Machine verification of typed process calculi

Research Proposal

Uma Zalakain

August 1, 2019

1 Introduction

During the last decades, while the frequency at which processors run has peaked, the number of available processing units has kept growing. Computing has therefore shifted its focus into making processes safely communicate with each other — no matter if they run concurrently on different CPU cores or on different hosts. As a result, the interest in the formalisation and verification of *communicating concurrent* systems has grown in this modern computing era. *Process calculi* model the communication between these concurrent processes — which share no state, and change as communication occurs.

Communicating concurrent processes must satisfy some safety properties, such as following a pre-established communication protocol (where all messages sent by one process are expected by the other and vice versa), communicating over private channels only known to the involved participants, or sometimes more advanced liveness properties, such as ensuring the absence of deadlocks or livelocks, or guaranteeing progress. To make properties like these easier to prove, formal models such as the π-calculus [WMP89, Mil89, Mil91, SW01] abstract real-world systems into suitable mathematical representations.

The properties of a formal system can be verified either *dynamically*, by monitoring processes at runtime, or *statically*, by reasoning on the definition of the processes themselves. Static guarantees — while harder to define and sometimes more conservative than dynamic ones — are *total*, and thus satisfied regardless of the execution path. The basis of static verification is comprised of *types* and *type systems*, which are also the basis of programming languages and tools, making type-based verification techniques transferable to practical applications. An example of this are the plethora of types for communication and process calculi: from standard channel types, as you can find in e.g., Erlang or Go, to *session types* [Hon93, THK94, HVK98], a formalism used to specify and verify communication protocols.

The mechanised formalisation and verification of programming languages and calculi is an ongoing community effort in securing existing work: humans are able to check proofs, but they are very likely to make mistakes; machines can verify proofs mechanically. A remarkable example of a community effort towards machine verification is RustBelt

[JJKD17], a project that aims to formalise and machine-check the ownership system of the programming language Rust. Not only does mechanisation increase confidence in what is mechanised, but also in all other derived work that is yet unverified: proving the correctness of Rust's type system immediately increases the confidence in all software written in it.

RESEARCH STATEMENT

My PhD research proposal focuses on the **machine verification** of the fundamental features and properties of typed process calculi, more specifically the **session-typed π-calculus** with its multiple extensions.

We choose Coq [CP89, Coq] to machine-verify the session-typed π-calculus, mainly due to its widespread use as a proof assistant. A first challenge with Coq is that it offers *no* support for *linearity*, which is at the very heart of session types (as communication occurs, a session type must transition through each of its stages exactly once). As a result, extra work will be required to simulate the linearity of the terms in the object language (§2.1). This will then be used to formalise and verify the properties of the linear π-calculus (§2.2). Building on top of that, session types and some of their more advanced extensions will be formalised and verified as well (§2.3). Once all these formalisms have been mechanically verified in Coq, the focus of this project will shift to the translation of the accomplished work into proof assistants with support for linear types (§2.4).

Structure of the proposal Brief introductions to the π-calculus, session types, and the proof assistant Coq are provided in §1.1. Tasks central to this research are outlined in §2 together with a brief overview of my personal suitability. Finally, a quick overview of the related work can be found in §3.

1.1 Background

π-Calculus The π-calculus [WMP89, Mil89, Mil91, SW01] is introduced as a way of modelling processes that change their structure by communicating with each other. The π-calculus features *channel mobility*, which allows communication channels to be sent over communication channels. An overview of the FAQs can be found in [Win02].

The following example creates two linked channel endpoints x and y and composes two processes in parallel: one that uses x to send integers 3 and 4, and then expect a response bound as r, do some P, then end; another that uses y to receive a and b, then send $a+b$, then end. Both processes communicate with one another when composed in parallel, changing their structure.

$$\begin{aligned}
(\boldsymbol{\nu} xy)\,(\overline{x}\langle 3\rangle.\overline{x}\langle 4\rangle.x(r).P.\mathbf{0} \mid y(a).y(b).\overline{y}\langle a+b\rangle.\mathbf{0}) &\to \\
(\boldsymbol{\nu} xy)\,(\overline{x}\langle 4\rangle.x(r).P.\mathbf{0} \mid y(b).\overline{y}\langle 3+b\rangle.\mathbf{0}) &\to \\
(\boldsymbol{\nu} xy)\,(x(r).P.\mathbf{0} \mid \overline{y}\langle 3+4\rangle.\mathbf{0}) &\to \\
(\boldsymbol{\nu} xy)\,(P[3+4/r].\mathbf{0} \mid \mathbf{0}) &\equiv \\
P[3+4/r]
\end{aligned}$$

In the π-calculus any number of processes can communicate over a channel. The type of a channel does not evolve as communication occurs: it only specifies the type of data sent over it (in the example above both x and y are of type `#Int`).

Session types. Session types [Hon93, THK94, HVK98] are sequences of actions, each representing the type and the direction of the data exchanged. Processes must use session-typed channels according to their specified protocol. Instead of being shared and static, session types are linear, private to the communicating processes, and change as communication occurs. The process above introduced as an example of the π-calculus would have the session-types of its channels evolve through communication as follows (`!` denotes sending, `?` receiving):

$$
\begin{gathered}
x : \texttt{!Int.!Int.?Int.End},\ y : \texttt{?Int.?Int.!Int.End} \rightarrow \\
x : \texttt{!Int.?Int.End},\ y : \texttt{?Int.!Int.End} \rightarrow \\
x : \texttt{?Int.End},\ y : \texttt{!Int.End} \rightarrow \\
x : \texttt{End},\ y : \texttt{End}
\end{gathered}
$$

It is worth noting that the session types of x and y must be *dual*: when one channel sends a type `T`, the other must receive `T`, and then both must continue dually. Session types guarantee *communication privacy* (at any given moment exactly two processes communicate over a channel), *session fidelity* (processes follow session types sequentially), and *communication safety* (processes only send what their counterpart is expecting to receive). At the basis of such gurantees are linearity and duality of session types. A comprehensive introduction to session types can be found in [Vas09], while answers to FAQs are compiled in [DD10].

Coq proof assistant. Coq [Coq] is a popular proof assistant and dependently typed functional language based on the calculus of inductive constructions [CP89] (which adds inductive data types to the calculus of constructions [CH85]), a type theory isomorphic to intuitionistic predicate calculus — a constructive logic with quantified statements.

Since types may contain arbitrary definitions, definitions in Coq must exhibit termination — recursion must occur on structurally smaller terms. Coq supports inductive and coinductive data types, and features proof irrelevance for proofs (in `Prop`) and a cumulative set of universes (in `Type`).

Coq allows users to build proofs using *tactics*: programs written in L_{tac} that manipulate hypotheses and transform goals. While these programs might be incorrect, or not terminate, their outcome is ultimately checked by *Gallina*, the specification language of Coq.

In Coq, simultaneously pattern matching on multiple indexed data types can be rather clunky and arduous. The *Equations* package eases this inconvenience by adding equational definitions, pattern matching on the left, and `with` constructs ([MM04]), making Coq as convenient for dependent pattern matching as Agda.

2 Research proposal

1. *Simulate linearity in an unobstructive manner.*

   Linearity lies at the very heart of session types: a channel must transition through each of its states *exactly* once. Unfortunately, proof assistants with linear type systems are rare (refer to §3): for them linearity must be simulated in the object language — or in predicates over it.

   (i) One approach to simulating linearity is traversing processes *a posteriori*, after they have been defined, to check that each channel is used exactly once. This can be done by making the type of channels parametric, and then instantiating it to $\mathbb{B}$ and *marking* each channel for inspection. This allows both channel creation and message input to be modelled as function abstraction — channels of a parametric type cannot be forged. However, to be able to traverse processes where message passing is modelled as function abstraction, one has to be able to create all types of messages. To elude this problem, message types can be parametrised over a Type $\rightarrow$ Type function and then projected to the unit type. *This approach is successfully implemented as part of my MSc project.*

   (ii) Unfortunately, the approach in point (i) makes it impossible for processes to use any logic that is external to the calculus and depends on the type of messages. An alternative approach for simulating linearity is by doing it *a priori*, at construction time, by keeping track of the linearly available channels through a context by which processes are indexed. This means that channel creation cannot be represented through function abstraction, that process composition needs to explicitly split the context, and that there must be a way of addressing a particular channel within a context — strings with the Barendregt convention [Bar84], De Bruijn indices [de 72], locally nameless De Bruijn indices, or a parametric HOAS [Chl08] — since only channels need to be used linearly, message input can still be represented as function abstraction whenever the message does not contain a channel. On the bright side, this approach allows processes to use logic that external to the calculus and depends on the types of messages.

   While the latter approach has more appealing properties, its mechanics negatively affect usability: can it be equipped with the usability of the former approach? I intend to create a Coq library that abstracts away the simulation of linearity.

2. *Explore the foundations of session types.*

   The theory of session types can be supported through embeddings into simpler calculi, or through correspondences with logics.

   (i) *Encoding session types into more primitive types.*

   Session types can be embedded into a π-calculus with linear channels: each state transition through a session type is encoded into linear channels by *continuation-passing* [DGS12, Dar16]. Describing session types in terms of a simpler calculi has several benefits: it is easier to make adaptations to the typing rules of session types without incurring into unnecessary work duplication; and it relegates the simulation of linearity far from session type representation and to the π-calculus, allowing languages with linear type systems to replace the linear π-calculus while keeping the encoding of session types unaltered. As part of my MSc project, I type-checked the session-typed π-calculus as one single joint calculus where all channels are session-typed. This implementation is however based on a continuation-passing principle, which lies at the basis of the encoding of session types [DGS12, Dar16]. So, the next step is to formalise the linear π-calculus, encode session types in it, and verify the properties of the encoding. This work would allow for the benefits of the encoding to be transferred into the mechanically verified session-typed π-calculus and will form the basis for further extensions (refer to 2.3).

(ii) *Session types and linear logic.*

Session types gave rise to a line of research on Curry-Howard correspondences [CHS80] between concurrency and linear logic.

A correspondence between session types and *intuitionistic* linear logic is constructed in [CP10]. Based on sequent calculus, output is represented by $\otimes$ on the right and $\multimap$ on the left, while input is represented by $\multimap$ on the right and $\otimes$ on the left. This means that certain combinations of input and output constructs cannot be connected through cut elimination.

In [Wad14], Wadler builds a correspondence between session types and *classical* linear logic. Without a sequent calculus with left and right rules, Wadler uses $\otimes$ for output and $\parr$ for input, and restricts any two communicating processes to share a *single* channel to communicate over, thus avoiding races and deadlocks. Through Wadler's lens, cut elimination is isomorphic to process reduction.

In [DG18] the authors present a more expressive correspondence between session types and classical linear logic, allowing processes to share more than one channel to communicate over, while still guaranteeing deadlock-freedom by construction.

As part of my PhD, I intend to formalise these correspondences and machine-check their properties.

3. *Formalise the more advanced extensions to session types.*

Considerable research has been conducted in extending session types with more advanced features, which lead to more flexible concurrent programming.

Adding subtyping to session types enables clients to only know about a subset of the services offered by a server [GH05]. In combination with subtyping, polymorphism results in bounded polymorphism, where session types can be specialised to a type implementing a specific supertype [Gay08].

Asynchronous communication has been explored through the permutation of actions [MYH09]. Deadlock freedom [Kob02, Kob06] and progress [DdY08] have also been added into the type system, leading to necessarily terminating processes. Typing rules for the higher-order π-calculus — capturing *process mobility*, where processes are sent over channels — are given in [MY07].

In [DGS12, Dar16, Kob03, Kob07], the authors show that the encoding of session types into linear π-calculus types is robust by extending it to accomodate subtyping, polymorphism and higher-order processes, hence the properties of the linear π-calculus can be used to prove the properties of session types.

During my PhD, building on 2.2 (i) I will use this formalisation into the π-calculus to verify the properties of the aforementioned extensions to session types.

4. *Translate representation into other proof assistants.*

   Coq is one of the current most popular proof assistants, and as such I consider encoding session types in Coq a priority. However, translating this encoding into other proof assistants widens the reach of this project and provides opportunities for further learning and experimentation. Possible candidate proof assistants that have linearity built into their type systems are ATS, Idris2, Granule, Ling and Celf. Encoding session types into a linear π-calculus will help to make this translation easier.

2.1 Suitability and personal background

My first contact with dependent types and machine verification was during the last year of my undergraduate degree at the University of Strathclyde, through Conor McBride's introductory class to Agda. I also took the introductory class to Haskell taught by Conor McBride, Bob Atkey and Fredrik Nordvall. I thoroughly enjoyed both classes. Under McBride's supervision, I worked on evidence-providing problem solvers in Agda [Zal18], where I provided a verified procedure for equations on monoids and an incomplete verified solver for Presburger arithmetic.

As part of the Programming Languages Mentoring Workshop, in September 2018 I attended the ICFP in St. Louis. I then started my master's degree at the University of Glasgow, where I had my first contact with the π-calculus and session types as part of Ornela Dardha's and Simon Gay's Theory of Computation course. Under Dardha's supervision, I am currently working on my master's thesis, where I type-check the session-typed π-calculus in Coq.

I attended the BehAPI 2019 summer school in Leicester in July 2019, where I got a wider view of the past and ongoing work on process calculi. The school was organised by the EU Horizon2020 RISE Action BehAPI, where Dardha is a UoG site leader. I will be attending the upcoming SPLV 2019 summer school, at University of Strathclyde in August 2019.

This mix of background, of both machine verification and formalisation of process calculi, self-study and guidance from my supervisor Dardha, allow me not to start from scratch, but to quickly dive into research and address the above questions.

3 Related work

Session types and linearity. Linearity is strongly connected to session types: a session type must transition through each of its stages *exactly* once. Session types can be encoded into a π-calculus with linear types, as shown by [KPT96, DGS12, Dar16]. As shown in [VDG], in systems where session types are shared, the tokens allowing access to the session-typed channels must still be linear.

The connection between session types and linearity can be drawn even further, at the logical level, where an isomorphism between linear logic and session types can be shown [CP10] [Wad14]. In [LM15] the operational semantics for a session-typed functional language that builds on Wadler's isomorphism are given. This work is continued in [LM], where the language is extended with polymorphism, row types, subkinding, and non-linear data types. [GV10] uses a linear type-system to encode asynchronous session types with buffers — and then verify properties of those buffers.

Proof assistants with linear type systems. The linear consumption of channels can be represented natively in a type system with linear types. Celf [SS08] is an implementation of the logical framework CLF, which extends LF with linear types. Idris2 uses quantitative type theory [McB16] to add proof irrelevance and linear types to the dependent types already available in Idris. ATS is a concurrency-ready programming language and proof assistant that combines dependent types and linear types [Xi17], and provides an ATS/LF subsystem to encode and prove the metaproperties of type systems. Granule [OLI] is a programming language based on bounded linear logic [GSS92] where the user can define the exact number of times a term is to be used by its context. Ling is a language which natively supports parallelism, sequencing and concurrency and includes process calculi as a first order construct.

Simulating linearity. In type systems with no linear types the linearity of channels has to be simulated. In these type systems, modelling channels through a parametric higher order abstract syntax [Chl08] is not possible per se: the host language is unable to check whether the channels passed along as arguments are used linearly. This means that typing judgments must happen at the object language, through the use of a context that keeps track of linear resources. This context is usually tracked at the type level, using inductive *families* [Dyb94] indexed by a context of linear resources [PW00] — though there are approaches that keep track of context through type-classes and use monadic binding to embed a linear calculus within non-linear hosts [PZ17].

Unlike channels, messages containing base types do not need to have their linearity checked. Modelling message input of base types through function abstraction is therefore doable — and interesting, as all substitution machinery, including variable references, is lifted to the host language.

Machine-verification of process calculi. The π-calculus has been an extensive subject of machine verification: [HM99] proofs subject reduction for it; [Des00] proofs subject reduction as well, but uses a higher order syntax; [AK08] provides proofs of fairness and confluence; [HMS01] formalises the bisimilarity proofs found in [WMP89]; [Gay01] provides a framework for formalisation on the π-calculus with linear channels in Isabelle/HOL.

In [XRWB16] session types are formalised in ATS, providing type preservation and global progress proofs. [BBMS16] uses Celf to represent session types in intuitionistic linear logic.

References

[AK08] Reynald Affeldt and Naoki Kobayashi. A Coq Library for Verification of Concurrent Programs. *Electronic Notes in Theoretical Computer Science*, 199:17–32, February 2008.

[Bar84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Number v. 103 in Studies in Logic and the Foundations of Mathematics. North-Holland ; Sole distributors for the U.S.A. and Canada, Elsevier Science Pub. Co, Amsterdam ; New York : New York, N.Y, rev. ed edition, 1984.

[BBMS16] Peter Brottveit Bock, Alessandro Bruni, Agata Murawska, and Carsten Schürmann. Representing Session Types. *Dale Miller's Festschrift*, 2016.

[CH85] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2):95–120, 1985.

[Chl08] Adam Chlipala. Parametric Higher-Order Abstract Syntax for Mechanized Semantics. In *ACM SIGPLAN Notices*, volume 43, pages 143–156, September 2008.

[CHS80] Haskell B. Curry, J. Roger Hindley, and J. P. Seldin, editors. *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism.* Academic Press, London ; New York, 1980.

[Coq] Coq Developer Community. The Coq Proof Assistant. https://coq.inria.fr/.

[CP89] Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88*, Lecture Notes in Computer Science, pages 50–66. Springer Berlin Heidelberg, 1989.

[CP10] Luís Caires and Frank Pfenning. Session Types as Intuitionistic Linear Propositions. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Paul Gastin, and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory*, volume 6269, pages 222–236. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[Dar16] Ornela Dardha. Session Types Revisited. In *Type Systems for Distributed Programs: Components and Sessions*. Springer, January 2016.

[DD10] Mariangiola Dezani-ciancaglini and Ugo De'Liguoro. Sessions and Session Types: An Overview. pages 1–28, August 2010.

[DdY08] Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro, and Nobuko Yoshida. On Progress for Structured Communications. In Gilles Barthe and Cédric Fournet, editors, *Trustworthy Global Computing*, pages 257–275. Springer Berlin Heidelberg, 2008.

[de 72] Nicolaas Govert de Bruijn. Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.

[Des00] Joëlle Despeyroux. *A Higher-Order Specification of the π-Calculus.* 2000.

[DG18] Ornela Dardha and Simon J. Gay. A New Linear Logic for Deadlock-Free Session-Typed Processes. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures*, volume 10803, pages 91–109. Springer International Publishing, Cham, 2018.

[DGS12] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In Danny De Schreye, Gerda Janssens, and Andy King, editors, *Principles and Practice of Declarative Programming, PPDP'12, Leuven, Belgium - September 19 - 21, 2012*, pages 139–150. ACM, 2012.

[Dyb94] Peter Dybjer. Inductive Families. *Formal Aspects of Computing*, 6:440–465, January 1994.

[Gay01] Simon J. Gay. A Framework for the Formalisation of Pi Calculus Type Systems in Isabelle/HOL. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science, pages 217–232. Springer Berlin Heidelberg, 2001.

[Gay08] Simon Gay. Bounded Polymorphism in Session Types. *Mathematical Structures in Computer Science*, 18:895–930, October 2008.

[GH05] Simon Gay and Malcolm Hole. Subtyping for Session Types in the Pi Calculus. *Acta Informatica*, 42(2):191–225, November 2005.

[GSS92] Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. Bounded linear logic: A modular approach to polynomial-time computability. *Theoretical Computer Science*, 97(1):1–66, April 1992.

[GV10] Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, January 2010.

[HM99] Loïc Henry-Gérard and Projet Meije. *Proof of the Subject Reduction Property for a π-Calculus in COQ*. 1999.

[HMS01] Furio Honsell, Marino Miculan, and Ivan Scagnetto. π-calculus in (Co)inductive-type theory. *Theoretical Computer Science*, 253(2):239–285, February 2001.

[Hon93] Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR'93*, Lecture Notes in Computer Science, pages 509–523. Springer Berlin Heidelberg, 1993.

[HVK98] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Chris Hankin, editors, *Programming Languages and Systems*, volume 1381, pages 122–138. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[JJKD17] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.*, 2(POPL):66:1–66:34, December 2017.

[Kob02] Naoki Kobayashi. A Type System for Lock-Free Processes. *Information and Computation*, 177:122–159, September 2002.

[Kob03] Naoki Kobayashi. Type Systems for Concurrent Programs. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Bernhard K. Aichernig, and Tom Maibaum, editors, *Formal Methods at the Crossroads. From Panacea to Foundational Support*, volume 2757, pages 439–453. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[Kob06] Naoki Kobayashi. A New Type System for Deadlock-Free Processes. pages 233–247, January 2006.

[Kob07] Naoki Kobayashi. Type systems for concurrent programs. Technical report, Tohoku University, 2007. Extended version of Kobayashi 2003.

[KPT96] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. *Linearity and the Pi-Calculus*. 1996.

[LM] Sam Lindley and J Garrett Morris. Lightweight Functional Session Types. page 20.

[LM15] Sam Lindley and J. Garrett Morris. A Semantics for Propositions as Sessions. In Jan Vitek, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 560–584. Springer Berlin Heidelberg, 2015.

[McB16] Conor McBride. I Got Plenty o' Nuttin'. In Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella, editors, *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pages 207–233. Springer International Publishing, Cham, 2016.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., January 1989.

[Mil91] Robin Milner. Operational and algebraic semantics of concurrent processes. In *Handbook of Theoretical Computer Science (Vol. B)*, pages 1201–1242. MIT Press, February 1991.

[MM04] Conor McBride and James McKinna. The View from the Left. *Journal of functional programming*, 14(1):69–111, 2004.

[MY07] Dimitris Mostrous and Nobuko Yoshida. Two session typing systems for higher-order mobile processes. In *In TLCA'07, LNCS*, pages 321–335. Springer-Verlag, 2007.

[MYH09] Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global Principal Typing in Partially Commutative Asynchronous Sessions. pages 316–332, August 2009.

[OLI] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades Iii. Quantitative Program Reasoning with Graded Modal Types. 3:30.

[PW00] James Power and Caroline Webster. Working with Linear Logic in Coq. July 2000.

[PZ17] Jennifer Paykin and S Zdancewic. The Linearity Monad. *ACM SIGPLAN Notices*, 52:117–132, September 2017.

[SS08] Anders Schack-Nielsen and Carsten Schürmann. Celf – A Logical Framework for Deductive and Concurrent Systems (System Description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning*, pages 320–326. Springer Berlin Heidelberg, 2008.

[SW01] Davide Sangiorgi and David Walker. *PI-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2001.

[THK94] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An Interaction-Based Language and Its Typing System. In Costas Halatsis, Dimitrios Maritsas, George Philokyprou, and Sergios Theodoridis, editors, *PARLE'94 Parallel Architectures and Languages Europe*, pages 398–413. Springer Berlin Heidelberg, 1994.

[Vas09] Vasco Vasconcelos. Fundamentals of Session Types. In *Information and Computation*, volume 217, pages 158–186. May 2009.

[VDG] A Laura Voinea, Ornela Dardha, and Simon J Gay. Resource Sharing via Capability-Based Multiparty Session Types. page 26.

[Wad14] Philip Wadler. Propositions as Sessions. *Journal of Functional Programming*, 24(2-3):384–418, 2014.

[Win02] Jeannette M. Wing. FAQ on Pi-Calculus. December 2002.

[WMP89] David Walker, Robin Milner, and Joachim Parrow. *A Calculus of Mobile Processes (Parts I and II)*, volume 100. June 1989.

[Xi17] Hongwei Xi. Applied Type System: An Approach to Practical Programming with Theorem-Proving. March 2017.

[XRWB16] Hongwei Xi, Zhiqiang Ren, Hanwen Wu, and William Blair. Session Types in a Linearly Typed Multi-Threaded Lambda-Calculus. *arXiv:1603.03727 [cs]*, March 2016.

[Zal18] Uma Zalakain. *Evidence-Providing Problem Solvers in Agda*. Dissertation, University of Strathclyde, Glasgow, 2018.