
3. if $Q \xrightarrow{\tau} Q'$ then $P \xrightarrow{\tau} P'$ for some P' with $P' \dot{\sim}\mathcal{S}\dot{\sim} Q'$.

Lemma 10. *If $\mathcal{S}$ is a strong barbed bisimulation up to $\dot{\sim}$, $(P, Q) \in \mathcal{S}$ implies $P \dot{\sim} Q$.*

Proof. We check that $\dot{\sim}\mathcal{S}\dot{\sim}$ is a strong barbed bisimulation and is thus included in $\dot{\sim}$.

Theorem 6. *$P \simeq^c Q$ if, for any process R and substitution σ, $P\sigma \mid R \dot{\sim} Q\sigma \mid R$.*

Proof. Since $\simeq^c$ is the largest congruence included in $\dot{\sim}$, it suffices to show that if $P\sigma \mid R \dot{\sim} Q\sigma \mid R$ for any R and σ, then $C[P]\sigma \mid R \dot{\sim} C[Q]\sigma \mid R$ for any C, R and σ. We proceed by induction on C.

$C = x?(z).C'$ Let $\mathcal{S} = \{(C[P]\sigma \mid R, C[Q]\sigma \mid R) \mid R \text{ and } \sigma \text{ arbitrary}\} \cup \dot{\sim}$. We can easily check that $\mathcal{S}$ is a strong barbed bisimulation, noting that $\dot{\sim}$ is preserved by restriction and is contained in $\mathcal{S}$.

$C = C' \mid S$ Then by the induction hypothesis,

$$C[P]\sigma \mid R \dot{\sim} C'[P]\sigma \mid (S\sigma \mid R) \dot{\sim} C'[Q]\sigma \mid (S\sigma \mid R) \dot{\sim} C[Q]\sigma \mid R$$

for any R and σ.

$C = (\boldsymbol{\nu} z)\ C'$ Then by the induction hypothesis we have $C'[P]\sigma \mid R \dot{\sim} C'[Q]\sigma \mid R$ for any R and σ. Without loss of generality, we assume that $z \notin \mathrm{fn}(R) \cup \mathrm{n}(\sigma)$. Then, using that $\dot{\sim}$ is preserved by restriction, we have

$$C[P]\sigma \mid R \dot{\sim} (\boldsymbol{\nu} z)\ (C'[P]\sigma \mid R) \dot{\sim} (\boldsymbol{\nu} z)\ (C'[Q]\sigma \mid R) \dot{\sim} C[Q]\sigma \mid R$$

$C = !C'$ Let $\mathcal{S} = \{(C[P]\sigma \mid R, C[Q]\sigma \mid R) \mid R \text{ and } \sigma \text{ arbitrary}\}$. Using lemma 10, it suffices to show that $\mathcal{S}$ is a strong barbed bisimulation up to $\dot{\sim}$. To this end, let

$$A = C'[P]\sigma, \qquad A' = C[P]\sigma$$
$$B = C'[Q]\sigma, \qquad B' = C[Q]\sigma,$$

noting that $A' = !A$ and $B' = !B$.
Suppose $R \mid A' \xrightarrow{\tau} S$ for some S. Then we can show by a case analysis on the derivation of this transition that there exists a T such that $R \mid (A \mid A) \xrightarrow{\tau} T$ and $S \dot{\sim} T \mid A'$. Using the induction hypothesis twice, we note that $R \mid B' \dot{\sim} R \mid (A \mid A) \mid B'$. Since by rule Par-L, $R \mid (A \mid A) \mid B' \xrightarrow{\tau} T \mid B'$, there must thus exist a U such that $R \mid B' \xrightarrow{\tau} U$, $U \dot{\sim} T \mid B'$ and $S \dot{\sim}\mathcal{S}\dot{\sim} U$ as required. The proof for $R \mid B' \xrightarrow{\tau} S$ is analogous.

The remaining cases are similar.