University of Glasgow | School of Computing Science

Type-checking session-typed π-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

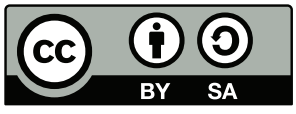
This project formalises the session-typed π-calculus in Coq using a mix of continuation passing, parametric HOAS, dependent types and ad-hoc linearity checks. Each action a process takes requires a channel capable of that action. The head of that channel's type is then stripped off and its continuation 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 *π-calculus* [WMP89, Mil89, Mil91, SW01] abstract real-world systems into suitable mathematical representations. §2.1 provides a brief overview of the π-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).

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.

This project formalises and verifies a subset of the session-typed π-calculus.

We choose Coq [**?**, Coq] to machine-verify the session-typed π-calculus, mainly due to its 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.

We use a higher order abstract syntax [Chl08] to lift bindings (of both channels and messages) in the object language into bindings in Coq. As a result, the object language requires *no typing contexts* and *no substitution lemmas*. Linearity is simulated through an inductive predicate on processes (§3.7). Dependent types ensure that processes are correct by construction up to linearity: processes satisfying the linearity predicate are guaranteed to use session-typed channels according to their specification.

We provide basic background on the π-calculus, session types and Coq in §2. We then introduce our design in §3 and comment on the results in §4. Alternative approaches are introduced in §5. Closing, §6 suggests future work of interest, and offers conclusions on what this project has achieved.

2. Background

2.1 π-calculus

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

The π-calculus [WMP89, Mil89, Mil91, SW01] models processes that progress and change their structure by using *channels* to communicate with one another. The π-calculus features *channel mobility*, which allows channels to be sent over channels themselves. In the π-calculus any number of processes can communicate over a channel. While the π-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 π-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
	$x \triangleright \{l_i : P_i\}_{i \in I}$	branching
	$x \triangleleft l_j.P$	selection
	$P \mid Q$	parallel composition

Figure 2.1: Grammar describing the syntax of the π-calculus

The syntax of the π-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 π-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 π-calculus

The operational semantics of the π-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. Finally, 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 π-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 π-calculus

The grammar for the π-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 π-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 π-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 π-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`

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

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 π-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 Γ. 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 Γ_1 or Γ_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.

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

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

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

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

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

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 π-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 [**?**] (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 $\mathbb{P}$) 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, also known as the unit type. 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 below, 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. The constructors of an inductive data type can contain recursive references to the data type — as long as they are strictly positive, that is, do not appear inside an argument to the left of an arrow [Dyb94]. The type of an inductive data type can accept both parameters and indices as arguments. Parameters appear on the left hand side of the colon, indices appear on the right hand side; parameters are fixed throughout the constructors, indices may change.

The example below defines vectors: lists that keep track of their length on the type level. The parameter `A` refers to the type of elements within the list, and therefore stays fixed through the induction. However, the index (of type $\mathbb{N}$) changes through induction: `nil` initialises it to 0; `cons` is provided with a term of type `A` and a vector of length `n`, and results in a vector of length `n + 1`.

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

The next example encodes proofs of a natural number n being less than or equal to a natural number m. These two naturals are kept as indices in `lte`. The constructor `zlte` initialises n to 0 and m to any number — 0 is less than or equal to any natural. The constructor `slte` increments both n and m. Every proof of $n \leq m$ can thus be encoded into exactly one recursive combination of the contructors `zlte` and `slte`.

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

3. Design

This chapter describes how we encode the session-typed π-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 π-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 §**??**. fixme

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 π-calculus [KPNT99, DGS12, 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 a higher order abstract syntax to lift binding of channels and messages in the object language into bindings in the host language. This approach offers several advantages, but it also creates several challenges that must be addressed: We do not do open processes

Advantages

1. *The user can refer naturally to both messages and channels*
   Our approach lifts all variable references into Coq. The example session-typed π-calculus process bellow has two branches: one that outputs `true` and receives some `m`; the other that receives some `m` and outputs that same `m`. Note that `m` gets bound as a variable in Coq. Alternative approaches that encode variable references in the object language itself need to use de Bruijn indices [de 72] or similar techniques, resulting in a processing overhead for users.

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

2. *There is no need for machinery that avoids variable capture* Explanation
3. *There is no need for substitution lemmas* Explanation

Problems

1. *Channels can be manufactured outside the calculus*
   The π-calculus provides two ways to obtain a channel: by using scope restriction, and by receiving a message that contains a channel. The user must be restricted to use these constructs. If the type of channels is defined inductively, the user is not prevented from forging channels. Our solution to this problem is to parametrise processes with an unknown channel type, and to define our calculus and its theorems parametrised by this unknown type. §3.3 covers our solution in depth.
2. *The type of messages must be tracked*
   The π-calculus allows messages to contain both terms of a given base type, and channels of a given session type. Messages must track information about the type of their content. Moreover, this information must be kept at the type level, so that the typechecker is able to verify that processes make correct use of messages. Dependent types allow session types to be defined as an inductive type, and to then index the type of messages by the type of value they contain — whether this is a session type or a base type. See §**??** and §3.4 for more details.
3. *Channels can be used more than once, or not used at all*
   Our approach encodes session-typed channels as single-use channels that are created and destroyed with every action taken by a process. To invoke these actions, an appropriate session-typed channel has to provided. The action then use function abstraction to provide an environment in which a channel with the tail of the original session type is received as an argument. This channel passed as an argument must be used exactly once. Unfortunately, Coq has no support for linearity, and thus this property has to be checked ad-hoc. How to do this is covered in §3.7.
4.

introduce congruence and reduction

The next sections cover the different components that make up our encoding. We start with type definitions and defining the notion of duality of session types (§3.2). We then encode messages and processes (§3.4), for which we define a linearity predicate (§3.7) to ensure that they are well-typed. Finally, we define structural congruence (§3.5) and reduction (§3.6), and we then prove that reduction preserves linearity, and hence well-typedness.

3.2 Types

We start by defining `MType` (the type of messages) and `SType` (the types of channels) by mutual induction, as messages can contain channels and channels can send and receive messages. This definition encodes type type grammar in Figure 2.6. We admit every Coq `Type` as a base type: anything can be sent over a channel. We use vectors for branching and selection so that we can keep track in the types of the number of options available.

```
Inductive MType : Type :=
| Base : Type → MType
| Channel : SType → MType

with SType : Type :=
| ∅ : SType
| Send : MType → SType → SType
```

```
| Receive : MType → SType → SType
| Branch: ∀ {n}, Vector.t SType n → SType
| Select : ∀ {n}, Vector.t SType n → SType
.
```

Coq does not directly admit misfix notation like Agda does, and so we need to introduce convenient notation separately:

```
Notation "B[ s ]" := (Base s).
Notation "C[ s ]" := (Channel s).
Notation "! m ; s" := (Send m s) (at level 90, right associativity).
Notation "? m ; s" := (Receive m s) (at level 90, right associativity).
Notation "&{ ss }" := (Branch ss) (at level 5, right associativity).
Notation "⊕{ ss }" := (Select ss) (at level 5, right associativity).
```

We then introduce the notion of duality of session types, modelling the definition in Figure 2.7 through an inductive proposition. The constructor suffixes `Right` and `Left` signify the direction of information flow. For branching and selection, we require both option vectors to be of the same length, and for every ith selection option and every ith branching option to be dual — `Forall2 Duality xs ys` encodes evidence that that is the case.

```
Inductive Duality : SType → SType → ℙ :=
| Ends : Duality ∅ ∅
| MRight : ∀ {m x y}, Duality x y → Duality (! m; x) (? m; y)
| MLeft : ∀ {m x y}, Duality x y → Duality (? m; x) (! m; y)
| SRight : ∀ {n} {xs ys : Vector.t SType n},
    Forall2 Duality xs ys → Duality ⊕{xs} &{ys}
| SLeft : ∀ {n} {xs ys : Vector.t SType n},
    Forall2 Duality xs ys → Duality &{xs} ⊕{ys}
.
```

Finally, we prove that duality is commutative by providing a recursive program of the type `duality_comm {s r : SType} (d : Duality s r) : Duality r s.`

3.3 Polymorphic messages

Programs that depend on unconstrained types are said to be parametrically polymorphic. Terms inhabiting those types cannot possibly be constructed or eliminated by pattern matching, and are therefore said to be *opaque*. In a purely functional language like Coq, programs of an unconstrained polymorphic type give rise to important theorems that stem purely from the polymorphism of the type [Wad89].

In our project, we parametrise several inductive definitions with polymorphic types. We do so by using the `Section Processes.` directive, instructing Coq to parametrise with parameters `ST` and `MT` all the definitions within the section. These variables are then only used by the inductive type `Message`, which is in turn used by other several definitions within the section. The type `Message` captures the values sent between processes, and is indexed by the type of the value being sent. The two constructors `V` and `C` construct values of a given base type and channels of a given session type, respectively.

```
Section Processes.
  Variable ST : Type.
  Variable MT : Type → Type.

  Inductive Message : MType → Type :=
  | V : ∀ {M : Type}, MT M → Message B[M]
  | C : ∀ {S : SType}, ST → Message C[S]
  .
```

The linearity predicate introduced in §3.7 is defined recursively on processes. Processes (§3.4) contain functions that take messages as arguments and return processes; to be able to traverse processes where message passing is modelled as function abstraction, one has to be able to provide messages of any given type:

Messages containing channels The `C` constructor expects a token of the undefined type `ST`. Channels cannot be constructed as long as the type `ST` is made concrete. This prevents channels from being built outside of the operations provided by the calculus. To traverse a process, it is enough to set `ST` to the unit type, or any other type that can be constructed.

Messages containing base types The `V` constructor expects a value of the undefined type `MT M`: `M` represents the type of the value sent as a message; `MT` serves as a projection function from types to types. If `MT` is set to `id`, then a value of type `M` must be provided; if `MT` is set to the projection `λ t ⇒ unit`, then a value of the unit type will be demanded instead. This makes messages always constructable, and therefore processes always traversable.

The users define processes without knowledge of the concrete instantiation of `ST` and `MT`. As a result, they have no information about the type `MT M` required to build messages of type `B[M]`. To remedy this, `PProcess` is provided the assumption `mt`: for any type `S`, no matter what `MT` is, a message of base type `S` can be constructed. Processes can use this assumption to create messages out of Coq terms. For convenience, we introduce a shortcut that ignores `ST` and `MT`:

```
Definition PProcess :=
  ∀ ST MT (mf : ∀ (S: Type), S → Message ST MT B[S]),
    Process ST MT.
```

3.4 Processes

We encode processes into a higher order abstract syntax where messages are parametric. Higher order abstract syntaxes use binders in the host language to encode binding in the object language [Chl08]. Here, we use it both to bind incoming messages, and to bind channels generated as a result of continuation passing. The constructors of the `Process` type are in one to one correspondence with typing rules for session types found in Figure 2.8. In this section we will argue that, **provided all channels are used exactly once, these constructors allow only correct processes to be built**. The assumption that all channels are used exactly one is later discharged on the linearity predicate defined on processes (§3.7).

We start defining process termination (T-INACT), which requires exactly one channel to be in context, and for that channel to have the session type `End`. The constructor `PEnd` requires a single

argument: a channel with session type `End`. The linearity predicate guarantees that all other channels in Coq's context are used.

```
Inductive Process : Type :=
| PEnd : Message C[∅] → Process
```

Scope restriction (T-RES) adds two channels of dual types to the context. The constructor `PNew` expects two session types, a proof of their duality, and a function that uses channels of those two session types to type a process. The linearity predicate needs to ensure that these channels are used.

```
| PNew
  : ∀ (T dT : SType)
  , Duality T dT
  → (Message C[T] → Message C[dT] → Process)
  → Process
```

Parallel composition (T-PAR) expects two subprocesses. The linearity predicate needs to ensure that the context is split between these two processes.

```
| PComp : Process → Process → Process
```

Input (T-IN) requires a channel of type $?T.S$, and a process where the continuation channel x of type S and the input y of type T are put in context. The `PInput` constructor uses function abstraction to model the addition of x and y to the context. The linearity predicate needs to ensure that context is split between these two premises, and that the consumed channel cannot be used again.

```
| PInput
  : ∀ {T : MType} {S : SType}
  , (Message T → Message C[S] → Process)
  → Message C[? T; S]
  → Process
```

Similarly, output (T-OUT) requires a channel of type $!T.S$, a message of type T, and a process where the continuation channel x of type S is put in context. The `POutput` constructor uses function abstraction to model the addition of x to the context. The linearity predicate needs to ensure that context is split between these three premises, and that the consumed channel cannot be used again.

```
| POutput
  : ∀ {T : MType} {S : SType}
  , Message T
  → (Message C[S] → Process)
  → Message C[! T; S]
  → Process
```

Branching (T-BRANCH) requires a channel of type $\&\{l_i : S_i\}_{i \in I}$, and for each continuation channel $x : S_i$, a process where x is put in context. The `PBranch` constructor requires a channel capable of branching on a vector of session types. For each of those session types, a process where a channel of the given session type is put in context is demanded — again, this is modelled through function abstraction. The linearity predicate needs to ensure that context is split between the two premises, and that the consumed channel cannot be used again.

```
| PBranch
  : ∀ {n : ℕ} {S : Vector.t SType n}
  , Forall (λ Si ⇒ Message C[Si] → Process) S
```

```
    → Message C[&{S}]
    → Process
```

Selection (T-SELECT) requires a channel of type $Select\{l_i : S_i\}_{i \in I}$, a choice j, and a process where the continuation channel $x : S_j$ is put in context. The `PSelect` constructor uses finite sets to ensure that the choice is a valid one, and function abstraction to model the addition of a channel of the chosen session type to the context. The linearity predicate needs to ensure that context is split between the two premises, and that the consumed channel cannot be used again.

```
  | PSelect
    : ∀ {n : ℕ} {S : Vector.t SType n} (j : Fin.t n)
    , Message C[⊕{S}]
   → (Message C[S[@j]] → Process)
   → Process
```

To make things easier for the end user, we provide convenient notations for process constructors. Refer to §3.9 to see example processes written using this notations.

```
Notation "'(new' s ← S , r ← R , SdR ) p" :=
  (PNew S R SdR (λ s r ⇒ p))(at level 90).
Notation "P <|> Q" := (PComp P Q)(at level 80).
Notation "![ m ]; p" := (POutput m p)(at level 80).
Notation "c ![ m ]; p" := (POutput m p c)(at level 79).
Notation "?[ m ]; p" := (PInput (λ m ⇒ p))(at level 80).
Notation "c ?[ m ]; p" := (PInput (λ m ⇒ p) c)(at level 79).
Notation "◁ i ; p" := (λ c ⇒ PSelect i c p)(at level 80).
Notation "c ◁ i ; p" := (PSelect i c p)(at level 79).
Notation "▷{ x ; .. ; y }" :=
  (PBranch (Forall_cons _ x .. (Forall_cons _ y (Forall_nil _)) ..))
  (at level 80).
Notation "c ▷{ x ; .. ; y }" :=
  (PBranch (Forall_cons _ x .. (Forall_cons _ y (Forall_nil _)) ..) c)
  (at level 79).
Definition ϵ {ST : Type} {MT: Type → Type} := @PEnd ST MT.
```

3.5 Structural congruence

3.6 Reduction

We use an inductive family of types to describe the reduction rules (listed in Figure 2.3, extended with session types in §2.2) of session-typed π-calculus processes. The data type is indexed by two processes: the former is the redex, the latter the reduction target.

```
  Reserved Notation "P ⇒ Q" (at level 60).
  Inductive Reduction : Process → Process → ℙ :=
```

The `RComm` constructor expects two channel endpoints created as a result of scope restriction to be used by two parallel processes: one that outputs `m` and continues as `Q`, the other that expects input in `P`. The reduction creates a new scope restriction, and puts in parallel a process with the continuation

Q and the process P fed with the input m. Note that this encoding works thanks to Coq's ability to pattern match on the body of functions.

```
| RComm {mt s r sDr P Q} {m : Message mt} :
    PNew (! mt; s) (? mt; r) (MRight sDr)
         (λ a b ⇒ PComp (POutput m Q a) (PInput P b)) ⇒
    PNew s r sDr
         (λ a b ⇒ PComp (Q a) (P m b))
```

Similarly to RComm, the RCase constructor transmits choice from one process to the other. Note, however, that thanks to the typing rules for the constructors PBranch and PSelect, the vectors Ps and Qs are guaranteed to be of the same length, and the choice i is guaranteed to be a valid index within the vectors. We therefore just need to select the ith vector of Qs as the continuation process after reduction.

```
| RCase {n mt} {i : Fin.t n} {ss rs : Vector.t SType n}
       {sDr} {Ps Qs} {m : Message mt} :
    PNew (Select ss) (Branch rs) (SRight sDr)
         (λ a b ⇒ PComp (PSelect i a Ps) (PBranch Qs b)) ⇒
    PNew ss[@i] rs[@i] (nthForall2 sDr i)
         (λ a b ⇒ PComp (Ps a) (nthForall Qs i b))
```

The last three constructors are inductively defined, and serve to encode that reduction goes under scope restriction and parallel composition, and that reduction is defined up to structural congruence.

```
| RRes {s r P Q} :
    (∀ (a : Message C[s]) (b : Message C[r]), P a b ⇒ Q a b) →
    (∀ (sDr : Duality s r), PNew s r sDr P ⇒ PNew s r sDr Q)

| RPar {P Q R} :
    P ⇒ Q → PComp P R ⇒ PComp Q R

| RStruct {P Q R} :
    P ≡ Q → Q ⇒ R → P ⇒ R

where "P ⇒ Q" := (Reduction P Q)
.
```

Lastly, we define an inductive type family that encodes big step reduction (reduction in zero or more steps):

```
Reserved Notation "P ⇒* Q" (at level 60).
Inductive RTReduction : Process → Process → ℙ :=
| RRefl P : P ⇒* P
| RStep P Q R : P ⇒ Q → Q ⇒* R → P ⇒* R
where "P ⇒* Q" := (RTReduction P Q)
.
```

reduction tactic

3.7 Linearity

The process constructors introduced in §3.4 rely on a linearity predicate that makes sure that every created channel is used exactly once. We define this proposition by induction on processes, in

particular processes where the type for channels (ST) is a boolean, and the function on message types (MT) is a projection to the unit type. The idea is to mark each newly created channel in turn (by setting it to true), and to then check that exactly one marked channel is found in the subprocesses.

```
Definition TMT : Type → Type := λ _ ⇒ unit.
Definition fMT : ∀ (S: Type), S → Message 𝔹 TMT B[S] := λ _ _ ⇒ V tt.
```

The predicate lin is defined by mutual recursion with single_x. Both predicates use the Equations package to dependently pattern match on channel types.

- lin (P: Process 𝔹 TMT) : ℙ recursively checks that if all newly created channels are unmarked, no marked channels are found. For every newly created channel, it also checks that marking the channel while leaving the rest unmarked results in single_x finding a single marked channel.
- single_x (P : Process 𝔹 TMT) : ℙ recursively checks that if newly created channels are unmarked, a single marked channel is found. Once that channel is found, it uses lin to check that continuing down recursively finds no marked channels. Worth nothing is the case of parallel composition, where single_x checks that (lin P ∧ single_x Q) ∨ (single_x P ∧ lin Q), that is, the marked channel must be in either P or Q, but not in both.

linearity tactic

3.8 Subject reduction

As shown in §3.4, provided that channels are used linearly, types the process constructors are accurate enough to rule out ill-typed processes. Here we show that well-typedness is preserved through reduction by showing that linearity is preserved through reduction. The main theorems to prove are therefore the following:

```
Theorem subject_reduction P Q : P ⇒ Q → Linear P → Linear Q.

Theorem big_step_subject_reduction P Q : P ⇒* Q → Linear P → Linear Q.
```

The proofs are by induction on processes. As linearity is defined on processes where channels are represented as booleans and message types are projected to the unit type (§3.7), proofs showing that linearity is carried through reduction need to specialise processes. To make the induction hypothesis stronger, we prove both that linearity is preserved, and that the presence of a marked channel is preserved.

```
Lemma reduction_linear {P Q} : Reduction _ _ P Q →
  (single_x P → single_x Q) ∧ (lin P → lin Q).
```

Reduction has two cases that need special attention: case selection (RCASE) needs to show that if an entire set of branches is considered to be linear, then so is any specific branch in the set; structural congruence (RSTRUCT) needs to show that congruence preserves linearity too. Again, we strengthen the induction hypotheses for these lemmas with assertions of single marked channel preservation.

```
Lemma congruence_linear {P Q} : Congruence _ _ P Q →
  (single_x P → single_x Q) ∧ (lin P → lin Q).
```

```
Lemma branches_linear {n} (i : Fin.t n) {xs : Vector.t SType n}
      {Ps : Forall (λ s ⇒ Message _ _ C[s] → Process _ _) xs} :
  (single_x (PBranch Ps (C false)) → single_x (nthForall Ps i (C false))) ∧
  (lin (PBranch Ps (C false)) → single_x (nthForall Ps i (C true))) ∧
  (lin (PBranch Ps (C false)) → lin (nthForall Ps i (C false))).
```

3.9 Examples

Provide better example code, the things we do do not use the theorems

Type inference

do not dedicate a section, move to subsections

4. Results

5. Alternative approaches and related work

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 [de 72], locally nameless De Bruijn indices, or a parametric HOAS [Chl08] — since only channels need to be used linearly, message input can still be represented as function abstraction whenever the message does not contain a channel. On the bright side, this approach allows processes to use logic that external to the calculus and depends on the types of messages.

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

Using de Bruijn indices

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

Linearity is strongly connected to session types: a session type must transition through each of its stages *exactly* once. Session types can be encoded into a π-calculus with linear types, as shown by [**?**, DGS12, 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* [Dyb94] 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 π-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 π-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/.

[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.

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

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

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

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

[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.

[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.