Engineering the Meta-Theory of Session Types

An experience report

DAVID CASTRO, Imperial College London
FRANCISCO FERREIRA*, Imperial College London
NOBUKO YOSHIDA, Imperial College London

Session types provide a principled programming discipline for structured interactions. They are used to statically check the safety of protocol composition and the absence of communication errors. These properties depend upon the meta-theory of the typing discipline, usually a type safety proof. These proofs, while conceptually simple, are very delicate and error prone due to the presence of linearity and name passing and have been falsified in several works in the literature. In this work, we explore mechanised proofs in theorem assistants as tools to develop trustworthy proofs, and at the same time to guide extensions that do not violate type safety. To that end, we study the meta-theory of two of the most used binary session types systems, the Honda-Vasconcelos-Kubo system and the more flexible revisited system by Yoshida and Vasconcelos. Additionally, we show the subtlety of representing the first system in α-equivalent representations. We develop these proofs in the Coq proof assistant, using a locally nameless representation for binders, and small scale reflection and overloaded lemmas to simplify the handling of linear typing environments.

Additional Key Words and Phrases: Message Passing Concurrency, Session Types, Proof Assistants, Subject Reduction.

1 INTRODUCTION

Given the prevalence of distributed computing and multi-core processors, concurrency is a key aspect of modern computing. There are huge practical and theoretical consequences of the transition from sequential models of computation to concurrent systems. Message passing calculi (like the π-calculus) have been used to model these systems since their introduction by Milner et al. [1992]. More recently, the π-calculus is getting used "directly" for guiding correct language design, implementation and verification. For example, Lange et al. [2017] used a variant of the π-calculus for Go program analysis; Scalas et al. [2017] used an encoding into the linear π-calculus to correctly implement multiparty session protocols; Scalas et al. [2019] implemented an extension of Dotty[1] modelled by a dependently typed Higher Order π-calculus; and Liao et al. [2019] proposed an affine π-calculus to build an executable implementation of the universal composable (UC) security protocols[2]. Remarkably, all the work above uses *types* in order to control concurrent and distributed behaviour. Certifying basic typed π-calculi is more and more crucial, as it has impact on the correctness of such implementations.

In this work, we concentrate on *session types* [Honda 1993], a typing discipline for structured interactions in distributed systems. They are applied to a wide range of problems, and their properties, such as deadlock-freedom, are well studied. Foundational works [Honda et al. 1998; Takeuchi et al. 1994] describe a basic approach to typing communicating systems together with the type-safety proof. Many other papers are built on this foundation and develop their own extensions. These calculi are very expressive, and rather complex, with features like: shared and linear communication channels, name passing, and fresh name generation. Given this complexity,

*corresponding author
[1]https://dotty.epfl.ch/
[2]https://en.wikipedia.org/wiki/Universal_composability

Authors' addresses: David Castro, Imperial College London, d.castro-perez@imperial.ac.uk; Francisco Ferreira, Imperial College London, f.ferreira-ruiz@imperial.ac.uk; Nobuko Yoshida, Imperial College London, n.yoshida@imperial.ac.uk.

it is not surprising that some innocent looking extensions violated the type safety properties of the calculus in several literature as pointed out by [Yoshida and Vasconcelos 2007]. As a consequence, the interest for mechanisation and formal proofs has risen significantly.

Type systems offer certain security properties by construction. These guarantees are backed by rigorous meta-theoretical proofs. These proofs have three particular characteristics: first they are rather cumbersome to write; second, they are boring to read and validate; and finally they are continuously extended to more expressive type systems. In this work, we study the mechanisation of proofs using a proof assistant with the aim of alleviating all three pain points: first to have proofs that are as easy to specify as possible. Second by the nature of proof assistants, the proofs are easy to trust without having to read them, instead they are mechanically validated. Finally, we aim to provide a description of a methodology and tools that make validating extensions to theories more straightforward.

Proof assistants facilitate writing and validating formal proofs. They define a logic formalism that the proof will be expressed in, offer tools to help building the proof, and check the correctness of the proof. They vary in the underlying logic system they implement and the automation mechanisms they offer. These systems have reached a certain level of maturity and are widely used in programming languages research. As the beginning of a project on the mechanisation of π-calculi, the choice of proof assistant reflects this objectives and our sensibilities. Furthermore, an important aspect is that we need an industrial strength, general purpose, and well established proof assistant that is appropriate for meta-theoretical proofs (but preferably not just that). While other options exist, among the well established ones that fill our objectives are: Coq [The Coq Development Team 2016], Isabelle/HOL [Nipkow and Paulson 1992] and Agda [Norell 2007]. We present this development in Coq, because of its strong support in the community, suitability for proofs in the locally nameless style, support for code extraction (that keeps our options open down the road for executing our code) and finally, small scale reflection of proofs using the `Ssreflect` [Gonthier and Mahboubi 2010] library. We use boolean reflection to mirror some of the inductive predicates, and we try to leverage in our advantage the proof style favoured by `Ssreflect`.

Modelling the properties of session types in existing proof assistants is challenging (even when compared to more habitual theories based on the λ-calculus) due to the presence of linearity and name passing calculi with complex binding structures. In this paper we want to propose an extension to common techniques for formalising the meta-theory of session types, that are both appropriate for session types and expressive enough to implement other systems too. To that effect, we use the Coq proof assistant [The Coq Development Team 2016] to study the representation and meta-theory of the two systems described in [Yoshida and Vasconcelos 2007]. Crucially, we discuss issues around the representation of common calculi using well known techniques.

We use *locally nameless* [Aydemir et al. 2008; Charguéraud 2012] binders to represent our syntax. In our opinion, locally nameless is at the same time a high-level representation (avoiding reasoning about de Bruijn indices and shifts as much as possible), and well suited to our setting with easy support for resource sensitivity (i.e: linearity) and name creation. Additionally, locally nameless is implementable in Coq (and possibly other systems) without requiring complicated extensions. And finally, it is amenable to the `Ssreflect` proof style. For example, the predicate to establish well-formed terms can be easily implemented as a boolean function that reflects the inductive definition. We use multiple disjoint sets of names to simplify the reasoning about different binding scopes (e.g: to avoid mixing expression variables and channel variables). This approach is suitable for resource sensitive calculi (support for this is implemented in the file: `theories/Env.v` of our Coq development).

This paper aims, on one hand, to be a case study on mechanising proofs in Coq and, on the other, to provide a library and tools to mechanise similar proofs, namely proofs about concurrent systems

that include the notion of names together with sub-structural contexts. As a case study, we choose to mechanise the Send-Receive system introduced in [Honda et al. 1998] as presented in [Yoshida and Vasconcelos 2007] (the *original* and *revised* presentations). For this purpose, we develop a library using the locally nameless [Charguéraud 2012] representation for binders (described in Section 3). The library supports multiple scopes of names (that we take advantage of to distinguish between names and variables) and a robust definition of contexts and environments that support easy splitting of contexts together with the necessary lemmas to easily implement the particular context splitting notion of a system.

Yoshida and Vasconcelos [2007] present two systems, the first one that we will refer to as *the original* and the second that we will refer to as *the revised* systems. Additionally, this work discusses a seemingly straightforward extension that breaks type safety. Notably, for the original system, we discuss how the way it is defined makes its representation impossible when using intrinsically α-convertible terms (e.g: locally nameless, de Bruijn indices, and others). In Section 4.1, we discuss this problem, and its relation to the unsound extended system. We also discuss how to fix the system when we implement and prove type preservation for the revisited system in Section 5. In hindsight this may seems obvious, but it is a unexpected consequence, and shows that mechanising proofs brings further understanding even to well-established and thoroughly studied systems.

The contributions of this paper are fourfold:

(1) The first fully mechanised proof of type preservation for a typing discipline with session types and linear and shared channels based on the calculus presented in [Yoshida and Vasconcelos 2007]. Available at: https://github.com/emtst/emtst-proof.
(2) The presentation of a technique suitable to mechanise resource sensitive systems. Concretely, we combine locally nameless [Aydemir et al. 2008; Charguéraud 2012] for syntax representation, `Ssreflect` [Gonthier and Mahboubi 2010] as proof style, and support for resource sensitivity inspired by [Nanevski et al. 2010] and [Gonthier et al. 2013].
(3) A reusable framework for representing typing environments with the supporting lemmas to implement resource sensitive context handling rules (i.e: variants of linear and affine logic that require context splits). For example, this framework is suitable for implementing richer systems like multiparty session types systems [Coppo et al. 2015; Honda et al. 2008, 2016].
(4) We discuss the problem of representing calculi like the original type system using syntax encodings that provide built-in α-equality. Furthermore, we discuss how the revised system's presentation addresses this issue.

The rest of the paper is structured in the following way: in the next section we introduce the original binary session types system as presented in [Honda et al. 1998]. Then in Section 3, we provide an introduction to the locally nameless (LN) representation roughly following [Aydemir et al. 2008]. Subsequently, we define the system from Section 2 using LN and we implement it in Coq in Section 4. There we discuss the difficulty of implementing the original system in LN (and other α-equivalent representations). And in Section 5 we define and implement the revisited binary session types system from [Yoshida and Vasconcelos 2007] and mechanise its proof of subject reduction. We describe the artefact in Section 6, and then finalise with the related work and conclusion sections.

2 BINARY SESSION TYPES: THE SEND-RECEIVE SYSTEM

Honda, Vasconcelos and Kubo's binary session types system [Honda et al. 1998] is a milestone in the development of type systems for concurrent process calculi. This system types structured interaction between processes and supports channel mobility, that is higher-order sessions.

Process						
P, Q, R	::=					
	|	$\texttt{request}\ a\,(k).P$	session request	|	$\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q$	conditional
	|	$\texttt{accept}\ a\,(k).P$	session accept	|	$P \mid Q$	parallel
	|	$k\,![e];\ P$	data send	|	$\texttt{inact}$	inaction
	|	$k\,?(x).P$	data receive	|	$\nu_n\,(a).P$	name hiding
	|	$k \triangleleft m; P$	selection	|	$\nu_c\,(k).P$	channel hiding
	|	$k \triangleright \{\texttt{l} : P [] \texttt{r} : Q\}$	branching	|	$!P$	replication
	|	$\texttt{throw}\ k\,[k']; P$	channel send			
	|	$\texttt{catch}\ k\,(k').P$	channel receive			
Expressions						
e	::=					
	|	$\texttt{true} \mid \texttt{false}$	boolean		$m ::= \texttt{l} \mid \texttt{r}$	labels
	|	$\ldots$				

Fig. 1. Syntax using names

The syntax appears in Figure 1, processes are ranged by P, Q, names are ranged by a, b, c,..., channels are ranged by k and k'. Notice that all the places where there are variable binderss are denoted with parenthesis followed by a dot (e.g: $k\,?(x).P$).

Sessions are produced by pairing a $\texttt{request}\ a\,(k).P$ with a $\texttt{accept}\ a\,(k).P$, data is communicated (values from expressions) by pairing $k\,![e];\ P$ and $k\,?(x).P$. We have a minimal set of expressions in this presentation, as we concentrate on the presentation of processes. Similarly, for label selection and branching, we simple fix two labels (i.e. $\texttt{l}$ and $\texttt{r}$) for convenience. Channels are communicated by pairing appropriately a $\texttt{throw}\ k\,[k']; P$ with a $\texttt{catch}\ k\,(k').P$. We have conventional `if` processes to allow the control flow to depend on expressions. Parallel composition and inactive processes are respectively: $\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q$, $P \mid Q$ and $\texttt{inact}$. For clarity reasons, we split name restriction in two, when restricting over a name, and over a channel, they are respectively: $\nu_n\,(a).P$ and $\nu_c\,(k).P$. Finally, we replace the original treatment of recursion with a simpler process replication model. This way we simplify the presentation of the calculus while remaining expressive. We use process replication (i.e: $!P$) to represent potentially infinitely many copies of P. With this, we avoid having an extra kind of variables (recursion variables are not needed). Replication works by equating the replicated process with a process in parallel with the replication of that process, as shown in the structural congruence rules in Figure 4.

2.1 Typing Discipline

The typing discipline we present follows [Honda et al. 1998] and [Yoshida and Vasconcelos 2007]. Figure 2 shows the syntax of types. First, we present *Sorts* that are the classifiers for shared channel endpoints (that type session requests and accepts), and expressions. And second, *Types* that classify the interactions of channels. There are types for sending and receiving expressions ($![S]; \alpha$ and $?[S]; \alpha$), for sending and receiving channels ($![\alpha]; \beta$ and $?[\alpha]; \beta$), for finished processes ($\texttt{end}$ and $\bot$), and for typing choice, both offered and taken ($\&\{\texttt{l} : \alpha, \texttt{r} : \beta\}$ and $\oplus\{\texttt{l} : \alpha, \texttt{r} : \beta\}$). Types have a natural notion of duality: sending and receiving, offering and taking a choice. Duality of types is defined in the usual way and the dual of type α is represented as $\overline{\alpha}$. Note that $\bot$ does not have a dual.

Expressions are typed by *sorts*, and processes are typed by *typings* that describe the type of all the channels involved in them. The type system is defined by two main judgements $\Gamma \vdash e : S$ and

$$\begin{array}{llll} \text{Sort} & S & ::= & \langle \alpha, \overline{\alpha} \rangle \mid \texttt{bool} \mid \ldots \\ \text{Type} & \alpha, \beta & ::= & ![S];\alpha \mid ?[S];\alpha \mid ![\alpha];\beta \mid ?[\alpha];\beta \mid \texttt{end} \mid \bot \mid \&\{\texttt{l} : \alpha, \texttt{r} : \beta\} \mid \oplus\{\texttt{l} : \alpha, \texttt{r} : \beta\} \\ \text{Sorting} & \Gamma & ::= & \cdot \mid \Gamma, x : S \qquad \text{Typing} \quad \Delta ::= \cdot \mid \Delta, k : \alpha \end{array}$$

Fig. 2. Syntax of types

$\Gamma \vdash P \triangleright \Delta$ that respectively type an expression and a process in a sorting context. Figure 3 defines these judgements as in [Honda et al. 1998]. Rule [Repl] requires that a process have its interface (i.e. typing) empty (or completed) to be able to be duplicated. The fact that it may be typed by a completed typing is a technicality that enables the system to support weakening by ended channels.

Sortings and typings support look-ups ($\Gamma(x) = S$) and a function to compute the domain ($\text{dom}(\Delta)$) in the conventional way. Because typings are resource sensitive, the rule for [Conc] defines the following two operations on typings, to establish compatibility ($\Delta \asymp \Delta'$) and composition ($\Delta \circ \Delta'$) among two of them. The definition is as follows:

$$\Delta \asymp \Delta' = \text{if } \Delta(k) = \overline{\Delta'(k)} \text{ for all } k \in \text{dom}(\Delta \bigcap \Delta')$$

$$(\Delta \circ \Delta')(k) = \begin{cases} \bot & \text{if } k \in \text{dom}(\Delta \bigcap \Delta') \\ \Delta(k) & \text{if } k \in \text{dom}(\Delta) \text{ and } k \notin \text{dom}(\Delta') \\ \Delta'(k) & \text{if } k \in \text{dom}(\Delta') \text{ and } k \notin \text{dom}(\Delta) \end{cases}$$

2.2 Structural Congruence and Reduction

Reduction and structural congruence rules are presented in Figure 4. They closely follow those from [Honda et al. 1998]. We call attention to the [Pass-Nm] rule stating that that in order to have a reduction sending a channel both sides communicate exactly the same channel (not only they communicate over a channel both processes know, but they also communicate a channel they both know of). Intuitively, the restriction means that the sent channel should be free enough in the receiving process, that is possible to use congruence (concretely α-equality) to rename the channels to be equal. However, we should see this rule as a red flag when we consider it from the point of view of syntax up-to α-conversion. The problem is eloquently articulated in [Pollack 1994] with the slogan: "the names of bound variables are not meant to be taken seriously". The rule [Pass-Nm] requires the sent channel name, and the received channel name (a bound variable) to be equal, this rule takes the bound name of the received channel very seriously indeed. This is a key aspect of the definition of this first system. It may seem natural to try to relax the [Pass-Nm] rule into this rule:

$$[\textsc{WrongPass}] \qquad \texttt{throw}\ k\,[k'];P \mid \texttt{catch}\ k\,(k'').Q \longrightarrow P \mid Q[k'/k'']$$

However, as shown in Section 3 of [Yoshida and Vasconcelos 2007] this relaxation breaks subject reduction. Therefore, they propose a system that has a more liberal [Pass-Nm] rule. We call it the revisited system, and discuss it in Section 5 together with its meta-theory.

3 A LOCALLY NAMELESS PRIMER

Locally nameless (LN) is a style of representation for syntax with binders that provides an α-equivalent representation of terms while still retaining names for free variables. The key concept is to use de Bruijn indices [de Bruijn 1972] for bound variables and names for free variables, these ideas were initially proposed by [Gordon 1994; McBride and McKinna 2004; McKinna and Pollack 1999], and more recently further developed in [Aydemir et al. 2008; Charguéraud 2012].

An important consideration is that locally nameless is easy to use and implement in existing proof assistants. Moreover, there are implementations of helper libraries that provide support for

$$\text{[BoolT]}\ \frac{}{\Gamma \vdash \mathtt{true} : \mathtt{bool}} \qquad \text{[BoolF]}\ \frac{}{\Gamma \vdash \mathtt{false} : \mathtt{bool}} \qquad \text{[NameI]}\ \frac{\Gamma(x) = S}{\Gamma \vdash x : S} \qquad \text{[Bot]}\ \frac{\Gamma \vdash P \triangleright \Delta, k : \mathtt{end}}{\Gamma \vdash P \triangleright \Delta, k : \bot} \qquad \text{[Inact]}\ \frac{\Delta \text{ completed}}{\Gamma \vdash \mathtt{inact} \triangleright \Delta}$$

$$\text{[Acc]}\ \frac{\Gamma(a) = \langle \alpha, \overline{\alpha} \rangle \qquad \Gamma \vdash P \triangleright \Delta, k : \alpha}{\Gamma \vdash \mathtt{accept}\ a\,(k).P \triangleright \Delta} \qquad \text{[Req]}\ \frac{\Gamma(a) = \langle \alpha, \overline{\alpha} \rangle \qquad \Gamma \vdash P \triangleright \Delta, k : \overline{\alpha}}{\Gamma \vdash \mathtt{request}\ a\,(k).P \triangleright \Delta}$$

$$\text{[Send]}\ \frac{\Gamma \vdash e : S \qquad \Gamma \vdash P \triangleright \Delta, k : \alpha}{\Gamma \vdash k\,![e];\, P \triangleright \Delta, k : ![S]; \alpha} \qquad \text{[Rcv]}\ \frac{\Gamma, x : S \vdash P \triangleright \Delta, k : \alpha}{\Gamma \vdash k\,?(x).P \triangleright \Delta, k : ?[S]; \alpha}$$

$$\text{[Br]}\ \frac{\Gamma \vdash P \triangleright \Delta, k : \alpha \qquad \Gamma \vdash Q \triangleright \Delta, k : \beta}{\Gamma \vdash k \triangleright \{\mathtt{l} : P [\!] \mathtt{r} : Q\} \triangleright \Delta, k : \&\{\mathtt{l} : \alpha, \mathtt{r} : \beta\}} \qquad \text{[SelL]}\ \frac{\Gamma \vdash P \triangleright \Delta, k : \alpha}{\Gamma \vdash k \triangleleft \mathtt{l}; P \triangleright \Delta, k : \oplus\{\mathtt{l} : \alpha, \mathtt{r} : \beta\}}$$

$$\text{[SelR]}\ \frac{\Gamma \vdash P \triangleright \Delta, k : \beta}{\Gamma \vdash k \triangleleft \mathtt{r}; P \triangleright \Delta, k : \oplus\{\mathtt{l} : \alpha, \mathtt{r} : \beta\}} \qquad \text{[Thr]}\ \frac{\Gamma \vdash P \triangleright \Delta, k : \beta}{\Gamma \vdash \mathtt{throw}\ k\,[k']; P \triangleright \Delta, k : ![\alpha]; \beta, k' : \alpha}$$

$$\text{[Cat]}\ \frac{\Gamma \vdash P \triangleright \Delta, k : \beta, k' : \alpha}{\Gamma \vdash \mathtt{catch}\ k\,(k').P \triangleright \Delta, k : ?[\alpha]; \beta} \qquad \text{[Conc]}\ \frac{\Gamma \vdash P \triangleright \Delta \qquad \Gamma \vdash Q \triangleright \Delta' \qquad \Delta \asymp \Delta'}{\Gamma \vdash P \mid Q \triangleright \Delta \circ \Delta'}$$

$$\text{[If]}\ \frac{\Gamma \vdash e : \mathtt{bool} \qquad \Gamma \vdash P \triangleright \Delta \qquad \Gamma \vdash Q \triangleright \Delta}{\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \triangleright \Delta} \qquad \text{[NRes]}\ \frac{\Gamma, a : S \vdash P \triangleright \Delta}{\Gamma \vdash \nu_n\,(a).P \triangleright \Delta} \qquad \text{[CRes]}\ \frac{\Gamma \vdash P \triangleright \Delta, k : \bot}{\Gamma \vdash \nu_c\,(k).P \triangleright \Delta}$$

$$\text{[Repl]}\ \frac{\Gamma \vdash P \triangleright \cdot \qquad \Delta \text{ completed}}{\Gamma \vdash\ !P \triangleright \Delta}$$

Fig. 3. Original system in [Honda et al. 1998] - the typing judgement

LN, such as the Metalib[3] library originally distributed with [Aydemir et al. 2008]. However, for this development we roll out our own implementation, mostly due to the fact that we use several disjoint nominal sets to draw names from, with the intention of separating the different kinds of binders present in the π-calculus and session types calculi.

3.1 Example: the linear λ-calculus

As an illustration, Figure 5 uses locally nameless to represent the syntax of the linear λ-calculus in order to start with a simple and familiar example. As expected for variables (ranged over by x), we distinguish between bound variables (ranged over by i, and j) that are encoded as de Bruijn indices, and free variables (ranged over by n) encoded as names from a set that contains countably many distinct names. Abstractions are as usual, noticing that they introduce an anonymous variable (as

[3]https://github.com/plclub/metalib

$$P \equiv Q \text{ if } P \equiv_\alpha Q \qquad P \mid \mathtt{inact} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R)$$

$$\nu_{\{n,c\}}\,(u).P \mid Q \equiv \nu_{\{n,c\}}\,(u).(P \mid Q) \text{ if } u \notin \mathrm{fn}(Q) \qquad \nu_{\{n,c\}}\,(u).\mathtt{inact} \equiv \mathtt{inact} \qquad !P \equiv P \mid !P$$

[Link]
$$\mathtt{accept}\ a\,(k).P \mid \mathtt{request}\ a\,(k).Q \longrightarrow \nu_c\,(k).(P \mid Q)$$

[Com]
$$\frac{e \downarrow c}{k\,![e];\, P \mid k\,?(x).Q \longrightarrow P \mid Q[c/x]}$$

[Left]
$$k \triangleleft \mathtt{l}; P \mid k \triangleright \{\mathtt{l} : Q [\!] \mathtt{r} : R\} \longrightarrow P \mid Q$$

[Right]
$$k \triangleleft \mathtt{r}; P \mid k \triangleright \{\mathtt{l} : Q [\!] \mathtt{r} : R\} \longrightarrow P \mid R$$

[Pass-Nm]
$$\mathtt{throw}\ k\,[k'];P \mid \mathtt{catch}\ k\,(k').Q \longrightarrow P \mid Q$$

[If1]
$$\frac{e \downarrow \mathtt{true}}{\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \longrightarrow P}$$

[If2]
$$\frac{e \downarrow \mathtt{false}}{\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \longrightarrow Q}$$

[Scop]
$$\frac{P \longrightarrow P'}{\nu_{\{n,c\}}\,(u).P \longrightarrow \nu_{\{n,c\}}\,(u).P'}$$

[Par]
$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}$$

[Str]
$$\frac{P \equiv P' \qquad P' \longrightarrow Q' \qquad Q' \equiv Q}{P \longrightarrow Q}$$

Fig. 4. Congruence and reduction using names

$$\begin{array}{llll}
\text{Terms} & t, s & ::= & x \mid \lambda.t \mid t\,s \mid c \\
\text{Types} & T, S & ::= & S \multimap T \mid C
\end{array}
\qquad
\begin{array}{llll}
\text{Variables} & x & ::= & i \mid n \quad \text{where } i \in \mathbb{N} \text{ and } n \in \mathbb{A} \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, n : T
\end{array}$$

Open for variables:
$$\begin{array}{lcl}
\{j{\to}s\}\,i & = & s \text{ with } j = i \\
\{j{\to}s\}\,x & = & x \text{ otherwise}
\end{array}$$

Open for terms:
$$\begin{array}{lcl}
\{j{\to}s\}\,\lambda.t & = & \lambda.\{(j+1){\to}s\}\,t \\
\{j{\to}s'\}\,t\,s & = & (\{j{\to}s'\}\,t)\,(\{j{\to}s'\}\,s)
\end{array}$$

Top variable open: $t^s \equiv \{0{\to}s\}\,t$

Locally closed terms:
$$\frac{}{\mathtt{lc}\ n} \qquad \frac{\forall n \notin L,\ \mathtt{lc}\ t^n}{\mathtt{lc}\ (\lambda.t)} \qquad \frac{\mathtt{lc}\ t \qquad \mathtt{lc}\ s}{\mathtt{lc}\ (t\,s)} \qquad \frac{}{\mathtt{lc}\ c}$$

Fig. 5. Example: the syntax of the linear λ-calculus

expected for the de Bruijn index representation that we use here for bound variables). Then we have applications, and a constant to inhabit a base type. For the types we simply have the linear function space, and a base type.

Opening variables. We define the operation $\{k{\to}s\}\,t$ to mediate between bound and free variables. This is intended to keep the invariant that free variables are represented by names when going under a binder. The operation $\{k{\to}s\}\,t$ replaces all the occurrences of bound variable j with the term s in term t. We define the more common operation to open the most recent variable t^s as a shortcut for $\{0{\to}s\}\,t$. There is a natural dual operation to *close* free names when they become

$$\frac{}{\cdot, n : T \vdash n : T} \quad \frac{\forall n \notin L,\ (\Gamma, n : S) \vdash t^n : T}{\Gamma \vdash \lambda.t : S \multimap T} \quad \frac{\Gamma_1 \vdash t : S \multimap T \qquad \Gamma_2 \vdash s : S \qquad \Gamma_1 \cap \Gamma_2 = \varnothing}{\Gamma_1, \Gamma_2 \vdash t\, s : T} \quad \frac{}{\cdot \vdash c : C}$$

$$\frac{t \longrightarrow t' \qquad \mathtt{lc}\ s}{t\, s \longrightarrow t'\, s} \quad \frac{\mathsf{body}\ t \qquad \mathtt{lc}\ s}{(\lambda.t)\, s \longrightarrow t^s} \quad \frac{\mathsf{body}\ t}{\text{value}\ \lambda.t} \quad \frac{}{\text{value}\ c}$$

Fig. 6. Example: the typing and operational semantics of the linear λ calculus

bound. This operation is mostly used when constructing terms. We omit its definition and refer the interested reader to [Charguéraud 2012].

The locally closed predicate. In Figure 5, we also introduce the locally closed (i.e: $\mathtt{lc}$) predicate over terms that validates the invariant that all bound variables are represented by indices and all the free variables are represented by names (i.e: there are no "free" de Bruijn indices). Of particular interest is the rule for abstractions, where the body may in general contain a bound variable. In this case, we need that the body of the abstraction is locally closed after opening the term with a name n that is *sufficiently fresh.* To express this notion of sufficient freshness we quantify over all the names that are not in some finite set L, this style of quantification is commonly called co-finite quantification and its use is proposed and justified in [Aydemir et al. 2008]. We may shorten this by defining:

$$\mathsf{body}\ t \equiv \forall n \notin L,\ \mathtt{lc}\ t^n$$

We will discuss co-finite quantification later in this section. Only locally closed terms adequately represent terms in the linear λ-calculus. Adequacy, or the fact that the system we implement corresponds to the system we have defined, is an important aspect of mechanisation efforts. Section 3.4 of [Aydemir et al. 2008] discusses how this relates to locally nameless representations, one approach is to show that your formalisation supports that properties you expect from it. They also suggest the approach, taken in [Harper and Licata 2007], of proving the bijection between the on-paper definition and the mechanised one. Give that, this approach requires a lengthy on-paper proof, it is easy to see why it is not chosen often. However, this is an important aspect of this work and we will revisit this topic in Section 4.1 when we consider the original binary session types system using the locally nameless representation.

The typing judgement. Figure 6 shows the typing rules and the operational semantics of the example calculus defined in Figure 5. Largely these rules are exactly what one would expect. However, we want to call the reader's attention to the following four aspects.

(1) Fresh Names. The typing rule for abstraction uses co-finite quantification to choose a fresh name. More common approaches are either to universally quantify over all possible free names, or state that exists one free name such that the body is well typed in the extended context. These two approaches have dual strengths and weaknesses. Having a single name existentially quantified makes it easy for introduction forms, but provides a weak induction principle. Conversely, having the premise hold for all the free variables provides a strong and useful induction principle but it is hard to use for introductions. [Aydemir et al. 2008] and [Charguéraud 2012] argue for co-finite quantification over a finite set of names. This is easy to use in introductions, and provides a strong enough induction principle. Our experience here validates their claim. We used co-finite quantification on all the inductive definitions, and it proved to be easy to deal with and sufficiently expressive for all the challenges we faced.

(2) Application. The rule for application is explicit on the fact that the split contexts need to be disjoint (a fact that can always be achieved using alpha conversion), that we will model by providing contexts that contain each name only once, and become undefined otherwise. Thus, we track these constraints by requiring the joint contexts to be defined. These definitions and companion lemmas (in the file `theories/Env.v`) form the basis of our reusable support for resource sensitive definitions (e.g. it is possible to support linear and affine type systems, and also the nuanced typing contexts of binary session types that unlike linear logic admit weakening under the condition that all added channels are of type `end`).

(3) β-reduction. We use a minimal operational semantics in a call by name style, and we keep in mind that we only want to reduce well-formed terms (i.e: locally closed terms), so we add premises to support the property that if a term reduces the reduced term is also well-formed. That's why the rule for application requires s to be locally closed, and the rule for β-reduction requires that the abstraction's body is locally closed when opened with a fresh name, and similarly for abstractions as values.

(4) Substitution with a fresh name. While the β-reduction rule is usually defined in terms of a substitution, in this setting we could open the body with a sufficiently fresh name, and immediately substitute it away. Instead, it is much simpler to just open the bound variable in the abstraction with the applied parameter, that has the same effect as the substitution, in a straightforward manner.

3.2 Coq and Locally Nameless

```
Module Type ATOM.
  Parameter atom : Set.
  Definition t := atom.

  (* atoms can be compared to booleans *)
  Parameter eq_atom : atom → atom → bool.
  Parameter eq_reflect : ∀ (a b : atom),
    ssrbool.reflect (a = b) (eq_atom a b).
  Parameter atom_eqMixin : Equality.mixin_of atom.

  Canonical atom_eqType := EqType atom
     atom_eqMixin.

  Parameter fresh : seq atom → atom.
  Parameter fresh_not_in : ∀ l, (fresh l) ∉ l.
  (* ... *)
End ATOM.
```

Fig. 7. The type of atoms

Proof formalisations give us confidence in the results and often result in new insights about the problem. This is due to the fact that successful mechanisations require very precise specifications and careful thought to define and revisit all the concepts. When starting a formalisation, the first choice is the particular proof assistant. These days there are many appropriate options, among them: some are general purpose like: Coq [The Coq Development Team 2016], Isabelle/HOL [Nipkow and Paulson 1992], Agda [Norell 2007] and others are more specialised like Abella [Gacek 2008] or Beluga [Pientka and Cave 2015]. As mentioned before, for this formalisation we chose Coq as it offers large community support, has a proven record for large developments, and implements a very powerful logic.

Locally nameless implementation. The implementation of locally nameless is in three files[4]: `theories/Atom.v`, `theories/AtomScopes.v` and `theories/Env.v`. Where the first provides the basic definition and specification of atoms to act as names, the second one, provides a way to create multiple disjoint sets of names for representing the different kinds of names that session types require (e.g. variables and channel names), and finally the last one implements finite maps used for contexts and typings, with emphasis on supporting the linearity requirements of various session typing disciplines.

[4]As mentioned before, the repository is available at https://github.com/emtst/emtst-proof

```
Section Environment.
  Context (K : ordType).
  Context (V : eqType).

  Inductive env := Undef
  | Def of {finMap K →  V}.

  (* Operations: add, def,
     dom, subst_env,... *)

  Lemma def_addb k t E:
   def(add k t E) = def E&&(k ∉ dom E).
  (* ... *)
```

```
Lemma domP x D: look_spec x D (x ∈ dom D).
(* ... *)
Lemma add_union k T D D':
  ((add k T D) ∪ D') = (add k T (D ∪ D')).

(* ... *)
Lemma subst_union c c' D1 D2 :
subst_env c c' (D1 ∪ D2)
  = (subst_env c c' D1 ∪
     subst_env c c' D2).
(* ... *)
End Environment.
```

Fig. 8. Environments to represent contexts and typings

We use module types and parametrised modules to have an abstract type of atoms together with their supported operations. Figure 7 shows the implementation of atoms, and the expected operations: how to compare them and functions to obtain a fresh atom given a finite sequence of atoms (definition: `fresh`), to have proof that the fresh atom is actually fresh (definition: `fresh_not_in`).

An important aspect of the formalisation is dealing with contexts and typings. As we saw in Section 2, processes are typed in context Γ, and classified by typing Δ, that specifies the types of all the channels a process uses. Moreover, typings are resource sensitive (i.e. linear), channels have to be used exactly once, unless they have performed all their communications. Similarly, weakening of typings is generally not admissible, except for channels that have finished their interactions. Our approach is to define an environment type that is suitable to implement intuitionistic contexts like Γ or resource sensitive typings like Δ.

Environments. Figure 8 shows the definition of environments, they are parametrised over two types, `K` for the keys, and `V` for the type of keys. Environments `env` are either undefined, or a finite map of unique keys and values. All the operations keep the invariant that any operation that would lead to a duplicated entry key makes the tree undefined. We define the expected operations over the type `env`. Important operations are: `add` to add a new element, `def` a predicate for defined environments, `dom` to obtain the predicates domain, `subst_env` for substitutions in environments, `look` to look up keys, etc. But more importantly, it provides dozens of lemmas to support common proofs that involve environments. In Figure 8, we illustrate some of these theorems with: `def_andb` that shows if adding an element to an environment is defined, that is equivalent to saying that the environment is defined and the element was not in the environment before. Also, `domP` that relates statements about looking up in an environment and its domain. Additional important lemmas are the ones that ease dealing with the union of contexts (these unions commonly arise in linear calculi), for example `add_union`, `subst_union` that respectively show that adding an element and substitution commute with unions. These are just some examples, there many more lemmas are proven.

The current implementation of environments together with their lemmas are used in the two formalisations in Sections 4.2 and 5.1 and they are suitable for other mechanisations where resource sensitivity and locally nameless are required.

Process

$$
\begin{array}{lcll}
P, Q, R & ::= & & \\
& \mid & \texttt{request}\ a\ ().P & \text{session request} \\
& \mid & \texttt{accept}\ a\ ().P & \text{session accept} \\
& \mid & k\,?().P & \text{data receive} \\
& \mid & \texttt{catch}\ k\ ().P & \text{channel receive} \\
& \mid & \nu_n\ ().P & \text{name hiding} \\
& \mid & \nu_c\ ().P & \text{channel hiding} \\
& \mid & \ldots &
\end{array}
$$

Names/Channels

$$
\begin{array}{lcll}
a, k & ::= & & \\
& \mid & n & \text{where } n \in \mathbb{A} \text{ (Free)} \\
& \mid & i & \text{where } i \in \mathbb{N} \text{ (Bound)}
\end{array}
$$

Fig. 9. Anonymous binders using locally nameless

[Acc]
$$\frac{\Gamma(a) = \langle \alpha, \overline{\alpha} \rangle \qquad (\forall k \notin L,\ \Gamma \vdash P^k \triangleright \Delta, k : \alpha)}{\Gamma \vdash \texttt{accept}\ a\ ().P \triangleright \Delta}$$

[Req]
$$\frac{\Gamma(a) = \langle \alpha, \overline{\alpha} \rangle \qquad (\forall k \notin L,\ \Gamma \vdash P^k \triangleright \Delta, k : \overline{\alpha})}{\Gamma \vdash \texttt{request}\ a\ ().P \triangleright \Delta}$$

[Rcv]
$$\frac{\forall x \notin L,\ \Gamma, x : S \vdash P^x \triangleright \Delta, k : \alpha}{\Gamma \vdash k\,?().P \triangleright \Delta, k : ?[S]; \alpha}$$

[Cat]
$$\frac{\forall k' \notin L,\ \Gamma \vdash P^k \triangleright \Delta, k : \beta, k' : \alpha}{\Gamma \vdash \texttt{catch}\ k\ ().P \triangleright \Delta, k : ?[\alpha]; \beta}$$

[NRes]
$$\frac{\forall a \notin L,\ \Gamma, a : S \vdash P^a \triangleright \Delta}{\Gamma \vdash \nu_n\ ().P \triangleright \Delta}$$

[CRes]
$$\frac{\forall k \notin L,\ \Gamma \vdash P^k \triangleright \Delta, k : \bot}{\Gamma \vdash \nu_c\ ().P \triangleright \Delta}$$

Fig. 10. Typing System, rules that change with LN

4 THE SEND-RECEIVE SYSTEM IN LOCALLY NAMELESS REPRESENTATION

As discussed in Section 3, the LN representation offers α-equivalent terms by using anonymous (i.e: de Bruijn indices) for bound variables. We choose it for the mechanisation due to its friendliness and suitability for calculi with linear resources. As a consequence, we need to adapt the presentation of our syntax and typing rules. We use co-finite quantification to ensure name freshness. In Figure 9 we show the changes to the syntax presented in the locally nameless representation. Notice that the only difference is that the terms with binders have anonymous binders and that names and channels can be either free or bound.

Typing rules. Figure 10 shows the typing rules that change with the LN representation, all of which are the rules with binders. There are two changes: the obvious one is that bound variables are now nameless, and the other one is the use of co-finite quantification. At first sight, there are two options for choosing a fresh name when opening a term. The first is using an existential to say that there must exist a name that is free in the term. And the second is to use a universal quantifier over all the names that do not appear in the term or context. For example, we could revisit [Rcv] in the following ways:

[Rcv-Ex]
$$\frac{x \notin \mathrm{fv}(P) \cup \mathrm{dom}(\Gamma) \qquad \Gamma, x : S \vdash P^x \triangleright \Delta, k : \alpha}{\Gamma \vdash k\,?().P \triangleright \Delta, k : ?[S]; \alpha}$$

[Rcv-All]
$$\frac{\forall x \notin \mathrm{fv}(P) \cup \mathrm{dom}(\Gamma),\ \Gamma, x : S \vdash P^x \triangleright \Delta, k : \alpha}{\Gamma \vdash k\,?().P \triangleright \Delta, k : ?[S]; \alpha}$$

These two rules both have their advantages and disadvantages. On one hand, the existential rule ([Rcv-Ex]) makes introduction easy, as one only needs to show that one name is free. On the other hand, the universal rule ([Rcv-All]) is more convenient as an elimination rule, given that it immediately shows that the body is well typed for any fresh name. On both cases, if a rule is good for introductions is less good for eliminations and vice versa. Aydemir et al. [2008]; Charguéraud [2012] propose the co-finite quantification approach, which compromises between these approaches. Our representation of typing rules use it (e.g: see [Rcv] in figure 3). For eliminations we get that there are infinitely many names that we can choose from, and as an introduction rule it is better than the universal rule in that we are able to exclude some names.

4.1 Reduction rules and a name handling problem

Figure 11 shows the changes for the reduction rules. Besides the lack of names for bound variables, we add premises to ensure that whenever a process reduces, both processes are locally closed. The congruence rules are largely unchanged, except for the α-conversion rule that holds implicitly for terms in LN representation since only α-equivalent terms can be represented. However, there is one more change to call our attention to, the rule for [Pass] went from:

[Pass-Nm]
$$\texttt{throw}\ k\,[k'];P \mid \texttt{catch}\ k\,(k').Q \longrightarrow P \mid Q$$

[Pass-LN]
$$\frac{\texttt{lc}\ P \qquad \texttt{body}\ Q}{\texttt{throw}\ k\,[k'];P \mid \texttt{catch}\ k\,().Q \longrightarrow P \mid Q^{k'}}$$

The original rule using names, enforces that the communicated channel to be the same as the channel that is expected in the receiving process. This is a way to say that the receiving process has to be α-equivalent to the process that uses that specific name. This policing on the name of a bound variable is in a sense pushing the meaning of *syntax up-to α-equivalence*. In the locally nameless setting, and in any setting that supports bindings with intrinsic α-equivalence (such as de Bruijn indices, higher-order abstract syntax, etc.) it is not possible to represent this constraint. In rule [Pass-LN] there is no name to *constrain* for the bound variable, and moreover it does not really make sense to talk about α-equivalence in the congruence rules (a locally nameless term represents the whole α-equivalent class of terms). Therefore, the natural representation of the rule ([Pass-LN]), simply opens the receiving variable with the communicated name. This seems a natural representation. Nevertheless, this presentation of the rule eminently becomes rule [WrongPass] in Section 2.2. And with that we have the key insight of this section: the straightforward representation of the original system using locally nameless allows the counterexample from [Yoshida and Vasconcelos 2007]. Moreover, this problem would also arise in any other representation that provides α-equivalent syntax and it does not have an easy solution. Notably, it is not possible to discuss the name of a bound variable in LN. Instead of reverting to the *names as strings* representation, we address this problem with the same solution offered in [Yoshida and Vasconcelos 2007]. We describe the solution and our implementation in Section 5. In the remainder of this section we discuss the Coq mechanisation, and provide a proof that the counterexample provided in the paper breaks type safety, once one settles on rule [Pass-LN].

As we have seen, the locally nameless representation provides an α-equivalent approach to syntax with binders that is simple to understand and suitable for *on-paper* presentations of meta-theory. This change is really motivated by being able to mechanise the definition and meta-theory of session types in Coq.

$$[\textsc{Link}]\ \frac{\mathsf{body}\ P \qquad \mathsf{body}\ Q}{\mathtt{accept}\ a\ ().P \mid \mathtt{request}\ a\ ().Q \longrightarrow \nu_c\ ().(P \mid Q)}$$

$$[\textsc{Com}]\ \frac{e \downarrow c \qquad \mathtt{lc}\ P \qquad \mathsf{body}\ Q}{k\,![e];\ P \mid k\,?().Q \longrightarrow P \mid Q^c}$$

$$[\textsc{Left}]\ \frac{\mathtt{lc}\ P \qquad \mathtt{lc}\ Q \qquad \mathtt{lc}\ R}{k \triangleleft \mathtt{l}; P \mid k \triangleright \{\mathtt{l} : Q \| \mathtt{r} : R\} \longrightarrow P \mid Q}$$

$$[\textsc{Right}]\ \frac{\mathtt{lc}\ P \qquad \mathtt{lc}\ Q \qquad \mathtt{lc}\ R}{k \triangleleft \mathtt{r}; P \mid k \triangleright \{\mathtt{l} : Q \| \mathtt{r} : R\} \longrightarrow P \mid R}$$

$$[\textsc{Pass-LN}]\ \frac{\mathtt{lc}\ P \qquad \mathsf{body}\ Q}{\mathtt{throw}\ k\ [k'];P \mid \mathtt{catch}\ k\ ().Q \longrightarrow P \mid Q^{k'}}$$

$$[\textsc{If1}]\ \frac{e \downarrow \mathtt{true} \qquad \mathtt{lc}\ P \qquad \mathtt{lc}\ Q}{\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \longrightarrow P}$$

$$[\textsc{If2}]\ \frac{e \downarrow \mathtt{false} \qquad \mathtt{lc}\ P \qquad \mathtt{lc}\ Q}{\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \longrightarrow Q}$$

$$[\textsc{Scop}]\ \frac{\forall u \notin L,\ P^u \longrightarrow P'^u}{\nu_{\{n,c\}}\ ().P \longrightarrow \nu_{\{n,c\}}\ ().P'}$$

$$[\textsc{Par}]\ \frac{P \longrightarrow P' \qquad \mathtt{lc}\ Q}{P \mid Q \longrightarrow P' \mid Q}$$

$$[\textsc{Str}]\ \frac{\mathtt{lc}\ P \qquad P \equiv P' \qquad P' \longrightarrow Q' \qquad Q' \equiv Q}{P \longrightarrow Q}$$

Fig. 11. Reduction using locally nameless

4.2 Mechanising the meta-theory in an α-equivalent representation

Figure 12 shows the representation of the syntax as a locally nameless definition in Coq (for full details please check the aforementioned repository). We call attention that due to bound variables being nameless variables they simply do not appear in the syntax.

Syntax definition. The binding structure is defined both by the opening operations as described in Section 3 and by the *locally closed* predicate, this predicate states that all free variables are represented by names, therefore no de Bruijn index escapes its binder. For this calculus we choose to represent all free variables with the same set of names, and we define several opening functions relevant to expression variables in expressions, and to opening names and expressions in processes. This is because, given the problem stated in Section 4.1, for this system we only want to show that the counterexample becomes admissible due to the changes needed as a consequence of having anonymous bound variables.

Recall that in the definition of LN syntax in Figure 5, we define the operation to open variables, and the predicate for well-formed terms (i.e: `lc` t). We proceed in the same manner for the binary session calculus, except that now we have expressions and processes. In the latter, we have binders for channels (e.g: `catch` k $().P$) and binders for variable expressions (e.g: $k\,?().P$). To cope with this additional complexity we define three opening operations, the first to open variables in expressions (i.e: function `oe` defined in file `theories/Syntax0.v`), and two more opening operations, one to open an expression variable in process (defined as: `{ope k ~> u } t`) and the final one to open names in a process (defined as: `{op k ~> u } t`). Finally, there are two predicates to show that expressions and processes are locally closed (defined as: `lc_exp` and `lc` respectively). We omit the definition of locally closed expressions as it is trivial. On the other hand, we show a few significant cases of the predicate for processes in Figure 13.

```
Inductive proc : Set :=
| request : name → proc → proc                (* request a ().P *)
| accept : name → proc → proc                 (* accept a ().P *)

| send : channel → exp → proc → proc          (* k ![e]; P *)
| receive : channel → proc → proc             (* k ?().P *)

| select : channel → label → proc → proc      (* k ◁ m; P *)
| branch : channel → proc → proc → proc       (* k ▷ {l : P[]r : Q} *)

| throw : channel → channel → proc → proc     (* throw k [k′]; P *)
| catch : channel → proc → proc               (* catch k ().P *)

| ife : exp → proc → proc → proc              (* if e then P else Q *)
| par : proc → proc → proc                    (* P | Q *)
| inact : proc                                (* inact *)

| nu_nm : proc → proc                         (* ν_n ().P *)
| nu_ch : proc → proc                         (* ν_c ().P *)
| bang : proc → proc                          (* !P *)
.
```

Fig. 12. Syntax representation

```
Inductive lc : proc → Prop :=
|lc_send : ∀ k e P,
  lc_nm k → lc_exp e →
  lc P →
  lc (send k e P)

|lc_receive : ∀ (L : seq atom) k P,
  lc_nm k →
  (∀ x, x ∉ L → lc (open P x)) →
  lc (receive k P)

|lc_inact : lc inact
(* ... *).

Inductive oft_exp (G : sort_env) :
  exp → sort → Prop := (*...*)

Inductive oft:
  sort_env → proc → tp_env → Prop :=
```

```
  binds a (end_points T t) G →
  (∀ k, k ∉ L →
    oft G (open P k) (add k t D)) →
  oft G (request a P) D

|t_send G k e P D S T:
  oft_exp G e S → oft G P (add k T D) →
  oft G (send k e P) (add k (output S T) D)

|t_receive (L : seq atom) G k P D S T:
  (∀ x, x ∉ L →
    oft (add x S G) P (add k T D)) →
  oft G (receive k P) (add k (input S T) D)

|t_par ∀ G P Q D1 D2:
  oft G P D1 → oft G Q D2 →
  compatible D1 D2 →
  oft G (par P Q) (D1 o D2)
(*...*).
```

Fig. 13. Locally closed and the typing judgement

On one hand, for simple processes without binders like `lc_inact` (i.e: inact), or `lc_send` (i.e: $k\,![e];\,P$), the predicate simply states that all the channels, expressions and process continuations are all recursively locally closed. On the other hand, processes with binders, need to *open* the terms in order to assign a name to the bound variable and show that the result is now locally closed. For example, this is what rules like `lc_receive` (i.e: $k\,?().P$) do. This is another example of co-finite

quantification, akin to the definition of the typing rules in Figure 10. Co-finite quantification ensures the name we use for opening is fresh.

Typing system. Following habitual practice, the typing relations are encoded as inductive predicates, some cases are shown in Figure 13. For expressions they relate a context (`sort_env` in the code), an expression (`exp` in the code) and their sort, and similarly processes replacing sorts by typings (`tp_env`). The encoding of the typing rules is generally straightforward. The notions of compatible typings and their composition are implemented in the `compatible` inductive predicate and the `compose` function respectively (note that for convenience, composition has an inline operator notation `D1 o D2` instead of the longer: `compose D1 D2`).

Operational semantics and the counterexample. Recall the discussion in Section 4.1 of the difficulty of encoding the reduction semantics in a representation with intrinsic α-equality. This is also present in the Coq representation. We represent the reduction and congruence relations with `P⟶Q` and `P≡Q` for (`P --> Q` and `P === Q` without notational embellishments). At this point we need to prove type safety, however, first we need to address the concerns about the counterexample. As it turns out, this definition breaks subject reduction. It is important to notice that the original system as defined in [Yoshida and Vasconcelos 2007] is type safe, but its reduction semantics are not directly representable in a locally nameless syntax (or any representation where α-equality is built in).

For the straightforward representation of the original system using locally nameless, we define the counterexample. And subsequently prove that it breaks subject reduction. The counterexample proof is in the file: `theories/Types0.v` in code section named `CounterExample`.

We define the process `counter` as:

$$\texttt{throw}\ k\,[k'];\texttt{inact} \mid \texttt{catch}\ k\,(0\,?().k'\,![\texttt{true}];\ \texttt{inact}).\texttt{inact}$$

and the process `reduced` as:

$$k'\,?().k'\,![\texttt{true}];\ \texttt{inact}$$

These processes are expected to be typed under the empty contexts and typings where `k` and `k'` are bottom. We represent this typing (`D`) as the concatenation of two typings that type the processes composed in `counter` (`D` that is defined as `D1 o D2`). With these definitions we prove three simple goals: First, that the counterexample admits the typing. Second, that it reduces. And finally, that there is no type derivation that allows the reduced process to admit the typing. Therefore, subject reduction does not hold for this definition of the calculus.

In Figure 14, we show the statements for the three theorems. These statements show the counter-example implemented in our syntax, and two goals[5]. The first is that `counter` is well typed, and the fact that it computes to `reduced`. Notice that before proving lemma `oft_reduced` we show a partial proof and where we get stuck trying to prove that the reduced process admits the same typing as the original. Finally, in lemma `oft_reduced` we prove that there are no derivations for `reduced` with typing `D`. In conclusion, our representation of the original system using a syntax representation that provides α-equality is not adequate. This result may seem discouraging, but we show it for two reasons; first, it is an important cautionary tale; and second, most of tools and techniques shown here will be applicable to represent and study the meta-theory of the revisited system. We discuss that in the next section.

[5] `Goal` is used to introduce an anonymous theorem

```
Goal oft nil counter D. (* the process is well typed *)
(* proof omitted.*)

Goal (counter ⟶ reduced).(* the process reduces *)
(* proof omitted.*)

(* we try and fail to type check the reduced process *)
Goal oft nil reduced D.
  rewrite/reduced/D/D1/D2.
  (* apply: t_receive.
     it fails, k' cannot be bot and input *)
Abort.

(* we prove that the reduced counter example process does not admit D. *)
Lemma oft_reduced : ~ oft nil reduced D.
(* proof omitted.*)
```

Fig. 14. Proof the Counterexample Breaks Subject Reduction

Polarities

$$p ::= + \mid -$$

Endpoints

$$\begin{array}{lll} k ::= & & \\ \mid & x & \text{endpoint variables} \\ \mid & \kappa^{p} & \text{channel names} \end{array}$$

Endpoint var.

$$\begin{array}{lll} x ::= & & \\ \mid & n & \text{where } n \in \mathbb{A}_{\mathsf{LC}} \text{ (free)} \\ \mid & i & \text{where } i \in \mathbb{N} \text{ (bound)} \end{array}$$

Endpoint name

$$\begin{array}{lll} \kappa ::= & & \\ \mid & \underline{n} & \text{where } n \in \mathbb{A}_{\mathsf{CN}} \text{ (free)} \\ \mid & \underline{i} & \text{where } i \in \mathbb{N} \text{ (bound)} \end{array}$$

Fig. 15. Syntactic Changes - Endpoints

5 THE REVISITED SEND RECEIVE SYSTEM

As discussed in Section 2, in the original binary session types system, as described in [Honda et al. 1998], one has to be very careful when extending or relaxing the channel passing rule to avoid breaking type preservation, as described in [Yoshida and Vasconcelos 2007]. In Section 4.1, we discussed that when representing the original system with syntax that intrinsically equates α-equivalent terms, the counter example to subject reduction becomes typable. This is a rather inconvenient and (somewhat) unexpected side-effect of the very particular way of ensuring safety in the communications chosen in the original system's description. Fortunately, the revisited system in [Yoshida and Vasconcelos 2007], inspired by [Gay and Hole 2005], proposes another solution that we will describe in this section, and for which we present the Coq full-development of the type-preservation proof.

The key insight in the design of the revisited system is considering *channel endpoints* instead of just *channels*. As before, a new channel is created when a requested session is accepted, and each continuation gets one of the *endpoints* of the newly created channel. Eventually, process synchronisation over a channel must occur over dual endpoints. It is not necessary for the endpoints to have dual types. The changes to the syntax are minor, specially in this setting where we use a locally nameless representation and binders do not appear in the syntax.

$$\frac{\Gamma(a) = \langle \alpha, \overline{\alpha} \rangle \qquad (\forall k \notin L_{\mathbb{LC}},\ \Gamma \vdash P^k \triangleright \Delta, k : \alpha)}{\Gamma \vdash \texttt{accept}\ a\ ().P \triangleright \Delta}\ [\text{Acc}]$$

$$\frac{\Gamma(a) = \langle \alpha, \overline{\alpha} \rangle \qquad (\forall k \notin L_{\mathbb{LC}},\ \Gamma \vdash P^k \triangleright \Delta, k : \overline{\alpha})}{\Gamma \vdash \texttt{request}\ a\ ().P \triangleright \Delta}\ [\text{Req}]$$

$$\frac{\forall x \notin L_{\mathbb{EV}},\ \Gamma, x : S \vdash P^x \triangleright \Delta, k : \alpha}{\Gamma \vdash k\,?().P \triangleright \Delta, k : ?[S]; \alpha}\ [\text{Rcv}]$$

$$\frac{\forall k' \notin L_{\mathbb{LC}},\ \Gamma \vdash P^k \triangleright \Delta, k : \beta, k' : \alpha}{\Gamma \vdash \texttt{catch}\ k\ ().P \triangleright \Delta, k : ?[\alpha]; \beta}\ [\text{Cat}]$$

$$\frac{\Gamma \vdash P \triangleright \Delta \qquad \Gamma \vdash Q \triangleright \Delta'}{\Gamma \vdash P \mid Q \triangleright \Delta, \Delta'}\ [\text{Conc}]$$

$$\frac{\forall \kappa \notin L_{\mathbb{CN}},\ \Gamma \vdash P^\kappa \triangleright \Delta, \kappa^+ : \alpha, \kappa^- : \overline{\alpha}}{\Gamma \vdash \nu_c\ ().P \triangleright \Delta}\ [\text{CRes}]$$

$$\frac{\forall \kappa \notin L_{\mathbb{CN}},\ \Gamma \vdash P^\kappa \triangleright \Delta \qquad \kappa \notin \Delta}{\Gamma \vdash \nu_c\ ().P \triangleright \Delta}\ [\text{CRes'}]$$

$$\frac{\forall a \notin L_{\mathbb{SC}},\ \Gamma, a : S \vdash P^a \triangleright \Delta}{\Gamma \vdash \nu_n\ ().P \triangleright \Delta}\ [\text{NRes}]$$

Fig. 16. Type System, rules that change in the revisited system

In Section 2, we used one set of names for channels and expressions. For the revisited system's formalisation we choose to distinguish our binders in four categories:

- expression variables, with names from the set $\mathbb{A}_{\mathsf{EV}}$
- shared channel variables, from $\mathbb{A}_{\mathsf{SC}}$
- linear channel variables, from $\mathbb{A}_{\mathsf{LC}}$
- channel names from $\mathbb{A}_{\mathsf{CN}}$ (notice that these names can also be bound in restrictions). Channel names are not variables, but objects that exist at run-time.

The motivation for having multiple disjoint sets of atoms is to simplify reasoning about free names (concretely to not have to avoid freshness problems among different kinds of binders). These choices are different to what we used in the formalisation of the original system. It is motivated by the different theorems that proven for each system (for the original system we proved that the counter example became admissible, for this system we prove the more challenging subject reduction theorem and its many required lemmas). This represents an engineering compromise, having more binders duplicates some easy theorems but they simplify some harder ones. Other compromises are possible. The usual one is having less complex calculi that completely avoid this problem, in our case we focus on the style of calculi the concurrency community works on.

Typing rules. Figure 15 shows the representation of channels, its endpoints and their names and variables. The last two use the familiar notion of names for free variables, and de Bruijn indices for bound variables. Figure 16 presents the changes to the typing rules, to accommodate for the new binding structure. And crucially, it clarifies the binding structure, by showing which set the co-finite quantification draws its names from (where $L_{\mathbb{EV}} \subset \mathbb{A}_{\mathsf{EV}}$, and respectively for the other three sets of names). Rules [Conc] and [Cres] are also different. In this system parallel composition is replaced by a new rule, where we just concatenate disjoint typings. Finally, name restriction, introduces a new channel name with its two endpoints. The rule [Cres'] allows the typing of process that do not use the channel in the restriction, this is necessary to preserve typing under the $\nu_{\{n,c\}}\ (u).\texttt{inact} \equiv \texttt{inact}$ congruence rule. This is one of the situations in which we see that

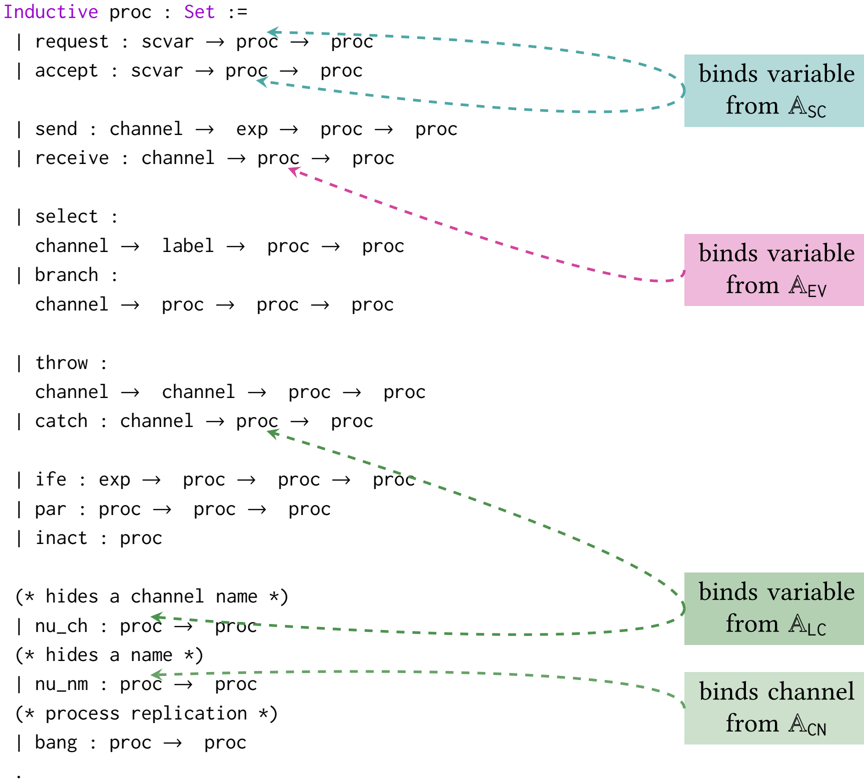

Fig. 17. Syntax representation annotated with binders

this calculus is resource sensitive something commonly associated with linearity. However, the calculus is not exactly linear (e.g: linear channel names can be bound and never used).

5.1 The Coq Mechanisation

Structure of the implementation. The mechanisation of the revisited system is done in three files: First, the file: `theories/SyntaxR.v` contains the definitions of the syntactic categories, like processes, expressions and channels, together with operations on them, like opening and substitutions, the locally closed predicate and operational semantics. Second, the file: `theories/TypesR.v` contains the definitions of contexts and environments, together with the typing judgements. It also contains some lemmas regarding the different kinds of opening and substitutions commuting, the strengthening and weakening lemmas, and notably the substitution lemmas for expressions and channels into processes. Finally, the file: `theories/SafetyR.v` defines the concept of balanced typings, and shows the two main results, that congruence preserves types, and finally the subject reduction theorem for processes.

Syntax and operational semantics. The syntactic changes of the revised system are minor as we can see in Figure 17. As explained, the binders are separated in four categories. The visible changes in the type definition, are twofold: in `request` and `accept` the names of shared channels are variables that use atoms from $\mathbb{A}_{SC}$ (type `scvar`). And second, channels now are not only names, but they follow the syntax from Figure 15. In locally nameless, besides the syntax, the binding structure

and the well-formedness of terms is established in the definitions of the opening operations and the locally closed predicate. As discussed before, in the original system's mechanisation, we used only one scope of atoms, but still had several opening operations, in this case, we use a single open per kind of atoms that appear in the syntactic category. For example, there is one opening operation for expression variables in expressions (`ope`), and there are four for processes: `open_c` to open linear channel variables with channels, `open_n` to open shared channels with shared channel variables, `open_e` to open variable expressions with expressions, and finally `open_k` to open channel name variables. For locally closed, we define only one predicate per syntactic category (`lc_exp` for expressions, `lc` for processes), as it does not make sense to be locally closed with regard to some variable kinds and not others.

Given our choice of `Ssreflect`, we have some overhead due to implementing processes, channels and expressions as equality types, and because we implement the locally closed predicate as a boolean function and show that it reflects the inductive definition. This enables us to use computation with these functions throughout our proofs. To implement locally closed as a function, we use a dual concept. We simply check that there are no de Bruijn indices that point to non-existent binders (i.e. we check for the absence of out of scope indices).

The rest of the file contains: theorems about the interactions of substitutions and opening, and the definitions of the congruence and reduction operational semantics for the calculus.

Type system. The type system is defined in the `theories/TypesR.v` file. The first concern is to use the environment type `env` from file: `theories/Env.v` to define the sorting context and the typings. A sorting (represented by Γ) contain names from expression variables, and from shared channel variables. This means that the key is atoms from $\mathbb{A}_{\mathsf{SC}} + \mathbb{A}_{\mathsf{EV}}$. Similarly, for typings, the keys are either linear channel variables, or actual endpoints that contain a channel name and a polarity, the type of keys should be $\mathbb{A}_{\mathsf{LC}} + \mathbb{A}_{\mathsf{CN}}^{p}$ where p is the polarity of endpoints.

The file: `theories/TypesR.v` contains the definitions of the typing judgements (`oft` for processes and `oft_exp` for expressions in the code). But before we tackle them, we need to think about co-finite quantification when we bind endpoints. Concretely, rules [Acc], [Req] and [Rcv] from Figure 16 show in the premise, that the process needs to admit typing when opened with a sufficiently fresh channel variable name. In our implementation, we generalise this from sufficiently fresh channel variable name to sufficiently fresh channel. That is, we open with a channel that can be a sufficiently fresh channel variable name (an element of $\mathbb{A}_{\mathsf{LC}}$ not in $L_{\mathbb{LC}}$), or a sufficiently fresh channel name (an element of $\mathbb{A}_{\mathsf{CN}}$ not in $L_{\mathbb{CN}}$). This is implemented as the relation `free_chan` and it is used in the aforementioned rules as shown Figure 18. `free_chan` works together with the function `chan_of_entry` that produces a channel from a typing entry (it either produces a channel variable, or a channel end point depending on its parameter) . The motivation for this generalisation is that, the induction hypothesis for these rules is stronger this way; we learn that the premise is well-formed when opened with both a channel variable or a channel name. As a consequence, the subject reduction proof is simpler.

Substitution lemma. After defining the typing judgements for processes and expressions, the file contains the proofs of the substitution lemmas (for the substitution principles defined in file: `theories/SyntaxR.v`). Concretely, the important lemmas are `SubstitutionLemmaExp` for showing the substitution of expressions in expressions, `ExpressionReplacement` for the substitution of expressions in processes. And finally `ChannelNameReplacement` for the substitution of channels in processes.

Subject reduction. Finally, the file: `theories/Safety.v` contains the proofs for the main result of this formalisation. Following [Yoshida and Vasconcelos 2007], subject reduction only holds for *balanced* typings. A balanced typing is one for which all the dual endpoints it contains are

```
Definition free_chan (c : tp_env_entry) (L : seq LC.atom) (L' : seq CN.atom) :=
  match c with
  | inr (k, _) => k ∉ L'
  | inl c => c ∉ L
  end.

Inductive oft : sort_env → proc → tp_env → Prop :=
| t_request (L : seq LC.atom) (L' : seq CN.atom) G a P D t:
    binds (inl a) (end_points t (dual t)) G →
    (∀ c, free_chan c L L' →
      oft G (open_c0 P (chan_of_entry c)) (add c (dual t) D)) →
    oft G (request (SC.Free a) P) D
| t_accept : ∀ (L : seq LC.atom) (L' : seq CN.atom) G a P D t,
    binds (inl a) (end_points t (dual t)) G →
    (∀ x, free_chan x L L' →
      oft G (open_c0 P (chan_of_entry x)) (add x t D)) →
    oft G (accept (SC.Free a) P) D

| t_catch : ∀ (L : seq LC.atom) (L' : seq CN.atom) G k P D T T',
    (∀ x, free_chan x L L' →
      oft G (open_c0 P (chan_of_entry x)) (add x T (add k T' D))) →
    oft G (catch (chan_of_entry k) P) (add k (ch_input T T') D)
(* ... *).
```

Fig. 18. Typing Judgement and Free Channel Relation

Theorem (Subject Congruence). *If* $\Gamma \vdash P \triangleright \Delta$ *with* balanced Δ *and* $P \equiv Q$ *then* $\Gamma \vdash Q \triangleright \Delta'$ *and* balanced Δ'.

```
Theorem CongruencePreservesOft G P Q D :
  P ≡ Q →  oft G P D →  oft G Q D.
  (* ... *)
```

Theorem (Subject Reduction). *If* $\Gamma \vdash P \triangleright \Delta$ *with* balanced Δ *and* $P \longrightarrow^* Q$ *then* $\Gamma \vdash Q \triangleright \Delta'$ *and* balanced Δ'.

```
Theorem SubjectReduction G P Q D:
  oft G P D →  balanced D →  P ⟶* Q →  exists D', balanced D' /\ oft G Q D'.
(* ... *)
```

Fig. 19. Congruence and Subject Reduction

associated with dual types. It is related to the concept of compatible from Section 2.1, but instead of relating to compatible typings, it ensures that the endpoints of a channel are typed by dual types. This condition is not enforced anywhere else and in our code is defined by the `balanced` predicate. Together with some lemmas required by the proofs that follow.

At this point, there remain the two important theorems: first: `CongruencePreservesOft` that states that congruent processes have the same type, and finally the subject reduction theorem: `SubjectReduction` (that is proven using a lemma about single small step subject congruence). Below we reproduce the statements of the theorems, and we remark on how similar the final statement is to Theorem 3.3 from [Yoshida and Vasconcelos 2007]. Figure 19 contains their statements in the formalisation together with their *on-paper* statement counterparts.

This concludes the technical development, and represents a full proof of subject reduction for binary types, following the revised system as defined in [Yoshida and Vasconcelos 2007]. The minor difference is that we use a simpler version of recursion compared to the original paper. This results in an expressive calculus with all the key constructs: shared channels, linear channels, choice, conditional processes, etc. We remark that implementing recursion as in the original paper is possible and it would not require additional techniques. Implementing recursion variables would make this proof more complex because of the need to add one more kind of variable, recursion variables.

6 IMPLEMENTATION

The locally nameless support, and the proof we develop in this work are available[6] at:

https://github.com/emtst/emtst-proof

The source code for the development is in the `./theories` directory. Locally nameless support is defined in the following files: `Atom.v`, `AtomScopes.v` and `Env.v`. Then the development of the counter example for the locally nameless version of the original system is in files: `SyntaxO.v`, and `TypesO.v`. Additionally, the type preservation proof for the revisited system is in `SyntaxR.v`, `TypesR.v` and `SafetyR.v`. When compiled using the make file, the compilation ends by Coq checking that the safety proof does not depend on any axioms or admitted theorems and it prints the reassuring message: `Closed under the global context`.

7 RELATED WORK

Following the general trend in the programming language community, the concurrency community is also keenly interested in mechanising their results. A recent example of this is the BeHAPI Workshop at ETAPS 2019 [BeH 2019], that started with a full morning session on different aspects of mechanical proofs and behavioural types. From proving important theorems about linear logic, to novel representation techniques, theoretical discussions of higher order abstract syntax. And it even included talks on a couple of work in progress mechanisations of session types.

While this work represents the first formalisation of binary session types as presented in [Honda et al. 1998; Yoshida and Vasconcelos 2007], there have been many other works in related areas. For example, Bengtson and Parrow [2009] mechanise the meta-theory of the untyped π-calculus in Isabelle/HOL. The Abella [Gacek 2008] proof assistant uses its λ-tree representation[7] [Miller and Palamidessi 1999] to implement some meta-theory of the untyped π-calculus [Abe 2019]. Goto et al. [2016] present a session types system with polymorphism and use Coq to prove type soundness of their system. There are several technical differences between their calculus and ours. First, theirs is affine, so that it does not provide deadlock freedom (between two parties) unlike ours; and secondly, they do not support neither shared channels nor expressions which are crucial to model session initiations and sequential computation. They are essential elements for implementing session-based programming languages, e.g. [Ancona et al. 2016; Gay and Ravera 2017; Hu et al. 2008]. Crucially, our objective differs from the aim in [Goto et al. 2016], which proves the correctness for the minimum calculus with a specialised extension. Our target is the "core" calculus which has all the essential features to model session-based programming and tools, and it is the most well-recognised and extended in the past. We formalise a standard binary session calculus with shared sessions, expressions and the habitual binding structure present in the literature. We strive for the extensibility and adaptability of our results to mechanise the many related session calculi already developed or to be developed in future.

[6] This repository is anonymous.

[7] also referred to as higher order abstract syntax

Also related are works that use and implement session types as part of their development (while not necessarily being the focus), two examples are: First, Tassarotti et al. [2017] where they show the correctness (in the Coq proof assistant) of a compiler that uses an intermediate language based on a simplified version the GV system [Gay and Vasconcelos 2010] to add session types to a functional programming language. And second, [Orchard and Yoshida 2015] that discusses the relation between session types and effect systems, and they implement their code in the Agda proof assistant. However, their formalisation concentrates on showing that their translation between effect systems and session types are type preserving.

Regarding our choice of technology to use, we follow the presentation of locally nameless from: [Aydemir et al. 2008; Charguéraud 2012]. We implement the proofs in the Coq proof assistant using the `Ssreflect` [Gonthier and Mahboubi 2010] library. And the design of environments and some of the automation is inspired from [Gonthier et al. 2013; Nanevski et al. 2010]. However, these choices are far from unique, while this is not an exhaustive bibliography, we provide some related approaches that could also be considered. Other options include: Isabelle/Nominal [Nipkow and Paulson 1992; Urban 2008] is an industrial strength proof assistant, and as we mentioned, it has been successfully used to implement the meta-theory of the untyped π-calculus. Our choice of Coq over Isabelle is out of familiarity and the authors' preference for constructive proofs as we are interested in future work where we take advantage of the Coq extraction mechanism.

Higher order abstract syntax (HOAS) [Church 1940; Pfenning and Elliott 1988] is in many ways the gold standard for representing languages and logics with binding. The Abella [Gacek 2008] and Beluga [Pientka and Cave 2015] systems implement powerful support for HOAS. Nevertheless, a lack of support for linearity and code extraction made us prefer using the more common Coq proof assistant. However, support for linearity and HOAS is being explored in projects like the LINCX [Georges et al. 2017] linear logical framework.

Additionally, other representations could be used, for example a weaker form of HOAS that can be directly used in Coq is championed by Parametric HOAS [Chlipala 2008]. And other forms of embedding specification logics like Hybrid and its enrichment with linearity are discussed in Felty and Momigliano [2012]; Felty [2019].

Finally, Benton et al. [2012] propose a powerful representation of intrinsically typed de Bruijn indices that can be readily implemented in many proof assistants (including Coq and Agda). And there is ongoing work [Kokke 2019] to extend this with support for linearity by following ideas from [Atkey 2018]. This is a promising direction that could bring the proof style from [Wadler and Kokke 2019] to the linear setting.

Important aspects that lead to our choices in this work were: current availability, good support for the technique in one of the leading general purpose proof assistants (we considered Coq, Agda and Isabelle). The results of the proof leave us with a valuable artefact, and a reusable implementation of locally nameless that can be used for other projects.

8 CONCLUSIONS AND FUTURE WORK

We set out to study the mechanisation of the meta-theory of binary session types as presented in [Honda et al. 1998; Yoshida and Vasconcelos 2007]. On the way, we learned a valuable lesson on adequacy, as we managed the difficulties representing the original system using the locally nameless representation. On the adequacy issue, we fully agree with Aydemir et al. [2008]: "First, note that it [adequacy] is an informal question, because it involves the relationship between an informal thing and a formal thing. No matter how much faith you put in Coq, no Coq proof will completely settle this question." Moreover, we propose that the canonical definition of a system should be the one that has the most properties proved in the most dependable way. This experience report is a call to arms to make that version a mechanised version. Therefore, we believe that

we should strive for developing our theories together with their mechanised meta-theory. This experience report, and its resulting tools and techniques are meant to help in that direction. Our effort represents a step in an ongoing community wide effort. However small it may be, we think it is a step in the right direction.

Our future work can be split in two aspects. First, we want to further develop and simplify the locally nameless and the generic environment implementation that our system supports. The engineering effort for this proof was significant, but besides having this result, automation will make it easier to have more results like this. The second aspect is to extend our results to other calculi, notably we are interested in implementing multiparty session types [Honda et al. 2008] and their meta-theory. This would be an important result on its own, but we also want to extract a library of certified operations on these types (e.g: code for projections, and state machine generation). Concretely, our future goal is to certify the protocol description language, Scribble[8] which has been used for various endpoint programming languages [Gay and Ravera 2017], including Java [Hu and Yoshida 2016, 2017; Kouzapas et al. 2016], Go [Castro et al. 2019], Scala [Scalas et al. 2017], F# [Neykova et al. 2018], MPI-C [Ng et al. 2015; Ng and Yoshida 2014], Erlang [Neykova and Yoshida 2017a], and Python [Demangeon et al. 2015; Neykova et al. 2017; Neykova and Yoshida 2017b]. This would provide a certified tool-chain, that is developed together and at the same time as its own meta-theory.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. nnnnnnn and Grant No. mmmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

2019. Abella Examples. http://abella-prover.org/examples/index.html. Accessed: 2019-07-04.

2019. BehAPI Workshop @ ETAPS 2019. https://www.um.edu.mt/projects/behapi/behapi-workshop-etaps-2019/. Accessed: 2019-07-04.

Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Denielou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. 2016. Behavioral Types in Programming Languages. *FTPL* 3(2-3) (2016), 95–230.

Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '18)*. ACM, New York, NY, USA, 56–65. https://doi.org/10.1145/3209108.3209189

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. Engineering Formal Metatheory. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM, New York, NY, USA, 3–15. https://doi.org/10.1145/1328438.1328443

Jesper Bengtson and Joachim Parrow. 2009. Formalising the pi-calculus using nominal logic. *Logical Methods in Computer Science* Volume 5, Issue 2 (June 2009). https://doi.org/10.2168/LMCS-5(2:16)2009

Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. 2012. Strongly Typed Term Representations in Coq. *J. Autom. Reasoning* 49, 2 (2012), 141–159.

David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. 2019. Distributed Programming Using Role-parametric Session Types in Go: Statically-typed Endpoint APIs for Dynamically-instantiated Communication Structures. *Proc. ACM Program. Lang.* 3, POPL, Article 29 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290342

Arthur Charguéraud. 2012. The Locally Nameless Representation. *Journal of Automated Reasoning* 49, 3 (01 Oct 2012), 363–408.

Adam J. Chlipala. 2008. Parametric higher-order abstract syntax for mechanized semantics. In *13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, James Hook and Peter Thiemann (Eds.). ACM, 143–156.

[8]https://www.scribble.org

Alonzo Church. 1940. A formulation of the simple theory of types. *The Journal of Symbolic Logic* 5 (6 1940), 56–68. Issue 02.

Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. 2015. A Gentle Introduction to Multiparty Asynchronous Session Types. In *15th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Multicore Programming (LNCS)*, Vol. 9104. Springer, 146–178.

N.G. de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indag. Math* 34, 5 (1972), 381–392.

Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. 2015. Practical interruptible conversations: distributed dynamic verification with multiparty session types and Python. *FMSD* 46, 3 (2015), 197–225. https://doi.org/10.1007/s10703-014-0218-8

Amy Felty and Alberto Momigliano. 2012. Hybrid: A Definitional Two-Level Approach to Reasoning with Higher-Order Abstract Syntax. 48, 1 (2012), 43–105.

Amy P. Felty. 2019. A Linear Logical Framework in Hybrid (Invited Talk). In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2019)*. ACM, New York, NY, USA, 14–14. https://doi.org/10.1145/3293880.3294088

Andrew Gacek. 2008. The Abella Interactive Theorem Prover (System Description). In *Proceedings of IJCAR 2008 (Lecture Notes in Artificial Intelligence)*, A. Armando, P. Baumgartner, and G. Dowek (Eds.), Vol. 5195. Springer, 154–161.

Simon Gay and Malcom Hole. 2005. Subtyping for session types in the pi calculus. *Acta Informatica* 42, 2 (01 Nov 2005), 191–225. https://doi.org/10.1007/s00236-005-0177-z

Simon Gay and Antonio Ravera (Eds.). 2017. *Behavioural Types: from Theory to Tools*. River Publishers.

Simon J. Gay and Vasco T. Vasconcelos. 2010. Linear type theory for asynchronous session types. *Journal of Functional Programming* 20, 1 (2010), 19âĂŞ50. https://doi.org/10.1017/S0956796809990268

Aina Linn Georges, Agata Murawska, Shawn Otis, and Brigitte Pientka. 2017. LINCX: A Linear Logical Framework with First-Class Contexts. In *Programming Languages and Systems*, Hongseok Yang (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 530–555.

Georges Gonthier and Assia Mahboubi. 2010. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning* 3, 2 (2010), 95–152. https://doi.org/10.6092/issn.1972-5787/1979

Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. 2013. How to make ad hoc proof automation less ad hoc. *Journal of Functional Programming* 23, 4 (2013), 357âĂŞ401. https://doi.org/10.1017/S0956796813000051

Andrew D. Gordon. 1994. A mechanisation of name-carrying syntax up to alpha-conversion. In *Higher Order Logic Theorem Proving and Its Applications*, Jeffrey J. Joyce and Carl-Johan H. Seger (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 413–425.

Matthew Goto, Radha Jagadeesan, Alan Jeffrey, Corin Pitchar, and James Riely. 2016. An extensible approach to session polymorphism. *Mathematical Structures in Computer Science* 26, 3 (2016), 465âĂŞ509. https://doi.org/10.1017/S0960129514000231

Robert Harper and Daniel R. Licata. 2007. Mechanizing metatheory in a logical framework. *Journal of Functional Programming* 17, 4-5 (2007), 613âĂŞ673. https://doi.org/10.1017/S0956796807006430

Kohei Honda. 1993. Types for dyadic interaction. In *CONCUR'93*, Eike Best (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 509–523.

Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems*, Chris Hankin (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 122–138.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *Proc. of 35th Symp. on Princ. of Prog. Lang. (POPL '08)*. ACM, New York, NY, USA, 273–284.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* 63, 1 (2016), 9:1–9:67. https://doi.org/10.1145/2827695

Raymond Hu and Nobuko Yoshida. 2016. Hybrid Session Verification Through Endpoint API Generation. In *Fundamental Approaches to Software Engineering - 19th International Conference, FASE 2016,Eindhoven, The Netherlands (Lecture Notes in Computer Science)*, Perdita Stevens and Andrzej Wasowski (Eds.), Vol. 9633. Springer, 401–418.

Raymond Hu and Nobuko Yoshida. 2017. Explicit Connection Actions in Multiparty Session Types. In *FASE (LNCS)*, Vol. 10202. 116–133. https://doi.org/10.1007/978-3-662-54494-5_7

Raymond Hu, Nobuko Yoshida, and Kohei Honda. 2008. Session-Based Distributed Programming in Java. http://www.doc.ic.ac.uk/~rhu. In *ECOOP'08 (LNCS)*, Vol. 5142. Springer, 516–541.

Wen Kokke. 2019. Formalising session-typed languages without worries. Invited talk at Workshop on Behavioural APIs.

Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. 2016. Typechecking protocols with Mungo and StMungo. In *PPDP*. 146–159. https://doi.org/10.1145/2967973.2968595

Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. 2017. Fencing off Go: Liveness and Safety for Channel-based Programming. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL*

2017). ACM, 748–761. https://doi.org/10.1145/3009837.3009847

Kevin Liao, Matthew A. Hammer, and Andrew Miller. 2019. ILC: A Calculus for Composable, Computational Cryptography. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, New York, NY, USA, 640–654. https://doi.org/10.1145/3314221.3314607

Conor McBride and James McKinna. 2004. Functional Pearl: I Am Not a Number–i Am a Free Variable. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell (Haskell '04)*. ACM, New York, NY, USA, 1–9. https://doi.org/10.1145/1017472.1017477

James McKinna and Robert Pollack. 1999. Some Lambda Calculus and Type Theory Formalized. *Journal of Automated Reasoning* 23, 3 (01 Nov 1999), 373–409. https://doi.org/10.1023/A:1006294005493

Dale Miller and Catuscia Palamidessi. 1999. Foundational Aspects of Syntax. *ACM Comput. Surv.* 31, 3es, Article 11 (Sept. 1999).

Robin Milner, Joachim Parrow, and David Walker. 1992. A Calculus of Mobile Processes, Parts I and II. *Info.& Comp.* 100, 1 (1992).

Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. 2010. Structuring the Verification of Heap-manipulating Programs. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. ACM, New York, NY, USA, 261–274. https://doi.org/10.1145/1706299.1706331

Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. 2017. Timed runtime monitoring for multiparty conversations. *Formal Asp. Comput.* 29, 5 (2017), 877–910. https://doi.org/10.1007/s00165-017-0420-8

Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. 2018. A Session Type Provider: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F#. In *27th International Conference on Compiler Construction*. ACM, 128–138. https://doi.org/10.1145/3178372.3179495

Rumyana Neykova and Nobuko Yoshida. 2017a. Let it Recover: Multiparty Protocol-Induced Recovery. In *CC*. ACM, 98–108.

Rumyana Neykova and Nobuko Yoshida. 2017b. Multiparty Session Actors. *Logical Methods in Computer Science* 13, 1 (2017). https://doi.org/10.23638/LMCS-13(1:17)2017

Nicholas Ng, Jose G.F. Coutinho, and Nobuko Yoshida. 2015. Protocols by Default: Safe MPI Code Generation based on Session Types. In *CC 2015 (LNCS)*, Vol. 9031. Springer, 212–232.

Nicholas Ng and Nobuko Yoshida. 2014. Pabble: parameterised Scribble. *SOCA* (2014), 1–16.

Tobias Nipkow and Lawrence C. Paulson. 1992. Isabelle-91. In *Proceedings of the 11th International Conference on Automated Deduction, Saratoga Springs, NY (Lecture Notes in Artificial Intelligence (LNAI) vol. 607)*. Springer-Verlag, 673–676.

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph.D. Dissertation. Department of Computer Science and Engineering, Chalmers University of Technology. Technical Report 33D.

Dominic A. Orchard and Nobuko Yoshida. 2015. Using session types as an effect system. In *Proceedings Eighth International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES 2015, London, UK, 18th April 2015*. 1–13. https://doi.org/10.4204/EPTCS.203.1

Frank Pfenning and Conal Elliott. 1988. Higher-Order Abstract Syntax. In *ACM SIGPLAN Symposium on Language Design and Implementation (PLDI'88)*. 199–208.

Brigitte Pientka and Andrew Cave. 2015. Inductive Beluga: Programming Proofs. In *Automated Deduction - CADE-25*, Amy P. Felty and Aart Middeldorp (Eds.). Springer International Publishing, Cham, 272–281.

Randy Pollack. 1994. Closure under alpha-conversion. In *Types for Proofs and Programs*, Henk Barendregt and Tobias Nipkow (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 313–332.

Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. 2017. A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming. In *ECOOP'17 (LIPIcs)*, Vol. 74. Sch. Dag., 24:1–24:31.

Alceste Scalas, Nobuko Yoshida, and Elias Benussi. 2019. Verifying Message-passing Programs with Dependent Behavioural Types. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, New York, NY, USA, 502–516. https://doi.org/10.1145/3314221.3322484

Kaku Takeuchi, Kohei Honda, and Makoto Kubo. 1994. An interaction-based language and its typing system. In *PARLE'94 Parallel Architectures and Languages Europe*, Costas Halatsis, Dimitrios Maritsas, George Philokyprou, and Sergios Theodoridis (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 398–413.

Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A Higher-Order Logic for Concurrent Termination-Preserving Refinement. In *Programming Languages and Systems*, Hongseok Yang (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 909–936.

The Coq Development Team. 2016. *The Coq Proof Assistant Reference Manual v. 8.6.1.* Institut National de Recherche en Informatique et en Automatique.

Christian Urban. 2008. Nominal Techniques in Isabelle/HOL. 40, 4 (2008), 327–356.

Philip Wadler and Wen Kokke. 2019. *Programming Language Foundations in Agda*. Available at http://plfa.inf.ed.ac.uk/.

Nobuko Yoshida and Vasco T. Vasconcelos. 2007. Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited: Two Systems for Higher-Order Session Communication. *Electronic Notes in Theoretical*

Computer Science 171, 4 (2007), 73 – 93. Proceedings of the First International Workshop on Security and Rewriting Techniques (SecReT 2006).