Mechanising the Meta-Theory of Session Types in Rocq: a Tutorial

Marco Carbone[0000−0001−9479−2632] and Alberto Momigliano[0000−0003−0942−4777]

Abstract This tutorial presents a mechanization of the meta-theory of binary session types in the Rocq proof assistant, with the goal of making the formalization of concurrent calculi more accessible to researchers and practitioners. We focus on two central difficulties: the treatment of names and binders inherited from the π-calculus, and the enforcement of a linear type discipline. To address the latter, we adopt the linearity predicate, which separates typing from resource management and integrates smoothly with de Bruijn indices, which is the way we handle name and binders. The tutorial is aimed at readers familiar with session types and Rocq, and is intended as a guided companion to the second chapter of Gay & Vasconcelos' recent book, highlighting the design choices and proof principles that arise when carrying out machine-checked developments of concurrency theory. The Rocq code is available in full, and the approach should carry over to other mainstream assistants.

1 Introduction

While the use of proof assistants to verify programming language theory has a well-established history [7], the mechanization of formal models for concurrent and distributed languages is less developed. Concurrent calculi are particularly subtle; for instance, it took several years before an error in the type preservation proof of the original session subtyping paper [4] was discovered and subsequently corrected [5], underscoring the importance of machine-checked proofs. Although some results in concurrent formalisms have already been mechanized, we believe that the process remains unnecessarily difficult to approach. This is precisely the motivation behind the "concurrent

Marco Carbone
IT University of Copenhagen, Copenhagen, Denmark e-mail: maca@itu.dk

Alberto Momigliano
DI, Università degli Studi di Milano, Milano, Italy e-mail: momigliano@di.unimi.it

calculi formalization benchmark" [2], which aims to promote the same spirit of collaboration and progress inspired by the original POPLMark challenge.

A particularly illustrative case study—also the focus of [2]'s first benchmark—is the meta-theory of binary session types [6], specifically their soundness with respect to standard reduction semantics. This area presents two interrelated challenges:

1. the formalization of free and bound names and their interaction under communication, inherited from the π-calculus—a longstanding challenge in mechanization [9];
2. the enforcement of a *linear* type discipline, which introduces non-trivial constraints on context management; we discuss its specific difficulties in Section 5.

To address the second aspect, we adopt the recently rediscovered *linearity* predicate [13] (LP for short), which cleanly separates typing rules from the management of linear resources. A crucial side effect is that LP avoids interfering with the traditional encoding of binders in mainstream proof assistants—specifically, the use of de Bruijn indices.

The main goal of this tutorial is to provide an understanding of the formal machinery involved in mechanizing the meta-theory of binary session types within a Rocq-like proof assistant. We concentrate on fundamental issues such as name management and the typing discipline, with a particular emphasis on the interplay between typing derivations and linearity assumptions. Rather than offering a general-purpose proof framework, this tutorial aims to clarify the design choices and theoretical principles underlying the implementation. While we do not claim significant novelty relative to the state of the art, we do introduce a few technical devices—such as injective renamings and the non-monotonicity of typing contexts—that have not previously been explored in this context.

Target Audience: This tutorial is intended for readers already familiar with:

- the motivation and mathematical development of binary session types,
- basic experience with Rocq, roughly at the level of the second volume of Software Foundations [11],
- a working understanding of de Bruijn indices, as introduced e.g. in [10].

We do not assume prior knowledge of advanced techniques for managing binders or of linearity in mechanised proofs. While our formalization mostly uses *Ssreflect* [8], this is purely for convenience; the development could be carried out in plain Rocq. We have not attempted to optimise the proofs—either through automation or through custom tactics. The material should be accessible to users of other proof assistants, with one caveat: our adoption of well-scoped de Bruijn indices relies on dependent types.

As far as session types are concerned, our treatment follows the recent book by Gay and Vasconcelos [6], to which we refer for both the underlying motivations and the technical details that we take as known. As we shall see, somewhat to our surprise, the mechanization process has led us to introduce slight modifications to the system presented in the book—changes that, in our view, improve it. Such refinements are a welcome outcome of formalization efforts. We leave to future work the rest of the book, in particular handling of unrestricted channels, recursion and subtyping and possible future surprises included.

The Rocq code discussed in this paper can be found in its entirety at this address:

`https://github.com/carbonem/rocq-session-types-tutorial`

In the next sections, the reader will find code snippets that are taken verbatim from the repository and can also click on the icon (🐓) to directly reach the encoding of selected definitions or statements.

2 Syntax

The syntax of our (finitary) process calculus is given by the following grammar:

$$P, Q \quad ::= \quad \mathbf{0} \mid x!.P \mid x?.P \mid x!y.P \mid x?(y).P \mid P \parallel Q \mid (\boldsymbol{\nu} xy)\ P$$

We assume a countable set of variables (also called names) for channel endpoints, denoted by x, y, etc. Our syntax includes standard process constructs such as parallel composition, output, and input. As in Gay and Vasconcelos [6], we have separate processes for closing a channel (wait and close). Following Vasconcelos's idea [15], channel restriction $(\boldsymbol{\nu} xy)\ P$ connects two endpoints to form a communication channel. Input and restriction are *binders*: in particular y is bound in P in $x?(y).P$ while x and y are bound in P in $(\boldsymbol{\nu} xy)\ P$, yielding the usual definition of free and bound variable.

2.1 Representation in Rocq

Names, Binders, and Processes. To implement this syntax in a proof assistant such as Rocq, we must make a fundamental design decision: how to represent names and binders. In this tutorial, we adopt (well-scoped) De Bruijn indices. De Bruijn indices replace variable names with numeric indices that count the number of binders between a variable occurrence and its binder. This avoids issues related to variable shadowing and α-conversion, at the cost of having to update these indices when communication occurs.

For example, consider the process $x?(y).\ y!.P$, where name y is bound in $y!.P$. In the de Bruijn representation, the term would be encoded as $x?().\ 0!.P'$, where y has become the index 0, referring to the most recently bound variable and P' is the encoding of P. Any other occurrences of variables in P' that are bound outside this scope must be incremented (shifted) to account for the new binding. This explicit shifting ensures that each index correctly refers to its intended binder, preserving the intended scoping of all variables.

This representation trades one difficulty for another: substitution reduces to replacing an index, but one must carefully manage shifts whenever new binders are introduced or terms are moved under additional binders.

The idea of *well-scoped* de Bruijn syntax is to associate each syntactic object with the size of its surrounding context. Concretely, the type of processes, written $proc^k$, carries an *upper bound* k on the admissible free names. This principle extends to richer

signatures, yielding an algebra of terms and substitutions indexed by scope. In such a representation, Rocq's type checker automatically rejects forgotten "shifts", since each constructor explicitly specifies how the scope changes. Closed processes, for instance, inhabit the type $proc^0$.

We have used the `Autosubst2` library to automatically generate from a signature of our calculus boilerplate code for de Bruijn syntax, including renaming and substitution. While this tool is convenient, its use is not essential for understanding the key ideas presented here. Below, we highlight the core concepts needed for the remainder of the tutorial, independently of `Autosubst2`.

In a well-scoped representation, processes have this BNF:

$$c \in \mathsf{ch}^k \quad ::= x \quad (x \in \texttt{fin k})$$
$$P, Q \in \mathsf{proc}^k ::= P^k \parallel Q^k \mid \nu.P^{k+2} \mid x^k\,?(\_).P^{k+1} \mid x^k!y^k.P^k \mid x^k!.P^k \mid x^k?.P^k \mid \mathbf{0}$$

where `fin` is the finite type of indices with at most k elements (🐓). For example, `fin 3` has three elements: `0`, `1`, and `2`. In `Autosubst2`, it is built as the n-fold iteration of the `Option` type, although other representation as inductive definitions or dependent sums are also possible. Some elements are:

```
Definition var_zero {n : nat} : fin n.+1 := None.
Definition var_one {n : nat} : fin n.+2 := Some None.
```

The typical use of `var_zero` is as a freshly bound variable.

Well-scopedness is enforced through dependent types: a process of type proc^k is guaranteed to only reference endpoints in the range `fin k`, thus preventing dangling references by construction.

The syntax of processes and channels is defined inductively via two key types:

- `ch n` (🐓): endpoints, represented using de Bruijn indices with scope `n`: these are just coercions `var_ch` from elements of the `fin` type;[1]
- `proc n` (🐓): processes, similarly well-scoped.

In the following we will heavily use Rocq's **`Notation`** mechanism to pretty print most of the informal judgments, some of those notations being auto-generated by `Autosubst2`.

Substitutions The (simultaneous) substitution operation is central to reasoning about process behavior. In our setting, a substitution on endpoints is a function of type `fin n -> ch m`. This means that for each index in the context, the substitution provides a replacement endpoint, possibly with a different scope. We can think of it as a stream of endpoints with length `n`.

We identify three elementary operations:

identity	`idren: fin k -> fin k` (🐓)
shifting	`shift: fin k -> fin k.+1` (🐓)
extension	`scons: X -> (fin n -> X) -> fin n.+1 -> X` (🐓)

[1] These coercions are introduced by `Autosubst2`, but could be omitted by working directly with elements of `fin`, since our calculus has only a single sort of variables.

In the extension operation we will instantiate `x` as `ch k`: hence we "cons" a new element of type `ch k` to the stream of type `fin n -> ch k` at its first position `var_zero`.

Process substitution, denoted by $\langle\sigma\rangle P$, is directly defined by structural recursion. This is notably different from the λ-calculus where the definition has to go through renaming to be accepted as total [1]. We first need to define a substitution function for endpoints, via `subst_ch : (fin m -> ch n) -> ch m -> ch m` (notation $x[\sigma]$), which simply applies the function under the coercion (🐓).

A typical use of such substitution over endpoints is when descending into a binder. Suppose we have a judgment `J` over an endpoint and process. Then, the restriction case will have this shape:

```
Inductive J {n : nat} : ch n → proc n → Prop :=
| JRes : forall (x : ch n) (P : proc n.+2),
  J (x[fun i ⇒ var_ch (shift (shift i))]) P → J x ((ν) P)
  ...
```

Above, the validity of the predicate depends on the subterm `P`, but we must shift `x` by two because of the binder in the initial term.

Finally, process substitution is defined as follows in our mechanization (🐓):

```
Fixpoint subst_proc {m : nat} {n : nat}
(σ : fin m -> ch n) (p : proc m) : proc n :=
  match p with
  | ∅ => ∅
  | p0 ? . p1 =>  p0 [σ] ? . subst_proc m n σ p1
  | p0 ! . p1 =>  p0 [σ] ! . subst_proc m n σ p1
  | (ν) p0 => (ν) subst_proc m.+2 n.+2 ↑(__ch ↑(__ch σ)) p0
  | p0 ‖ p1 => subst_proc m n σ p0 ‖ subst_proc m n σ p1
  | p0 ? (_).p1 => p0 [σ] ? (_).subst_proc m.+1 n.+1 ↑(__ch σ) p1
  | p0 ! p1 . s2 => p0 [σ] ! p1 [σ] . subst_proc m n σ s2
  end
```

Listing 1 Processes substitution

where `↑__ch` is notation for a function

```
up_ch_ch: (fin m → ch n) → fin n.+1 → ch n.+1
```

that shifts a substitution (🐓).

3 Operational Semantics

The dynamics of processes are described through a small step operational semantics. We base our mechanization on the one given by Gay and Vasconcelos [6], which we report in Fig. 1. As usual, this definition uses a *structural congruence* relation that equates processes deemed to be indistinguishable. Gay and Vasconcelos define structural congruence as the smallest congruence relation that satisfies the axioms in the top part of Fig. 1. Then, reduction is defined in the bottom part.

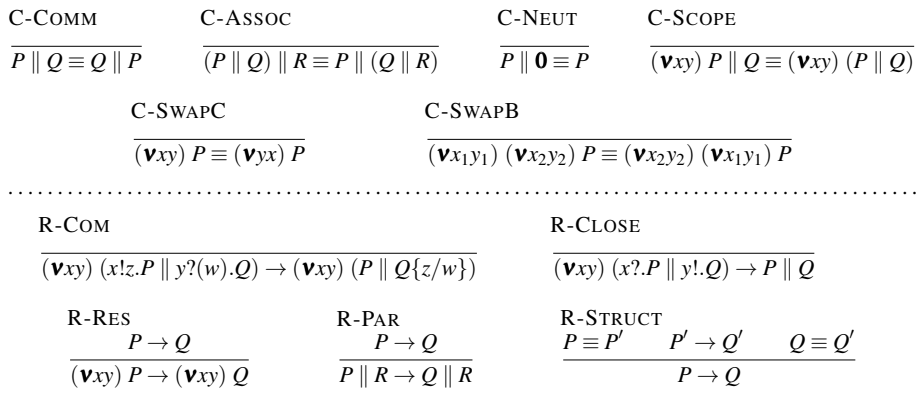

Fig. 1 Structural Congruence and Reduction [6]

While any mathematician would understand what we mean by "the smallest congruence relation closed under given rules" — as indeed defined in [15] — this is too hand-wavy for a mechanization. One way to make it more formal is to introduce the notion of "process contexts", as in [12]: these are defined as in the PL literature since Felleisen and Hieb's work from the BNF of processed adding a "hole"; this hole, when filled, specifies the location where a particular congruence rule will be applied. This approach has its merits, as it separates the action itself from the position where the action takes place. It is also modular, since extending the language will only require to extend the syntax of processes and contexts, rather than adding more rules to the notion of equivalence. However, it is rather complicated to implement: not only it duplicates the syntax of processes and all its related notions, but contexts have their own peculiar substitution operation that does not respect α-conversion: free names can become bound when filling a hole.

We therefore recommend formalizing closure compatibility within structural congruence by explicitly adding the equivalence relation conditions alongside a compatibility condition for each construct, as suggested in [6]. This makes some proofs longer (e.g., type preservation of congruence), but these extra cases are not the difficult ones. A caveat is that, when inducting on such a judgment, certain theorems must be stated as "iff" so that the inductive hypothesis applies symmetrically — for an example, see Lemma 4.2. Finally, note that the de Bruijn discipline resolves ambiguities arising from the Barendregt convention, which was previously used to omit some conditions on bound names in the congruence rule, e.g., C-SCOPE.

An excerpt of our encoding of congruence can be found here:

```
Inductive struct_eq {n:nat} : proc n -> proc n -> Prop :=
    SC_Par_Com : forall P Q : proc n,
                 P ‖ Q ≅ Q ‖ P
  | SC_Par_Assoc : forall P Q R : proc n,
                   P ‖ Q ‖ R ≅ P ‖ (Q ‖ R)
  | SC_Par_Inact : forall P : proc n,
                   P ‖ ∅ ≅ P
  | SC_Res_Scope : forall (P : proc n.+2) (Q : proc n),
                   (ν) P ‖ Q ≅  (ν) (P ‖ shift_two_up Q)
  | SC_Res_SwapC : forall P : proc n.+2,
                   (ν) P ≅ (ν) (< swap_ch var_zero var_one >) P
  | SC_Res_SwapB : forall P : proc n.+4,
                   (ν) (ν) P ≅ (ν) (ν) (< swap_ch var_one var_three >)
                     ((< swap_ch var_zero var_two >) P)
```

Listing 2 Selected congruence rules (🏂)

The monoidal cases are immediate. In the scope extrusion case, we use the function `shift_two_up : : proc m -> proc m.+2` (🏂) to adjust the scope of `Q`. Similarly, in the swap rules the function `swap_ch` (🏂) is mapped over `Q` to exchange the appropriate indexes. Note that its signature requires decidability of the `fin` type.

The encoding of reduction looks rather straightforward, save for one non trivial technical issue brought in by rule R-CLOSE: this rule will gladly produce processes that make little sense. For example, if x occurs freely in P, an application of the rule will result in a term with a (new) free variable, since the restriction is deleted. This is exactly where the type system steps in, ruling out such a process as ill-typed; in fact, it is a theorem that free names of a well-typed process are preserved under reduction.

While it may be defensible on philosophical grounds to allow such a rule in an informal development, the rule is incompatible with a well-scoped implementation, where the context size must remain invariant under reduction. The same issue was observed in [17] and more radically addressed by forbidding free occurrences of x and y in P and Q, respectively. However, our well-scoped implementation would require complex adjustments to reconcile this extra condition with the context-size checking that is automatically enforced by Rocq. We modify R-CLOSE to ensure the restriction is preserved in the reductum, thereby also preserving the set of free names. To compensate, we add a further congruence rule allowing us to remove restrictions over inactive processes:

```
  | SC_Res_Inact : (ν) ∅ ≅ ∅
```

The encoding of the rest of the rules is unsurprising: note that in the last constructor `x..` is a notation for `scons x ids`, which is the `Autosubst2` way to encode endpoint substitution in rule R-COM. In the mathematical notation in Fig. 1, the substitution $Q\{z/w\}$ is the traditional way of denoting $\langle\cdot\{z/w\}\rangle Q$ where $\{z/w\}$ is the identity substitution on all inputs except for w which is instead mapped to z.

```
Inductive reduce {n:nat} : proc n -> proc n -> Prop :=
  | R_Res : forall P Q : proc n.+2,
            P ⇛ Q -> (ν) P ⇛ (ν) Q
  | R_Par : forall P Q R : proc n,
            P ⇛ Q -> P ‖ R ⇛ Q ‖ R
  | R_Struct : forall P P' Q Q' : proc n,
               P ≅ P' -> P' ⇛ Q' -> Q' ≅ Q -> P ⇛ Q
  | R_Close : forall P Q : proc n.+2,
              (ν) (1 ! . P ‖ 0 ? . Q) ⇛ ((ν) (P ‖ Q))
  | R_Com : forall (x : ch n.+2)(P : proc n.+2) (Q : proc n.+3),
            (ν) (1 ! x . P ‖ 0 ? (_).Q) ⇛ (ν) (P ‖ (< x.. >) Q)
```

Listing 3 Reduction rules ( )

As a roadmap for the following sections, the reader should keep in mind that we will need to establish preservation under both congruence and reduction for:

1. free names,
2. the linearity predicate, and
3. type derivations.

This in turn requires proving the corresponding substitution lemmas, as well as their inverses (arising from the symmetry built into the definition of congruence). All these proofs will be carried out by rule induction on the definitions of congruence and reduction, combined with inversion or case analysis on the judgments being preserved. Finally, since reduction is defined in terms of congruence, any preservation result for reduction will depend on the corresponding result for congruence.

4 Free Names

One of the main hassles in a de Bruijn encoding of a π-calculus is the handling of free names. While it is straightforward to implement a function that collects the set of free names of a given process, we then have to prove a variety of trivial but annoying results about this datatype. We instead use the predicate (actually implemented as a function) `free_in : ch n -> proc n -> Prop` ( ) that abstracts from the representation of the collection of those names by just checking if a name does occur freely in a process. This predicate will be crucial in the definition of the linearity predicates.

Free names have an interesting relationship with substitutions, in order to guarantee properties such as linearity. Recall that a substitution is a function from indexes to endpoints. While those are arbitrary in `Autosubst2`, we need them to satisfy *injectivity*. The latter, however, is a strong property since it must hold over the entire (possibly infinite) domain of the function. As we explore in more detail later, communication in well-typed process is *not* fully injective. To address this, we introduce a weaker notion, dubbed *injectiveNS*, where we check injectivity from the perspective of a particular name n, and only with respect to a set S of free names. Since we have adopted an implicit definition of free names, one of the argument is the process P.

Definition 4.1 (🐓) The predicate $\mathsf{injectiveNS}(n, P, \sigma)$ holds if and only if, for every j, whenever $\mathsf{free\_in}(j, P)$ and $\sigma(n) = \sigma(j)$, then $n = j$.

This definition is particularly useful as it allows us to verify whether a specific name maintains linearity relative to the free names of a process. Note also that adopting injectivity properties of substitutions is quite common in the π-calculus — see Sangiorgi and Walker [12] (page 48) stronger notion of injectivity over a set X.

Our definition of injectivity satisfies some (low level) results that we leave to the accompanying code, including:

- inversion principles over the process constructors;
- the fact that swapping substitutions are indeed injective.

Armed with this definition, we now tackle the three preservation results with respect to free names. While formal proofs can be found in the github repository, we give an intuition of the main aspects of each proof.

Lemma 4.1 (Preservation of free names under substitution)

1. *Assume that* $\mathsf{injectiveNS}(i, P, \sigma)$ *holds. If* $\mathsf{free\_in}(i[\sigma], \langle\sigma\rangle P)$ *then* $\mathsf{free\_in}(i, P)$ *(🐓).*
2. *If* $\mathsf{free\_in}(i, P)$ *then* $\mathsf{free\_in}(i[\sigma], \langle\sigma\rangle P)$ *(🐓).*

Proof Intuition.

1. We prove, by induction on P, the following auxiliary result, from which the main result follows by injectivity: $\forall\sigma\ P\ x, \mathsf{free\_in}(x, \langle\sigma\rangle P) \to \exists y, \mathsf{free\_in}(y, P) \wedge \sigma y = x$.
2. By a straighforward induction on P.

□

As to why the first direction of the lemma requires $\mathsf{injectiveNS}(i, P, \sigma)$, consider this (counter)example: given the process $P = x!.Q \parallel x?.R$ such that $\neg\mathsf{free\_in}(y, P)$, any substitution that maps both x and y to z — for which *injectiveNS* does not hold — would not satisfy the lemma. In fact, $y[\sigma]$ is a free name in $\langle\sigma\rangle P$, while y is not free in P.

Next, we show that structural congruence preserves the free names of a process:

Lemma 4.2 (Preservation of free names under congruence (🐓)) *Assume that* $P \equiv Q$ *holds:* $\mathsf{free\_in}(x, P)$ *iff* $\mathsf{free\_in}(x, Q)$.

Proof Intuition. The proof proceeds by structural induction on $P \equiv Q$. Because of scope extrusion and shifting, in rules such as C-SCOPE, C-SWAPC, and C-SWAPB, we use both parts of Lemma 4.1. Note that all substitutions used in structural congruence are injective, therefore *injectiveNS* always holds. □

We conclude this part with the following result that states that, under reduction, no new free name is generated:

Lemma 4.3 (Preservation of free names under reduction (🐓)) *If* $P \to Q$ *and* $\mathsf{free\_in}(x, Q)$, *then,* $\mathsf{free\_in}(x, P)$.

Proof Intuition. By structural induction on the derivation of $P \to Q$. The R-STRUCT case relies on Lemma 4.2. The most interesting case is R-COM, where $(\boldsymbol{\nu} xy)\ (x!z.P \parallel y?(w).Q) \to (\boldsymbol{\nu} xy)\ (P \parallel Q\{z/w\})$. This case uses the first part of Lemma 4.1. The difficulty arises in the subcase where x is free in Q. If x coincides with the communicated endpoint z, then x must be present in the redex, since it is the object of the output. Otherwise, there is no clash: we are checking $\mathsf{injective}\mathsf{NS}(x, P, \{z/w\})$, which holds because x and z are distinct. □

5 Towards Session Types

The main challenge in mechanizing session types lies in handling *linearity* within the type system. In session-typed languages, linearity of an endpoint ensures the absence of races: no two processes can run in parallel while attempting to communicate (input or output) over the same endpoint. This contrasts with the linear π-calculus, where a linear channel is restricted to a single use—that is, it must be used and cannot be duplicated.

Most formalizations of linearity in proof assistants—whether of type-theoretic frameworks, sequent calculi, or operational semantics—have approached resource management at the data-type level: this is typically done by modeling contexts as lists or multisets and developing libraries of theorems (often in the thousands lines of code) to manage their structural properties; for a guided tour of the literature, please see [17].

The central idea of our formalization is to use the linearity predicate in the typing rules in lieu of explicit management of resources. However, for pedagogical reasons, we will first present the typing rules as in the book in this Section, which use multisets and splitting, and then reformulate them with LP (Section 6). As hinted in the introduction, the two systems are *not* equivalent; in fact, ours is slightly more liberal, in so far that it types certain processes such as $(\boldsymbol{\nu} xy)\ \mathbf{0}$ — this is due to our choice of representation of the R-CLOSE reduction rule.

Type Syntax. We start with some basic common notions. The type system does not type-check processes directly, but instead focuses on the endpoints used in the process. The syntax of *session types* S, T and linear type contexts Δ is as follows:

$$\begin{array}{rcl} S, T & ::= & \mathbf{end}_! \mid \mathbf{end}_? \mid ?T.S \mid !T.S \\ \Delta & ::= & \cdot \mid \Delta, x : T \end{array}$$

The *end types* describe communications that have been ended, where the only remaining operations are closing the channel from one of the endpoints. The *input type* $?T.S$ describes endpoints used for receiving a message and then behaving according to S. The *output type* $!T.S$ describes endpoints used for sending a message and then behaving according to S.

(Linear) contexts associate a type to endpoints. We use the comma to add an entry and $\cdot$ as the empty context. We assume that all entries are distinct. Note that the order in which information is added to a type context does not matter.

The type system maintains two invariants:

$$\text{T-INACT}\ \frac{}{\cdot \vdash \mathbf{0}} \qquad \text{T-PAR}\ \frac{\Delta_1 \vdash P \qquad \Delta_2 \vdash Q}{\Delta_1, \Delta_2 \vdash P \parallel Q} \qquad \text{T-RES}\ \frac{\Delta, x : T, y : \overline{T} \vdash P}{\Delta \vdash (\boldsymbol{\nu} xy)\, P}$$

$$\text{T-WAIT}\ \frac{\Delta \vdash P}{\Delta, x : \mathbf{end}_? \vdash x?.P} \qquad \text{T-CLOSE}\ \frac{\Delta \vdash P}{\Delta, x : \mathbf{end}_! \vdash x!.P}$$

$$\text{T-SEND}\ \frac{\Delta, x : U \vdash P}{\Delta, x : !T.U, y : T \vdash x!y.P} \qquad \text{T-RECV}\ \frac{\Delta, x : T, y : U \vdash P}{\Delta, x : ?T.U \vdash x?(y).P}$$

Fig. 2 Linear typing rules

1. No endpoint is used simultaneously by parallel processes;
2. The two endpoints of the same session have dual types, where duality is defined as:

$$\overline{?S.S'} = !S.\overline{S'} \qquad \overline{!S.S'} = ?S.\overline{S'} \qquad \overline{\mathbf{end}_!} = \mathbf{end}_? \qquad \overline{\mathbf{end}_?} = \mathbf{end}_!$$

The second invariant is maintained by requiring duality when typing restrictions. The way the first is preserved varies according to the approach adopted to encode linearity.

Standard Typing Rules. Looking at the rules listed in Figure 2, linearity shows up as follows:

- in *splitting* the type context when typing the composition of processes (T-PAR) — this is what enforces the first invariant:
- requiring the null process to be typed in the *empty* context (T-INACT);
- by implicitly implementing lookup, consumption and update of an entry in the structure of the context (T-SEND,T-RECV). A more explicit version of e.g. rule T-SEND would be:

$$\text{T-SEND'}\ \frac{\Delta_1 \vdash x : !T.U \qquad \Delta_2 \vdash y : T \qquad \Delta_3, x : U \vdash P}{\Delta_1, \Delta_2, \Delta_3 \vdash x!y.P} \qquad \text{T-VAR}\ \frac{}{x : T \vdash x : T}$$

  where the comma is overloaded to denote context split and "consing" — we have also added that standard axiom rule for linear variables.

6 The Linearity Predicate

The idea of a linearity predicate was formulated by Crary [3] to enable the encoding of the meta-theory of substructural systems in a logical framework such as LF (and its implementation, Twelf) without modifying the framework itself. LF is well known to

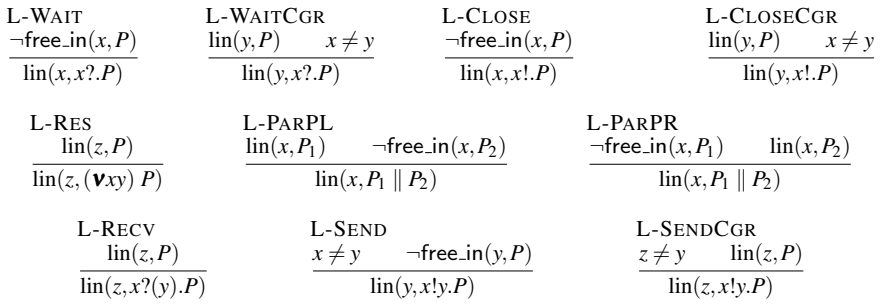

Fig. 3 The linearity predicate ( )

support higher-order abstract syntax (HOAS), an encoding technique where the object logic context is kept *implicit*, via hypothetical judgments; this has several benefits, the foremost the relegation of the proof of the substitution lemma to the substitution property of the framework. However, since LF is an intuitionistic logic, its contexts are *monotonic* and do not support substructural use. Crary's idea was to *separate* typing from resource management, the latter handled by a linearity judgment: the typing rules do not account for linearity, but they are decorated with assumptions indicating when (bound) variables are introduced in a substructural way. Crary's main application was the full linear lambda calculus and its subject reduction property. The idea has recently been applied to session types, more specifically to Wadler's CP [16], by Sano et al. [13] using the Beluga proof assistant, which is also based on LF.

In this tutorial, we show that there is no obstacle in porting Crary's approach to any mainstream proof assistant supporting inductive predicates. While for long standing reasons discussed elsewhere [7] we have to abandon HOAS, it turns out that LP fits particularly well with well-known techniques to encode binder signatures, namely well-scoped de Bruijn indexes and explicit contexts as total maps [14] — although other approaches such as unscoped indexes and contexts as association lists would also work.

The rules defining the relation $\mathrm{lin}(x,P)$, which specifies when an endpoint x is linear in a process P, are given in Figure 3. As noted earlier, this discipline enforces endpoint usage that guarantees the absence of races.

Most rules come in a base and a congruence case; let us discuss some cases and how we have mechanized them. The term $x!.P$ closes a session with endpoint x. Therefore, the predicate ensures that x is no longer used in the continuation P (rule L-CLOSE). However, any other y different from x is linear if it is linear in P (rule L-CLOSECGR). In Rocq, this is simply mechanised as:

```
Inductive lin {n : nat} : ch n → proc n → Prop :=
| LClose : forall (x : ch n) (P : proc n),
           ¬ free_in x P → lin x (x ! . P)
| LCloseCgr : forall (x y : ch n) (P : proc n),
               lin y P → x <> y → lin y (x ! . P)
```

The rule for restriction L-RES is straightforward. However, its representation in Rocq needs shifting of de Bruijn indices:

```
| LRes : forall (x : ch n) (P : proc n.+2),
      lin x [fun i ⇒ var_ch (shift (shift i))] P → lin x ((v) P)
```

Restriction binds two variables, therefore the endpoint x must be shifted up twice.

A key case for linearity is parallel composition $P_1 \parallel P_2$. Since an endpoint x can be used in P_1 or in P_2 but not in both, we use two rules: L-PARPL checks that x is used linearly in P_1 but does not occur in P_2 while symmetrically LPARPR makes sure that x does not occur in P_1 but then is linear in P_2. The mechanisation is easy:

```
| LParPL : forall (x : ch n) (P1 P2 : proc n),
          lin x P1 → ¬ free_in x P2 → lin x (P1 ∥ P2)
 (* plus symmetric rule *)
```

The receive case, captured by rule L-RECV, is similar to restriction. Since there is only a single binder—the placeholder for the received value—we shift by one.

```
| LRecv : forall (x y : ch n) (P : proc n.+1),
        lin y [fun i ⇒ var_ch (shift i)] P → lin y (x ? (_).P)
```

Rule L-SEND enforces linearity in endpoint delegation. In the term $x!y.P$, the endpoint y cannot be used within P, since it will be transferred to the peer of x and subsequently used there. Rule L-SENDCGR covers the case where the linearity of the delegated endpoint y is not being checked. Importantly, a term is never linear in the case of self-delegation, i.e., when $x = y$. In our code, this is formalized as:

```
| LSend : forall (x y : ch n) (P : proc n),
          x <> y → ¬ free_in y P → lin y (x ! y . P).
| LSendCgr: forall (x y z : ch n) (P : proc n),
          z <> y → lin z P → lin z (x ! y . P)
```

7 Typing with the Linearity Predicates

We argue that the linearity predicate simplifies the formulation of standard type systems as the one presented in Fig. 2. But how does one turn such a linear proof system into one featuring LP? Following on [13], we list a "recipe" to decorate the rules in a type system with the appropriate LP annotations.

1. If endpoints should not to be shared in the typing rule, then the linearity predicate must ensure that no sharing occurs for the construct. This is indeed taken care by LP for parallel composition, where the two rules L-PARPL and L-PARPR handle context splitting.
2. If the construct binds any new linear endpoint, its typing judgment must check its linearity by requiring the appropriate LP in the premise(s) of the rules. This is the case with restriction and input.

3. If the construct requires the absence of other linear assumptions, then there should be no congruence rules. This is typically the case for axioms and does not apply here, since the only axiom is for the null process for which congruence does not apply.

These heuristics need to be fine tuned for calculi with both additive and multiplicative connectives, as well as exponentials, such as in [3].

A key departure from previous literature is that our linearity predicate is coupled with the notion of *updating* endpoint types. The need for update is visible for example in rule T-SEND, where the type of x changes from the premise to the conclusion. In our setting, where the context is explicit, this is readily achieved by standard functional update (denoted by $\Delta + x : T$).

This contrasts with the mechanization of CP in Beluga [13], where the authors introduced a different calculus with "continuation channels" to avoid updating the type of the same channel. Their solution, though equivalent, requires a change in the syntax and semantics, whereas our approach maintains a more standard, direct representation. In passing, note that this issue was not manifest in Crary's account, since in the lambda calculus resources are consumed but not updated.

Figure 4 depicts our typing rules featuring the linearity predicate (notation $\Delta \Vdash_l P$) and a context that is never split, but can be updated, meaning a context where the entry for x has been updated with T— note that that *domain* of Δ stays monotonic. We also use the functional notation $\Delta(x) = T$ for looking up a context entry.

Some further remarks: the reader may be surprised by the non-empty context in rule TL-INACT. However this does not make the system *affine*, since there is no LP for $\mathbf{0}$ — see Exercise 7.2. In rules TL-PAR, TL-WAIT, TL-CLOSE, we thread the same context, which is neither split nor consumed. Rules TL-SEND, TL-RECV both update the context, the latter also extending it. Rule TL-RES has a special premise. Specifically, the linearity predicate for the fresh variables x and y is only checked if x

$$\text{TL-INACT}\ \frac{}{\Delta \Vdash_l \mathbf{0}} \qquad \text{TL-PAR}\ \frac{\Delta \Vdash_l P \qquad \Delta \Vdash_l Q}{\Delta \Vdash_l P \parallel Q}$$

$$\text{TL-RES}\ \frac{\Delta, x : T, y : \overline{T} \Vdash_l P \quad \mathsf{free\_in}(x,P) \Rightarrow \mathrm{lin}(x,P) \quad \mathsf{free\_in}(y,P) \Rightarrow \mathrm{lin}(y,P)}{\Delta \Vdash_l (\boldsymbol{\nu} xy)\, P}$$

$$\text{TL-WAIT}\ \frac{\Delta(x) = \mathbf{end}_? \quad \Delta \Vdash_l P}{\Delta \Vdash_l x?.P} \qquad \text{TL-CLOSE}\ \frac{\Delta(x) = \mathbf{end}_! \quad \Delta \Vdash_l P}{\Delta \Vdash_l x!.P}$$

$$\text{TL-SEND}\ \frac{\Delta(x) = !T.U \quad \Delta(y) = T \quad \Delta + x : U \Vdash_l P}{\Delta \Vdash_l x!y.P} \qquad \text{TL-RECV}\ \frac{\Delta(x) = ?T.U \quad (\Delta + x : T), y : U \Vdash_l P \quad \mathrm{lin}(y,P)}{\Delta \Vdash_l x?(y).P}$$

Fig. 4 Reformulating the typing rules with the linearity predicate

and y are free in P. This is due to the predicate not holding when x and y do not occur. This is necessary because of the way our semantics handles closing of a session. For example, the typable term $(\boldsymbol{\nu} xy)\ x?.\mathbf{0} \parallel x!.\mathbf{0}$ reduces to $(\boldsymbol{\nu} xy)\ \mathbf{0} \parallel \mathbf{0}$ and, by structural congruence, to $(\boldsymbol{\nu} xy)\ \mathbf{0}$: for the latter term to be typable rule TL-RES must do without checking linearity of x and y.

Exercise 7.1 (Choice). *Consider the standard typing rule for binary internal choice as in [6], for $i = 1, 2$.*

$$\text{T-SEL}\quad \frac{\Delta, x : T_i \vdash P}{\Delta, x : T_1 \oplus T_2 \vdash x \lhd i.P}$$

Reformulate it with the LP. Keep in mind the methodology delineated above. What happens if we instead consider n-ary choice? Now add also external choice. Does it bring in any additional insights?

Exercise 7.2 (Affine typing). *One way to make the type system* affine *is to generalize the axiom rule for* $\mathbf{0}$ *as follows:*

$$\text{T-INACT'}\quad \frac{}{\Delta \vdash \mathbf{0}}$$

But, wait! This is the rule we already have! How would you account for that?

We now look at the encoding of the typing rules. After this discussion, it is rather uneventful and we leave it to the accompanying code (🐓). Note that, as we have anticipated, the typing context is defined as a total map:

```
Definition env (n:nat) := ch n → sType.
```

This can be seen as a stream of types, where the empty map is represented as `env 0`. We define two main operations over environments:

1. *update* (🐓): this is the usual functional overriding;
2. *shift_env* (🐓), notation `T ::: Δ`: this is analogous to the *extension* operation to type environments and amounts to consing a new type at the top of the stream.

A technicality: we could have defined the typing environment as the map `fin n -> sType`, making the connection to `Autosubst2`'s parallel substitutions more explicit. In this case *shift_env* would be just an instantiation of the `scons` operation.

8 Preservation Results

In this section, we present the various results concerning both the linearity predicate and the typing rules. First, we show that linearity is preserved under substitution and structural congruence. Then, we move to the typing rules and show preservation results for substitution, congruence, and reduction.

8.1 Linearity Preservation

Linearity interacts subtly with substitution. If two distinct endpoints are identified by a non-injective substitution, linearity can be lost: two occurrences of distinct variables in a process may collapse into a single endpoint after substitution, creating multiple uses of the same channel. Injectivity with respect to the free occurrences of x in P prevents this collapse. The following lemma formalizes this preservation:

Lemma 8.1 (Preservation of linearity under substitution) *Let x be an endpoint, P a process, and σ be a substitution such that* $\mathsf{injectiveNS}(x,P,\sigma)$. *Then,*

1. *If* $\mathrm{lin}(x[\sigma],\langle\sigma\rangle P)$ *then* $\mathrm{lin}(x,P)$ *(🐢).*
2. *If* $\mathrm{lin}(x,P)$ *then* $\mathrm{lin}(x[\sigma],\langle\sigma\rangle P)$ *(🐢).*

Proof Intuition. The proof proceeds by structural nduction on P and inversion on the linearity predicate. Both directions need injectivity over x and the free names of P as well as Lemma 4.1, which depends on injectivity. For example, in the first case, consider a substitution σ that clashes endpoints x and y, e.g., $x[\sigma]=z$ and $y[\sigma]=z$. Then, it is clearly the case that $\mathrm{lin}(x[\sigma],y[\sigma]!.\mathbf{0})$ holds, while $\mathrm{lin}(x,y!.\mathbf{0})$ does not.

In the other direction, in order to show that $\mathrm{lin}(x[\sigma],x[\sigma]!.\langle\sigma\rangle P)$, it must be the case that $\neg\mathsf{free\_in}(x[\sigma],\langle\sigma\rangle P)$. By inverting the assumption $\mathrm{lin}(x,x!.P)$, we obtain $\neg\mathsf{free\_in}(x,P)$. That means that we must show that $\mathsf{free\_in}(x[\sigma],\langle\sigma\rangle P)$ implies $\mathsf{free\_in}(x,P)$, which follows from Lemma 4.1 (part 1).

□

Lemma 8.2 (Preservation of linearity under congruence (🐢).) *Assume that $P\equiv Q$ holds:* $\mathrm{lin}(x,P)$ *iff* $\mathrm{lin}(x,Q)$.

Proof Intuition. By induction on the derivation of $P\equiv Q$. Because of the bi-implication, we must use both directions of Lemma 8.1. It is interesting to observe that all substitutions used by structural congruence, namely swap and extrusion (shift up by two) are fully injective functions. Other relevant lemmas are 4.1 and 4.2. □

Our next proof obligation is the preservation property for the linearity predicate under reduction: if a process P reduces to Q, and x is linear in P, then x should also be linear in Q. It is important to notice, however, that this property only holds when P is well-typed. If we not check for well-typedness, linearity may not be preserved by reduction. For example, consider the process

$$x!y.P \parallel x?(z).Q$$

with $\neg\mathsf{free\_in}(y,P)$. After communication, y is substituted for z in Q. If z is not linear in Q, then y will not be linear in the resulting process either. Typing ensures that all bound names, namely those bound by restriction and input, are linear. Hence, we will be able to prove this lemma only after we have established preservation of types under congruence.

8.2 Type Preservation

Before tackling type preservation, we need some further definitions. We adapt the idea of *context morphism* [14] to our setting, which is central to state and prove the substitution lemma for our typing discipline. A context morphism is just a well-typed (parallel) substitution between contexts, say $\sigma : \Gamma \to \Delta$, meaning "$\sigma$ turns Δ-assumptions into Γ-assumptions". For our purposes, we first need to relativize the action of σ to the free names of P. This yield the following definition:

Definition 8.1 (🐓) $\mathsf{ltc}(\Delta,\Gamma,\sigma,P)$ iff $\forall i, \mathsf{free\_in}(i,P) \to \Gamma(i) = \Delta(i[\sigma])$.

There is a further twist: we need to make the action of the substitution compatible with *updating* types, as required by our typing rules. This means that the ltc predicate must be preserved by the update function. This property, however, is guaranteed only if the substitution is injective.

As a counter-example, consider a *non-injective* substitution σ defined by $x[\sigma] = x$, $y[\sigma] = x$, and $z[\sigma] = y$. Assume that $\Delta = \{x : T, y : T, z : T'\}$ and $\Gamma = \{x : T, y : T'\}$. Then, for any P, $\mathsf{ltc}(\Delta,\Gamma,\sigma,P)$ clearly holds because $\Delta(x) = \Gamma(x[\sigma]) = T$, $\Delta(y) = \Gamma(y[\sigma]) = T$, and $\Delta(z) = \Gamma(z[\sigma]) = T'$. However, if we update Δ and Γ according to the substitution, we obtain the environments:

$$\Delta' = \Delta + x : T' = \{x : T', y : T, z : T'\},$$
$$\Gamma' = \Gamma + x[\sigma] : T' = \{x : T', y : T'\}.$$

In this case, $\mathsf{ltc}(\Delta',\Gamma',\sigma,P)$ does not hold, since $\Delta'(x) = \Gamma'(x[\sigma]) = T'$, but $\Delta'(y) = T \neq \Gamma'(y[\sigma]) = T'$.

Unfortunately, the definition of $\mathsf{injectiveNS}(x,P,\sigma)$ will not do. For this purpose, we use a more standard definition, indeed the one by Sangiorgi & Walker [12], which refers to injectivity over the (finite) set of free names within a process:

Definition 8.2 (🐓) $\mathsf{injectiveS}(P,\sigma)$ iff $\forall ij,\ \mathsf{free\_in}(j,P) \to \mathsf{free\_in}(i,P) \to \sigma i = \sigma j \to i = j$

Again, we need to state and prove the substitution lemma in both directions — this is the equivalent of the *context morphism lemma* in [14]:

Lemma 8.3 (Preservation of substitution under types) *Assume that* $\mathsf{injectiveS}(P,\sigma)$ *and* $\mathsf{ltc}(\Delta,\Gamma,\sigma,P)$ *hold:*

1. *If* $\Delta \Vdash_l P$ *then* $\Gamma \Vdash_l \langle\sigma\rangle P$ *(🐓).*
2. *if* $\Gamma \Vdash_l \langle\sigma\rangle P$ *then* $\Delta \Vdash_l P$ *(🐓).*

Proof Intuition. By structural induction on P. The proof relies on several inversion lemmas about the ltc relation as well as the substitution properties for free names (Lemma 4.1) and the linearity predicate (Lemma 8.1). □

Note that this property (as the later case of weakening) can also be proven by rule induction on the given type derivation without much difference; this is due to the fact that the type system is syntax-directed, that is, there is exactly one rule for each process constructor.

Theorem 8.1 (Preservation of types under congruence (🐓).) *Assume that $P \equiv Q$ holds: $\Delta \Vdash_l P$ iff $\Delta \Vdash_l Q$.*

Proof Intuition. By induction on the derivation of $P \equiv Q$.All substitutions used by congruence are injective therefore all the previous lemmas assuming injectivity can be easily applied. □

Now, we can go back and establish:

Lemma 8.4 (Preservation of linearity under reduction (🐓).) *Assume that $\Delta \Vdash_l P$ and $P \to Q$ hold: if* $\mathrm{lin}(x,P)$ *then* $\mathrm{lin}(x,Q)$.

Proof Intuition. By induction on the derivation of $P \to Q$ and inversion on those of $\Delta \vdash P$ and $\mathrm{lin}(x,P)$. Case R-PAR requires Lemma 4.3. Congruence calls the related congruence Lemmas 8.2 and 8.1. The communication case R-COM makes use of the substitution Lemmas 4.1 (part 1) and 8.1 (part 2). □

The next lemma allows us to make arbitrary updates over endpoints that do not occur in a typed process.

Lemma 8.5 (Weakening (🐓)) *Assume that* $\neg\mathsf{free\_in}(x,P)$ *holds. Then, if $\Delta \Vdash_l P$ then $\Delta + x : T \Vdash_l P$.*

Proof Intuition. By structural induction on P and inversion on the derivation of $\Delta \Vdash_l P$ using properties of the `update` function. □

We stress that this weakening lemma is a property of the typing rules with respect to the update operation. This differs from the standard weakening lemma, which allows the addition of arbitrary endpoints (of terminated sessions) to a linear context. In our setting, however, the context itself is not linear (see rules T-PAR, T-END, and T-DEL). Accordingly, the lemma instead ensures that one may assign any type to endpoints not occurring in a typed process.

Weakening is needed in the proof of type preservation (subject reduction), specifically for handling the R-COM case, which we tackle next.

Theorem 8.2 (Type Preservation (🐓)) *Assume that* $\mathsf{free\_in}(x,P)$ *entails* $\mathrm{lin}(x,P)$. *If $\Delta \Vdash_l P$ and $P \to Q$, then $\Delta \Vdash_l Q$.*

Proof Intuition. The proof proceeds by induction on $P \to Q$, yielding five cases corresponding to the rules R-RES, R-PAR, R-STRUCT, R-CLOSE, and R-COM. Most cases are straightforward applications of the induction hypothesis or, in the case of R-STRUCT, of preservation of congruence (Lemma 8.1).

The most interesting case is R-COM, where the linearity of the communicated endpoint plays a crucial role in ensuring that typability is preserved. From the hypothesis

$$\Delta \Vdash_l (\boldsymbol{\nu} xy)\ \left(x!z.P \parallel y?(w).Q\right)$$

we must prove that

$$\Delta \Vdash_l (\boldsymbol{\nu} xy)\ \left(P \parallel Q\{z/w\}\right)$$

Applying inversion several times on the hypothesis, we obtain that the two endpoints x and y are linear in both P and Q, respectively, i.e., $\mathrm{lin}(x, P)$ and $\mathrm{lin}(y, Q)$. Additionally, each subterm is typable as:

$$\Delta, x : T, y : ?T'.\overline{T} \Vdash_l P \qquad \Delta, x : ?T'.T, y : \overline{T}, w : T' \Vdash_l Q$$

Here we observe that, since our type system does not directly split the type environment, the type of x is not updated while typing Q and similarly for y with respect to P. However, thanks to linearity of both x and y, we know that x does not occur in Q and y does not occur in P. Therefore, we can apply weakening (Lemma 8.5) to update their type correctly.

Another key observation is the requirement that the communicated endpoint z, which is free in the original term before reduction, is assumed to be as in the hypothesis of the theorem. In fact, thanks to that, we know that z is not free in P and, since it occurs in $x!z.P$, it cannot be free in Q. This makes the substitution $Q\{z/w\}$ injective, ensuring that no clashes occur which could compromise typability.

□

9 Conclusions

In this tutorial, we have mechanised the meta-theory of binary session types in the Rocq proof assistant, focusing on two of the central challenges of the area: the treatment of binders inherited from the π-calculus, and the enforcement of a linear type discipline. Our approach combines well-scoped de Bruijn indices with the linearity predicate, thus separating typing from resource management while ensuring well-formedness by construction. Along the way we have clarified the design space underlying the mechanisation of concurrency calculi, showing how concepts such as injective renamings, context updates, and preservation lemmas interact to yield machine-checked proofs of type preservation. Although our development does not claim novelty with respect to the theory itself, we believe that the Rocq formalisation contributes a useful reference point for researchers interested in the practicalities of mechanising session types.

The mechanisation highlights a number of lessons. First, the use of dependent types to encode scope information pays off in the form of strong guarantees against dangling references and shorter proofs. Second, the linearity predicate provides a modular and proof-friendly account of resource usage, avoiding the heavy infrastructure often associated with explicit context splitting. Finally, the process of formal development inevitably exposes small inconsistencies or oversights in the literature, which can be corrected or refined in the course of mechanisation.

The present work has concentrated on the core calculus of binary session types, but natural extensions include recursion, unrestricted channels, and subtyping, all of which present additional difficulties. Another relevant direction is to pursue more automation for repetitive reasoning steps via Rocq's tactic language and proof-search plugins. Finally, the ongoing development of the "concurrent calculi formalisation benchmark"

provides an ideal setting in which to situate and extend our contributions, fostering a more systematic understanding of the mechanisation of concurrent type systems.

In summary, we hope that this tutorial will serve both as a gentle entry point for newcomers to mechanised concurrency theory and as a platform for further explorations into the design and verification of session-typed languages.

References

1. Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. Strongly typed term representations in coq. *J. Autom. Reason.*, 49(2):141–159, 2012.
2. Marco Carbone, David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, Frederik Krogsdal Jacobsen, Alberto Momigliano, Luca Padovani, Alceste Scalas, Dawit Tirore, Martin Vassor, Nobuko Yoshida, and Daniel Zackon. The concurrent calculi formalisation benchmark. In Ilaria Castellani and Francesco Tiezzi, editors, *Proc. COORDINATION '24*, volume 14676 of *Lect. Notes Comput. Sci.*, pages 149–158, 2024.
3. Karl Crary. Higher-order representation of substructural logics. In *Proc. ICFP '10*, pages 131–142, 2010.
4. Simon J. Gay and Malcolm Hole. Types and subtypes for client-server interactions. In S. Doaitse Swierstra, editor, *ESOP '99: Proc. 8th European Symp. on Programming*, volume 1576 of *Lect. Notes Comput. Sci.*, pages 74–90. Springer, 1999.
5. Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3):191–225, 2005.
6. Simon J. Gay and Vasco T. Vasconcelos. *Session Types*. Cambridge University Press, 2025.
7. Robbert Krebbers, Alberto Momigliano, and Brigitte Pientka. Formalization of programming languages. In Jasmin Blanchette and Assia Mahboubi, editors, *Handbook of proof assistants*. Springer, 2025. Forthcoming.
8. Assia Mahboubi and Enrico Tassi. Mathematical components. In Bruno Woltzenlogel Paleo and David Delahaye, editors, *All about Proofs, Proofs for All*, volume 55 of *Studies in Logic*, pages 33–43. College Publications, 2010.
9. T. F. Melham. A mechanized theory of the π-calculus in HOL. *Nordic J. of Computing*, 1(1):50–76, March 1994.
10. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, February 2002.
11. Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. *Programming Language Foundations*, volume 2 of *Software Foundations*. Electronic textbook, 2024. Version 6.7, `http://softwarefoundations.cis.upenn.edu`.
12. Davide Sangiorgi and David Walker. *The π-Calculus - a Theory of Mobile Processes*. Cambridge University Press, 2001.
13. Chuta Sano, Ryan Kavanagh, and Brigitte Pientka. Mechanizing session-types using a structural view: Enforcing linearity without linearity. *Proc. ACM Program. Lang.*, 7(OOPSLA):235:374–235:399, 2023.
14. Kathrin Stark, Steven Schäfer, and Jonas Kaiser. Autosubst 2: reasoning with multi-sorted de bruijn terms and vector substitutions. In Assia Mahboubi and Magnus O. Myreen, editors, *CPP*, pages 166–180. ACM, 2019.
15. Vasco T. Vasconcelos. Fundamentals of session types. *Inf. Comput.*, 217:52–70, 2012.
16. Philip Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3):384–418, 2014.
17. Daniel Zackon, Chuta Sano, Alberto Momigliano, and Brigitte Pientka. Split decisions: Explicit contexts for substructural languages. In Kathrin Stark, Amin Timany, Sandrine Blazy, and Nicolas Tabareau, editors, *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2025, Denver, CO, USA, January 20-21, 2025*, pages 257–271. ACM, 2025.

A Additional Exercises and Selected Solutions

Exercise A.1 (Sending a channel to itself). *Give a typing rule so that* `x!x.P` *is well-typed. Do you need to modify the LP? If so, how? Reprove type preservation.*

Solution to Exercise 7.1. The typing rule is unchanged. Add the LP:

$$\textsc{L-IChoice}\ \frac{\neg \mathsf{free\_in}(x,P)}{\mathrm{lin}(x, x \lhd i.P)} \qquad \textsc{L-IChoiceCgr}\ \frac{\mathrm{lin}(y,P) \qquad x \neq y}{\mathrm{lin}(y, x \lhd .P)}$$

Solution to Exercise 7.2. You have to modify the LP by adding the obvious clause — arguably, now the name "linear" is a stretch:

$$\textsc{L-Inact}\ \frac{}{\mathrm{lin}(x, \mathbf{0})}$$

You need to reprove preservation of linearity.