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

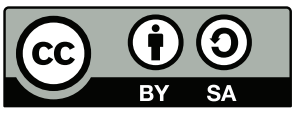
This project formalises the session-typed π-calculus in the proof assistant Coq using a mix of continuation passing, parametric HOAS, dependent types and ad-hoc linearity checks. Each action a process takes requires a channel capable of that action. The action strips off the head of that channel's type and passes its continuation to the next action taken by the process. 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 variable references are lifted to Coq, no typing contexts are required. 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 recursively traversing processes. Ultimately, the claim is this: if the definition of a process typechecks in Coq, and the process uses channels linearly, then well-typedness through reduction holds.

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

The source code for this project can be found at: `https://github.com/umazalakain/session-types-coq/`.



Acknowledgements

This thesis has been months in the making. All the way though, but particularly during the early days, the care and guidance that my supervisor Ornela Dardha has offered has played a crucial role in this project. Everything I learned about the π-calculus and session types I learned through her.

Although limited, my ability to do dependently typed programming and to think of propositions as types and of proofs as programs is entirely due to Conor McBride and the amazing people at Strathclyde (especially Fredrik Nordvall and Bob Atkey). Indirectly, others have inspired me as well, especially Wen Kokke and James Wood.

The nice people from the #coq, #agda and #dependent channels on Freenode are to be thanked for answering many of my (often silly) questions. Special thanks to Paolo Giarrusso. I also want to thank the developers of the Equations package for Coq: it has avoided me many dependently typed headaches.

Lastly, and perhaps most importantly, I want to thank the people close to me: my girlfriends Mia and Tilde, my friends, and my family. They are the ones that have provided the care and affection that has kept me going.

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 [Rey02] and the proof assistant Coq [CP90, 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 [CP90, Coq] to machine-verify the session-typed π-calculus, mainly due to its widespread use as a proof assistant (refer to §2.5 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 in the use of channels in the object language.

We use a parametric higher order abstract syntax [Chl08] (§3.2) 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 a recursive predicate on processes (§3.6). Dependent types ensure that processes (§3.3) are correct by construction up to linearity: processes satisfying the linearity predicate are guaranteed to use session-typed channels according to their specification (§3.7).

We provide basic background on the π-calculus, session types, linearly typed π-calculus, continuation passing, and Coq in §2. We then introduce our design in §3 and introduce alternative approaches in §4. Closing, §5 offers conclusions on what this project has achieved and suggests future work of interest.

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, both 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 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 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 an occurrence of $\mathbf{0}$ is substituted by a *hole* that can then be filled in with another 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 are 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 the variable 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 process that is choosing in parallel with the chosen continuation of the process that is offering the choice – as long as the choice itself is a valid one (R-Case). Reduction goes under both restriction (R-Res) and parallel composition (R-Par), but not under output, input, selection or branching – these constructs impose an order to 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 structures.

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

Figure 2.4: Example process in the π-calculus

2.2 Session types

Scope This section covers a subset of session types: the diadic (shared by two processes), finite (no replication nor recursion), deterministic (no indeterministic choice), and synchronous session types.

The grammar of the π-calculus allows well-formed processes with no semantic meaning (e.g. $(\boldsymbol{\nu} xy)\ \overline{x}\langle true\rangle.\mathbf{0} \mid y(u).(u+3).\mathbf{0}$). These meaningless terms can be discarded using type systems. Session types [Hon93, THK94, HVK98] provide a type system that encodes sequences of actions, each containing the type and the direction of the data exchanged. Processes must use session-typed channels according to their specified protocol. 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 2.5. 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 contain the continuation that will be chosen at runtime. 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`

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

Figure 2.5: 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 2.6. 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 2.6: 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 2.7. 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 these 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 these 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.

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

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

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

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

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

Figure 2.7: 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 those channels created by message input as well as those created 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 which 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. That is, if $P \to Q$ and $\Gamma \vdash P$, then $\Gamma \vdash Q$. This property is a consequence of linearity and the way in which the typing rules in Figure 2.7 deconstruct session types by taking their continuations apart.

2.3 Linearly typed π-calculus

In this section we present a linearly typed π-calculus in which scope restriction creates *two* opposite linear channel endpoints. This differs from the standard literature as found in [DGS17], where a single channel capable of linearly sending and then receiving is created by scope restriction. In doing so, our aim is to avoid the introduction of machinery that we will not use.

The linearly typed π-calculus adds linear channel types to the shared channel types of the standard π-calculus. These linear channel types specify the direction in which data is exchanged, and must be used exactly once. Their grammar is given in Figure 2.8. Channels of type $\ell_i T$ must be used exactly once to receive T, while channels of type $\ell_o T$ must be use exactly once to send T. The variant type $\langle l_1 T \dots l_n T \rangle$ is a disjoint union of labelled types in which order is irrelevant.

$$
\begin{array}{lr}
T ::= \ell_i T & \text{linear input} \\
\quad \ell_o T & \text{linear output} \\
\quad \langle l_1 T \dots l_n T \rangle & \text{variant type} \\
\quad \dots & \text{base type}
\end{array}
$$

Figure 2.8: Grammar for linear π-calculus types

Typing rules (shown in Figure 2.9) must handle typing contexts with care: linear channels must not be duplicated nor discarded. The context split operator $\uplus$ takes care of split the linear resources into two disjoint sets. Shared resources can however freely be copied or ignored.

$$
\frac{\Gamma, x : \ell_i T, y : \ell_o T \vdash P}{\Gamma \vdash (\boldsymbol{\nu} x : \ell_i T \; y : \ell_o T) \; P} \quad (\mathrm{T}\pi\text{-}\textsc{Res}) \qquad \frac{\Gamma_1 \vdash P \qquad \Gamma_2 \vdash Q}{\Gamma_1 \uplus \Gamma_2 \vdash P \mid Q} \quad (\mathrm{T}\pi\text{-}\textsc{Par})
$$

$$
\frac{\Gamma_1 \vdash x : \ell_i T \qquad \Gamma_2, y : T \vdash P}{\Gamma_1 \uplus \Gamma_2 \vdash x(y).P} \quad (\mathrm{T}\pi\text{-}\textsc{In})
$$

$$
\frac{\Gamma_1 \vdash x : \ell_o T \qquad \Gamma_2 \vdash v : T \qquad \Gamma_3 \vdash P}{\Gamma_1 \uplus \Gamma_2 \uplus \Gamma_3 \vdash \overline{x}\langle v \rangle.P} \quad (\mathrm{T}\pi\text{-}\textsc{Out})
$$

Figure 2.9: Some of the typing rules for linearly typed π-calculus

2.4 Encoding session types into linear π-calculus types

The encoding of the session-typed π-calculus into the linearly-typed π-calculus is introduced in [Kob03, Kob07, DGS17], where both session types and their processes are encoded into the linear π-calculus. This project encodes session types into linear π-calculus types as shown in Figure 2.10 — note that termination gets encoded as a channel with no input or output capabilities. However, it differs from the existing work in that processes do not exchange continuations, and thus we have no need to encode processes. More on this can be found in §3.5.

$$
\begin{aligned}
[\![\mathtt{End}]\!] &= \ell_{\varnothing}[] \\
[\![?T.S]\!] &= \ell_i[[\![T]\!], [\![S]\!]] \\
[\![!T.S]\!] &= \ell_o[[\![T]\!], [\![S]\!]] \\
[\![\{l_1 : S_1\&, \ldots, \&l_n : S_n\}]\!] &= \ell_i[\langle l_1 : [\![S_1]\!], \ldots, l_n : [\![S_n]\!]\rangle] \\
[\![\{l_1 : S_1\oplus, \ldots, \oplus l_n : S_n\}]\!] &= \ell_o[\langle l_1 : [\![S_1]\!], \ldots, l_n : [\![S_n]\!]\rangle]
\end{aligned}
$$

Figure 2.10: Encoding of session types into linear types

2.5 Coq proof assistant

Coq [Coq] is a popular proof assistant and dependently typed functional language based on the calculus of inductive constructions [CP90] (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` $\rightarrow$ `Zero` represents negation, that `P` is provably false. Conversely, from a proof of falsity one might conclude anything: `Zero` $\rightarrow$ `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 — even contain — programs. Since that is the case, programs in Coq must exhibit termination — recursion must occur on structurally smaller terms, otherwise type checking would diverge. The example below 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 using the induction hypothesis 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 our encoding of the session-typed π-calculus into Coq works. This encoding is limited to 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. Our design is based on *continuation passing* (§2.4), where channels are **used exactly once**: channels get destroyed and created with every action taken by a process. The main characteristic of our design is the use of a higher order abstract syntax to lift the bindings of channels and messages in the object language into bindings in the host language. As a result of this approach, we limit our encoding to closed processes — processes where all variables are bound. This, however, in no way limits expressivity: after all, free variables are only meaningful if they are at some point bound. The key parts of our design focus around solving the following issues:

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 using 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. §3.2 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 the value they contain — whether this is a session type or a base type. See §3.1 and §3.3 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 channel has to provided. The action then uses function abstraction to provide an environment in which a channel with the tail of the original session type is available 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 §3.6.

The advantages of our design over other alternative approaches (introduced in §4) are listed below. The coming sections cover the different components that make up our encoding. We start defining types and the notion of duality of session types (§3.1). We then encode messages (§3.2) and processes (§3.3), for which we define a linearity predicate (§3.6) to ensure that they are well-typed. Finally, we define structural congruence (§3.4) and reduction (§3.5), after which we prove that reduction preserves linearity, and hence well-typedness (§3.7).

Users can refer naturally to both messages and channels Our approach lifts all variable references into Coq. The example process below has two communicating parallel subprocesses: 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.
```

No typing contexts are required Using dependent types, our approach keeps track of the types of both channels and messages in the host language. A linearity predicate ensures that each channel is used exactly once. Common alternative approaches index object terms with an explicit context of channel types, and handle linearity at construction time, resulting in complex typing rules.

No need for capture avoiding machinery Techniques for representing variable references need to avoid the capture of free variables. This involves some extra machinery: it is minimal in the case of de Bruijn indices [de 72] or locally nameless representations [Cha12], considerable in the case of names with the Barendregt convention [Bar84]. This extra machinery in turn requires one to prove substitution lemmas. As a result of lifting references into the host language, in our case, no such need exists.

3.1 Types

We start by defining `MType` (the type of messages) and `SType` (the session types of channels) by mutual induction, as messages can contain channels and channels can send and receive messages. This definition encodes the grammar for types found in Figure 2.5. We admit every Coq `Type` as a base type: anything can be sent as a message. We use vectors for branching and selection so that we can keep track in their 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, 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 2.6 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 this 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.`

Examples. The examples below introduce several session types using the aforementioned notations. We then prove that `type1` and `type2` are dual, while `type1` and `type3` are not. These proofs execute specialised tactics which are found in a hint database by `auto`.

```
Example type1 : SType := ? B[𝔹] ; ? B[ℕ] ; ! C[? B[𝔹] ; ∅] ; ∅.
Example type2 : SType := ! B[𝔹] ; ! B[ℕ] ; ? C[? B[𝔹] ; ∅] ; ∅.
Example type3 : SType := ! B[𝔹] ; ! B[ℕ] ; ? C[! B[𝔹] ; ∅] ; ∅.
Example duality1 : Duality type1 type2. auto. Qed.
Example duality2 : ~ Duality type1 type3. auto. Qed.
```

3.2 Polymorphic messages

Programs that depend on unconstrained types are said to be parametrically polymorphic. Terms inhabiting those types cannot possibly be constructed nor 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 several other 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 §3.6 is defined recursively on processes. Processes (§3.3) 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 not 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` or `MT`. As a result, they have no information about the type `MT M` required to build messages of type `B[M]`. To remedy this and allow them to create messages of base types, `PProcess` is provided the assumption `mt`: for any base type `M`, no matter what `MT` is, a message of type `M` can be constructed. Processes can use this assumption to embed Coq terms in messages. 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.
Notation "[ f ]> P" := (λ _ _ f ⇒ P)(at level 80).
```

3.3 Processes

We encode processes into a higher order abstract syntax where messages are parametrised. Higher order abstract syntaxes use binders in the host language to encode binding in the object language [Chl08]. Here, we use it to bind both incoming messages and channels. The constructors of the `Process` type are in one to one correspondence with the typing rules for session-typed processes found in Figure 2.7. 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 (§3.6).

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 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 the 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 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 $\oplus\{l_i : S_i\}_{i \in I}$, 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:

```
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.
```

Examples. Below is a simple process definition that creates two channel endpoints `o` and `i`, and composes in parallel two processes that exchange a boolean value back and forth, and then end. Note the inline use of tactics to prove the duality of the channels. The function υ is used to *lift* the boolean value `true` into a message.

```
Example example1 : PProcess :=
  [υ]> (new o ← ! B[𝔹] ; ? B[𝔹] ; ∅, i ← ? B[𝔹] ; ! B[𝔹] ; ∅, ltac:(auto))
    (o![υ _ true]; ?[m]; ϵ) <|> i?[m]; ![m]; ϵ.
```

The next example showcases type inference for session types. We introduce two parallel processes in where the session types for the channels `i` and `o` are left undefined. We use the tactic `refine` to leave the proof of their duality for later, as it is yet too early to know about the types of `i` and `o`. Once the bodies of the processes force their types, we are able to provide a proof of their duality with `auto`.

```
Example example2 : PProcess.
  refine
  ([υ]> (new i ← _, o ← _, _)
    (i?[m]; ![m]; ϵ) <|> (o![υ _ true]; ?[m]; ϵ)).
  auto.
Defined.
```

When we print the process above, we can see that Coq has automatically deduced the required session types of `i` and `o`:

```
example2 =
λ (ST : Type) (MT : Type → Type) (υ : ∀ S : Type, S → Message ST MT B[ S]) ⇒
(newi ← ? B[ 𝔹]; ! B[ 𝔹]; ∅, o ← ! B[ 𝔹]; ? B[ 𝔹]; ∅,
MLeft (MRight Ends)) i ?[ m ]; (![ m ]; ϵ) <|> o ![ υ 𝔹 true ]; (?[ _ ]; ϵ)
     : PProcess
```

Finally, the example below demonstrates channel mobility (sending a channel over another channel):

```
Example channel_over_channel : PProcess :=
  [υ]>
    (new x ← ? C[! B[𝔹] ; ∅] ; ∅, y ← ! C[! B[𝔹] ; ∅] ; ∅, MLeft Ends)
    (new w ← ? B[𝔹] ; ∅, z ← _, MLeft Ends)

    (x?[c]; λ a ⇒ (ϵ a <|> c![υ _ true]; ϵ))
    <|>
    (y![z]; λ a ⇒ (ϵ a <|> w?[_]; ϵ)).
```

3.4 Structural congruence

Structural congruence is an equivalence relation, and as such it is reflexive, symmetric and transitive. In this project, we were **unable to add symmetry to structural congruence and still prove subject reduction** (see §3.7 for details). This difficulty was only noticed during the final stages of the project. We have therefore limited the relation to a **structural precongruence** relation where only reflexivity and transitivity are defined. This limitation entails the following: the equivalence rules `CCompAssoc` and `CScopeExpa` (asymmetric in their structure) can only be used to rewrite from left to right, and not the other way around. As a workaround, `Axiom CSymm : ∀ {P Q}, Congruence P Q → Congruence Q P` can be postulated.

We use an inductive family of types to describe the structural congruence rules of session-typed π-calculus processes. The data type is indexed by the two structurally equivalent processes.

```
  Reserved Notation "P ≡ Q" (no associativity, at level 80).
  Inductive Congruence : Process → Process → ℙ :=
```

Our encoding does not discard any channels, including those of type `End`: they still need a process ϵ to close them. For this reason, the equivalence rules of the π-calculus that have to do with process termination will be absent from our encoding. Note however that this in no way impacts expressivity: it is possible to decide whether a process has terminated by traversing it and making sure that scope restriction, parallel composition and termination are the only present constructs.

The `CComp` rules define commutativity and associativity for process compositition, respectively. The `CScope` rules express scope expansion, scope commutativity, and session type commutativity, respectively.

```
| CCompComm {P Q} :
    PComp P Q ≡ PComp Q P

| CCompAssoc {P Q R} :
    PComp (PComp P Q) R ≡ PComp P (PComp Q R)

| CScopeExpa {s r sDr P Q} :
    PComp (PNew s r sDr P) Q ≡ PNew s r sDr (λ a b ⇒ PComp (P a b) Q)

| CScopeComm {s r sDr p q pDq P} :
    PNew s r sDr (λ a b ⇒ PNew p q pDq (λ c d ⇒ P a b c d)) ≡
    PNew p q pDq (λ c d ⇒ PNew s r sDr (λ a b ⇒ P a b c d))

| CScopeTypesComm {s r sDr P} :
    PNew s r sDr (λ a b ⇒ P a b) ≡
    PNew r s (duality_comm sDr) (λ b a ⇒ P a b)
```

If processes P and Q are structurally congruent, then replacing them within the same context $C[]$ results in two processes $C[P]$ and $C[Q]$ that are structurally congruent as well. Contexts are defined as processes *with holes*: every possible place where $\mathbf{0}$ can appear can be replaced by a hole. This includes continuation processes coming after input, output, branching, and selection. However, in these cases congruence is not a necessity: reduction needs to operate on the outer layer first, then congruence can be used on the inner layer. Although omitting congruence relations under these constructs means that not all structurally congruent processes can have their equivalence expressed, it does in no way affect the operational semantics on which our final proof of subject reduction is based upon. We therefore only define congruence under parallel composition and scope restriction.

```
| CCompCong {P Q R S} :
    P ≡ Q → R ≡ S → PComp P R ≡ PComp Q S

| CScopeCong {s r sDr P Q} :
    (∀ a b, P a b ≡  Q a b) → PNew s r sDr P ≡ PNew s r sDr Q
```

Finally, we define reflexivity and transitivity as a separate inductive data type. This is so that transitivity can be defined asymmetrically, making automatic proof search easier.

```
Reserved Notation "P ≡* Q" (at level 60).
Inductive RTCongruence : Process → Process → ℙ :=
| CRefl P : P ≡* P
| CStep {P} Q {R} : P ≡ Q → Q ≡* R → P ≡* R
where "P ≡* Q" := (RTCongruence P Q)
```

Examples. The following example proves that `Example1` and `Example2` introduced in §3.3 are structurally congruent. The congruence proof performs breath first search at depth 10. If the automatic tactic fails, the user is still able to provide an explicit proof of structural congruence.

```
Example congruent_example1 : example1 ≡* example2. eauto 0 10. Qed.
```

3.5 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 in 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 through 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 the process `Q` and the process `P` fed with the input `m`. Note that unlike the formalisation presented in [DGS17], processes do not send over continuation channels: it is rather the reduction rule that rewrites in the processes the channels created through scope restriction. This encoding works thanks to Coq's ability to pattern match against 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 of the constructors `PBranch` and `PSelect`, the reduction rule can require the vectors `Ps` and `Qs` to be of the same length. Note also that the choice `i` is guaranteed to be a valid index within both vectors. We therefore just need to select the `i`th 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 the fact that reduction goes under scope restriction and parallel composition, and that reduction is defined up to structural congruence.

```
| RRes {s r P Q} :
    (∀ a b, 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 family of types that encodes big step reduction (reduction in zero or more steps). We do so asymmetrically, in a way that makes big step reduction proofs unique.

```
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)
.
```

Examples. The following example defines the process `Example3`, and proves that `Example2` introduced in §3.3 reduces in one step to `Example3`. The reduction proof uses breath first search of depth 10 and executes specialised tactics found in a hint database. The tactic is not guaranteed to succeed: reduction might make use of zero or more steps of structural congruence rewriting, resulting in a wide search space. If the automatic tactic fails, the user is still able to provide an explicit proof of reducibility.

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

Example reduction_example1 : example2 ⇒ example3. eauto 0 10. Qed.
```

The next example goes further and shows that `Example2` reduces to `Example5` (a final irreducible value) in zero or more steps:

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

Example reduction_example2 : example4 ⇒ example5. eauto 0 10. Qed.
```

3.6 Linearity

The process constructors introduced in §3.3 rely on a linearity predicate that makes sure that every created channel is used exactly once. We define this predicate by recursion on processes, in particular processes where the type of channels (`ST`) is boolean, and the function on message types (`MT`) is a projection to the unit type. The idea is to mark each newly created channel one by one (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.
Definition Linear (P : PProcess) : ℙ := lin (P 𝔹 TMT fMT).
```

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, then 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 still found. Once that channel is found, it uses `lin` to check that continuing down recursively finds no other marked channels. Worth nothing is the case for parallel composition, where `single_x` checks that `(lin P ∧ single_x Q) ∨ (single_x P ∧ lin Q)`, that is, that the marked channel must be in either `P` or `Q`, but not in both.

Examples. The next two examples (of example processes defined in §3.3) introduce a tactic that proves that the process uses channels linearly. This tactic is called by `auto` through its hint database.

```
Example linear_example1 : Linear example1. auto. Qed.
Example linear_channel_over_channel : Linear channel_over_channel. auto. Qed.
```

As a counterexample, we introduce a process for which we prove that its use of channels is *not* linear.

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

    (* Cheat the system by using the channel o twice *)
    i?[_]; ϵ <|> o![v _ true]; (λ _ ⇒ o![v _ true]; ϵ)
    .
Example nonlinear_example1 : ~ (Linear nonlinear_example). auto. Qed.
```

3.7 Subject reduction

As shown in §3.3, assuming that channels are used linearly, process constructors are accurate enough to rule out ill-typed processes by construction. Here, we show that well-typedness is preserved through reduction, i.e. that linearity is preserved through reduction: if P reduces to Q and P is linear (well-typed), then Q is linear (well-typed) as well:

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

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

As linearity is defined on processes where channels are represented as booleans and message types are projected to the unit type (§3.6), proofs that show that linearity is carried through reduction need to specialise their processes as well. The proofs themselves are by induction on processes. To make the induction hypothesis stronger, proofs show that both linearity and the presence of a marked channel are preserved.

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

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

```
Lemma rtcongruence_linear {P Q} : RTCongruence _ _ 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))).
```

As mentioned in §3.4, we were unable to add symmetry to structural congruence. To prove that linearity is preserved through a symmetric structural congruence, we would need to show that both implications above are in fact bidirectional:

```
Lemma rtcongruence_linear {P Q} : RTCongruence _ _ P Q →
  (single_x P ↔ single_x Q) ∧ (lin P ↔ lin Q).
```

Proving implication in the other direction has proven difficult in the case of scope expansion: it involves showing that if `PNew s r sDr (λ a b ⇒ PComp (P a b) Q` has a single marked variable, so does `PComp (PNew s r sDr P) Q`. This boils down to showing that `lin (P o o) ∧ single_x Q` implies `lin (P o o) ∧ single_x (P x o) ∧ single_x (P o x) ∧ single_x Q`. This stems from the way in which the linearity predicate is defined. We were unable to find a quick workaround.

Examples. We can apply the proof of linearity preservation through reduction to specific processes. The example below proves that any process that `example1` reduces to in zero or more steps must make a linear use of channels.

```
Example big_step_subject_reduction_example1 {P : PProcess}
  : example1 ⇒* P → Linear example1 → Linear P.
Proof.
  apply big_step_subject_reduction.
Qed.
```

4. Related Work

The π-calculus has been an extensive subject of machine verification: [HM99] proves subject reduction; [Des00] proves subject reduction as well, but uses a higher order syntax; [AK08] provides proofs of fairness and confluence; [HMS01] formalises the bisimilarity proofs found in [WMP89].

Work on the machine verification of session types is more scarce. Session types are strongly connected to linearity: a session type must transition through each of its stages *exactly* once. In type systems with no linear types, linearity has to be simulated. In these type systems, modelling channels through a parametric higher order abstract syntax [Chl08] (like we did) is not possible per se: the host language is unable to check whether the channels are used linearly. In our case, we used a predicate on processes to simulate linearity. In [Pet], an HOAS is used as well, in this case to model attestation protocols using session types, but linearity is not considered by the author.

It is usually typing contexts that are used to keep track of linear resources. The context is usually kept at the type level, using inductive type *families* [Dyb94] indexed by a context of linear resources [PW00], though there are approaches that keep track of context through monadic binding and embed linear calculi within non-linear hosts [PZ17]. In [Kok19], the author proposes to use ideas from quantitative type theory [McB16] to model processes with linear contexts.

Session types can also be encoded into an intermediate π-calculus with linear types, as shown in [DGS17]. Using Isabelle/HOL, [Gay01] formalises the linear π-calculus using de Bruijn indices [de 72] for references. Using Coq, [Dil19] encodes session types into a linear π-calculus with de Bruijn references, provides some of the preliminary lemmas required for a proof of subject reduction, and hints towards the use of a parametric higher order abstract syntax.

Cutting out this intermediary formalisation into the linear π-calculus, and using explicit contexts and a locally-nameless [Cha12] representation, [GJJ$^+$16] formalises polymorphic session types in Coq, but their work is considerably complex. Using a locally-nameless representation as well, [Fer] formalises and proves subject reduction for session types in Coq in a considerably simpler way.

In contrast with type systems with no linear types, type systems with linear types allow linearity constraints to be encoded into the host language. In [XRWB16] session types are formalised in ATS, providing type preservation and global progress proofs. [BBMS16] uses Celf to represent session types in intuitionistic linear logic.

5. Conclusion and Future Work

We have encoded a subset of the session-typed π-calculus into the proof assistant Coq. We have used dependently typed channels to model continuation passing. We have used polymorphism to prevent the user from forging channels. We have lifted variable references in the object language into variable references in Coq. We have simulated linearity (which Coq lacks) in the use of channels through a recursive predicate on processes. We have proven that the operational semantics respects such linearity predicate, and therefore the well-typedness of processes.

The present work can be expanded in several different directions. Currently, the subject reduction theorem is only proven for reduction using structural precongruence. A first addition would be to extend this proof and include symmetry in the structural congruence relation. This would likely imply changes to the current linearity predicate.

The type system can be extended in several directions as well. Shared channel types from the standard π-calculus can be added to the language. These channels are non-linear and do not evolve as communication occurs. They could probably be easily added through extra process constructors, that could then be hidden through appropriate notation. The linearity predicate would ignore these shared channels, which would have no effect on the subject reduction proof neither, as they would be guaranteed to be correct by construction.

Without explicit environments, adding process replication seems more challenging: the types of the channels created outside of a replicating process but used inside of it, need to have the replicating capability as well. Without explicit environments, making sure that this typing obligation is fulfilled requires an approach similar to our linearity predicate. At this point, it would be probably best to switch to a representation with explicit typing contexts.

Slightly branching off from this project, we can also formalise the operational semantics presented in §3.5, where we work with session types encoded as linear π-calculus types, but avoid the need of sending over continuation channels. We do so by modifying the operational semantics of the linear π-calculus instead, giving it the knowledge about how to manage session types encoded as linear types with heads and tails.

As part of a wider effort, I intend to continue working on the machine verification of process calculi during my oncoming PhD. Some of the aspects I plan to work on are: encoding session types into more primitive types in simpler calculi; formalising and machine verifying the properties of the correspondences between session types and linear logic; representing the more advanced extensions to session types; and translating these representations into other proof assistants.

6. Bibliography

[AK08] Reynald Affeldt and Naoki Kobayashi. A Coq Library for Verification of Concurrent Programs. *Electronic Notes in Theoretical Computer Science*, 199:17–32, February 2008. http://www.sciencedirect.com/science/article/pii/S1571066108000765.

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

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

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

[Cha12] Arthur Charguéraud. The Locally Nameless Representation. *Journal of Automated Reasoning*, 49(3):363–408, October 2012. http://link.springer.com/10.1007/s10817-011-9225-2.

[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/.

[CP90] 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, 1990.

[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.

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

[DGS17] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. *Inf. Comput.*, 256:253–286, 2017. https://doi.org/10.1016/j.ic.2017.06.002.

[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.

[Fer] Francisco Ferreira. Adventures in Formalising the Meta-Theory of Session Types. page 27.

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

[GJJ+16] Matthew Goto, Radha Jagadeesan, Alan Jeffrey, Corin Pitcher, and James Riely. An extensible approach to session polymorphism. *Mathematical Structures in Computer Science*, 26(3):465–509, March 2016. https://www.cambridge.org/core/product/identifier/S0960129514000231/type/journal_article.

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

[HMS01] Furio Honsell, Marino Miculan, and Ivan Scagnetto. π-calculus in (Co)inductive-type theory. *Theoretical Computer Science*, 253(2):239–285, February 2001. http://www.sciencedirect.com/science/article/pii/S0304397500000955.

[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. http://link.springer.com/10.1007/BFb0053567.

[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.

[Kob03] Naoki Kobayashi. Type Systems for Concurrent Programs. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Bernhard K. Aichernig, and Tom Maibaum, editors, *Formal Methods at the Crossroads. From Panacea to Foundational Support*, volume 2757, pages 439–453. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. http://link.springer.com/10.1007/978-3-540-40007-3_26.

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

[Kok19] Wen Kokke. Formalising Session Types With Fewer Worries. https://www.um.edu.mt/projects/behapi/wp-content/uploads/2019/04/Wen-Kokke-Formalising-Session-Typed-Languages-Without-Worries.pdf, 2019.

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

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., January 1989. http://dl.acm.org/citation.cfm?id=534666.

[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. http://dl.acm.org/citation.cfm?id=114891.114910.

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

[Pet] Adam Petz. A Semantics for Attestation Protocols using Session Types in Coq. page 61.

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

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

[Rey02] J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Copenhagen, Denmark, 2002. IEEE Comput. Soc. http://ieeexplore.ieee.org/document/1029817/.

[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.

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