
University of Glasgow | School of Computing Science

# Type-checking session-typed $\pi$-calculus with Coq

Uma Zalakain

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

A dissertation presented in part fulfilment of the requirements of the Degree of Master of Science at The University of Glasgow

2019-09-06

**Abstract**

This project formalises the session-typed $\pi$-calculus in Coq using a mix of continuation passing, parametric HOAS, dependent types and ad-hoc linearity checks. Each action a process takes requires a channel capable of that action. The head of that channel's type is then stripped off and its continuation is passed to the next action the process takes. Dependent types guarantee this continuation passing is correct by construction. The type of channels is parametrised over, so that users are unable to skip the proper mechanisms to create channels. The HOAS makes the syntax easy to use for both the end user and the designer: all variables are lifted to Coq, no typing contexts are required. The continuation passing always creates channels that must be used exactly once, but unfortunately Coq has no support for linearity, so this check needs to happen ad-hoc, by traversing processes. Ultimately, the claim is this: if the definition of a process typechecks in Coq, and the process uses channels linearly, then type safety and type preservation through reduction hold.

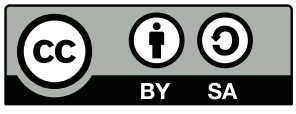
go over this again

## Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: Uma Zalakain Signature: Uma Zalakain





## Acknowledgements

Acknowledgements go here

# Contents

# 1. Introduction

During the last decades, while the frequency at which processors run has peaked, the number of available processing units has kept growing. Computing has consequently shifted its focus into making processes safely communicate with one another — no matter if they run concurrently on different CPU cores or on different hosts. The interest in the formalisation and verification of *communicating concurrent* systems (where processes share no state and change as communication occurs) has grown as a result.

Communicating concurrent processes must satisfy some safety properties, such as following a pre-established communication protocol (where all messages sent by one process are expected by the other and vice versa) or communicating over private channels only known to the involved participants. To make properties like these easier to prove, formal models such as the *$\pi$-calculus* [WMP89, Mil89, Mil91, SW01] abstract real-world systems into suitable mathematical representations. §2.1 provides a brief overview of the $\pi$-calculus.

The properties of a formal system can be verified either *dynamically*, by monitoring processes at runtime, or *statically*, by reasoning on the definition of the processes themselves. Static guarantees — while harder to define and sometimes more conservative than dynamic ones — are *total*, and thus satisfied regardless of the execution path. The basis of static verification is comprised of *types* and *type systems*, which are also the basis of programming languages and tools, making type-based verification techniques transferable to practical applications. An example of this are the plethora of types for communication and process calculi: from standard channel types, as found in e.g., Erlang or Go, to *session types* [Hon93, THK94, HVK98], a formalism used to specify and verify communication protocols (more in §2.2).

move session type definition and properties up here?

The mechanised formalisation and verification of programming languages and calculi is an ongoing community effort in securing existing work: humans are able to check proofs, but they are very likely to make mistakes; machines can verify proofs mechanically. A remarkable example of a community effort towards machine verification is RustBelt [JJKD17], a project that aims to formalise and machine-check the ownership system of the programming language Rust with the help of separation logic [] and the proof assistant Coq. Not only does mechanisation increase confidence in what is mechanised, but also in all other derived work that is yet unverified: proving the correctness of Rust's type system immediately increases the confidence in all software written in it.

make this short, take it out of the box, just centered and in italics

**THESIS STATEMENT**

This project formalises the *session-typed $\pi$-calculus* in such a way that variable references in the object language are lifted into variable references in the host language. The use of channels in the object language is then restricted to be linear, ensuring *communication privacy*, *communication safety* and *session fidelity* (refer to §**??**). Lastly, we machine-verify *subject reduction* for the resulting language.

We choose Coq [CP89, Coq] to machine-verify the session-typed $\pi$-calculus, mainly due to its

mention we have no shared channels and no recursive

widespread use as a proof assistant (refer to §2.3 for an overview). A first challenge with Coq is that it offers *no* support for *linearity*, which is at the very heart of session types (as communication occurs, a session type must transition through each of its stages exactly once). As a result, extra work is required to simulate the linearity of the terms in the object language.

The present work simulates linearity by defining it as an inductive predicate on processes (§3.5). Once such a predicate establishes that a process uses channels linearly, the use of channels according to their specification is guaranteed by construction. References to both channels and messages are lifted into the host language, making the resulting syntax amicable to the user. As a consequence of this approach the object language requires *no typing contexts* and *no substitution lemmas*.

Other approaches to formalising process calculi are briefly exposed in §5. Closing, §6 suggests future work that might be of interest, and offers conclusions on what this project has achieved.

# 2. Background

## 2.1 $\pi$-calculus

**Scope** This section provides an overview of the $\pi$-calculus as introduced in [SW01]. However, it deliberately ignores replication and indeterministic choice, features part of the $\pi$-calculus that are not covered by this project. Additionally, and as preliminary preparation for the introduction of session types, this section defines channel restriction by introducing two channel *endpoints*, instead of the usual single variable used for channels.

polarity

The $\pi$-calculus [WMP89, Mil89, Mil91, SW01] models processes that progress and change their structure by using *channels* to communicate with one another. The $\pi$-calculus features *channel mobility*, which allows channels to be sent over channels themselves. In the $\pi$-calculus any number of processes can communicate over a channel. While the $\pi$-calculus can be typed, the type of a channel does *not* evolve as communication occurs: it only specifies the type of data sent over it. An overview of the FAQs can be found in [Win02].

The syntax of the $\pi$-calculus is given by the grammar in Figure 2.1. Inaction denotes the end of a process, and has therefore no continuation. Scope restriction creates a new communication channel between endpoints $x$ and $y$, which are bound in $P$. Output sends $u$ over the channel endpoint $x$, and then continues as $P$. Input waits to receive $u$ on the endpoint $y$; upon reception $u$ is bound in $P$. Selection sends the choice of process $l_j$ over $x$, and then continues as $P$. Branching offers choices over $I$, where the choice $l_i$ selects the continuation process $P_i$. Parallel composition runs processes $P$ and $Q$ in parallel, allowing these processes to communicate over shared channels.

| | | |
|---|---|---|
| $P, Q ::= \mathbf{0}$ | | inaction |
| | $(\boldsymbol{\nu} xy)\ P$ | scope restriction |
| | $\overline{x}\langle u \rangle.P$ | output |
| | $y(u).P$ | input |
| | $x \triangleleft l_j.P$ | selection |
| | $x \triangleright \{l_i : P_i\}_{i \in I}$ | branching |
| | $P \mid Q$ | parallel composition |

Figure 2.1: Grammar describing the syntax of the $\pi$-calculus

The syntax of the $\pi$-calculus captures undesired syntactical properties of processes (e.g. associativity should not matter when three processes are composed in parallel). Structural congruence is introduced as a way to abstract over these unintended differences in syntax. It is defined by the smallest congruent equivalence relation that satisfies the inference rules in Figure 2.2 – a congru-

ent equivalence relation in itself is the smallest relation that is reflexive, symmetric, transitive and congruent. Worth noting is the structural congruence rule for scope expansion: the scope of bound variables can include or exclude a process at will, as long as the bound variables do not appear free in that process. The congruence rule states that if two processes considered equal are placed within a common context, then the resulting contexts are equal as well (a context is a process where some occurrence of $\mathbf{0}$ is substituted by a *hole* that can then be filled in with a process). Said otherwise, structural congruence *goes under* the syntactic constructs of the $\pi$-calculus.

$$\frac{}{P \mid Q \equiv Q \mid P} \quad \text{(C-CompComm)}$$

$$\frac{}{(\boldsymbol{\nu} xy)\ (\boldsymbol{\nu} zw)\ P \equiv (\boldsymbol{\nu} zw)\ (\boldsymbol{\nu} xy)\ P} \quad \text{(C-ScopeComm)} \qquad \frac{}{P \mid \mathbf{0} \equiv P} \quad \text{(C-Comp0)}$$

$$\frac{}{(P \mid Q) \mid R \equiv P \mid (Q \mid R)} \quad \text{(C-CompAssoc)} \qquad \frac{}{(\boldsymbol{\nu} xy)\ \mathbf{0} \equiv \mathbf{0}} \quad \text{(C-Scope0)}$$

$$\frac{}{(\boldsymbol{\nu} xy)\ P \equiv (\boldsymbol{\nu} yx)\ P} \quad \text{(C-ScopeSwap)} \qquad \frac{x, y \notin fn(Q)}{((\boldsymbol{\nu} xy)\ P) \mid Q \equiv (\boldsymbol{\nu} xy)\ P \mid Q} \quad \text{(C-ScopeExp)}$$

Figure 2.2: Structural congruence rules for the $\pi$-calculus

The operational semantics of the $\pi$-calculus is specified by reduction rules, defined in Figure 2.3. Two parallel processes communicating over the same channel (by using opposite endpoints to send and receive a message) get reduced to the parallel composition of their continuations, with the continuation of the receiving process having all the references to the message substituted by the message term itself (R-Comm). Similarly, a process that makes a choice put in parallel with a process that offers a choice gets reduced to the continuation of the choosing process and the chosen continuation of the process offering the choice – so long as the choice itself is valid (R-Case). Reduction goes under both restriction (R-Res) and parallel composition (R-Par), but not under output, input, selection or branching – constructs that impose order in the communication. Lastly, reduction is defined up to structural congruence: any amount of syntax rewriting can be performed before and after reduction (R-Struct).

$$\frac{}{(\boldsymbol{\nu} xy)\ (\overline{x}\langle a\rangle.P \mid y(b).Q) \to (\boldsymbol{\nu} xy)\ (P \mid Q[a/b])} \quad \text{(R-Comm)}$$

$$\frac{j \in I}{(\boldsymbol{\nu} xy)\ (x \triangleleft l_j.P \mid y \triangleright \{l_i : Q_i\}_{i \in I}) \to (\boldsymbol{\nu} xy)\ (P \mid Q_j)} \quad \text{(R-Case)}$$

$$\frac{P \to Q}{(\boldsymbol{\nu} xy)\ P \to (\boldsymbol{\nu} xy)\ Q} \quad \text{(R-Res)} \qquad \frac{P \to Q}{P \mid R \to Q \mid R} \quad \text{(R-Par)}$$

$$\frac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q} \quad \text{(R-Struct)}$$

Figure 2.3: Reduction rules for the $\pi$-calculus

As an example, Figure 2.4 creates two linked channel endpoints $x$ and $y$ and then composes two processes in parallel: one that uses $x$ to send integers 3 and 4, and then expect a response bound as $r$, do some $P$, then end; another that uses $y$ to receive $a$ and $b$, then send $a + b$, then end. Both processes communicate with one another when composed in parallel, changing their structure.

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

Figure 2.4: Example process in the $\pi$-calculus

The grammar for the $\pi$-calculus allows well-formed processes with no semantic meaning (e.g. $(\boldsymbol{\nu} xy)\ \overline{x}\langle true\rangle.\mathbf{0} \mid y(u).(u+3).\mathbf{0}$). The syntax rules for the construction of $\pi$-calculus terms can be refined to discard some of these constructs. One such refinement are shared types (Figure 2.5), which ensure that the types of channels and messages match – in the example in Figure 2.4, channel endpoints $x$ and $y$ would need to be of the shared base type `Int` for the processes to be well-typed. The formation of $\pi$-calculus terms can be further restricted with *session types*: types that serve to specify communication protocols.

| | |
|---|---|
| $T ::= \texttt{Chan}[T]$ | channel type |
| $\ldots$ | base type |

Figure 2.5: Shared types for the $\pi$-calculus

## 2.2 Session types

**Scope** This section covers a subset of session types: the diadic (shared by two processes), finite (no replication or recursion), deterministic (no indeterministic choice), and synchronous session types.

Session types [Hon93, THK94, HVK98] encode sequences of actions, each action containing the type and the direction of the data exchanged. Processes must use session-typed channels according to their specified protocol. Instead of being shared and static, session types are linear, private to the communicating processes, and changing as communication occurs. A comprehensive introduction to session types can be found in [Vas09], while answers to FAQs are compiled in [DD10].

The grammar of session types is listed in the Figure 2.6. Channel termination admits no continuation. For sending and receiving, the type of the transmitted data and the session type of the continuation are required. The types transmitted can either be base types or session types. Branching and selection both expect a set of session types which contains the continuation that will be chosen. In

the example process introduced in Figure 2.4, the session type of $x$ is `!Int.!Int.?Int.End`, while the one of $y$ is `?Int.?Int.!Int.End`

| | | |
|---|---|---|
| $S ::=$ | `End` | termination |
| | $!T.S$ | send |
| | $?T.S$ | receive |
| | $\oplus\{l_i : S_i\}_{i \in I}$ | select |
| | $\&\{l_i : S_i\}_{i \in I}$ | branch |
| $T ::=$ | $S$ | session type |
| | $\ldots$ | base type |

Figure 2.6: Grammar for session types

Note that the session types of $x$ and $y$ must be *dual*: when one channel sends a type `T`, the other must receive `T`, and then both must continue dually. The precise definition of duality is given in Figure 2.7. Duality is one of the core principles of session types, as it guarantees *communication safety*.

$$\overline{!T.S} = ?T.\overline{S} \qquad \overline{?T.S} = !T.\overline{S} \qquad \overline{\&\{l_i : S_i\}_{i \in I}} = \oplus\{l_i : \overline{S_i}\}_{i \in I}$$

$$\overline{\oplus\{l_i : S_i\}_{i \in I}} = \&\{l_i : \overline{S_i}\}_{i \in I} \qquad \overline{\texttt{End}} = \texttt{End}$$

Figure 2.7: Duality on session types

Session-typed channels impose typing rules on the syntactical constructs of the $\pi$-calculus: a process can perform an action on a channel only if that channel is capable of the action. The typing rules for processes using session typed channels are shown in Figure 2.8. The judgment $\Gamma \vdash P$ signifies that $P$ is well typed under the context $\Gamma$. Contexts are linear: their elements cannot be duplicated nor discarded. The disjoint union operator $\circ$ represents context split: every element in the context $\Gamma_1 \circ \Gamma_2$ must appear exactly once in one of $\Gamma_1$ or $\Gamma_2$, but not in both.

A process can be terminated if the only channel in context is a channel that is expecting to be terminated — thus premature termination is avoided. Restriction creates two linked channel endpoints, where these endpoints are dual. Parallel composition splits the linear context in two. Input requires a channel capable of receiving and a continuation process typed under the continuation channel and the received input — context is split between those two. Output requires a channel capable of sending, a value to be sent, and a continuation process typed under the continuation channel — context is split between those three. Selection requires a channel capable of selecting, a process typed under the internally selected continuation channel, and a valid selection —— context is split between the first two. Branching requires a channel capable of branching, and for every possible external choice, a process typed under the externally chosen continuation channel — context is split between the two.

$$\frac{}{x : \mathtt{End} \vdash \mathbf{0}} \text{ (T-INACT)} \qquad \frac{\Gamma, x : T, y : \overline{T} \vdash P}{\Gamma \vdash (\boldsymbol{\nu} xy)\ P} \text{ (T-RES)} \qquad \frac{\Gamma_1 \vdash P \qquad \Gamma_2 \vdash Q}{\Gamma_1 \circ \Gamma_2 \vdash P \mid Q} \text{ (T-PAR)}$$

$$\frac{\Gamma_1 \vdash x : ?T.S \qquad \Gamma_2, x : S, y : T \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash x(y).P} \text{ (T-IN)}$$

$$\frac{\Gamma_1 \vdash x : !T.S \qquad \Gamma_2 \vdash v : T \qquad \Gamma_3, x : S \vdash P}{\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash \overline{x}\langle v \rangle.P} \text{ (T-OUT)}$$

$$\frac{\Gamma_1 \vdash x : \oplus\{l_i : S_i\}_{i \in I} \qquad \Gamma_2, x : S_i \vdash P_i \qquad \exists j \in I}{\Gamma_1 \circ \Gamma_2 \vdash x \triangleleft l_j.P} \text{ (T-SELECT)}$$

$$\frac{\Gamma_1 \vdash x : \&\{l_i : S_i\}_{i \in I} \qquad \Gamma_2, x : S_i \vdash P_i \qquad \forall i \in I}{\Gamma_1 \circ \Gamma_2 \vdash x \triangleright \{l_i : P_i\}_{i \in I}} \text{ (T-BRANCH)}$$

Figure 2.8: Typing rules for processes using session typed channels

The reduction of session types is a byproduct of these rules, and follows the operational semantics of the $\pi$-calculus listed in Figure 2.3: when two processes either communicate (R-COMM) or make a choice (R-CASE) over linked dual channel endpoints, the *head* of those channel endpoints (their first action) is consumed.

As a result of this setup, session types guarantee three properties:

**Communication privacy** Every channel endpoint is used exactly by one process. This applies to channels created by message input as well as by scope restriction. This property is a byproduct of the **linearity** of contexts: parallel composition splits the linear context into two disjoint unions, effectively deciding which process gets to use the endpoint.

**Communication safety** Values sent by one process are expected by the process on the other side of the channel. This property arises as a result of **duality**: only processes that communicate over channel endpoints linked through restriction can be reduced, and restriction requires channel endpoints to have dual session types.

**Session fidelity** Processes follow session types sequentially. This property is a consequence of linearity and the way in which the typing rules in Figure 2.8 deconstruct session types by taking their continuations apart.

## 2.3 Coq proof assistant

Coq [Coq] is a popular proof assistant and dependently typed functional language based on the calculus of inductive constructions [CP89] (which adds inductive data types to the calculus of constructions [CH85]), a type theory isomorphic to intuitionistic predicate calculus — a constructive logic with quantified statements. Coq features proof irrelevance for proofs (in `Prop`) and a cumulative set of universes (in `Type`).

The following example introduces some of the basic building blocks of dependent type programming. The type `Zero` (also known as $\bot$) has no constructors: there exist no programs that inhabit it. The proposition `P → Zero` represents negation, that `P` is provably false. Conversely, from a proof of falsity one might conclude anything: `Zero → P`, for any `P`. The dual of $\bot$ is $\top$, here represented as `One`: the trivial type that contains no information. Sigma is the existential type, or dependent tuple: for some $A$ and some predicate $P$, the first element of the tuple is an $A$; the second element of the tuple is a proof that $P$ holds for that particular $A$.

```
Inductive Zero : Type :=.
Definition ¬ (P : Type) : Type := P → Zero.
Inductive One : Type := tt.

Inductive Sig (A : Type) (P : A → Type) : Type :=
  sig : ∀ (a : A), P a → Sig A P.
Arguments sig {A P}.
```

Coq is dependently typed: types can depend on — or even contain — programs. Since that is the case, programs in Coq must exhibit termination — recursion must occur on structurally smaller terms. The example bellow introduces the recursive function `Even`, which given a natural number returns a type — that is to say, it relates each natural number with a proposition, in these case capturing their evenness. Below it, a proof that uses sigma types to show that there is at least one natural number that is even.

```
Fixpoint Even (n : ℕ) : Type :=
  match n with
  | Z       ⇒ One
  | (S Z)   ⇒ Zero
  | (S (S n)) ⇒ Even n
  end.

Example _ : Sig ℕ Even := sig 42 tt.
```

Coq allows users to build proofs using *tactics*: programs written in $L_{tac}$ that manipulate hypotheses and transform goals. While these programs might be incorrect, or not terminate, their outcome is ultimately checked by *Gallina*, the specification language of Coq. The example below proves for every natural number $n$ that if $n$ is even, then $n + 1$ is not. It does so through the sequential application of tactics. Each tactic manipulates the goal and context of the proof (see annotations in comments). The proof proceeds by induction on $n$: the proof obligation in the base case reduces to `¬ (Even (S Z))`, which by definition of `Even` reduces to `¬ Zero`, that is, `Zero → Zero`, the identity function. The inductive step is provable thanks to the fact that `Even (S (S n))` reduces to `Even n`.

```
Lemma SEven0 (n : ℕ) : Even n → ¬ (Even (S n)).
Proof.
  (* n : ℕ
     --------------------
     Even n → ¬ (Even (S n)) *)
  intros En ESn.

  (* n : ℕ
     En : Even n
     ESn : Even (S n)
     --------------------
     Zero *)
```

```
  induction n.

  (* En : Even 0
     ESn : Even 1
     --------------------
     Zero *)
  contradiction.

  (* n : ℕ
     En : Even (S (S n))
     ESn : Even (S (S n))
     IHn : Even n → Even (S n) → Zero
     --------------------
     Zero *)
  apply (IHn ESn En).
Qed.
```

In Coq, simultaneously pattern matching on multiple indexed data types can be extremely clunky and arduous. The *Equations* package eases this inconvenience by enabling an equational definition style, pattern matching on the left, and `with` constructs ([MM04]), making Coq as convenient for dependent pattern matching as Agda. The theorem `SEven0` is proven again here, this time using dependent pattern matching. Coq is able to use tactics to solve the base case automatically, as `ESn` is uninhabited.

```
From Equations Require Import Equations.

Equations SEven1 (n : ℕ) : Even n → ¬ (Even (S n)) := {
SEven1 Z     En ESn := _ ;
SEven1 (S n) En ESn := SEven1 n ESn En}.
```

Coq supports inductive and coinductive data types,

indexed families of types

```
Inductive lte : ℕ → ℕ → Type :=
| zlte : ∀ {n}, lte Z n
| slte : ∀ {n m}, lte n m → lte (S n) (S m)
.

Inductive Vec (A : Type) : ℕ → Type :=
| nil  : Vec A Z
| cons : ∀ {n}, A → Vec A n → Vec A (S n)
.
```

# 3. Design

This chapter describes how we encode the session-typed $\pi$-calculus into Coq. Our encoding only handles session-typed channels, specifically channels with finite (non-replicating, non-recursive) session types. The encoding has no intermediary models — for instance, the polarised linear $\pi$-calculus. The main characteristic of our design is the use of function abstraction to pass around both messages and channels (see §3.1); alternative approaches are listed in §3.2.

## 3.1 Overview

Our design is based on *continuation passing*, where channels are **used exactly once**: channels get destroyed and created with every action taken by a process. This approach has its origin in the encoding of session types into the polarised linear $\pi$-calculus [Dar16, **?**]: session-typed channels are transformed into single-use polarised channels, each containing the type of data being transmitted and the encoding of the channel's continuation. For instance, the session type $!A.?B.\texttt{End}$ is encoded as $\ell_o[A, \ell_i[B, \ell_\varnothing[]]]$ where $\ell_i[A, Cont]$ is a single-use channel signifying input $A$, then continue as $Cont$ — and similarly for output. When a process uses a channel with such a type to communicate, that channel gets destroyed, the *head* of the channel's type is consumed, and a new channel with the *tail* of the type ($\ell_i[B, \ell_\varnothing[]]$) is created.

The main characteristic of our design is the use of function abstraction as a way of passing around references to both messages and channels. That is, references are not modeled in the object language, but rather in the host language, in Coq.

### 3.1.1 Advantages

- The user can refer to both messages and channels naturally.
  Example
- There is no need for machinery that avoids variable capture.
  Explanation
- There is no need for substitution lemmas.
  Explanation

### 3.1.2 Drawbacks

This approach comes with several problems that need to be solved:

- Information about the session types of channels must be kept in the channels themselves.
  Explanation

Solution §3.3
- Channels can be manufactured without following the procedures in the calculus.
  Explanation
  Solution §3.4
- Channels can be used more than once, or not used at all.
  Explanation
  Solution §3.5

mention the prize to pay

## 3.2 Alternative encodings

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

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

#### Using de Bruijn indices

#### Adding shared channels

## 3.3 Dependent types

[**?**]

Assuming linearity, **processes are correct by construction**: the processes that can be constructed depend on the session types of the channels in the environment of the host language; an action strips off the outer layer of a channel's session type — modelling **continuation passing**.

## 3.4 Parametric HOAS

[Wad89] [Chl08]

We use polymorphism to make channels opaque

We do

Polymorphism allows programs to depend on types. If those types are unconstrained — if the universe of types is infinite — then programs are unable pattern match on them (as they cannot possibly produce infinitely many cases) and unable to create terms inhabiting them. Such unconstrained polymorphic types are said to be *opaque*. Programs depending on opaque types exhibit common general properties that are highlighted in [Wad89]. This project makes use of polymorphism to prevent the creation and inspection of terms of a polymorphic type.

## 3.5 Linearity

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

[KPNT99] [TCP11]

## 3.6 Subject reduction

Ensuring that linearity is preserved through reduction is therefore essential:

**Theorem 1** $lin(P) \Rightarrow P \to Q \Rightarrow lin(Q)$.

# 4. Implementation

## 4.1 Session types

## 4.2 Messages and processes

## 4.3 Linearity check

## 4.4 Linearity preservation

## 4.5 Examples

Type inference

```
Example example1 : PProcess.
  refine
  ([v]> (new i ← _, o ← _, _)
    (i?[m]; ![m]; ϵ) <|> (o![v _ true]; ?[m]; ϵ)).
  auto.
Defined.
Print example1.

Example example2 : PProcess :=
  ([v]> (new o ← ! B[bool] ; ? B[bool] ; ø, i ← ? B[bool] ; ! B[bool] ; ø, ltac:(auto)
    (o![v _ true]; ?[m]; ϵ) <|> i?[m]; ![m]; ϵ).

Example congruent_example1 : example1 ≡ example2. auto. Qed.

Example example3 : PProcess :=
  ([v]> (new o ← ? B[bool] ; ø, i ← ! B[bool] ; ø, ltac:(auto))
    (o?[m]; ϵ) <|> i![v _ true]; ϵ).

Example reduction_example1 : example2 ⇒ example3. auto. Qed.

Example subject_reduction_example1 : example2 ⇒ example3 → Linear example2 →
Linear example3.
eauto.
Qed.

Example example4 : PProcess :=
  ([v]> (new i ← ! B[bool] ; ø, o ← ? B[bool] ; ø, ltac:(auto))
    (i![v _ true]; ϵ <|> o?[m]; ϵ)).

Example congruent_example2 : example3 ≡ example4. auto. Qed.
```

```
Example example5 : PProcess :=
  ([v]> (new i ←ø , o ←ø , Ends) (ϵ i <|> ϵ o)).

Example reduction_example2 : example4 ⇒ example5. auto. Qed.

Example big_step_reduction : example1 ⇒* example5. auto. Qed.

Example big_step_subject_reduction_example1
  : example1 ⇒* example5 → Linear example1 → Linear example5.
eauto.
Qed.

Example channel_over_channel : PProcess :=
  [v]>
    (new i ← ? C[ ! B[bool] ; ø ] ; ø, o ← ! C[ ! B[bool] ; ø ] ; ø, MLeft Ends)
    (new i' ← ? B[bool] ; ø, o' ← _, MLeft Ends)

    (i?[c]; fun a ⇒ ϵ a <|> c![v _ true]; ϵ)
    <|>
    (o![o']; fun a ⇒ ϵ a <|> i'?[_]; ϵ)
.

Example channel_over_channel1 : PProcess :=
  [v]>
    (new i' ← ? B[bool] ; ø, o' ← ! B[bool] ; ø, MLeft Ends)
    (new i ← ? C[ ! B[bool] ; ø ] ; ø, o ← ! C[ ! B[bool] ; ø ] ; ø, MLeft Ends)

    (i?[c]; fun a ⇒ c![v _ true]; ϵ <|> ϵ a)
    <|>
    (o![o']; fun a ⇒ i'?[_]; ϵ <|> ϵ a)
.

Example congruent_example3 : channel_over_channel ≡ channel_over_channel1. auto. Qed.

Example nonlinear_example : PProcess :=
  [v]> (new i ← ? B[bool] ; ø, o ← ! B[bool] ; ø, MLeft Ends)

    (* Cheat the system by using the channel o twice *)
    i?[_]; ϵ <|> o![v _ true]; (fun _ ⇒o![v _ true]; ϵ)
    .

Example linear_example1 : Linear example1. auto. Qed.

Example linear_channel_over_channel : Linear channel_over_channel. auto. Qed.

Example nonlinear_example1 :   (Linear nonlinear_example). auto. Qed.

Example branch_and_select : PProcess :=
  ([v]> (new
           i ← &{ (! B[bool] ; ø) :: (? B[bool] ; ø) :: [] },
           o ← ⊕{ (? B[bool] ; ø) :: (! B[bool] ; ø) :: [] },
           ltac:(auto))
         i⊳{(![v _ true]; ϵ) ; (?[m]; ϵ)} <|> o◁Fin.F1; ?[_]; ϵ).
```

# 5. Related work

fixme

Other efforts in formalising session types in Coq have created object languages and handled variable references, typing contexts, and typing judgments by hand [Dil19].

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

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

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

The $\pi$-calculus has been an extensive subject of machine verification: [**?**] proofs subject reduction for it; [**?**] proofs subject reduction as well, but uses a higher order syntax; [**?**] provides proofs of fairness and confluence; [**?**] formalises the bisimilarity proofs found in [WMP89]; [**?**] provides a framework for formalisation on the $\pi$-calculus with linear channels in Isabelle/HOL.

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

# 6. Conclusion

# 7. Bibliography


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

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

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

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

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

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

[Dil19] Eric Dilmore. *Pi-Calculus Session Types in Coq*. Master's Thesis, School of Computing Science, University of Glasgow, 2019.

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

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

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

[KPNT99] Naoki Kobayashi, Benjamin Pierce, and David N. Turner. Linearity and the Pi-Calculus. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21:914–947, December 1999.

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

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

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

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

[TCP11] Bernardo Toninho, Luís Caires, and Frank Pfenning. Dependent Session Types via Intuitionistic Linear Type Theory. In *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming*, PPDP '11, pages 161–172. ACM, 2011.

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

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

[Wad89] Philip Wadler. *Theorems for Free!* 1989.

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

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