HOπ in Coq

Guillaume Ambal, Sergueï Lenglet, Alan Schmitt

Abstract We present a formalization of HOπ in Coq, a process calculus where messages carry processes. Such a higher-order calculus features two very different kinds of binder: process input, similar to λ-abstraction, and name restriction, whose scope can be expanded by communication. For the latter, we compare four approaches to represent binders: locally nameless, de Bruijn indices, nominal, and Higher-Order Abstract Syntax. In each case, we formalize strong context bisimilarity and prove it is compatible, i.e., closed under every context, using Howe's method, based on several proof schemes we developed in a previous paper.

Keywords Higher-order process calculus, Howe's method, Coq

HOπ in Coq

Guillaume Ambal · Sergueï Lenglet · Alan Schmitt

Received: date / Accepted: date

1 Introduction

Process calculi aim at representing communicating concurrent systems, where agents are executed in parallel and exchange messages. When an input process $a?X.P$ is in parallel with an output process $\overline{a}!(M).Q$, communication takes place on the channel a, generating the process $P\{M/X\} \parallel Q$. If the message M is inert data, like a channel name in the π-calculus [37,49], the calculus is called first-order. Otherwise, if M is an executable process, the calculus is higher-order.

If the first-order π-calculus has been formalized in various proof assistants using different representations for binders (Gay [19] and Perera and Cheney [41] list some of them), only a few recent works propose a formalization of a higher-order calculus [40,33]. The semantics of the calculus of Parrow et al. [40] is based on triggers and clauses to enable the execution of transmitted processes. Maksimović and Schmitt [33] have formalized a minimal higher-order calculus which lacks name restriction $\nu a.P$, an operator widely used in process calculi to restrict the scope of channel names.

In this paper, we study the formalization in Coq of the Higher-Order π-calculus HOπ [48], a calculus with name restriction. From a formalization point of view, a higher-order calculus differs from a first-order calculus in their binding constructs. In the π-calculus, the entities bound by an input or a name restriction are names; a single representation of names can be used in a formalization as long as it suits both binders. In HOπ, an input binds a process variable while name restriction

Guillaume Ambal
Université de Rennes 1 - ENS Lyon
E-mail: guillaume.ambal@irisa.fr

Sergueï Lenglet
Université de Lorraine
E-mail: serguei.lenglet@univ-lorraine.fr

Alan Schmitt
Inria
E-mail: alan.schmitt@inria.fr

binds a name, so it makes sense to use distinct datatypes for process variables and names, giving us freedom to use different representations for each.

In addition, input and name restriction are quite different binding structures in HOπ. An input $a?X.P$ is similar to a λ-abstraction $\lambda x.t$, as the variable X is substituted with a process during communication. In contrast, restricted names are not substituted; moreover, the scope of an input is static, while the scope of a name restriction may change during a communication, a phenomenon known as *scope extrusion*. Indeed, when $a?X.P$ receives a message from $\nu b.\overline{a}!(Q).R$, the scope of b does not change if b does not occur in Q:

$$a?X.P \parallel \nu b.\overline{a}!(Q).R \longrightarrow P\{Q/X\} \parallel \nu b.R.$$

Otherwise, we extend the scope of b to include the receiving process, to keep the occurrences of b bound in Q:

$$a?X.P \parallel \nu b.\overline{a}!(Q).R \longrightarrow \nu b.(P\{Q/X\} \parallel R).$$

This assumes b does not occur in P, which may be achieved using α-conversion to rename b. Note that several names may be extruded at once, meaning that the binding representation should accommodate for sets of names.

Because of this discrepancy, we pick a single representation—de Bruijn indices as a nested datatype [7]—for the usual binding structure that is process input, but compare several representations—locally nameless [13], de Bruijn indices [16], nominal [45], and Higher-Order Abstract Syntax (HOAS) [42]—for the more unconventional name restriction. Considering several techniques allows us to illustrate the strengths and weaknesses of each of them when representing name restriction. The main result we formalize is showing that HOπ *context bisimilarity* [48] is *compatible* (i.e., preserved by the operators of the language) using *Howe's method* [27], a systematic proof technique to prove compatibility in a higher-order setting. Unlike Sangiorgi's original compatibility proof technique for HOπ [48], Howe's method scales to more expressive calculi, like calculi with passivation [32], a class of calculi we plan to formalize in the future. Our proofs follow a previous paper [30] in which we adapt Howe's method to context bisimilarity. This work is a first step in developing tools to work with higher-order process calculi in Coq.

The contributions of this paper are as follows.

- We propose the first formalization of HOπ, a calculus with name restriction and higher-order communication.
- We study several representations for channel names and pinpoint their respective strengths and weaknesses, in particular when dealing with the specificities of process calculi such as scope extrusion.
- Our test bed is Howe's method, a technique that has so far been formalized only for functional languages [1, 39, 52, 2].

We present HOπ in Section 2. We show how we formalize process variables in Section 3 before turning to channel names, for which we consider successively locally nameless (Section 4), de Bruijn indices (Section 5), nominal (Section 6), and HOAS (Section 7). We compare the four formalizations in Section 8, and we discuss related and future work in respectively Sections 9 and 10. The locally nameless formalization has been first presented at CPP [31]; the de Bruijn, nominal, and HOAS formalizations are new.

`fset A`	Finite sets with elements of type `A`
`\{}`	Empty set
`\{x}`	Singleton set
`\in`	Set membership
`\notin`	Negation of membership
`\c`	Set inclusion
`\u`	Set union
`\n`	Set intersection
`[=]`	Extensional equality on sets
`binary A`	Binary relations on elements of type `A`
∘	Relations composition
`tclosure Rel`	Transitive closure of the relation `Rel`

Table 1 Coq notations used in the paper for finite sets and relations

The formal developments are available at `http://passivation.inria.fr/hopi/`; a symbol 🐓 in the paper indicates a link to the online proofs scripts. We use the TLC library [12] which provides tools for classical reasoning in Coq. In particular, the conditional `If` (note the capital `I`) enables a choice on any proposition, and not just booleans. We rely on the excluded middle as a convenience; we believe our developments could be adapted to a pure constructive logic. The locally nameless development also uses the LN [11] library, while the de Bruijn and nominal ones rely on Metalib [51]. Both libraries provide predefined datatypes, such as finite sets and binary relations, as well as tools for automation to conduct proofs about formalized metatheory. In particular, they provide a representation of names (`var` for LN and `atom` for Metalib) with tactics to generate fresh names. To avoid switching notations mid-paper, we use the TLC and LN notations listed in Table 1 all along the paper, even when Metalib is used in the developments.

2 The Higher-Order π-Calculus

We recall the syntax, semantics, and bisimilarity of HOπ. This section is meant to be introductory. Thus, the syntax and notations we define are general and do not correspond to any of the Coq formalizations. We may adapt the syntax to be closer to the formalization under consideration when needed, like, e.g., in Section 4.5.

2.1 Syntax and Semantics

The syntax and semantics of the process-passing fragment of HOπ [48] are given in Figure 1. We use a, b to range over channel names, $\overline{a}$, $\overline{b}$ to range over conames, and X, Y to range over process variables. Multisets $\{x_1 \dots x_n\}$ are written $\widetilde{x}$.

We write $\oslash$ for the nil process that does nothing, $P \parallel Q$ for the parallel composition of the processes P and Q, $a?X.P$ for an input process which waits for a message on a, $\overline{a}!(P).Q$ for a sending process which emits P on a before continuing as Q, and $\nu a.P$ for the process where the scope of the name a is restricted to P. An input $a?X.P$ binds X in P, and a restriction $\nu a.P$ binds a in P. We write $\mathsf{fv}(P)$ for the free variables of a process P and $\mathsf{fn}(P)$ for its free names. A *closed process* has

Syntax

$$\begin{array}{rl} \text{Processes} & P ::= \oslash \mid X \mid P \parallel P \mid a?X.P \mid \overline{a}!(P).P \mid \nu a.P \\ \text{Agents} & A ::= P \mid F \mid C \\ \text{Abstractions} & F ::= (X)P \\ \text{Concretions} & C ::= \langle P \rangle Q \mid \nu a.C \end{array}$$

Structural congruence

$$(P \parallel Q) \parallel R \equiv P \parallel (Q \parallel R) \qquad P \parallel Q \equiv Q \parallel P \qquad P \parallel \oslash \equiv P \qquad \nu a.\nu b.P \equiv \nu b.\nu a.P$$

$$\nu a.(P \parallel Q) \equiv (\nu a.P) \parallel Q \text{ if } a \notin \mathsf{fn}(Q) \qquad \nu a.\oslash \equiv \oslash$$

Extension of operators to all agents

$$\begin{array}{ll} (X)Q \parallel P \triangleq (X)(Q \parallel P) & (\nu\widetilde{b}.\langle Q \rangle R) \parallel P \triangleq \nu\widetilde{b}.\langle Q \rangle (R \parallel P) \text{ if } \widetilde{b} \cap \mathsf{fn}(P) = \emptyset \\ P \parallel (X)Q \triangleq (X)(P \parallel Q) & P \parallel (\nu\widetilde{b}.\langle Q \rangle R) \triangleq \nu\widetilde{b}.\langle Q \rangle (P \parallel R) \text{ if } \widetilde{b} \cap \mathsf{fn}(P) = \emptyset \\ \nu a.(X)P \triangleq (X)\nu a.P & \nu a.(\nu\widetilde{b}.\langle Q \rangle R) \triangleq \nu a, \widetilde{b}.\langle Q \rangle R \text{ if } a \in \mathsf{fn}(\nu\widetilde{b}.Q) \\ & \nu a.(\nu\widetilde{b}.\langle Q \rangle R) \triangleq \nu\widetilde{b}.\langle Q \rangle \nu a.R \text{ if } a \notin \mathsf{fn}(\nu\widetilde{b}.Q) \end{array}$$

Pseudo-application

$$(X)P \bullet \nu\widetilde{b}.\langle R \rangle Q \triangleq \nu\widetilde{b}.(P\{R/X\} \parallel Q) \text{ if } \widetilde{b} \cap \mathsf{fn}(P) = \emptyset$$

LTS rules
$\alpha ::= \tau \mid a \mid \overline{a}$

$$a?X.P \xrightarrow{a} (X)P \ \textsc{In} \qquad \overline{a}!(Q).P \xrightarrow{\overline{a}} \langle Q \rangle P \ \textsc{Out} \qquad \frac{P \xrightarrow{\alpha} A}{P \parallel Q \xrightarrow{\alpha} A \parallel Q} \ \textsc{Par}$$

$$\frac{P \xrightarrow{\alpha} A \qquad \alpha \notin \{a, \overline{a}\}}{\nu a.P \xrightarrow{\alpha} \nu a.A} \ \textsc{Restr} \qquad \frac{P \xrightarrow{a} F \qquad Q \xrightarrow{\overline{a}} C}{P \parallel Q \xrightarrow{\tau} F \bullet C} \ \text{HO}$$

Fig. 1 Contextual LTS for HOπ

no free variable. We write $P\{Q/X\}$ for the usual capture-free substitution of X by Q in P.

Structural congruence $\equiv$ equates processes up to reorganization of their subprocesses and their name restrictions; it is the smallest congruence verifying the rules of Figure 1. Because the ordering of restrictions does not matter, we abbreviate $\nu a_1 \ldots \nu a_n.P$ as $\nu\widetilde{a}.P$.

The semantics is given by a labeled transition system (LTS), where closed processes transition to *agents*, namely processes, *abstractions* F of the form $(X)Q$, or *concretions* C of the form $\nu\widetilde{b}.\langle R \rangle S$. Like for processes, the ordering of restrictions does not matter for a concretion; therefore we write them using a set of names $\widetilde{b}$, except if $\widetilde{b} = \emptyset$, where we write $\langle R \rangle S$. Labels of the LTS are ranged over by α. Transitions are either an *internal action* $P \xrightarrow{\tau} P'$, a *message input* $P \xrightarrow{a} F$, or a *message output* $P \xrightarrow{\overline{a}} C$. The transition $P \xrightarrow{a} (X)Q$ means that P may receive a process R on a to continue as $Q\{R/X\}$. The transition $P \xrightarrow{\overline{a}} \nu\widetilde{b}.\langle R \rangle S$ means that P may send the process R on a and then continue as S, and the scope of the names $\widetilde{b}$

must be expanded to encompass the recipient of R. We expect $\widetilde{b}$ to contain names that are indeed in R, so that we only extend the scope of names for which it is necessary.

To write the LTS, we extend parallel composition and name restriction to abstractions and concretions, with side-conditions to avoid name capture. We remind that the LTS is defined on closed processes, so when we write $(X)Q \parallel P \stackrel{\Delta}{=} (X)(Q \parallel P)$, we assume P to be closed, and we do not need a side-condition to prevent the capture of X in P. When we define restriction on concretions, we distinguish between two cases, depending on whether the added restriction captures a name in the message. A higher-order communication takes place when a concretion C interacts with an abstraction F, resulting in a process written $F \bullet C$. The definition of the pseudo-application operator $\bullet$ and the LTS rules are given in Figure 1, except for the symmetric application $C \bullet F$ and the symmetric equivalent of rules Par and HO.

Remark 1 Our scope extrusion discipline is *lazy*, as we extend the scope of a name only when necessary. In contrast, *eager* scope extrusion always extends the scope of a restriction, meaning that adding a restriction on a around $\langle Q \rangle R$, using the extension of restriction to concretions (Figure 1), evaluates to $\nu a.\langle Q \rangle R$ in all cases, even when a is not free in Q. In HOπ, the two are equivalent, since $\nu a.(P\{Q/X\} \parallel R) \equiv P\{Q/X\} \parallel \nu a.R$ if $a \notin \mathsf{fn}(P\{Q/X\})$, but it is not true in all calculi; for instance, it does not hold in calculi with passivation [32], where restriction does not commute with localities. We use lazy scope extrusion because it appears to be the most commonly used in process calculi [15], and we want our formalization to scale to calculi with passivation.

2.2 Bisimilarity and Howe's Method

We relate processes with the same behavior using strong *context bisimilarity* [48], shortened as bisimilarity, defined as follows.

Definition 1 A relation $\mathcal{R}$ on closed processes is a *simulation* if $P \mathcal{R} Q$ implies:

- for all $P \xrightarrow{\tau} P'$, there exists Q' such that $Q \xrightarrow{\tau} Q'$ and $P' \mathcal{R} Q'$;
- for all $P \xrightarrow{a} F$, for all C, there exists F' such that $Q \xrightarrow{a} F'$ and $F \bullet C \mathcal{R} F' \bullet C$;
- for all $P \xrightarrow{\overline{a}} C$, for all F, there exists C' such that $Q \xrightarrow{\overline{a}} C'$ and $F \bullet C \mathcal{R} F \bullet C'$.

A relation $\mathcal{R}$ is a *bisimulation* if $\mathcal{R}$ and $\mathcal{R}^{-1}$ are simulations. Two processes P, Q are *bisimilar*, written $P \sim Q$, if there exists a bisimulation relating them.

We extend $\sim$ to open processes using *open extension*.

Definition 2 Given a relation $\mathcal{R}$ on closed processes and two processes P and Q, $P \mathcal{R}^{\circ} Q$ holds if $P\sigma \mathcal{R} Q\sigma$ holds for all process substitutions σ that close P and Q.

In the following, we use simulation up to structural congruence, a proof technique which allows us to use $\equiv$ when relating processes. Given two relations $\mathcal{R}$ and $\mathcal{S}$, we write $\mathcal{R}\mathcal{S}$ for their composition.

Definition 3 A relation $\mathcal{R}$ is a simulation up to $\equiv$ if $P \mathcal{R} Q$ implies the clauses of Definition 1, replacing $\mathcal{R}$ with $\equiv\mathcal{R}\equiv$.

Since $\equiv$ is a bisimulation, the resulting proof technique is sound.

Lemma 1 *If $\mathcal{R}$ is a bisimulation up to $\equiv$, then $\mathcal{R} \subseteq \sim$.*

The main result we formalize is compatibility of $\sim$ using Howe's method [27, 20], a systematic compatibility proof technique. The method can be divided in three steps: first, prove some basic properties on the *Howe's closure* $\sim^\bullet$ of the bisimilarity. By construction, $\sim^\bullet$ contains $\sim^\circ$ and is compatible. Second, prove a simulation-like property for $\sim^\bullet$. Finally, prove that $\sim$ and $\sim^\bullet$ coincide on closed processes. Since $\sim^\bullet$ is compatible, then so is $\sim$.

Given a relation $\mathcal{R}$, its Howe's closure is inductively defined as the smallest compatible relation closed under right composition with $\mathcal{R}^\circ$.

Definition 4 Howe's closure $\mathcal{R}^\bullet$ of a relation $\mathcal{R}$ is defined inductively by the following rules, where op ranges over the operators of the language.

$$\frac{P\ \mathcal{R}^\bullet\ P' \qquad P'\ \mathcal{R}^\circ\ Q}{P\ \mathcal{R}^\bullet\ Q} \qquad\qquad \frac{\widetilde{P\ \mathcal{R}^\bullet\ Q}}{\mathsf{op}(\widetilde{P})\ \mathcal{R}^\bullet\ \mathsf{op}(\widetilde{Q})}$$

The second rule of the definition ensures that $\mathcal{R}^\bullet$ is compatible. In this rule, the multiset $\widetilde{P\ \mathcal{R}^\bullet\ Q}$ contains as many premises as required by the arity of op. In particular, they are no premises for the base cases of the syntax of processes, namely $\oslash\ \mathcal{R}^\bullet\ \oslash$ and $X\ \mathcal{R}^\bullet\ X$, meaning that they are also the base cases of the definition of $\mathcal{R}^\bullet$.

Instantiating $\mathcal{R}$ as $\sim$, $\sim^\bullet$ is compatible by definition. The composition with $\sim^\circ$ enables a form of transitivity and additional properties. In particular, we can prove that $\sim^\bullet$ is *substitutive*: if $P \sim^\bullet Q$ and $R \sim^\bullet S$, then $R\{P/X\} \sim^\bullet S\{Q/X\}$. The closure $\sim^\bullet$ is also reflexive, which implies that $\sim^\circ \subseteq \sim^\bullet$; for the reverse inclusion to hold, we prove that $\sim^\bullet$ is a bisimulation, hence it is included in the bisimilarity. To this end, we first prove that $\sim^\bullet$ (restricted to closed terms) is a simulation, using a pseudo-simulation lemma. We then use the following result on the transitive closure $(\sim^\bullet)^+$ of $\sim^\bullet$.

Lemma 2 *If $\mathcal{R}$ is symmetric, then $(\mathcal{R}^\bullet)^+$ is symmetric.*

If $\sim^\bullet$ is a simulation, then $(\sim^\bullet)^+$ (restricted to closed terms) is also a simulation. By Lemma 2, $(\sim^\bullet)^+$ is in fact a bisimulation. Consequently, we have $\sim \subseteq \sim^\bullet \subseteq (\sim^\bullet)^+ \subseteq \sim$ on closed terms, and we conclude that $\sim$ is compatible.

The main challenge is to state and prove a simulation-like property for the Howe's closure $\sim^\bullet$. For higher-order process calculi equipped with a context bisimilarity, the difficulty is in the communication case. We propose in a previous paper [30] a formulation of the pseudo-simulation lemma which combines the input and output cases in a single clause, letting us deal with communication directly. We write $\sim_c^\bullet$ for the restriction of $\sim^\bullet$ to closed processes.

Lemma 3 (Pseudo-Simulation Lemma) *Let $P_1 \sim_c^\bullet Q_1$ and $P_2 \sim_c^\bullet Q_2$. If $P_1 \xrightarrow{\overline{a}} C_1$ and $P_2 \xrightarrow{a} F_1$, then there exist C_2, F_2 such that $Q_1 \xrightarrow{\overline{a}} C_2$, $Q_2 \xrightarrow{a} F_2$, and $F_1 \bullet C_1 \equiv \sim_c^\bullet \equiv F_2 \bullet C_2$.*

The proof of this result can be done by either *serialized* or *simultaneous* inductions. A proof by serialized induction proceeds in two steps, first proving an intermediary result by induction on the derivation of Howe's closure for the output processes $P_1 \sim_c^\bullet Q_1$, and then proving Lemma 3 by induction on $P_2 \sim_c^\bullet Q_2$. The reverse order (input then output) is also possible. A simultaneous induction proof considers the derivations of Howe's closure of the output and input processes together in the induction hypothesis, and proves Lemma 3 directly. Having several proof methods is convenient, as some of them cannot be applied in some calculi. If all the techniques apply in HOπ, only serialized proofs can be applied in a calculus with passivation, and only a simultaneous proof can be applied in a calculus with join patterns. We refer to [30] for more details.

3 Formalization of Process Variables

Because the representation of process variables is shared by our different formalizations, we discuss it beforehand. We use the Bird and Patterson encoding of de Bruijn indices as a nested datatype [7]. This representation enforces the set of free variables a process can be build on at the level of types: given a set `V`, `proc V` is the set of processes that can be built with variables taken from `V`.

```
Inductive incV (V:Set): Set :=
| VZ: incV V
| VS: V → incV V.

Inductive proc (V:Set): Set :=
| pr_nil: proc V                                    (* 'PO' *)
| pr_var: V → proc V
| pr_par: proc V → proc V → proc V                 (* P // Q *)
| pr_inp: name → proc (incV V) → proc V             (* a? P *)
| pr_snd: name → proc V → proc V → proc V        (* a!(P) Q *)
| pr_nu : ...
```

The datatype `name` and the representation of name restriction are discussed for each technique in its respective section. For all the constructors of `proc V` and `incV V`, we declare the parameter `V` as *implicit* (`:`), meaning that it can be omitted when writing terms if Coq is able to infer it from the context. With enough information, we can write `VZ` for `VZ Empty_set` or `pr_nil` for `pr_nil Empty_set`. To simplify further, we introduce Coq notations for each process construct, given in the comments of the code above. When Coq is unable to infer the parameter `V`, we can write it explicitly with `@`, e.g., `@VZ Empty_set` or `@pr_nil Empty_set`.

An input process `pr_inp` is built from a process of type `proc (incV V)`, where `incV V` extends `V` with an extra variable `VZ`, representing the index 0 of the new binder. Assuming we have some names a and b, the process $a?X.b?Y.(X \parallel Y)$ is thus written `a? b? (pr_var (VS VZ) // (pr_var VZ))` if we omit the implicit parameters. In the paper, we write these indices with a blackboard font, for instance $a?.b?.(\mathbb{1} \parallel \mathbb{0})$.

The benefit of this representation is that `proc` is parametric in its set of free variables. As a result, closed processes can easily be defined as processes built from the empty set, and abstractions as processes with at most one free variable.

```
Notation proc0 := (proc Empty_set).
Notation proc1 := (proc (incV Empty_set)).
```

Similarly, it is very simple to define a closing substitution, as shown in Section 4.3.

Substitution is defined in terms of shifting and monadic operations. To work under binders, we define a map operation (🦫)

```
Fixpoint mapV {V W:Set} (f:V → W) (P:proc V): proc W.
```

transforming the free variables of type `V` into variables of type `W`; note that `V` and `W` are declared as implicit in `mapV` because of the curly braces. We can shift variables using `mapV` as follows.

```
Notation shiftV := (mapV (@VS _)).
```

Next, we define lift and bind operations (🦫), to replace the variables of a process `proc V` with processes `proc W`.

```
Definition liftV {V W:Set} (f:V → proc W) (x:incV V): proc (incV W) :=
match x with
| VZ   ⟹ pr_var VZ
| VS y ⟹ shiftV (f y)
end.

Fixpoint bind {V W:Set} (f:V → proc W) (P:proc V): proc W :=
match P with
| pr_var x ⟹ f x
| a ? P ⟹ a ? (bind (liftV f) P)
(*...*)
end.
```

Lifting is necessary in the input case to prevent capture of process variables, but we also have to avoid channel name capture with name restriction; we discuss this case for each representation, in Section 4.1 for the locally nameless and de Bruijn formalizations, and Section 6.2 for the nominal one—HOAS prevents capture in this case by design.

Finally, we define substitution `subst P Q` which replaces the occurrences of `VZ` in `P` with `Q`. We do not define a more general operation that would replace any given variable (not only `VZ`) with a process as we do not need it.

```
Definition subst_func {V:Set} (Q:proc V) (x:incV V): proc V :=
match x with
| VZ   ⟹ Q
| VS y ⟹ pr_var y
end.

Notation subst P Q := (@bind _ _ (@subst_func _ Q) P).
```

In `subst P Q`, `P` is of type `proc (incV V)` while `Q` is of type `proc V` for some `V`.

Proofs usually require various properties on the relationships between `mapV`, `liftV`, and `bind` (🦫), including well-known monadic laws, such as the associativity of `bind` (🦫).

```
Lemma bind_bind {V1 V2 V3:Set}: ∀ (P:proc V1) (f:V1 → proc V2) (g:V2 → proc V3),
  bind g (bind f P) = bind (fun x ⟹ bind g (f x)) P.
```

The proofs of these results are usually by straightforward structural inductions on processes; the only difficult case is generally name restriction, depending on the chosen representation for channel names.

4 Locally Nameless

4.1 Syntax

The first technique we experiment with is the *locally nameless* representation [13], where bound channels are represented by de Bruijn indices and free channels by names. This representation has been originally proposed to benefit from a canonical representation of bound names thanks to de Bruijn indices, while avoiding the arithmetics on indices for free names. In our case, we define the datatype `name` as follows.

```
Inductive name: Set :=
| b_name: nat → name
| f_name: var → name.
```

We define name restriction for processes accordingly.

```
Inductive proc (V:Set): Set :=
(*...*)
| pr_nu: proc V → proc V        (*nu P*)
```

The name restriction construct `nu P` binds the occurrences of the bound name `0` in `P`. For example, $\nu a.(\overline{a}!(\oslash).\oslash \parallel \overline{b}!(\oslash).\oslash)$ is written

```
nu ((b_name 0)!(PO) PO // (f_name b)!(PO) PO)
```

assuming `b` is of type `var`. In the following, we omit `b_name` and `f_name` where it does not cause confusion,[1] and we use k to range over bound names and a, b to range over free names.

The grammar of processes allows ill-formed terms like `nu (b_name 1)!(PO) PO`, where the index `1` is a *dangling* name: it points to a non-existing restriction. We rule such terms out as usual in locally nameless by defining a predicate which checks if a process is *fully bound*.[2] The definition of the predicate relies on an *opening* operation, written $\{k \to a\}P$, which replaces a dangling name k with a free name a (👉). The converse *closing* operation, written $\{k \leftarrow a\}P$, replaces a free name a by a dangling name k (👈). The fully bound predicate `is_proc` is defined as follows.

```
Inductive is_proc {V:Set}: proc V → Prop :=
| proc_nil: is_proc PO
| proc_var: ∀ x, is_proc (pr_var x)
| proc_par: ∀ P Q, is_proc P → is_proc Q → is_proc (P//Q)
| proc_inp: ∀ (a:var) P, @is_proc (incV V) P → is_proc (a? P)
| proc_out: ∀ (a:var) P Q, is_proc P → is_proc Q → is_proc (a!(P) Q)
| proc_nu : ∀ (L:fset var) P, (∀ a, a \notin L → is_proc (open 0 a P)) →
                              is_proc (nu P).
```

In the input and output cases, a process is fully bound if the channel on which the communication happens is a free name. For name restriction, a process `nu P` is fully bound if opening `P` with a fresh name `a` generates a fully bound process. The name `a` should be fresh w.r.t. a finite set `L`; this *cofinite* quantification on `a`, usual in a locally nameless representation, gives a more tractable induction principle on

[1] In the code, we define coercions from respectively `nat` and `var` to `name`.

[2] Charguéraud [13] denotes this property as *locally closed*, but we prefer to use a different term, as our notion of closed process refers to process variables and not names.

fully bound processes, as it provides some leeway on the set `L` from which `a` should be fresh.

As explained by Charguéraud [13, Section 4.3], cofinite quantification has the drawback that we have to prove a result for infinitely many fresh names, while sometimes we can prove it for only one name `a`. We must then rely on a *renaming lemma* to change `a` into any name `b`, assuming `a` and `b` meet some freshness conditions. Such lemmas are traditionally consequences of more general lemmas showing that a property is preserved by substitution, because when variables are terms of the language, renaming is just a particular case of substitution. It is not the case here, as free names are not substituted in HOπ, so we have to write specific renaming lemmas. For example, `is_proc` is preserved by renaming in the most general sense (🐓).

```
Lemma is_proc_rename {V:Set}: ∀ (P:proc V) k a, is_proc (open k a P) →
                                                ∀ b, is_proc (open k b P).
```

There is no freshness conditions on neither `a` nor `b`. The proof is by induction on the derivation of `is_proc (open k a P)`.

In proofs, we may substitute processes that are not fully bound (see Remark 3), so we have to be careful to avoid capture of dangling names during substitution. We therefore define a map function on processes which operates on bound names, from which we can define a shift operation (🐓).

```
Fixpoint mapN {V:Set} (f:nat→nat) (P:proc V): proc V.

Notation shiftN n := (mapN (fun k ⟹ n+k)).
```

We can shift indices not only by 1 but by any n, as it will be useful later on. In the definition of `bind` in Section 3, the case for name restriction is

```
| nu P ⟹ nu (bind (fun x ⟹ shiftN 1 (f x)) P)
```

We also define the set of free names `fn` in a straightforward way (🐓) and then prove expected results in the relationship between `open`, `close`, `mapN`, and `fn` (🐓). For example, we prove that `open` and `close` are inverses of each other (🐓, 🐓).

```
Lemma close_open {V:Set}: ∀ (P:proc V) k a, a \notin fn P →
                                          close k a (open k a P) = P.

Lemma open_close {V:Set}: ∀ (P:proc V), is_proc P →
                                        ∀ k a, open k a (close k a P) = P.
```

In addition, we need to prove results on the interaction between the functions handling names and those manipulating process variables. For example, `open` and `close` commute with `mapV` (🐓), and they distribute over `bind` (🐓, 🐓). As a result, we have the following relationship between `open` and `subst`

```
Lemma open_subst {V:Set}: ∀ P (Q:proc V) k a, open k a (subst P Q) =
                                              subst (open k a P) (open k a Q).
```

and similarly for `close` (🐓, 🐓). These proofs are typically by structural induction on processes, or by induction on the derivation of the predicate `is_proc`.

4.2 Semantics

So far, the definitions are typical of a locally nameless representations of binders. We encounter issues more specific to HOπ when formalizing its semantics, in particular because of lazy scope extrusion.

As seen in Section 2, the LTS maps closed processes to agents, and language constructs are extended to agents (Figure 1). Representing abstractions is easy, as they are simply processes with at most one variable `proc1` (cf. Section 3).

```
Inductive abs: Set := | abs_def: proc1 → abs.
```

We define parallel composition for abstractions as follows

```
Definition abs_parl F (P:proc0) := match F with
| abs_def Q ⟹ abs_def (Q // shiftV P)
end.
```

with a symmetric function `abs_parr` (🦆). The LTS is defined only on closed processes; hence `P` is of type `proc0` in the above definition. As a result, `P` has no free variable, so `shiftV P` is in fact equal to `P`. However, shifting is necessary for typechecking: `abs_def` expects a process of type `proc1`, parallel composition expects two processes of the same type, and shifting transforms `P` into a process of type `proc1`. We also extend restriction to abstractions as expected (🦆).

The formalization of concretions is more involved because of scope extrusion.

```
Inductive conc: Set :=
| conc_def: nat → proc0 → proc0 → conc.
```

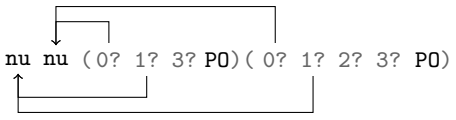
A concretion $\nu\widetilde{b}.\langle P\rangle Q$ is written `conc_def n P Q`, where `n` is the number of restrictions enclosing `P` and `Q`, i.e., the number of names in $\widetilde{b}$. Consequently, if $n > 0$, the processes `P` and `Q` are not fully bound. In particular, `P` contains all the extruded names, meaning all the bound names up to $n-1$. To enforce this condition, we define the set of dangling bound names `bn P` of a process `P` (🦆), and we define a well-formedness predicate on concretions as follows.

```
Definition conc_wf C := match C with
| conc_def n P Q ⟹ ∀ k, k < n → k \in bn P
end.
```

When extending parallel composition to concretions, we need to shift names if the process `P` we put in parallel is not fully bound (similarly for `conc_parr` (🦆)).

```
Definition conc_parl C (P:proc0) := match C with
| conc_def n P' Q ⟹ conc_def n P' (Q // (shiftN n P))
end.
```

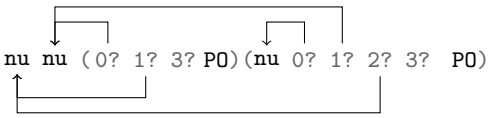
Defining name restriction for concretions implements lazy scope extrusion. If we add a restriction to the concretion `conc_def n P Q` such that `P` contains the bound name `n`, then `n` must be extruded, and the result is `conc_def (S n) P Q`. Otherwise, the added restriction needs to encompass `Q` only, but we must reorganize the names in `P` and `Q` accordingly. Indeed, for `conc_def 2 (0? 1? 3? PO)(0? 1? 2? 3? PO)`, the binding structure is originally



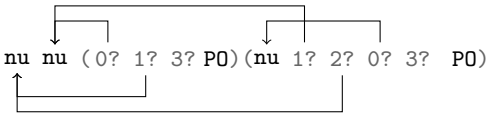
```
Definition down n k := If (k <= n) then k else k-1.

Definition permut n k := If (k < n) then (k+1) else If (k=n) then 0 else k.

Definition conc_new (C:conc) := match C with
| conc_def n P Q ⟹ If n \in bn P then conc_def (S n) P Q
                   else conc_def n (mapN (down n) P) (nu (mapN (permut n) Q))
end.
```

Fig. 2 Name restriction for concretions

with 2 and 3 left dangling. Suppose we want to bind 2. If we simply add a `nu` around the continuation, we get



The indices `0` and `1` in the message no longer match those in the continuation. We need to permute indices in the continuation to keep the correspondence.



The above term is still not quite the right result, as the index `3` in the message and continuation no longer designate the same channel: there are now three name restrictions around the continuation, but only two around the message. We therefore must change `3` in the message into `2`, and the correct result is `conc_def 2 (0? 1? 2? P0)(nu 1? 2? 0? 3? P0)`.

To summarize, if we add to a concretion `conc_def n P Q` a restriction which does not need to be extruded, we must permute the indices in `Q` that are smaller than `n`, and reduce by one the indices in `P` that are strictly greater than `n`. The definitions of these auxiliary operations and of the `conc_new` function are given in Figure 2.

We can now formalize the LTS of HOπ. We define agents and labels as follows.

```
Inductive agent :=
| ag_proc: proc0 → agent
| ag_abs : abs   → agent
| ag_conc: conc  → agent.
```

```
Inductive label: Set :=
| tau: label
| inp: var → label
| out: var → label.
```

The `out` and `inp` labels expect a `var`, meaning that only free names can be labels. We write `parl`, `parr`, and `new` the functions extending parallel composition and name restriction to agents (🦃, 🦃, 🦃), which are based on the previously defined extensions to concretions and abstractions. We compute the pseudo-applications $F \bullet C$ and $C \bullet F$ using respectively `appl` and `appr` (🦃, 🦃).

```
Fixpoint genNu {V:Set} n (P:proc V) :=
  match n with
  | 0 ⟹ P
  | S n' ⟹ nu (genNu n' P)
  end.

Definition appr F C := match C with
```

```
Inductive lts: proc0 → label → agent → Prop :=
| lts_out : ∀ (a:var) P Q, lts (a !(P) Q) (out a) (ag_conc (conc_def 0 P Q))
| lts_inp : ∀ (a:var) P,   lts (a ? P) (inp a) (ag_abs (abs_def P))
| lts_parl: ∀ P Q l A, lts P l A → lts (P // Q) l (parl A Q)
| lts_parr: ∀ P Q l A, lts P l A → lts (Q // P) l (parr Q A)
| lts_new : ∀ L P l A, (∀ a, a \notin L → a \notin fn_lab l →
                    lts (open 0 a P) l (open_agent 0 a A)) → lts (nu P) l (new A)
| lts_taul: ∀ P Q a F C, lts P (out a) (ag_conc C) → lts Q (inp a) (ag_abs F) →
                         lts (P // Q) tau  (ag_proc (appl C F))
| lts_taur: ∀ P Q a F C, lts P (out a) (ag_conc C) → lts Q (inp a) (ag_abs F) →
                         lts (Q // P) tau (ag_proc (appr F C)).
```

Fig. 3 Formalization of the LTS

```
  | conc_def n P Q ⟹ genNu n (asubst (fshiftN n F) P // Q)
  end.
```

The functions `asubst` (🐓) and `fshiftN` (🐓) are the straightforward extensions to abstractions of the corresponding operations `subst` and `shiftN` on processes. The function `genNu` recreates the n name restrictions of the concretion in front of the resulting process. We shift the abstraction n times to avoid unwanted captures, as `appr` and `appl` can be applied to abstractions and concretions with dangling bound names.

The LTS formalization is given in Figure 3. Because a label cannot be a bound name, in the output and input cases, we prevent a not fully bound process of the form `(b_name k)? P` or `(b_name k)!(P) Q` from reducing. In the name restriction case `nu P`, we open the process and instantiate the bound name `0` with a fresh name `a`, using cofinite quantification; `open 0 a P` should transition to an agent of the form `open_agent 0 a A`, where `open_agent` is the extension of `open` to all agents (🐓). We also forbid the label to be `a`, as in the Restr rule (cf. Figure 1); the function `fn_lab` returns either the empty set for τ or a singleton set otherwise.

The formalization of the LTS does not prevent a process with dangling names to reduce, as, e.g., `a? 0? P0` (where `a` is a free name) can do an input. However, a fully bound process should transition to a fully bound agent. We define `is_agent` as the extension of `is_proc` to agents (🐓). For an abstraction `abs_def P`, we just check that `P` is fully bound. For a concretion `conc_def n P Q`, the processes `P` and `Q` are not fully bound, but their dangling names should be smaller than `n`.

```
| conc_def n P Q ⟹ ∀ k, k \in bn P \u bn Q → k < n
```

If $n = 0$, then `bn P` and `bn Q` are empty, i.e., `P` and `Q` are fully bound. Note that in a well-formed fully bound concretion `conc_def n P Q`, the dangling names of `P` are exactly all k such that $0 \leq k < n$.

As wished, the LTS generates a fully bound agent from a fully bound process, and it also produces only well-formed concretions (🐓, 🐓).

```
Lemma lts_is_proc: ∀ P l A, is_proc P → lts P l A → is_agent A.

Lemma lts_conc_wf: ∀ P l (C:conc), lts P l (ag_conc C) → conc_wf C.
```

We also prove a renaming lemma for the LTS (🐓). We write `subst_lab l a b` for the function that replaces `a` with `b` in the label `l` (🐓).

```
Lemma lts_rename: ∀ P A l k a, lts (open k a P) l (open_agent k a A) →
                               is_proc (open k a P) → a \notin fn P →
                               a \notin fn_agent A → ∀ b, lts (open k b P)
                               (subst_lab l a b) (open_agent k b A).
```

As with `is_proc_rename`, there is no freshness condition on b. However, the condition $a \notin \mathsf{fn}(P)$ is necessary: if $P \stackrel{\Delta}{=} 0?.\mathbb{0} \parallel \overline{a}!(\oslash).\oslash$, then $\{0 \to a\}P$ can perform a communication, but $\{0 \to b\}P$ cannot for $b \neq a$.

Remark 2 We manipulate the bound names of a concretion `conc_def n P Q` directly when we define `conc_wf`, `conc_new`, and `is_agent`. This is unusual for a locally nameless formalization; a more standard way of defining these notions would have been to open the concretion with n fresh names and reason on these fresh names. We prefer to use bound names as it leads to very simple conditions to check (usually comparisons with n). The drawback is that we need lemmas relating `bn` to the other functions of the formalization (`mapN`, `open`, ...). For example, it is easy to show that P is fully bound iff $\mathsf{bn}(P)$ is empty. We can also prove a more general version of the lemma `open_close` using `bn`.

```
Lemma open_close_gen {V:Set}: ∀ (P:proc V) k a, k \notin bn P →
                                                open k a (close k a P) = P.
```

4.3 Bisimilarity

Even though bisimilarity can be defined using a coinductive datatype, we prefer to use the set theoretic approach, where two terms are bisimilar if there exists a bisimulation relating them. Not only it corresponds to Definition 1, but it is also more tractable in Coq [33].

In the following, `test_proc`, `test_abs`, and `test_conc` are notations representing the testing conditions of Definition 1 for each kind of agent. In `test_abs`, the concretion we use to compare abstractions is fully bound and well-formed, and similarly in `test_conc`, the testing abstraction is fully bound.

```
Definition simulation (Rel: binary proc0) :=
  is_proc_Rel Rel ∧
  ∀ P Q, Rel P Q → test_proc Rel P Q ∧ test_abs Rel P Q ∧ test_conc Rel P Q.
```

We restrict the notion of simulation to relations on fully bound processes thanks to the predicate `is_proc_Rel`. We then define bisimulation and bisimilarity as in Definition 1. The definition of open extension is simple thanks to the chosen representation of process variables.

```
Definition open_extension {V:Set} (Rel:binary proc0) (P Q:proc V) :=
  ∀ (f: V → proc0), (∀ v, is_proc (f v)) → Rel (bind f P) (bind f Q).
```

Since the free variables of `P` and `Q` are in `V`, a closing substitution is simply a function from `V` to `proc0`. The condition on `f` ensures that `f` maps variables to fully bound processes.

4.4 Structural Congruence

Structural congruence is denoted as `struct_congr` in the development. We prove that its restriction to fully bound closed processes, written `sc0`, is a bisimulation. The simulation proof is by induction on the derivation of $P \equiv Q$, and then by case analysis on the transition performed by P. The proof is quite lengthy because of the number of cases in the definition of $\equiv$ and the number of possible transitions. Most cases are straightforward, except for the ones manipulating name restrictions.

```
Inductive struct_congr {V:Set}: binary (proc V) :=
(*...*)
(*scope extrusion: nu a (P // Q) = (nu a P) // Q if a \notin fn Q*)
| sc_scope: ∀ P Q, struct_congr (nu (P // (shiftN 1 Q))) ((nu P) // Q)
(*permutation: nu a nu b P = nu b nu a P*)
| sc_nu_nu: ∀ P, struct_congr (nu nu P) (nu nu (mapN (permut 1) P))
```

As we can see from the definition of `permut` in Figure 2, `permut 1` exchanges `0` and `1` and leaves the other bound names unchanged. The difficulty with these two structural rules is when checking the output case, as we need to consider well-formed concretions with an arbitrary number of binders n. To test it, we pass it to an arbitrary abstraction, and we must verify that the resulting processes are still structurally congruent. This requires showing that the scope extrusion and permutation cases can be extended to an arbitrary number of restrictions. For example, to generalize the scope extrusion rule, we show that (🦕)

```
Lemma sc_scope_genNu {V:Set}: ∀ n (P Q:proc V),
  struct_congr (genNu n (P // (shiftN n Q))) ((genNu n P) // Q).
```

W.r.t. permutation, we show that (🦕)

```
Lemma sc_nuN_compM_permutN {V:Set}: ∀ n m (P:proc V),
  struct_congr (genNu (S n) P) (genNu (S n) (mapN (comp m (permut n)) P)).
```

where `comp m f` computes f^m for all function f and natural number m. The proofs of these results require some non-trivial arithmetic on de Bruijn indices, in particular to be able to compute `comp m (permut n) k` for all m, n, and k.

Once we prove `sc0` is a bisimulation (🦕), we can define bisimulation up to `sc0` as in Definition 3 (🦕), and prove it is a sound up-to technique in the sense of Lemma 1 (🦕).

4.5 Renaming Lemmas

The most intricate proof in the formalization outside of the proof of the main result is the renaming lemma for the bisimilarity; we sketch its proof here. We use mathematical notations for readability, but we modify the syntax of Figure 1 to stay faithful to the formalization. We use X, Y to range over indices representing process variables, written $\mathbb{0}$, $\mathbb{1}$, ..., we use k to range over indices representing bound names, written 0, 1, ..., and we use a, b, c to range over free names.

$$P ::= \oslash \mid X \mid P \parallel P \mid N?.P \mid \overline{N}!(P).P \mid \nu.P \qquad N ::= k \mid a$$

For example, the process $\nu a.(\overline{a}!(\oslash).\oslash \parallel b?X.X)$ is now written $\nu.(\overline{0}!(\oslash).\oslash \parallel b?.\mathbb{0})$. Abstractions F are just processes P such that $\mathsf{fv}(P) \subseteq \{\mathbb{0}\}$, and concretions C are now written $\nu^n\langle P\rangle Q$, where n is the number of name restrictions. We write $\mathsf{fn}(A)$ and $\mathsf{bn}(A)$ for the free names and dangling bound names of an agent A, and we write $\{k \to a\}A$ and $\{k \leftarrow a\}A$ for respectively the opening and closing operations, extended to all agents. We write $l\{b/a\}$ for the renaming operation on labels `subst_lab l a b`. Given two sets S_1, S_2, we write $S_1 \# S_2$ if $S_1 \cap S_2 = \emptyset$.

In this section, we use extensively the following decomposition result (🦕).

Lemma 4 *For all k, x, and A such that $k \notin \mathsf{bn}(A)$, there exists A' such that $A = \{k \to x\}A'$ and $x \notin \mathsf{fn}(A')$.*

Indeed, take $A' = \{k \leftarrow x\}A$, and then conclude with lemma `open_close_gen`. We also use this commuting property of opening (`open_open`, 🦆).

Lemma 5 *For all P, k_1, k_2, a, and b, $k_1 \neq k_2$ implies $\{k_1 \to a\}\{k_2 \to b\}P = \{k_2 \to b\}\{k_1 \to a\}P$.*

To prove a renaming lemma for the bisimilarity, we define a general renaming criterion on relations (🦆).

Definition 5 A relation $\mathcal{R}$ is stable by renaming if for all k, a, P, and Q such that $\{k \to a\}P$, $\{k \to a\}Q$ are fully bound and $a \notin \mathsf{fn}(P) \cup \mathsf{fn}(Q)$, $\{k \to a\}P \; \mathcal{R} \; \{k \to a\}Q$ implies that for all b such that $b \notin \mathsf{fn}(P) \cup \mathsf{fn}(Q)$, $\{k \to b\}P \; \mathcal{R} \; \{k \to b\}Q$.

This renaming notion is stricter than for the LTS (Lemma `lts_rename` in Section 4.2), as it requires b to be fresh from P and Q. This condition is not necessary for $\sim$ to be stable by renaming in HOπ, but it is needed as soon as we extend the language or if we consider weak bisimilarity. We discuss this issue in the appendix.

We show that bisimilarity is stable by renaming by proving a more general result. Given a relation $\mathcal{R}$ on fully bound processes, we define the renaming closure of $\mathcal{R}$ as follows (🦆).

$$\frac{P \; \mathcal{R} \; Q}{P \; \mathcal{R}^{\to} \; Q} \qquad \frac{\{a, b\} \# \mathsf{fn}(P) \cup \mathsf{fn}(Q) \qquad \{k \to a\}P \; \mathcal{R}^{\to} \; \{k \to a\}Q}{\{k \to b\}P \; \mathcal{R}^{\to} \; \{k \to b\}Q}$$

Lemma 6 *If $\mathcal{R}$ is a simulation, then so is $\mathcal{R}^{\to}$ (🦆).*

Proof Let P, Q, a, and b such that $a \neq b$, $\{a, b\} \# \mathsf{fn}(P) \cup \mathsf{fn}(Q)$, $\{k \to a\}P \; \mathcal{R}^{\to} \; \{k \to a\}Q$, and $\{k \to b\}P \; \mathcal{R}^{\to} \; \{k \to b\}Q$. We show that the transitions from $\{k \to b\}P$ are matched by $\{k \to b\}Q$. The proof is by induction on the derivation of $\mathcal{R}^{\to}$: assuming that the simulation tests hold for $\{k \to a\}P \; \mathcal{R}^{\to} \; \{k \to a\}Q$, we show that they hold for $\{k \to b\}P \; \mathcal{R}^{\to} \; \{k \to b\}Q$ as well. The interesting cases are the input and output tests; as they are dealt with similarly, we only discuss the former.

In that case, we have $\{k \to b\}P \xrightarrow{c} F_1$ for some F_1. Let C be a well-formed fully bound concretion. We want to find F_2 such that $\{k \to b\}Q \xrightarrow{c} F_2$ and $F_1 \bullet C \; \mathcal{R}^{\to} \; F_2 \bullet C$. The main idea is as follows. By Lemma 4, we have in fact $\{k \to b\}P \xrightarrow{c} \{k \to b\}F_1'$ for some F_1', so with `lts_rename`, we have $\{k \to a\}P \xrightarrow{c\{a/b\}} \{k \to a\}F_1'$. At this point, we want to use the induction hypothesis on $\{k \to a\}P \; \mathcal{R}^{\to} \; \{k \to a\}Q$ to get F_2' such that $\{k \to a\}Q \xrightarrow{c\{a/b\}} \{k \to a\}F_2'$ and $\{k \to a\}F_1' \bullet C \; \mathcal{R}^{\to} \; \{k \to a\}F_2' \bullet C$. We want to write this as $\{k \to a\}(F_1' \bullet C) \; \mathcal{R}^{\to} \; \{k \to a\}(F_2' \bullet C)$, to then use $\mathcal{R}^{\to}$ to rename a into b, but we need $\{a, b\} \# \mathsf{fn}(F_1' \bullet C) \cup \mathsf{fn}(F_2' \bullet C)$, and since C is completely arbitrary, it may contain a or b.

We modify C to remove a and b from it. Because k and $k+1$ are not dangling in C, using Lemma 4 twice, we can decompose C as $C = \{k \to b\}\{k+1 \to a\}C'$ for some C' such that $\{a, b\} \# \mathsf{fn}(C')$. We then apply the induction hypothesis on $\{k \to a\}P \; \mathcal{R}^{\to} \; \{k \to a\}Q$ not with C, but with the concretion $C'_{a,d} \triangleq \{k \to a\}\{k+1 \to d\}C'$, where d is a fresh name. We get F_2'' such that $\{k \to a\}Q \xrightarrow{c\{a/b\}} F_2''$ and $\{k \to a\}F_1' \bullet C'_{a,d} \; \mathcal{R}^{\to} \; F_2'' \bullet C'_{a,d}$. By Lemma 4, $F_2'' = \{k \to a\}F_2'$ for some F_2'

such that $a \notin \mathsf{fn}(F_2')$, so we have in fact $\{k \to a\}F_1' \bullet C'_{a,d} \mathrel{\mathcal{R}^{\to}} \{k \to a\}F_2' \bullet C'_{a,d}$. We write this as

$$\{k \to a\}(F_1' \bullet \{k+1 \to d\}C') \mathrel{\mathcal{R}^{\to}} \{k \to a\}(F_2' \bullet \{k+1 \to d\}C'). \tag{1}$$

We can prove that a and b do not occur in F_1', F_2', d, and C', either by construction or because $\{a, b\} \# \mathsf{fn}(P) \cup \mathsf{fn}(Q)$; we can thus use $\mathcal{R}^{\to}$ to rename a into b in (1):

$$\{k \to b\}(F_1' \bullet \{k+1 \to d\}C') \mathrel{\mathcal{R}^{\to}} \{k \to b\}(F_2' \bullet \{k+1 \to d\}C'). \tag{2}$$

Now we need to rewrite d back into a, but d is fresh from P and Q, so it does not occur in F_1' and F_2' (🦕). Because $k+1$ is not dangling in F_1' or F_2', we can rewrite (2) into

$$\{k+1 \to d\}\{k \to b\}(F_1' \bullet C') \mathrel{\mathcal{R}^{\to}} \{k+1 \to d\}\{k \to b\}(F_2' \bullet C'). \tag{3}$$

Again, d and a do not occur in F_1', F_2', b, and C', so we can rename d into a with $\mathcal{R}^{\to}$, and if we distribute back the opening operations, we get

$$\{k \to b\}F_1' \bullet \{k+1 \to a\}\{k \to b\}C' \mathrel{\mathcal{R}^{\to}} \{k \to b\}F_2' \bullet \{k+1 \to a\}\{k \to b\}C'. \tag{4}$$

But $\{k+1 \to a\}\{k \to b\}C' = C$, so we have $\{k \to b\}F_1' \bullet C \mathrel{\mathcal{R}^{\to}} \{k \to b\}F_2' \bullet C$, as needed.

What is left to prove is $\{k \to b\}Q \xrightarrow{c} \{k \to b\}F_2'$; but we know that $\{k \to a\}Q \xrightarrow{c\{a/b\}} \{k \to a\}F_2'$, so by `lts_rename`, we have $\{k \to b\}Q \xrightarrow{c\{a/b\}\{b/a\}} \{k \to b\}F_2'$; we can then prove that $c\{a/b\}\{b/a\} = c$ (🦕). □

We can then deduce from Lemma 6 the following result (🦕).

Theorem 1 *$\sim$ is stable by renaming.*

We also need to show that being stable by renaming is preserved by open extension (🦕).

Lemma 7 *If $\mathcal{R}$ is stable by renaming, then so is $\mathcal{R}^{\circ}$.*

The proof requires several renamings, as in that of Lemma 6. Indeed, let P, Q, a, and b such that $a \neq b$, $\{a, b\} \# \mathsf{fn}(P) \cup \mathsf{fn}(Q)$, and $\{k \to a\}P \mathrel{\mathcal{R}^{\circ}} \{k \to a\}Q$. We want to prove that $\{k \to b\}P \mathrel{\mathcal{R}^{\circ}} \{k \to b\}Q$, i.e., for all closing substitution σ, $(\{k \to b\}P)\sigma \mathrel{\mathcal{R}} (\{k \to b\}Q)\sigma$. To conclude, we would like to use the hypothesis that $\mathcal{R}$ is stable by renaming on $(\{k \to a\}P)\sigma \mathrel{\mathcal{R}} (\{k \to a\}Q)\sigma$, which we would like to rewrite into $\{k \to a\}(P\sigma) \mathrel{\mathcal{R}} \{k \to a\}(Q\sigma)$. But σ is arbitrary and may contain a or b, so we modify σ the same way we modify C in the proof of Lemma 6.

Remark 3 When we write, e.g., $\{k \to b\}(F_1' \bullet C')$ in the proof of Lemma 6, the agents F_1' and C' are not fully bound. This justifies why process substitution (performed in $\bullet$) should be defined on processes with dangling names.

```
Inductive howe {V:Set}: binary proc0 → proc V → proc V → Prop :=
| howe_comp: ∀ Rel P Q R, is_proc Q → howe Rel P R → open_extension Rel R Q →
                          howe Rel P Q
| howe_nil : ∀ Rel, howe Rel PO PO
| howe_var : ∀ Rel x, howe Rel (pr_var x) (pr_var x)
| howe_par : ∀ Rel P Q P' Q', howe Rel P Q → howe Rel P' Q' →
                            howe Rel (P // P') (Q // Q')
| howe_inp : ∀ Rel (a:var) P Q, @howe (incV V) Rel P Q → howe Rel (a ? P) (a ? Q)
| howe_out : ∀ Rel (a:var) P Q P' Q', howe Rel P Q → howe Rel P' Q' →
                                    howe Rel (a!(P) P') (a !(Q) Q')
| howe_nu  : ∀ L Rel P Q, (∀ x, x \notin L →
                    howe Rel (open 0 x P) (open 0 x Q)) → howe Rel (nu P) (nu Q).
```

Fig. 4 Formalization of Howe's closure

4.6 Howe's Closure: Formalization and Basic Properties

Figure 4 contains the formalization of Howe's closure. The closure relates only fully bound processes, because of the condition `is_proc Q` in the `howe_comp` case, and because the channel in the input and output cases must be a free name (🐓).

```
Lemma howe_implies_proc {V:Set}: ∀ Rel (P Q:proc V), howe Rel P Q →
                                                     is_proc P ∧ is_proc Q.
```

As a result, Howe's closure is reflexive on fully bound processes only (🐓).

We now prove a renaming lemma for Howe's closure, as we need it in the proof of Lemma 2 (🐓).

Lemma 8 *If $\mathcal{R}$ is stable by renaming, then so is $\mathcal{R}^\bullet$.*

Let P, Q, a, and b such that $a \neq b$, $a \notin \mathsf{fn}(P) \cup \mathsf{fn}(Q)$, $b \notin \mathsf{fn}(P) \cup \mathsf{fn}(Q)$, and $\{k \to a\}P \ \mathcal{R}^\bullet \ \{k \to a\}Q$. We prove that $\{k \to b\}P \ \mathcal{R}^\bullet \ \{k \to b\}Q$ by induction on the size of the derivation of $\{k \to a\}P \ \mathcal{R}^\bullet \ \{k \to a\}Q$. The induction is on the size of the derivation and not the derivation itself, as we need to rename twice in one of the cases, and therefore apply the induction hypothesis to processes that are not in the derivation of $\{k \to a\}P \ \mathcal{R}^\bullet \ \{k \to a\}Q$. In the formalization, we define a predicate `howe' Rel P Q n` where `n` is the size of the derivation (🐓), and the renaming lemma states that renaming preserves `n` (🐓).

The difficult case is `howe_comp`, where we have $\{k \to a\}P \ \mathcal{R}^\bullet \ R \ \mathcal{R}^\circ \ \{k \to a\}Q$ for some R. We would like to apply the induction hypothesis on $\{k \to a\}P \ \mathcal{R}^\bullet \ R$, to rename a into b. Even though $R = \{k \to a\}R'$ for some R' such that $a \notin \mathsf{fn}(R')$ with Lemma 4, we still cannot apply the induction hypothesis to rename a into b, as we may have $b \in \mathsf{fn}(R')$. Instead, we first apply the induction hypothesis to $\{k \to a\}P \ \mathcal{R}^\bullet \ R$ to rename b into a fresh name c. This leaves $\{k \to a\}P$ unchanged, since $b \notin \mathsf{fn}(\{k \to a\}P)$, so we get $\{k \to a\}P \ \mathcal{R}^\bullet \ R_c$ for some R_c. We then apply the induction hypothesis again to rename a into b to obtain $\{k \to b\}P \ \mathcal{R}^\bullet \ R'_c$ for some R'_c. We can do the same reasoning on $R \ \mathcal{R}^\circ \ \{k \to a\}Q$ using Lemma 7 to get $R'_c \ \mathcal{R}^\circ \ \{k \to b\}Q$. As a result, we have $\{k \to b\}P \ \mathcal{R}^\bullet \ R'_c \ \mathcal{R}^\circ \ \{k \to b\}Q$, i.e., $\{k \to b\}P \ \mathcal{R}^\bullet \ \{k \to b\}Q$, as wished.

We prove that $(\mathcal{R}^\bullet)^+$ is symmetric (Lemma 2 🐓) by showing that for all P and Q, $P \ (\mathcal{R}^\bullet)^+ \ Q$ implies $Q \ (\mathcal{R}^\bullet)^+ \ P$ by induction on $P \ (\mathcal{R}^\bullet)^+ \ Q$ (IH_1). The inductive case is straightforward. In the base case, we show that $P \ \mathcal{R}^\bullet \ Q$ implies

```
Notation pseudo_sim := ∀ {V W:Set} Rel (P1 Q1:proc V) (P2 Q2:proc W) a F1 C1
                         (f:V → proc0) (g: W → proc0),
   simulation Rel → refl0 Rel → rename_compatible Rel → (Rincl sc0 Rel) →
   trans Rel → howe Rel P2 Q2 → howe Rel P1 Q1 →
   (∀ v, is_proc (f v)) → (∀ v, is_proc (g v)) →
   lts (bind f P1) (inp a) (ag_abs F1) → lts (bind g P2) (out a) (ag_conc C1) →
   ∃ F2 C2, lts (bind f Q1) (inp a) (ag_abs F2) ∧
            lts (bind g Q2) (out a) (ag_conc C2) ∧
            (sc0 ° Rel ° sc0) (appr F1 C1) (appr F2 C2).
```

Fig. 5 Pseudo-simulation lemma

$Q\ (\mathcal{R}^\bullet)^+\ P$ by induction on $P\ \mathcal{R}^\bullet\ Q$ (IH_2). First, suppose $P\ \mathcal{R}^\bullet\ R\ \mathcal{R}^\circ\ Q$ for some R. Using (IH_2), we have $R\ (\mathcal{R}^\bullet)^+\ P$, and because $\mathcal{R}$ is symmetric, we have $Q\ \mathcal{R}^\circ\ R$, which in turn implies $Q\ \mathcal{R}^\bullet\ R$. We get $Q\ \mathcal{R}^\bullet\ R\ (\mathcal{R}^\bullet)^+\ P$, i.e., $Q\ (\mathcal{R}^\bullet)^+\ P$, as wished.

In the case where $P = \mathsf{op}(\widetilde{P'})$, $Q = \mathsf{op}(\widetilde{Q'})$, and $P'\ \widetilde{\mathcal{R}^\bullet}\ Q'$, then we have $Q'\ \widetilde{(\mathcal{R}^\bullet)^+}\ P'$ using (IH_2). We must show that $Q'\ \widetilde{(\mathcal{R}^\bullet)^+}\ P'$ implies $\mathsf{op}(\widetilde{Q'})\ (\mathcal{R}^\bullet)^+\ \mathsf{op}(\widetilde{P'})$, which is direct for all the operators, except name restriction. In that case we have $\forall a, a \notin L \Rightarrow \{0 \to a\}Q\ (\mathcal{R}^\bullet)^+\ \{0 \to a\}P$ for some L, and we must prove $\nu.Q\ (\mathcal{R}^\bullet)^+\ \nu.P$. We want to do an induction on $\{0 \to a\}Q\ (\mathcal{R}^\bullet)^+\ \{0 \to a\}P$ (IH_3), but we have to choose a fresh a first. As a result, in the base case we have $\{0 \to a\}Q\ \mathcal{R}^\bullet\ \{0 \to a\}P$ only for a given $a \notin L$, but to apply `howe_nu`, we want $\forall b, b \notin L' \Rightarrow \{0 \to b\}Q\ \mathcal{R}^\bullet\ \{0 \to b\}P$ for some L'. We need Lemma 8 to rename a into $b \notin L \cup \mathsf{fn}(Q) \cup \mathsf{fn}(P)$ to conclude.

Finally, to prove substitutivity, we show that `bind` preserves Howe's closure (🦕,🦕). Lemma `howe_subst` is then a direct consequence of Lemma `howe_bind` (🦕).

```
Lemma howe_bind {V W:Set}: ∀ Rel (P Q:proc V) (f g:V → proc W),
  (∀ x, howe Rel (f x) (g x)) → howe Rel P Q → howe Rel (bind f P)(bind g Q).

Lemma howe_subst {V:Set}: ∀ Rel (P' Q':proc V) P Q,
  howe Rel P Q → howe Rel P' Q' → howe Rel (subst P P')(subst Q Q').
```

4.7 Pseudo-Simulation Lemma

Figure 5 contains the formalization of Lemma 3, except we formulate it with any relation $\mathcal{R}$, and not just bisimilarity $\sim$. As a result, we can see the properties that $\mathcal{R}$ should satisfy for the lemma to hold, namely to be a stable by renaming, to be reflexive (on fully bound processes), and to be a transitive simulation that contains structural congruence. The other difference with Lemma 3 is that we consider open processes and use closing substitutions, instead of taking closed processes directly. The issue with the latter choice is that in the `howe_comp` case $P\ \mathcal{R}^\bullet\ R\ \mathcal{R}^\circ\ Q$, having P and Q closed would not necessarily imply that R is closed. We then would have to show that R can be closed without changing the size of the derivation, so the proofs would be done by induction on the size of the derivation of $P\ \mathcal{R}^\bullet\ Q$ instead of the derivation itself [30]. Using open processes and closing substitution is simpler as we can do a regular induction on the derivation in most cases.

We now present how to formalize the proofs of Lemma `pseudo_sim`, without detailing the proofs themselves, as the rationale behind these proof schemes is explained in our previous work [30]. We instead discuss how these schemes have been formalized, in particular how the formalization differs in the case of the simultaneous induction proof.

The formalization of the serialized proofs follows the pen-and-paper proofs. If we consider first the input processes P_1 and Q_1, we start by proving the following lemma (🦕).

```
Lemma pseudo_inp_first {V:Set}: ∀ Rel (P1 Q1:proc V) (P' Q':proc0) a F1
 (f:V → proc0), simulation Rel → refl0 Rel → rename_compatible Rel →
 (∀ P Q, Rel (P // PO) (Q // PO) → Rel P Q) → howe Rel P1 Q1 → howe Rel P' Q' →
 (∀ v, is_proc (f v)) → lts (bind f P1) (inp a) (ag_abs F1) →
 ∃ F2, lts (bind f Q1) (inp a) (ag_abs F2) ∧
       howe Rel (asubst F1 P') (asubst F2 Q').
```

We write σ_f and σ_g for the closing substitutions `f` and `g`. If $P_1\sigma_f \xrightarrow{a} F_1$, then there exists F_2 such that $Q_1\sigma_f \xrightarrow{a} F_2$ and $F_1 \bullet \nu^0\langle P'\rangle\oslash \mathcal{R}^\bullet F_2 \bullet \nu^0\langle Q'\rangle\oslash$. The condition $P \parallel \oslash \mathcal{R} Q \parallel \oslash \Rightarrow P \mathcal{R} Q$—which is weaker than containing structural congruence—then allows to remove any $\oslash$ in parallel. The proof is by induction on the derivation of $P_1 \mathcal{R}^\bullet Q_1$.

We then prove `pseudo_sim` by induction on the derivation of $P_2 \mathcal{R}^\bullet Q_2$ (🦕). The base cases are `howe_var`, where $P_2 = Q_2 = X$ and $X\sigma_g \xrightarrow{\overline{a}} C$ for some C, and `howe_out`, where $P_2 = \overline{a}!(P_2^1).P_2^2$, $Q_2 = \overline{a}!(Q_2^1).Q_2^2$, $P_2^1 \mathcal{R}^\bullet Q_2^1$, and $P_2^2 \mathcal{R}^\bullet Q_2^2$. In these cases, we know the messages are related by Howe's closure (either explicitly, or because Howe's closure is reflexive); therefore we can conclude with Lemma `pseudo_inp_first`.

If we start instead with the output processes P_2 and Q_2, we prove first the following lemma (🦕).

```
Lemma pseudo_out_first {V:Set}: ∀ Rel (P2 Q2:proc V) (P' Q':proc1) a C1
  (g:V → proc0), simulation Rel → refl0 Rel → trans Rel →
  rename_compatible Rel →  (Rincl sc0 Rel) → howe Rel P2 Q2 → howe Rel P' Q' →
  (∀ v, is_proc (g v)) → lts (bind g P2) (out a) (ag_conc C1) →
  ∃ C2, lts (bind g Q1) (out a) (ag_conc C2) ∧
        (sc0 ° howe Rel ° sc0) (appr (abs_def P') C1) (appr (abs_def Q') C2).
```

Unlike in Lemma `pseudo_inp_first`, we work up to structural congruence because we manipulate the scope of the restricted names of C_1 and C_2 in the proof. We then prove Lemma `pseudo_sim` by induction on $P_1 \mathcal{R}^\bullet Q_1$, using Lemma `pseudo_out_first` in the `howe_var` and `howe_inp` cases (🦕).

The formalization differs from [30] in how we handle the simultaneous induction proof, where we prove Lemma `pseudo_sim` directly by induction on the derivations of $P_1 \mathcal{R}^\bullet Q_1$ and $P_2 \mathcal{R}^\bullet Q_2$, considered together. There are four base cases, mixing the cases `howe_var` and `howe_inp` from $P_1 \mathcal{R}^\bullet Q_1$ and `howe_var` and `howe_out` from $P_2 \mathcal{R}^\bullet Q_2$, and the remaining cases are proved using the induction hypothesis.

This induction scheme is specifically tailored to prove Lemma `pseudo_sim`, as it relies on the fact that we do not need the induction hypothesis in the `howe_inp` and `howe_out` cases. Being ad hoc, Coq cannot generate such an induction principle automatically, so we would have to write by hand around forty-nine cases (seven

cases for $P_1\ \mathcal{R}^\bullet\ Q_1$ times seven for $P_2\ \mathcal{R}^\bullet\ Q_2$, although some can be factorized). We instead use a more tractable proof method.

Our formalized simultaneous proof (🐓) is by induction on the lexicographically-ordered couple (n_2, n_1), where n_2 is the size of the derivation of $P_2\ \mathcal{R}^\bullet\ Q_2$, and n_1 the size of the derivation of $P_1\ \mathcal{R}^\bullet\ Q_1$. Inside the induction, we proceed in two steps: first, show a preliminary result similar to `pseudo_out_first`, with the derivation of the output processes $P_2\ \mathcal{R}^\bullet\ Q_2$ of size n_2, and size of the derivation of the processes used as abstractions $P'\ \mathcal{R}^\bullet\ Q'$ is arbitrary. The proof is by case analysis on $P_2\ \mathcal{R}^\bullet\ Q_2$, and because n_2 strictly decreases, we can use the induction hypothesis.

We then prove the main result with a case analysis on $P_1\ \mathcal{R}^\bullet\ Q_1$. We can use the induction hypothesis because n_1 decreases, except for `howe_var`. In that case, we have $P_1 = Q_1 = X$, $n_1 = 0$, and $X\sigma_f \xrightarrow{a} F_1$; then $X\sigma_f\ \mathcal{R}^\bullet\ X\sigma_f$ holds by reflexivity of $\mathcal{R}^\bullet$, but the size of this proof can be any n_1'; this is why we need a lexicographic ordering on (n_2, n_1).We conclude in this case with the preliminary result.

We believe this proof scheme can be generalized to join patterns, where a receiver expects several messages. We cannot use a serialized proof in this case because there are several emitters: we cannot focus on a particular sender and need to consider them all at once. The pseudo-simulation lemma is then formulated with a list of output processes $(P_i\ \mathcal{R}^\bullet\ Q_i)_{1\leq i\leq n}$ as in [30], each derivation of size m_i. The decreasing measure is then $(\sum_{i=1}^{n} m_i, m)$, where m is the size of the derivation of the input process.

Once Lemma `pseudo_sim` has been proved, we can finish the proof by showing that $\mathcal{R}^\bullet$ restricted to closed processes is a simulation up to structural congruence (🐓).

```
Lemma simulation_up_to_sc_howe: ∀ Rel,
  simulation Rel → refl0 Rel → trans Rel → rename_compatible Rel →
  (Rincl sc0 Rel) → simulation_up_to_sc (howe Rel).
```

Because `simulation_up_to_sc` expects an argument of type `binary proc0`, writing `simulation_up_to_sc (howe Rel)` automatically restricts `howe Rel` to closed processes. We then show that $(\mathcal{R}^\bullet)^+$ restricted to closed processes is a bisimulation with the same hypotheses on $\mathcal{R}$ (🐓); because $\sim$ meets these conditions, $\sim_c^\bullet$ is a bisimulation, which implies $\sim = \sim_c^\bullet$, and $\sim$ is therefore compatible (🐓).

```
Theorem bis_howe: bisimilarity = howe bisimilarity.
```

In all these proofs, the name restriction case reveals to be quite intricate because of cofinite quantification. To see why, take, e.g., Lemma `pseudo_out_first`, and see how the name restriction case unfolds in a pen-and-paper proof. We have $P_2\ \mathcal{R}^\bullet\ Q_2$, $P'\ \mathcal{R}^\bullet\ Q'$, $\nu a.P_2 \xrightarrow{b} C_2$, and we want to show that there exists C_2' such that $\nu a.Q_2 \xrightarrow{b} C_2'$ and $(X)P' \bullet C_2 \equiv \mathcal{R}^\bullet \equiv (X)Q' \bullet C_2'$. The transition from $\nu a.P_2$ implies that $P_2 \xrightarrow{b} C$ for some C such that $\nu a.C = C_2$. We apply the induction hypothesis on P_2 and Q_2: there exists C' such that $Q_2 \xrightarrow{b} C'$ and $(X)P' \bullet C \equiv \mathcal{R}^\bullet \equiv (X)Q' \bullet C'$. The candidate concretion is then $C_2' \stackrel{\Delta}{=} \nu a.C'$, and indeed we can show that $\nu a.Q_2 \xrightarrow{b} \nu a.C'$ and $(X)P' \bullet \nu a.C \equiv \nu a.((X)P' \bullet C) \equiv \mathcal{R}^\bullet \equiv \nu a.((X)Q' \bullet C') \equiv (X)Q' \bullet \nu a.C'$, i.e., $(X)P' \bullet C_2 \equiv \mathcal{R}^\bullet \equiv (X)Q' \bullet C_2'$ as wished.

With a locally nameless representation, inverting the transition $\nu.P_2 \xrightarrow{b} C_2$ gives $\{0 \to x\}P_2 \xrightarrow{b} C_x$ for some cofinitely quantified x and for some C_x such that

$\nu.\{0 \leftarrow x\}C_x = C_2$. Therefore, we apply the induction hypothesis to $\{0 \to x\}P_2$ and $\{0 \to x\}Q_2$ and we get C'_x such that $\{0 \to x\}Q_2 \xrightarrow{b} C'_x$ and $(X)P' \bullet C_x \equiv\mathcal{R}^\bullet\equiv (X)Q' \bullet C'_x$. The candidate concretion becomes $C'_2 \stackrel{\Delta}{=} \nu.\{0 \leftarrow x\}C'_x$, but checking that $\nu.Q_2 \xrightarrow{b} \nu.\{0 \leftarrow x\}C'_x$ or $\nu.((X)P' \bullet \{0 \leftarrow x\}C_x) \equiv\mathcal{R}^\bullet\equiv \nu.((X)Q' \bullet \{0 \leftarrow x\}C'_x)$ can be tedious, as the LTS and Howe's closure are defined with cofinite quantification. As a result, we need to check that, e.g., $\{0 \to y\}Q_2 \xrightarrow{b} \{0 \to y\}\{0 \leftarrow x\}C'_x$ for a new cofinitely quantified y, which requires rewriting the open and close operations and applications of renaming lemmas. The main issue here is that we apply a simulation property (the induction hypothesis) to processes with a cofinitely quantified name x. The resulting entity (here C_x) necessarily depends on that x, which is problematic when using it in other cofinitely quantified statements.

4.8 Conclusion

De Bruijn indices seem well-suited to represent concretions, as a single natural number may stand for several binders. As a result, stating and proving properties such as `conc_wf` or `is_agent` usually amounts to simple comparisons on natural numbers.

Unfortunately, manipulating de Bruijn indices is not always as simple and an expected benefit of a locally nameless representation is to avoid most indices manipulation by using plain names for free channel names. It is the case when showing that Howe's closure is substitutive; since the closure is defined on fully-bound processes only, we do not manipulate indices for that proof. However, in general, we still need to define process substitution on processes with dangling bound names and do the required shifting machinery, as some proofs require substitution of such processes (cf. Remark 3).

Cofinite quantification is also problematic for a language where working under binders is commonplace. The issue is more acute with an existentially quantified property like simulation, because existential terms then depend on the choice of a cofinite name—an example is the proof of Howe's pseudo-simulation lemma, as discussed at the end of Section 4.7. We then have to rely on renaming lemmas, which are themselves difficult to prove (cf Section 4.5). In the end, the costs of using two different representations for bound and free channel names outweigh any of the benefits, which suggests that a representation using either only de Bruijn indices (Section 5) or names (Section 6) would be better.

5 De Bruijn Indices

5.1 From Locally Nameless to de Bruijn Indices

The locally nameless representation of Section 4 already defines the operations we need on de Bruijn indices, so we can derive a de Bruijn representation from it by removing what is no longer necessary.

First, the `name` datatype can be identified with natural numbers.

```
Definition name := nat.
```

We no longer need the `open` and `close` operations, the fully bound predicate `is_proc`, nor cofinite quantification. The free names `fn` (🐓) of a process in de Bruijn are the indices that do not refer to any binder, which is also how dangling bound names `bn` are defined in Section 4.2. Consequently, the developments on the latter can be easily reused to define the former. In particular, we reuse the same `mapN` function to manipulate indices (🐓), as well as any definition using dangling bound names, simply changing `bn` into `fn`. It is the case for instance of the functions on concretions `conc_wf` (🐓) and `conc_new` (🐓).

As a result, most of the formalization of the syntax and semantics in de Bruijn is the same as in locally nameless. Differences arise for notions whose definition depend on the representation of free names like, e.g., the LTS, where the labels are either free names or τ. We have to be careful when writing the LTS transition for name restriction (🐓).

```
| lts_new  : ∀ P l A, 0 \notin fn_lab l → lts P l A →
                      lts (nu P)(down_lab l)(new A)
```

The definition is simpler than in locally nameless because we no longer need cofinite quantification, but we should not forget to subtract 1 to the label in the input and output cases, which we do with the `down_lab` function (🐓). A more difficult task is to adapt the renaming lemmas of Section 4.5 so that they involve only indices. We present the issues and explain where such lemmas are needed in the next section.

5.2 Renaming Lemmas

In our de Bruijn representation, renaming lemmas become necessary to show that Howe's closure is substitutive, more precisely in the lemma stating that `bind` preserves Howe's closure (🐓).

```
Lemma howe_bind {V W:Set}:
  ∀ Rel (P Q:proc V) (f g:V → proc W), (∀ x, howe Rel (f x) (g x)) →
  howe Rel P Q → howe Rel (bind f P)(bind g Q).
```

We remind that `bind` shifts `f` and `g` in the name restriction case, so we need to show that $\mathcal{R}^\bullet$ is preserved by shift. But shifting also requires lifting, so for the proof to go through, we need a more general result stating that $\mathcal{R}^\bullet$ is preserved by any composition of shifting and lifting operations. It turns out that such a proof would be as difficult as proving a more general result, that $\mathcal{R}^\bullet$ is preserved by any injective total function on de Bruijn indices. Being total and injective ensures that two distinct indices are not mapped to the same index, thus avoiding name clashes. We call such a function a *renaming*, ranged over by ξ, and we write $a\xi$ and $A\xi$ for the application of ξ to a name a and an agent A.

Definition 6 (🐓) A relation $\mathcal{R}$ is stable by renaming is $P\ \mathcal{R}\ Q$ implies $P\xi\ \mathcal{R}\ Q\xi$ for all ξ.

We need to prove that the bisimilarity, its open extension, and its Howe's closure all share the above property.

Bisimilarity. Like in locally nameless, we define a renaming closure, as follows .

$$\frac{P\ \mathcal{R}\ Q}{P\ \mathcal{R}^{\rightarrow}\ Q} \qquad\qquad \frac{P\ \mathcal{R}^{\rightarrow}\ Q}{P\xi\ \mathcal{R}^{\rightarrow}\ Q\xi}$$

Lemma 9 () *If $\mathcal{R}$ is stable by renaming, then so is $\mathcal{R}^{\rightarrow}$.*

We proceed by induction on the derivation of $\mathcal{R}^{\rightarrow}$. The base case is easy; for the inductive case, let $P\xi\ \mathcal{R}^{\rightarrow}\ Q\xi$, with $P\ \mathcal{R}^{\rightarrow}\ Q$. We discuss only the input clause $P\xi \xrightarrow{a} F_1$. Let C be an arbitrary concretion; assume we can write it $C'\xi$ for some C'. Then the proof becomes as in Section 4.5. With a decomposition lemma on the LTS, we know that $P\xi \xrightarrow{b\xi} F_1'\xi$ for some b and F_1' such that $P \xrightarrow{b} F_1'$ and $b\xi = a$. By induction, there exists F_2' such that $Q \xrightarrow{b} F_2'$ and $F_1' \bullet C'\ \mathcal{R}^{\rightarrow}\ F_2' \bullet C'$. Then by applying ξ, we get $F_1'\xi \bullet C'\xi\ \mathcal{R}^{\rightarrow}\ F_2'\xi \bullet C'\xi$, i.e., $F_1'\xi \bullet C\ \mathcal{R}^{\rightarrow}\ F_2'\xi \bullet C$. To conclude, we need $Q\xi \xrightarrow{a} F_2'\xi$, which can be derived from $Q \xrightarrow{b} F_2'$ with a renaming lemma on the LTS.

The main difficulty is to show that for any ξ and C (or, more generally, any agent A), we can write C as $C'\xi$ for some C', the issue being that C may contain indices that are not in the image of ξ. We therefore build first a renaming ξ' such that all the indices of $C\xi'$ are the image of ξ. We can then find C' such that $C\xi' = C'\xi$. Introducing ξ' slightly changes the above proof: from $F_1' \bullet C'\ \mathcal{R}^{\rightarrow}\ F_2' \bullet C'$, we apply ξ to get $F_1'\xi \bullet C'\xi\ \mathcal{R}^{\rightarrow}\ F_2'\xi \bullet C'\xi$, and then we apply ξ'^{-1} (assuming it exists) to get $F_1'\xi\xi'^{-1} \bullet C\ \mathcal{R}^{\rightarrow}\ F_2'\xi\xi'^{-1} \bullet C$. But to conclude, we want $F_1'\xi \bullet C\ \mathcal{R}^{\rightarrow}\ F_2'\xi \bullet C$, so ξ'^{-1} should not change the names of $F_1'\xi$ nor $F_2'\xi$.

To satisfy these constraints, we prove the following lemma.

Lemma 10 () *Given two finite sets of names F and E, there exists ξ' such that*

- *$F\xi' = F'\xi$ for some F';*
- *$a\xi'\xi' = a$ for all a (ξ' is involutive);*
- *the set $\{a \mid a\xi' \neq a\}$ is finite;*
- *$a\xi' = a$ for all $a \in E$.*

The set F represents the free names of C, while E should contain the free names of $F_1'\xi$ and $F_2'\xi$. The third condition implies that ξ' is the identity, except on a finite set E''. We build ξ' incrementally, by induction on the size of the set F. If F is empty, take ξ' as the identity. Otherwise, pick $a \in F$. If $a = b\xi$ for some b, then there is no need to rename a: we just apply the induction hypothesis to $F \setminus \{a\}$ and $E \cup \{a\}$, and the ξ' we get satisfy the conditions of Lemma 10 for F and E as well.

Suppose $a \neq b\xi$ for all b. We first apply the induction hypothesis to $F \setminus \{a\}$ and E to get ξ'_a satisfying the conditions of Lemma 10 for these sets. Suppose we have b an index such that $b = c\xi$ for some c and for some $b \notin F\xi'_a \cup F \cup E$. Then the function

$$\xi \triangleq x \mapsto \begin{cases} b & \text{if } x = a \\ a & \text{if } x = b \\ x\xi'_a & \text{otherwise} \end{cases}$$

which extends ξ'_a with the swapping of a and b, is a renaming and satisfies the conditions for F and E.

What is left to prove is that we can find such an index b. To do so, we prove the following result: for all injective function f, for all n, there exist x and y such that $x > n$ and $x = f(y)$ (🦃). Indeed, suppose that there exists n such that for all x and y, $x = f(y)$ implies $x \leq n$. Let I be the interval $[\![0 \dots n+1]\!]$. Because f is injective, $f(I)$ is of cardinal $n+2$, and yet, for all $x \in f(I)$, $x \leq n$, a contradiction. Applying this result to the renaming ξ and to $n = \max(F\xi'_a \cup F \cup E)$ gives us indices b and c such that $b = c\xi$ and $b \notin F\xi'_a \cup F \cup E$, as wished.

This concludes the proof of Lemma 9, that we then apply to the bisimilarity.

Theorem 2 (🦃) *$\sim$ is stable by renaming.*

Open extension. The proof is similar for open extension.

Lemma 11 (🦃) *If $\mathcal{R}$ is stable by renaming, then so is $\mathcal{R}^\circ$.*

Let $P\ \mathcal{R}^\circ\ Q$. We want to prove that $P\xi\ \mathcal{R}^\circ\ Q\xi$ for any ξ, i.e., for all closing σ, $(P\xi)\sigma\ \mathcal{R}\ (Q\xi)\sigma$. We want to rewrite the latter into $(P\sigma')\xi\ \mathcal{R}\ (Q\sigma')\xi$ for some σ', to use the fact that $\mathcal{R}$ is stable by renaming. As before, σ may contain indices not in the image of ξ, so we need to rename these indices beforehand.

However we cannot use Lemma 10 directly with σ, as this lemma suppose a finite set of names. In our formalization, if P and Q are of type `proc V`, then σ is of type `V` $\to$ `proc0` for an arbitrary `V` which can be infinite. As a result, $\mathsf{fn}(\sigma) \triangleq \bigcup_{X \in V} \mathsf{fn}(X\sigma)$ may be infinite as well. Instead, we consider σ_r, the restriction of σ to the free variables of P and Q. Then $\mathsf{fn}(\sigma_r)$ is finite, $(P\xi)\sigma_r = (P\xi)\sigma$, and $(Q\xi)\sigma_r = (Q\xi)\sigma$. With Lemma 10, we build ξ' such that $\sigma_r\xi' = \sigma'_r\xi$ for some σ'_r. We then conclude the proof like with the bisimilarity.

To build σ_r in Coq, we have to define the set of free variables of a process `P`, written `fv P` (🦃). If `P` is of type `proc V`, then `fv P` contains elements of type `V`, which can be any set. As a result, we cannot reuse the finite sets libraries we use to define the set of free names, as they require some basic properties on the type of the elements. Instead, we use the `Ensemble` datatype from Coq standard library, so that `fv P` is of type `Ensemble V`. We then need to prove some basic results about `fv P`, in particular that `fv P` is finite for all `P` (🦃).

Howe's closure. With these results, we can easily show that Howe's closure is stable by renaming.

Lemma 12 (🦃) *If $\mathcal{R}$ is stable by renaming, then so is $\mathcal{R}^\bullet$.*

The proof is a simple induction on the definition of the closure, using Lemma 11 in the `howe_comp` case. We can then prove that $\mathcal{R}^\bullet$ is preserved by `bind` and therefore by substitution, but with the extra hypothesis that $\mathcal{R}$ is stable by renaming.

5.3 Conclusion

With substitutivity of Howe's closure proved, the rest of the formalization of Howe's method is the same as in locally nameless, but with much simpler proofs in the name restriction cases of the pseudo-simulation lemma. Indeed, in the same setting as at the end of Section 4.7, inverting $\nu.P_2 \xrightarrow{b} C_2$ gives us simply $P_2 \xrightarrow{b+1} C_2$,

on which we can apply the induction hypothesis to get $Q_2 \xrightarrow{b+1} C_2'$ for some C_2'. In turn, we deduce $\nu.Q_2 \xrightarrow{b} \nu.C_2'$ and conclude the proof in that case.

As predicted, going from a locally nameless representation to a plain de Bruijn makes the proof significantly shorter, going from 5k lines of code to 3k lines of code. The proof is also simpler in the cases involving name restriction, as we no longer have existentially quantified terms depending on the choice of cofinitely quantified names. But we still need renaming lemmas, and their proof is still difficult for the bisimilarity and its open extension, involving results about injective functions.

In the end, the difficulty of a de Bruijn representation (be it locally nameless or plain de Bruijn) depends on how comfortable the proof developer is to work with de Bruijn indices. Non-trivial computations on indices arise not only in the boilerplate part of the development—results about the binding structure itself—but also sometimes for the lemmas and theorems we want to prove in the first place. In our case, it happens not so much in Howe's proof, but more while showing that structural congruence is a bisimulation (cf Section 4.4).

6 Nominal

When working on paper, where binding constructs are represented using explicit names, we often assume Barendregt's conventions—stating that names under consideration are distinct—to simplify manipulations and avoid capture. The nominal representation [45] keeps the formalization of terms close to the standard pen-and-paper version, except that the implicit reasoning up to α-conversion and the naming conventions usually applied now need to be explicit.

6.1 Syntax

The `nu` constructor now takes as extra argument an explicit name.

```
Definition name := var.

Inductive proc (V:Set): Set :=
(* ... *)
| pr_nu  : name → proc V → proc V (*nu a, P*)
```

The process $\nu a.(\overline{a}!(\oslash).\oslash \parallel \overline{b}!(\oslash).\oslash)$ is written `nu a, (a!(PO) PO // b!(PO) PO)`.

Unlike the locally nameless and de Bruijn indices approaches, α-convertible processes like $\nu a.(a?\oslash)$ and $\nu b.(b?\oslash)$ are not equal. Instead, we define α-equivalence as a quotient structure that is built on top of *swapping*, a more general operation than renaming. The swapping of a and b is written $[a \leftrightarrow b]P$. While renaming applies only to free names, swapping also transforms bound names. For example, swapping a and b in $P \triangleq \nu a.(a?X.X \parallel b?Y.\oslash \parallel \overline{c}!(\oslash).\oslash)$ produces $[a \leftrightarrow b]P = \nu b.(b?X.X \parallel a?Y.\oslash \parallel \overline{c}!(\oslash).\oslash)$: we see that the bound name a has been exchanged with the free name b, while c has been left untouched. The resulting process is not α-equivalent to P, but we can obtain an α-equivalent process by swapping a with a name d that is not free in P. As swapping is more general and homogeneous than renaming, more properties are true for swapping than for renaming, and proofs are usually simpler.

We first define swapping on names and processes as follows (🐓).

```
Definition swap_aux (b:name) (c:name) (a:name) :=
  if (a == b) then c else if (a == c) then b else a.

Fixpoint swap {V:Set} (b:name) (c:name) (P:proc V) : proc V:=
  match P with
  | pr_var x ⟹ pr_var x
  | a ? P' ⟹ (swap_aux b c a) ? (swap b c P')
  | a !(P') Q ⟹ (swap_aux b c a) !(swap b c P') (swap b c Q)
  | P' // Q ⟹ (swap b c P') // (swap b c Q)
  | nu a, P' ⟹ nu (swap_aux b c a), (swap b c P')
  | PO ⟹ PO end.
```

We then define α-equivalence using the fact that P is equivalent to $[b \leftrightarrow c]P$ when b and c are not free in P (🐓).

```
Inductive aeq {V:Set}: proc V → proc V → Prop :=
  | aeq_var: ∀ x, aeq (pr_var x) (pr_var x)
  | aeq_inp: ∀ a (P Q : proc (incV V)), @aeq (incV V) P Q → aeq (a ? P) (a ? Q)
  | aeq_snd: ∀ a P1 P2 Q1 Q2, aeq P1 Q1 → aeq P2 Q2 →
             aeq (a !(P1) P2) (a !(Q1) Q2)
  | aeq_par: ∀ P1 P2 Q1 Q2, aeq P1 Q1 → aeq P2 Q2 → aeq (P1 // P2) (Q1 // Q2)
  | aeq_nu_same: ∀ a P Q, aeq P Q → aeq (nu a, P) (nu a, Q)
  | aeq_nu_diff: ∀ b c P Q, b<>c → b \notin fn Q → aeq P (swap b c Q) →
                 aeq (nu b, P) (nu c, Q)
  | aeq_nil: aeq PO PO.
```

Most rules defining α-equivalence are compatibility rules, except for name restriction, where we distinguish two cases. With the same name on each side (rule `aeq_nu_same`), we simply ask for the sub-terms to be equivalent. Otherwise (rule `aeq_nu_diff`), to equate $\nu b.P$ with $\nu c.Q$, we compare P and $[b \leftrightarrow c]Q$, provided b is fresh w.r.t. Q. Indeed, if b is fresh, then we know that $\nu c.Q =_\alpha [b \leftrightarrow c]\nu c.Q = \nu b.[b \leftrightarrow c]Q$, and so deciding α-equivalence comes down to comparing P and $[b \leftrightarrow c]Q$. The freshness condition is important, as we don't want processes such as $\nu b.(c?X.X)$ and $\nu c.(b?X.X)$ to be considered α-equivalent.

We use respectively `P =A= Q` and $P =_\alpha Q$ as notations for α-equivalence in Coq and in the paper. The main drawback of the nominal representation is the additional requirement that some functions preserves α-equivalence. To do so, we usually rely on *equivariance* results of the form $f\ [b \leftrightarrow c]x = [b \leftrightarrow c](f\ x)$ or $f\ [b \leftrightarrow c]x =_\alpha [b \leftrightarrow c](f\ x)$, stating that f commutes with swapping, like, e.g., for the `mapV` function (🐓,🐓) (cf. Section 3).

```
Lemma swap_mapV: ∀ (V W:Set) (P:proc V) b c (f:V → W),
  swap b c (mapV f P) = mapV f (swap b c P).
Lemma aeq_mapV : ∀ (V W:Set) (P Q:proc V) (f:V → W),
  P =A= Q → (mapV f P) =A= (mapV f Q).
```

6.2 Process Substitution

As before, process substitution relies on a `bind` operation such that `bind f P` replaces the free variables of P by their image by f. We have to be careful to avoid capturing the free names that appear in f by the name restrictions of P. In a de Bruijn indices based representation, it means shifting f; with nominal, we need to recursively rename the bound names of P before substituting, which leads to a more complex definition (🐓).

```
Fixpoint bind_rec {V W:Set} (n:nat) (f:V → proc W)
             (N:fset name) (P:proc V): proc W :=
  match n with
  | 0 ⟹ PO
  | S m ⟹  match P with
        | a ? P'     ⟹ a ? (bind_rec m (liftV f) N P')
        (* ... *)
        | nu a, P'   ⟹ let (b,_) := pick_fresh N in
                        nu b, (bind_rec m f (add b N) (swap a b P')) end
  end.

Definition bind {V W:Set} (f:V → proc W) (N:fset name) (P:proc V): proc W :=
  bind_rec (size P) f N P.

Definition subst {V: Set} (P: proc (incV V)) (Q: proc V): proc V :=
  bind (subst_func Q) (fn P \u fn Q) P.
```

In the case $P = \nu a.P'$, we avoid the capture of the names in f by swapping a with a fresh name b before performing the binding operation. As f is of type `V → proc W`, if `V` is infinite, the set of free names in the image of f may itself be infinite. The `bind` operation is defined only for a function f such that the free names of the image of f are in a finite set N, added as a parameter of the `bind` function. The name b is then chosen to be fresh from N—the term `pick_fresh N` generates a name `b` fresh from `N` as well as the proof that `b \notin N`. When doing a substitution `subst P Q`, the set N is instantiated with the free names of both processes. Note that `bind` is applied recursively to `swap a b P'`, which is not a subterm of P. We therefore define `bind` implicitly by induction on the size of the process, using an auxiliary function with a parameter n expected to be larger than the size of P.

The definition of `bind` is hard to reason about, mainly because of the additional set N tracking forbidden names. We want some flexibility about this set, whether when swapping names for equivariance results, or to increase it to encompass a new set of names. We expect `bind f N P` to behave as `bind f M P` as long as the two finite sets are "big enough", i.e., contains the free names of f and P (🦫).

```
Lemma aeq_bind_2: ∀ V W (P: proc V) (f: V → proc W) (N M: fset name),
  (∀ x, fn (f x) \c M)-> fn P \c M →
  (∀ x, fn (f x) \c N)-> fn P \c N →
  (bind f N P) =A= (bind f M P).
```

In practice, for every use of a lemma involving `bind`, we have to explicitly give the appropriate set N and check the "big enough" properties. See the explanations on `howe_bind` in Section 6.6 for an example.

6.3 Semantics

The main difficulty in formalizing the semantics is handling concretions. We represent $\nu b_1, \ldots, b_n.\langle P\rangle Q$ as `conc_def L P Q`, where L is the list $b_1, \ldots, b_n$ (🦫).

```
Inductive conc: Set :=
| conc_def: list name → proc0 → proc0 → conc.
```

Because we define concretions as new objects with a different binding structure than processes, we cannot reuse the α-equivalence we define on processes for these new terms. Thus we define two other swapping operations for lists and concretions (🦫,🦫) as well as a new α-equivalence relation for concretions (🦫).

```
Inductive aeq_conc: conc → conc → Prop := (* C =Ac= C' *)
  | aeq_conc_nil: ∀ P P' Q Q', P =A= P' →
      Q =A= Q' → aeq_conc (conc_def nil P Q) (conc_def nil P' Q')
  | aeq_conc_cons_1: ∀ L L' P P' Q Q' a,
      aeq_conc (conc_def L P Q) (conc_def L' P' Q') →
      aeq_conc (conc_def (a::L) P Q) (conc_def (a::L') P' Q')
  | aeq_conc_cons_2: ∀ L L' P P' Q Q' b c, b<>c →
      b \notin fn_agent (ag_conc (conc_def L' P' Q')) →
      aeq_conc (conc_def L P Q) (swap_c b c (conc_def L' P' Q')) →
      aeq_conc (conc_def (b::L) P Q) (conc_def (c::L') P' Q').
```

The definition is done by induction on the list of names: if it is empty, we simply ask for the messages and continuations to be α-equivalent. Otherwise, as for processes, there are two rules for adding a new restriction, depending on whether the same name is used or not. The rule `aeq_conc_cons_2` allows to add different names on each side, provided that the new name on the left is fresh w.r.t. the concretion on the right. We then extend α-equivalence to agents (🦕), noted `=Ag=`, using `=A=` and `=Ac=`.

We define concretions in Section 2 with a set of names and not a list because the ordering is not important for the semantics. In contrast, we formalize them with an ordered list to simplify the definition of α-equivalence. A proof of α-equivalence with sets would require an explicit matching between the sets, while lists induce a matching given by the ordering. For instance, it is easy to check that $\nu a.\nu b.\nu c.\langle a?X.X \parallel \overline{b}!(c?Y.Y).\oslash\rangle\oslash$ is α-equivalent to $\nu b.\nu a.\nu d.\langle b?X.X \parallel \overline{a}!(d?Y.Y).\oslash\rangle\oslash$ with the above rules. However, to verify that $\nu\{a,b,c\}.\langle a?X.X \parallel \overline{b}!(c?Y.Y).\oslash\rangle\oslash$ is α-equivalent to $\nu\{a,b,d\}.\langle b?X.X \parallel \overline{a}!(d?Y.Y).\oslash\rangle\oslash$, we need to construct the matching $\{(a,b),(b,a),(c,d)\}$, as the a on the left needs to be swapped with the b on the right.

Not all concretions represented by the `conc` datatype can be obtained in the semantics, so we filter out the invalid ones with a well-formedness predicate `conc_wf` (🦕).

```
Definition conc_wf C := match C with
| conc_def L P Q  ⟹  NoDup L ∧ list_to_fs L \c fn P end.
```

As in Section 4, we verify that C respects the lazy scope extrusion discipline by checking that the names in L are free in P; the function `list_to_fs` simply turns L into a finite set. The predicate `NoDup L` states that the names in L are pairwise distinct, as expected when binding several names at once.

Extending parallel composition and name restriction to concretions requires extra checks to avoid capture, as we can see in Figure 1. We define an auxiliary function `conc_convert` which α-converts a concretion C relatively to a finite set M, so that the bound names of the α-converted C are disjoint from M. It satisfies (🦕,🦕):

```
Lemma aeq_conc_convert: ∀ C M, C =Ac= conc_convert C M.
Lemma fn_conc_convert: ∀ C (M:fset name) (L:list name) P Q,
    conc_convert C M = conc_def L P Q → (list_to_fs L) \n M [=] \{}.
```

We then define parallel composition and name restriction by using first `conc_convert` to prevent any capture (🦕,🦕).

```
Definition conc_parl C P := match conc_convert C (fn P) with
  | conc_def L Q R  ⟹  conc_def L Q (R // P) end.
```

```
Definition conc_new a C := match conc_convert C \{a} with
  | conc_def L P Q ⟹ If a \in (fn P) then conc_def (a::L) P Q
                                     else conc_def L P (nu a, Q) end.
```

We show that these operations are equivariant and preserve α-equivalence. For instance we have:

```
Lemma swap_conc_parl: ∀ b c C R,
    swap_c b c (conc_parl C R) =Ac= conc_parl (swap_c b c C) (swap b c R).

Lemma aeq_conc_parl: ∀ C C' R R', C =Ac= C' → R =A= R' →
    conc_parl C R =Ac= conc_parl C' R'.
```

Lemmas on concretions using these extended operations can be tedious to prove, because we need to check that a property true for a given concretion still holds for its α-converted counterpart. As an example, here is a proof sketch for the α-equivalence result for name restriction (🦕).

```
Lemma aeq_conc_new: ∀ C C' a, C =Ac= C' → conc_new a C =Ac= conc_new a C'.
```

Proof (Sketch) By definition of `conc_new`, we first compute C_a and C'_a, the α-converted versions of C and C' with respect to a, for which we know that $C =_\alpha C_a$ and $C' =_\alpha C'_a$. We then distinguish cases based on whether a should be extruded or not: to this end, we prove that a should be extruded in C_a iff it is also extruded in C'_a. We do so by showing that a occurs in the message of C_a iff it occurs in the message of C'_a, based on the fact that $C_a =_\alpha C =_\alpha C' =_\alpha C'_a$.

While the proofs of these compatibility results may require some work, they are quite simple to use afterwards, as there is no freshness condition to check.

Once all these operations on concretions are defined, we can extend them to agents and then formalize the LTS itself by simply writing the rules of Figure 1. In particular, the name restriction case is straightforward (🦕).

```
Inductive lts: proc0  → label  → agent  → Prop :=
  (* ... *)
| lts_new : ∀ a P l A, a \notin fn_lab l → lts P l A →
      lts (nu a, P) l (new a A).
```

We then check that the LTS produces only well-formed concretions and respects swapping and α-equivalence (🦕,🦕,🦕).

```
Lemma lts_conc_wf: ∀ P l (C:conc), lts P l C → conc_wf C.

Lemma swap_lts: ∀ P l A b c, lts P l A →
    ∃ A', A' =Ag= (swap_ag b c A) ∧ lts (swap b c P) (swap_lab b c l) A'.

Lemma aeq_lts: ∀ P P' l A, P =A= P' → lts P l A → ∃ A', A =Ag= A' ∧ lts P' l A'.
```

6.4 Bisimulation

Bisimilarity is defined as in de Bruijn based representations. Proving that bisimilarity is stable by renaming is much simpler in nominal than in the other two formalizations, as we can show the more general result that bisimilarity is equivariant. A first step is to prove that α-equivalence itself is a bisimulation (🦕).

```
Lemma aeq_bisimulation : bisimulation aeq.
```

The proof is simple as we already know that the LTS itself respects α-equivalence. We then show that bisimilarity is preserved by swapping and α-equivalence (🐓).

```
Definition mod_aeq {V:Set} (Rel:binary (proc V)) := ∀ (P P' Q Q': proc V),
    Rel P Q → P=A=P' → Q=A=Q' → Rel P' Q'.
Definition mod_swap {V:Set} (Rel:binary (proc V)) := ∀ (P Q: proc V) b c,
    Rel P Q → Rel (swap b c P) (swap b c Q).

Lemma bisim_aeq: mod_aeq bisimilarity.
Lemma bisim_swap: mod_swap bisimilarity.
```

Compared to locally nameless or de Bruijn indices, there are no freshness conditions on b and c in definition of `mod_swap` (🐓). Carrying such side conditions throughout the proofs is what makes the proofs difficult in the other two representations. As explained before, swapping is more general than renaming, so b and c do not need to be fresh in nominal, and the proof of Lemma `bisim_swap` simply relies on the fact that the LTS is preserved by swapping. In the end, the proof of `bisim_swap` is 20 lines long (🐓), while the renaming proof is 300 lines long in locally nameless (🐓).

The definition of open extension is a bit more complex than in Section 4 or 5, because the `bind` function requires an additional parameter N and the assumption that this N is big enough (🐓).

```
Definition open_extension {V:Set} (Rel:binary proc0) (P Q:proc V) :=
    ∀ (f: V→ proc0) N, fn P \c N → fn Q \c N →
    (∀ x, fn (f x) \c N) → Rel (bind f N P) (bind f N Q).
```

With the previous results on `bind`, we show that the open extension $\mathcal{R}^\circ$ is stable by swapping and α-equivalence if $\mathcal{R}$ is also stable (🐓,🐓).

```
Lemma mod_swap_oe {V:Set}: ∀ Rel, mod_aeq Rel → mod_swap Rel →
    mod_swap (@open_extension V Rel).
Lemma mod_aeq_oe {V:Set} : ∀ Rel, mod_aeq Rel →
    mod_aeq (@open_extension V Rel).
```

If swapping makes the proofs simpler than with de Bruijn indices on one hand, the manipulations of the sets N required by the `bind` operation complexify them on the other hand.

6.5 Structural Congruence

We can see the benefits of the nominal approach when formalizing structural congruence, noted `struct_congr` or `=sc=`. Since the representation is so close to the pen-and-paper definitions, all we need to do is write the rules of Figure 1 and add an explicit α-equivalence rule `sc_aeq`, which is implicit in Figure 1 (🐓).

```
Inductive struct_congr {V:Set}: binary (proc V) := (* =sc= *)
| sc_scope: ∀ a P Q, a \notin fn Q → struct_congr (nu a, (P // Q)) ((nu a, P) // Q)
| sc_nu_nu: ∀ a b P, struct_congr (nu a, (nu b, P)) (nu b, (nu a, P))
(* ... *)
| sc_nu: ∀ a P Q, struct_congr P Q → struct_congr (nu a, P) (nu a, Q)
| sc_aeq: ∀ P Q, P =A= Q → struct_congr P Q.
```

As in the other formalizations, we prove that the restriction of structural congruence to closed processes, noted `=sc0=`, is a bisimulation. The proof proceeds by induction on the derivation of $P \equiv Q$ and by case analysis on the transition performed by P. We first need a couple lemmas showing the interactions between structural congruence and pseudo-application $F \bullet C$. Notably, we extend the rule `sc_scope` to abstractions and concretions (🦕) (we only show one such lemma here):

```
Lemma sc_appr_nu_indep_left: ∀ x (P:proc1) (C:conc), conc_wf C →
  x \notin fn_agent C → appr (nu x, P) C =sc= nu x, (appr P C).
```

The process `P` in Lemma `sc_appr_nu_indep_left` is of type `proc1`, meaning that it may have a free process variable, so it stands for an abstraction.

Most cases of the bisimulation proof are straightforward. The only difficulty is the freshness of names when dealing with rules `sc_scope` and `sc_nu_nu`: the lemma `sc_appr_nu_indep_left` above is only valid for a x fresh for C, a property that may not be verified by the concretions we use in the simulation clause for abstractions. For instance, when trying to prove $(\nu a.\nu b.F) \bullet C \equiv (\nu b.\nu a.F) \bullet C$ for an arbitrary C, we first have to pick fresh names x and y and transform the goal into $(\nu x.\nu y.[a \leftrightarrow x][b \leftrightarrow y]F) \bullet C \equiv (\nu y.\nu x.[a \leftrightarrow x][b \leftrightarrow y]F) \bullet C$. Then we can apply the aforementioned lemma to reduce it to $\nu x.\nu y.([a \leftrightarrow x][b \leftrightarrow y]F \bullet C) \equiv \nu y.\nu x.([a \leftrightarrow x][b \leftrightarrow y]F \bullet C)$ and conclude.

6.6 Howe's Method

Formalizing Howe's closure so that it includes α-equivalence is a bit more complicated than doing so for structural congruence. The reason is that Howe's closure is not a stand-alone relation, as it is built out of a given relation $\mathcal{R}$, so the closure $\mathcal{R}^\bullet$ is stable by α-equivalence only if $\mathcal{R}$ itself is stable (🦕).

```
Lemma mod_aeq_howe {V:Set}:
  ∀ Rel, refl Rel → mod_aeq Rel → mod_aeq (@howe V Rel).
```

If $\mathcal{R}$ contains α-equivalence, then we can already show that $P\ \mathcal{R}^\bullet\ Q$ and $Q =_\alpha Q'$ implies $P\ \mathcal{R}^\bullet\ Q'$, because of the built-in right-transitivity of Howe's closure with open extension. However, we cannot conclude on the left, as $P\ \mathcal{R}^\bullet\ Q$ and $P' =_\alpha P$ imply at best $P'\ \mathcal{R}^\bullet\ P\ \mathcal{R}^\bullet\ Q$, but Howe's closure is not transitive in general.

To be able to use α-equivalence on the left, we modify the compatibility rule for name restriction by adding an α-conversion on the first process (🦕):

```
Inductive howe {V:Set}: binary proc0 → proc V → proc V → Prop :=
| howe_comp: ∀ Rel P Q R, howe Rel P R →
      open_extension Rel R Q → howe Rel P Q
(* ... *)
| howe_nu: ∀ Rel b c P Q, c \notin fn (nu b, P) →
      howe Rel P Q → howe Rel (nu c, (swap b c P)) (nu b, Q).
```

Compared to the formalization of α-equivalence, the name restriction rule does not distinguish the case $b = c$ from the case $b \neq c$. In fact, it turns out that most inductive proofs on Howe's closure do not need a case disjunction on whether b and c are equal, so we factorize these two possibilities into one case for convenience.

This definition allows us to prove Lemma `mod_aeq_howe` by a straightforward induction on the definition of Howe's closure, and also that $\mathcal{R}^\bullet$ is equivariant

assuming $\mathcal{R}$ is equivariant. Proving that $\mathcal{R}^\bullet$ is substitutive requires more work, as always when `bind` is involved, because we need hypotheses stating that the set of forbidden names N required by the definition of `bind` is big enough (🦥).

```
Lemma howe_bind {V W:Set}: ∀ Rel (P Q:proc V) (f g:V → proc W) N,
    (∀ x, fn (f x) \c N) → (∀ x, fn (g x) \c N) →
    fn P \c N → fn Q \c N → refl Rel → mod_aeq Rel → mod_swap Rel →
    howe Rel P Q → (∀ x, howe Rel (f x) (g x)) →
    howe Rel (bind f N P) (bind g N Q).

Lemma howe_subst {V:Set}: ∀ Rel (P' Q':proc V) P Q,
    refl Rel → mod_aeq Rel → mod_swap Rel →
    howe Rel P Q → howe Rel P' Q' → howe Rel (subst P P')(subst Q Q').
```

Given $P\ \mathcal{R}^\bullet\ Q$ and a set N containing the free names of f, g, P and Q, we want to show that `howe Rel (bind f N P) (bind g N Q)`. The proof is by induction on $P\ \mathcal{R}^\bullet\ Q$, and the problematic case is `howe_comp`, i.e., when $P\ \mathcal{R}^\bullet\ R\ \mathcal{R}^\circ\ Q$ for a given R. We cannot apply the induction hypothesis for $P\ \mathcal{R}^\bullet\ R$, as we do not know if N contains the free names of R. We thus apply the induction hypothesis with $M \triangleq N \cup \mathsf{fn}(R)$, so that in the end we obtain `howe Rel (bind f M P) (bind g M Q)`. We can then deduce that `howe Rel (bind f N P) (bind g N Q)` with α-equivalence. All the manipulations involving α-equivalence and checking the hypotheses about the sets M and N makes this proof more tedious in nominal than in the other formalizations.

The proof of the pseudo-simulation lemma exhibits the same kind of issues. First of all, we have to include extra hypotheses about the finite set N (🦥). Second, the `howe_comp` case suffers from the same drawback as previously: we need to work with an intermediate bigger finite set, which leads to several α-equivalence and set manipulations. Apart from this, the proof is shorter than in locally nameless in the name restriction cases, as we do not need to open terms anymore, but slightly longer than with plain de Bruijn indices, because of the extra hypotheses on N we have to check when applying the induction hypothesis.

6.7 Conclusion

The formalization of the syntax and semantics of the language is straightforward with explicit channel names. In particular, we do not need to do arithmetic on indices or cofinite quantification to define the LTS or structural congruence. The main benefit of the nominal representation is that swapping lemmas are much easier to prove than renaming lemmas, because their statements do not require freshness hypotheses on names. This shortens some proofs considerably, especially for the bisimilarity.

These advantages are at the cost of two explicit α-equivalences, which should be preserved by any relation or operation on processes or concretions. These results are often simple, but still take an important part of the development and still need to be checked for any future definition. Besides, the nominal representation does not interact well with the nested datatypes representation of de Bruijn indices we use for process variables, as we can see with the `bind` function: the additional finite set and its manipulation complicate several proofs, notably the results about Howe's closure.

With 4k lines of code, the nominal formalization is shorter than the locally nameless formalization (5k lines of code). It is longer than the 3k lines of code of the de Bruijn representation; most of the difference is due to the definitions of the several α-equivalences and the necessary morphisms. Once this is set up, the effectiveness of the two formalizations are comparable for advanced results.

7 Weak Higher-Order Abstract Syntax

In Higher-Order Abstract Syntax (HOAS) [42], the binding constructs of the object language (here, HOπ) are encoded using the constructs of the meta-language (here, Coq), benefiting from the infrastructure of the theorem prover to get substitution and α-conversion for free. We first remind the principles behind *weak HOAS* [18, 25, 14], the HOAS variant we use to represent name restriction.

7.1 Syntax and Theory of Contexts

The restriction operator binds names, which are not themselves terms of the language in most process algebras. Such a binder can therefore be represented at the meta-level by a function from names to terms, as done in weak HOAS.[3] We fix a set of names and define name restriction as follows.

```
Parameter name : Set.

Inductive proc (V : Set) : Set :=
(* ... *)
| pr_nu  : (name → proc V) → proc V (*nu P*)
```

For example, the process $\nu a.(\overline{a}!(\oslash).\oslash \parallel \overline{b}!(\oslash).\oslash)$ is represented as

```
nu (fun a ⟹ a!(PO) PO // b!(PO) PO)
```

In the paper, we call functions of type `name → proc V` *process functions* and we denote them using blackboard letters $\mathbb{P}$, $\mathbb{Q}$, ..., writing $\nu.\mathbb{P}$ for a restricted process formalized in HOAS. We write $\mathbb{P}\ a$ for the application of such a function to a; we use η-expansion to make bound names apparent when needed, writing $\nu.\mathbb{P}$ also as $\nu.(a \mapsto \mathbb{P}\ a)$, assuming a fresh from $\mathbb{P}$.

Because of the expressiveness of Coq meta-language, `name` cannot be an inductive datatype, as it would then allow *exotic terms*, meta-level terms with no counterpart at the object level [18]. Indeed, a function `name → proc V` could then discriminate on its argument and return different processes. For example, taking `name` as `nat`, one could write a process `nu (fun a ⟹ if (a =? 0) then PO else a ? PO)` which cannot be written in HOπ.

Since `name` is arbitrary, we can no longer define the set of free names using a library like in the previous formalizations, as finite set libraries require properties of the base type `name` that we do not have. Instead, we define membership predicates `isin` (🐓) and `notin`, to state whether a name occurs in a process or not. In the paper, we still write these predicates as respectively $a \in \mathsf{fn}(P)$ or $a \notin \mathsf{fn}(P)$.

[3] Regular HOAS, which relies on functions from terms to terms, cannot be used in Coq to define the syntax of some object language L, as inductive types of the form $(L \to L) \to L$ are not allowed [18].

```
Inductive notin {V:Set} (b:name): proc V → Prop :=
| notin_var: ∀ v, notin b (pr_var v)
| notin_nil: notin b PO
| notin_inp: ∀ a P, a <> b → @notin (incV V) b P → notin b (a ? P)
| notin_snd: ∀ a P Q, a <> b → notin b P → notin b Q
                                    → notin b (a!(P) Q)
| notin_par: ∀ P Q, notin b P → notin b Q → notin b (P//Q)
| notin_nu : ∀ P, (∀ a, a<>b → notin b (P a)) → notin b (nu P).
```

The definition is by induction on the structure of the process, considering any possible instantiation in the restriction case. In the following, we often use the higher-order version of the `notin` predicate, written $a \notin \mathsf{fn}(\mathbb{P})$ in mathematical proofs.

```
Definition notin_ho {V:Set} a (P:name → proc V) := ∀ b, a<>b → notin a (P b).
```

We cannot prove results about names, e.g., on the relationship between `notin` and `isin`, without assuming properties for the type `name`. The *Theory of Contexts* [26] regroups the axioms about names and process functions[4] that we need for a weak HOAS formalization. The axioms have been proved to be consistent [8], but can also be derived from the axiom of choice [47]. For this paper, we write the axioms directly instead of deriving them, to keep the development as close to HOπ as possible and not make it artificially bigger with results not related to the calculus. As a result, we do not use the axiom of choice for the HOAS formalization, in contrast with our other developments.

For names, we assume that we can decide their equality, and that they form an infinite set.

```
Axiom dec_name: ∀ a b:name, a=b ∨ a<>b.
Axiom unsat: ∀ {V:Set} (P:proc V), ∃ a, notin a P.
```

The `unsat` (for "unsaturated") axiom allows to generate a name fresh w.r.t. a given process. We can generalize it and prove that we can pick a name fresh from a list of names `L`, as we can build a process containing the names in `L` (🐘).

We assume extensionality for process functions, meaning that two functions are equal if they are equal when applied to a fresh name. We also suppose that given a process P and a name a, we can write P as $\mathbb{P}\ a$ so that a does not occur in $\mathbb{P}$. We assume this β-expansion not only for processes but also for (second-order) process functions.

```
Axiom proc_ext: ∀ {V:Set} a  (P Q:name→proc V), notin_ho a P →
  notin_ho a Q → (P a = Q a) → P = Q.

Axiom beta_exp: ∀ {V:Set} a (P:proc V),
  ∃ P', notin_ho a P' ∧ P = P' a.
Axiom ho_beta_exp: ∀ {V:Set} a (P:name → proc V),
  ∃ P', notin_ho a (fun x ⟹ nu (P' x)) ∧ P = P' a.
Axiom ho_ho_beta_exp: ∀ {V:Set} a (P:name → name → proc V),
  ∃ P', notin_ho a (fun x ⟹ nu (fun y ⟹ nu (P' x y))) ∧ P = P' a.
```

The axioms enable some structural reasoning on process functions, but are still not enough to handle the name restriction case, where several renamings may be necessary. We therefore define a size on processes and prove it is preserved by renaming.

[4] Process functions can be seen as contexts, since they map name to terms.

```
Inductive size {V:Set} : proc V → nat → Prop :=
| sz_var: ∀ v, size (pr_var v) 0
| sz_nil: size (@pr_nil V) 0
| sz_inp: ∀ a P n, @size (incV V) P n → size (a ? P) (S n)
| sz_out: ∀ a P Q n m, size P n → size Q m → size (a!(P) Q) (S (n+m))
| sz_par: ∀ P Q n m, size P n → size Q m → size (P//Q) (S (n+m))
| sz_nu : ∀ P n, (∀ x, size (P x) n) → size (nu P) (S n).

Lemma size_rename {V:Set}: ∀ n (P:name→ proc V) a, notin_ho a P →
  size (P a) n → ∀ b, size (P b) n.
```

The size cannot be defined as a recursive function on processes using `Fixpoint`, because of the restriction case. Instead, we define a relational predicate `size P n`, meaning that the size of P is n. A drawback is that we have to prove that the relation is total (🦃), and size expressions cannot be reduced with the `simpl` tactic, but have to be inverted instead.

On the positive side, we see that writing a renaming lemma is very simple in HOAS, as we just apply the process function to the two different names. We do not have to define and prove properties of an auxiliary renaming function, like `open` and `close` in locally nameless, `mapN` in plain de Bruijn, or `swap` in nominal. The proofs also benefit from Coq built-in mechanisms. For example, rewriting a goal `P b c = Q b c` as `(fun a ⟹ P a c) b = (fun a ⟹ Q a c) b` is frequent to apply a renaming hypothesis. In HOAS, it can be done with the `change` tactic, which simply checks that the expressions are indeed β-convertible. Doing the same in locally nameless, plain de Bruijn, or nominal requires several rewritings and properties about the commutativity of their respective renaming functions.

The proof of `size_rename` (🦃) is typical of most HOAS proofs in Coq. We proceed by induction on n, as Coq do not provide a higher-order induction principle on $\mathbb{P}$. Inverting the size hypothesis, we get for instance in the parallel case P_1 and P_2, such that $\mathbb{P}\ a = P_1 \parallel P_2$. With the `beta_exp` axiom, we can decompose P_1 and P_2 into $\mathbb{P}_1\ a$ and $\mathbb{P}_2\ a$ so that $a \notin \mathsf{fn}(\mathbb{P}_1) \cup \mathsf{fn}(\mathbb{P}_2)$. We can then rewrite the previous equality into $\mathbb{P}\ a = (x \mapsto \mathbb{P}_1\ x \parallel \mathbb{P}_2\ x)\ a$, and deduce that $\mathbb{P} = x \mapsto \mathbb{P}_1\ x \parallel \mathbb{P}_2\ x$ with the `proc_ext` axiom. This sequence of three steps is so pervasive in any HOAS proof, that we define tactics to automate it as much as possible (see, e.g., (🦃)). We can then compute $\mathbb{P}\ b$ and conclude with the induction hypothesis, as $\mathbb{P}_1\ a$ and $\mathbb{P}_2\ a$ are smaller than $\mathbb{P}\ a$.

In the restriction case of the same proof, we have $\mathbb{P}\ a = \nu.\mathbb{P}'$ for some $\mathbb{P}'$. With `ho_beta_exp` and `proc_ext`, we know that $\mathbb{P} = x \mapsto \nu(\mathbb{P}''\ x)$ for some $\mathbb{P}''$ of type `name → name → proc V` such that a does not occur in $\mathbb{P}''$. We want to prove that $\mathbb{P}\ b = \nu(\mathbb{P}''\ b)$ is of size of $n+1$. By definition, we need to show that for all c, $\mathbb{P}''\ b\ c$ is of size n. We can conclude with the induction hypothesis, provided that a does not occur in $x \mapsto \mathbb{P}''\ x\ c$. But c is any name, and can be in particular a; therefore, we must use the induction hypothesis first to rename c into a fresh name (generated using `unsat`), and only then we can apply the hypothesis to change b.

This pattern of having a name not fresh enough is quite common when handling the restriction case. One way to carry on proofs is to systematically reason by induction on the size of terms, and apply renaming lemmas. When defining the semantics in the next section, we take a slightly different path, by using cofinite quantification, as done by Honsell et al. [25]. Doing so reduces (but does not completely eliminate) uses of renaming lemmas: in the above proof, it would allow us to take directly c fresh from a. However, as already experienced with locally nameless

(Section 4), cofinite quantification is useful when proving constructor introduction (as it would be in the above proof, where we proceed to a `size` introduction), but less for elimination. Indeed, inverting a cofinitely quantified hypothesis introduces the finite set in the proof environment, and using this hypothesis then require a name fresh from that set, which may demand extra renamings. Therefore, we do not use cofinite quantification when defining predicates that are often inverted, such as the membership and size ones.

With the size of processes defined, we can also prove renaming lemmas for the membership predicates (🐓,🐓) from which we can deduce expected properties, e.g., that we can decide membership 🐓. We also prove the relationship between these predicates and the monadic operators `mapV` and `bind` by straightforward structural induction on the processes (🐓, 🐓).

7.2 Semantics

The representation of concretions introduces a new binding structure of type `name` → `conc`, called *concretion functions* and ranged over by $\mathbb{C}$.

```
Inductive conc : Set :=
| conc_def: proc0 → proc0 → conc
| conc_nu : (name → conc) → conc.
```

The concretion $\nu ab.\langle\overline{a}!(\overline{b}!(\oslash).\oslash).\oslash\rangle a?X.X$ is written

```
conc_nu (fun a ⟹ conc_nu (fun b ⟹ conc_def (a!(b!(P0) P0) P0) (a ? VZ)))
```

The binding is defined for a new datatype, therefore we cannot reuse what has been defined for processes: like in nominal, we have to redevelop its theory. In particular, we define the non-membership predicate `conc_notin` (🐓) with its corresponding version for functions `conc_notin_ho` (🐓). However, we do not need the opposite membership predicate.

It is also necessary to restate the axioms `conc_unsat` (🐓), `conc_ext` (🐓), and `conc_beta_exp` (🐓) (and its higher-order variants), as well as the corresponding tactics for them (🐓). We define a size for concretions, which corresponds to the number of binders it contains.

```
Inductive sizec : conc → nat → Prop :=
| szc_def: ∀ P Q, sizec (conc_def P Q) 0
| szc_nu : ∀ C n, (∀ a, sizec (C a) n) → sizec (conc_nu C) (S n).
```

We then prove the same properties for `sizec` and `conc_notin` as with processes, in particular renaming lemmas (🐓, 🐓).

Unlike with the other formalizations, the representation of concretions is inductive in HOAS, and destructing a concretion C does not give directly access to the process being sent. It makes defining the well-formedness predicate or performing scope extrusion more difficult, as these operations rely on the names of the message. To check well-formedness, we instantiate the binders, accumulating the names we use in a list L. We then check that the names in L indeed occurs in the message, thanks to the predicate `list_in_proc` below.

```
Inductive list_in_proc {V:Set}: list name → proc V → Prop :=
| list_in_proc_nil: ∀ P, list_in_proc nil P
| list_in_proc_rec: ∀ a L P, list_in_proc L P → isin a P → list_in_proc (a::L) P.
```

```
Inductive cwf_aux: list name → conc → Prop :=
| cwf_def: ∀ L P Q, list_in_proc L P → cwf_aux L (conc_def P Q)
| cwf_nu : ∀ L C L', (∀ a, ¬In a L' → conc_notin_ho a C → cwf_aux (a::L)(C a)) →
                     cwf_aux L (conc_nu C).


Definition conc_wf C := cwf_aux nil C.
```

As hinted at the end of Section 7.1, we define the recursive case of `cwf_aux` using cofinite quantification: the list `L'` represents the names that `a` should be fresh from.

To perform scope extrusion, we no longer define a function, but instead a relational predicate `conc_new`, taking a function $a \mapsto \mathbb{C}\ a$ as argument and returning a concretion C where a is extruded iff it occurs in the message of $\mathbb{C}\ a$. For example, given a concretion $\nu b.\langle\overline{a}!(\overline{b}!(\oslash).\oslash).\oslash\rangle a?X.\oslash$, adding a restriction on a amounts to applying `conc_new` to the function $a \mapsto \nu b.\langle\overline{a}!(\overline{b}!(\oslash).\oslash).\oslash\rangle a?X.\oslash$.

If we consider a function of the form $a \mapsto \langle\mathbb{P}\ a\rangle\mathbb{Q}\ a$, then we compute the result based on whether a occurs in $\mathbb{P}\ a$ for any fresh a. It corresponds to the first two cases of the following definition.

```
Inductive conc_new: (name → conc) → conc → Prop :=
| cnew_extr   : ∀ P Q, (∀ a, notin_ho a P → isin a (P a)) →
                conc_new (fun a ⟹ conc_def (P a)(Q a))
                         (conc_nu (fun a ⟹ conc_def (P a)(Q a)))
| cnew_no_extr: ∀ P Q, (∀ a, notin_ho a P → notin a (P a)) → ∀ b,
                conc_new (fun a ⟹ conc_def (P a)(Q a)) (conc_def (P b) (nu Q))
| cnew_nu     : ∀ (C:name → name → conc) C' L,
                (∀ b, ¬In b L → conc_notin_ho b (fun a ⟹ conc_nu (C a)) →
                conc_new (fun a ⟹ C a b)(C' b)) →
                conc_new (fun a ⟹ conc_nu (C a)) (conc_nu C').
```

If scope extrusion is not needed (case `cnew_no_extr`), the result is $\langle\mathbb{P}\ b\rangle\nu.\mathbb{Q}$ for any b, fresh or not, as we know that the binder in $\mathbb{P}$ is useless. In fact, if for a given a, $a \notin \mathsf{fn}(\mathbb{P}\ a)$, then for all b and c, $\mathbb{P}\ b = \mathbb{P}\ c$ (🦃).

In the inductive case `cnew_nu`, we consider a function $a \mapsto \nu.(b \mapsto \mathbb{C}\ a\ b)$ where $\mathbb{C}$ is of type `name → name → conc`. In the recursive call, we instantiate the already existing binder on b, to focus on the binding on a being added: given a fresh (cofinitely quantified) b, the result of adding the restriction on a to $a \mapsto \mathbb{C}\ a\ b$ should be a concretion of the form $\mathbb{C}'\ b$, so that the result of restricting a in $a \mapsto \nu.(b \mapsto \mathbb{C}\ a\ b)$ is $\nu.\mathbb{C}'$.

For example, to restrict a in

$$\mathbb{C} \triangleq a \mapsto \nu.(b \mapsto \langle\overline{a}!(\overline{b}!(\oslash).\oslash).\oslash\rangle a?X.\oslash),$$

we consider in the recursive call $a \mapsto \langle\overline{a}!(\overline{b}!(\oslash).\oslash).\oslash\rangle a?X.\oslash$ for a given b; since a occurs in the message, we get $\nu.(a \mapsto \langle\overline{a}!(\overline{b}!(\oslash).\oslash).\oslash\rangle a?X.\oslash)$ as a result. Therefore, restricting a in the original function $\mathbb{C}$ generates

$$\nu.(b \mapsto \nu.(a \mapsto \langle\overline{a}!(\overline{b}!(\oslash).\oslash).\oslash\rangle a?X.\oslash)).$$

We see that the binders on a and b have been exchanged in the resulting concretion compared to $\mathbb{C}$. In general, a new binder is added at the innermost position, so that restricting an extruded name a_1 in a concretion $\nu a_2 \ldots a_n.\langle P\rangle Q$ produces $\nu a_2 \ldots a_n, a_1.\langle P\rangle Q$. This behavior differs from the other formalizations, where the

new binder is added at the outermost position, but remains a valid representation of the semantics of HOπ, as the order of the binders in a concretion $\nu a_1 \dots a_n.\langle P\rangle Q$ does not matter. Adding binders while preserving their order is technically possible, but would require a significant number of additional predicates and properties on these predicates. When scope extrusion does not occur, e.g., when restricting a in $a \mapsto \nu.(b \mapsto \langle \overline{b}!(\oslash).\oslash\rangle a?X.\oslash)$, we get $\nu.(b \mapsto \langle \overline{b}!(\oslash).\oslash\rangle \nu.(a \mapsto a?X.\oslash))$ as expected.

The predicate `conc_new` is defined by case analysis on a concretion function, but Coq does not discriminate on functions. As a result, inverting for instance an hypothesis `H: conc_new (fun a ⟹ conc_def (P a)(Q a)) C` generates three goals, each of them corresponding to a case defining `conc_new`. The first two have in their hypotheses an equality of the form

```
(fun a ⟹ conc_def (P a)(Q a)) = (fun a ⟹ conc_def (P' a)(Q' a))
```

while the last one has an equality

```
(fun a ⟹ conc_def (P a)(Q a)) = (fun a ⟹ conc_nu (C a))
```

Proving that `P=P'` and `Q=Q'` in the first two cases or that the last equality is absurd is not difficult and can be automatized (🐓) but it is still an inconvenience.

We prove for `conc_new` totality (🐓) but also uniqueness (🐓), making it effectively a functional predicate. We also prove a renaming lemma (🐓) and show it preserves free names (🐓). The proofs are by induction on the derivation of `conc_new`, except for totality, where we reason on size, as some renaming is necessary.

While extending name restriction to concretions requires some care, doing the same for parallel composition is simple (🐓,🐓), as there is no risk of capturing the names of the process put in parallel. We do not have to shift the process as in de Bruijn representations (cf Section 4.2) or to α-convert the concretion as in nominal (Section 6.3).

```
Fixpoint conc_parl C P: conc :=  match C with
| conc_def P' Q ⟹ conc_def  P' (Q // P)
| conc_nu C' ⟹ conc_nu (fun x ⟹ (conc_parl (C' x) P))
end.
```

Similarly, the pseudo-applications `appl` and `appr` are defined by straightforward induction on the concretion (🐓,🐓).

In contrast with the other formalizations, we do not define the `agent` datatype (which subsumes processes, abstractions, and concretions) in HOAS, as we would have to also manipulate agent functions `name` $\to$ `agent`, and in particular do case analysis on them. As already pointed out with `conc_new`, it is inconvenient to discriminate over functions in Coq. The consequence of this choice is that instead of having one LTS predicate relating processes and agents, we define three predicates between processes and respectively processes `ltsproc` (🐓), abstractions `ltsabs` (🐓), and concretions `ltsconc` (🐓), meaning that some results about the LTS have also to be duplicated. We believe this duplication is still more manageable than any overhead introduced by reasoning and discriminating over agent functions.

A consequence of dropping agents is that the `label` datatype is no longer necessary, as the `ltsabs` and `ltsconc` directly mention the name on which the communication is expected, while `ltsproc` does not need to mention the τ label. We show the restriction cases for `ltsabs` and `ltsconc`, the definition for `ltsproc` is the same as for `ltsabs`.

```
Inductive ltsabs: proc0 → name → abs → Prop :=
(*...*)
| ltsa_new: ∀ P a F L, (∀ b, ¬In b L → a<>b → notin_ho b P →
    notin_ho b F → ltsabs (P b) a (F b)) → ltsabs (nu P) a (nu F).

Inductive ltsconc: proc0 → name → conc → Prop :=
(*...*)
| ltsc_new: ∀ P a C L, (∀ b, ¬In b L → a<>b → notin_ho b P →
    conc_notin_ho b C → ltsconc (P b) a (C b)) →
    ∀ C', conc_new C C' → ltsconc (nu P) a C'.
```

The definitions follow the same structure, relying on a fresh enough, cofinitely quantified name b to instantiate the binder in P. The only difference is the use of `conc_new` in the concretion case to perform scope extrusion if necessary.

Like in the other formalizations, we prove that the LTS produces well-formed concretions (🦃). We also prove that the LTS preserves freshness (🦃, 🦃, 🦃) and renaming lemmas (🦃, 🦃, 🦃) similar to the following one.

```
Lemma ltsproc_rename: ∀ P P' b,
  notin_ho b P → notin_ho b P' → ltsproc (P b) (P' b) →
  ∀ c, ltsproc (P c) (P' c).
```

The proofs are by induction on the derivation of the LTS; we do not need to do several renamings in the name restriction cases thanks to cofinite quantification, as the names that are introduced to instantiate the binders can be chosen to be fresh from in particular b and c.

7.3 Bisimilarity and Structural Congruence

The formalization of the bisimilarity is the same as with the other techniques, except that we replace the `lts` predicate with its variants in the definitions of `test_proc` (🦃), `test_abs` (🦃), and `test_conc` (🦃). The renaming proof is exactly the same as in locally nameless (Section 4.5), with the same side conditions in the definition of `rename_compatible` (🦃), and the same scheme to prove Lemma 6 (🦃). The proof is significantly shorter in HOAS than in locally nameless with 120 lines vs 300 lines, simply because renaming is easier to manipulate in HOAS, as explained in Section 7.1.

The renaming proof for open extension is also the same as in locally nameless (🦃), except we rely on β-expansion axioms for functions (🦃). With the axiom of choice, we could instead extend the `beta_exp` axiom to functions using Hilbert's epsilon operator, but we prefer to use axioms for functions to be consistent with our initial decision of not using the axiom of choice for our HOAS development (cf Section 7.1).

With HOAS, it is straightforward to represent the structural congruence rule permuting name restrictions, and it is relatively simple to prove simulation in that case. The rules modifying scope require more work.

```
Inductive struct_congr {V: Set}: binary (proc V) :=
(*...*)
| sc_scope: ∀ P Q, (∀ a, notin_ho a Q → notin a (Q a)) →
            ∀ b, struct_congr (nu (fun a ⟹ (P a // Q a))) ((nu P) // Q b)
| sc_nu_nu: ∀ P, struct_congr (nu (fun a ⟹ nu (fun b ⟹ P a b)))
                              (nu (fun b ⟹ nu (fun a ⟹ P a b)))
(*...*)
```

In the rule `sc_scope`, we limit the scope of the name restriction to $\mathbb{P}$ provided it does not bind anything in $\mathbb{Q}$; we check the latter condition by verifying that $a \notin \mathbb{Q}\ a$ for all fresh a. Since the binder of $\mathbb{Q}$ is useless, we can instantiate it with any name b in the process on the right.

Proving simulation in the case of `sc_scope` is lengthy because of several factors. First of all, if $\mathbb{Q}$ is making a step (when instantiated), we have to check that the resulting agent has the same property as $\mathbb{Q}$ (🦕, 🦕, 🦕). For example, for processes, we have the following result.

```
Lemma ltsproc_unused_arg : ∀ Q, (∀ a, notin_ho a Q → notin a (Q a)) →
  ∀ Q' b, notin_ho b Q → notin_ho b Q' → ltsproc (Q b)(Q' b) →
  ∀ a, notin_ho a Q' → notin a (Q' a).
```

It is a consequence of the fact that the LTS preserves freshness.

Next, proving any structural congruence relation resulting from `sc_scope` which involves a concretion usually requires a non-trivial induction on the (size of) the concretion or on any `conc_new` hypothesis we might have. For example, if $\mathbb{P}$ is reducing to $\mathbb{C}$, then we have to show that for a given F and b, $F \bullet \nu.(a \mapsto \mathbb{C}\ a \parallel \mathbb{Q}\ a) \equiv F \bullet \nu.\mathbb{C} \parallel \mathbb{Q}\ b$ (🦕).

```
Lemma sc_F_newCQ_F_newC_Q: ∀ F C (C': name → conc) C'' Q,
  (∀ a, notin_ho a Q → notin a (Q a)) →
  conc_new (fun a ⟹ conc_parl (C' a) (Q a)) C → conc_new C' C'' →
  ∀ b, struct_congr (appr F C) (appr F (conc_parl C'' (Q b))).
```

The concretion C differs from C'' only by the process $\mathbb{Q}$ put in parallel in its continuation, but to show this, we have to reason by induction on the size of $\mathbb{C}'$, and do case analyses on the two `conc_new` hypotheses. In the end, the proof is 50 lines long, while it takes less than 10 lines in de Bruijn based formalizations (🦕, 🦕) or in nominal (🦕). We see differences of the same order of magnitude for the other subcases of the `sc_scope` case. In the end, the structural congruence simulation proof is quite long in HOAS (about 1000 lines), even though it benefits from more automation than the same proof with the other techniques.

7.4 Howe's Method

The proofs of the pseudo-simulation lemmas exhibit the same issues with HOAS than with locally nameless in the name restriction cases (cf the end of Section 4.7): inverting a transition $\nu.\mathbb{P} \xrightarrow{\alpha} A$ introduces a cofinitely quantified name b to instantiate the binder. Any entity introduced afterwards then depends on that b, and checking other cofinitely quantified properties for this entity thus requires to rename that b. Again, HOAS handles renaming better than locally nameless, so the proofs are shorter in HOAS.

An exception is the proof of `conc_to_proc`, which states that given a well-formed concretion C and a name a, we can build a process P such that $P \xrightarrow{\overline{a}} C$, and P does not emit an other message on a.

```
Lemma conc_to_proc: ∀ (C:conc), conc_wf C →
  ∀ a, ∃ P, ltsconc P a C ∧ (∀ (C':conc), ltsconc P a C' → C = C').
```

On paper, if $C \overset{\Delta}{=} \nu a_1 \dots a_n.\langle R \rangle Q$, then we pick $P = \nu a_1 \dots a_n.\overline{a}!(R).Q$. The proof is simple with plain de Bruijn or nominal (🦕, 🦕), a bit more involved in locally

nameless (🦕). In HOAS, `conc_new` is adding name restrictions at the innermost position, so the process we are constructing is $\nu a_n \dots a_1.\overline{a}!(R).Q$. As a result, we have to be careful in the proof, as we cannot simply proceed by induction on the concretion.

Indeed, if we apply the induction hypothesis to $\nu a_2 \dots a_n.\langle R\rangle Q$, we get a process P such that $P \xrightarrow{\overline{a}} \nu a_2 \dots a_n.\langle R\rangle Q$, but then $\nu a_1.P \xrightarrow{\overline{a}} \nu a_2 \dots a_n, a_1.\langle R\rangle Q$, because of `conc_new`. Therefore, we have to apply the induction hypothesis to $\nu a_1 \dots a_{n-1}.\langle R\rangle Q$, so that we get $P \xrightarrow{\overline{a}} \nu a_1 \dots a_{n-1}.\langle R\rangle Q$, and then $\nu a_n.P \xrightarrow{\overline{a}} \nu a_1 \dots a_{n-1}, a_n.\langle R\rangle Q = C$, as wished. Hence, the proof should be done by induction on the size of C, and not C itself.

To carry on the proof, we define a `pull` (🦕) operation which, given a well-formed concretion $C \triangleq \nu a_1 \dots a_n.\langle R\rangle Q$, pulls out the innermost binder, i.e., builds the function $\mathbb{C} = a_n \mapsto \nu a_1 \dots a_{n-1}.\langle R\rangle Q$.

```
Inductive pull: conc → (name → conc) → Prop :=
| pull_nudef: ∀ P Q, pull (conc_nu (fun a ⟹ conc_def (P a)(Q a)))
                          (fun a ⟹ conc_def (P a)(Q a))
| pull_nunu : ∀ (C C':name → name → conc) L, (∀ a, ¬In a L →
    conc_notin_ho a (fun b ⟹ conc_nu (C b)) →
    conc_notin_ho a (fun b ⟹ conc_nu (C' b)) →
    pull (conc_nu (C a))(C' a)) → pull (conc_nu (fun b ⟹ conc_nu (C b)))
                                       (fun c ⟹ conc_nu (fun b ⟹ C' b c)).
```

We see that `pull` is defined for concretions with at least one binder. In the `pull_nunu` case, we instantiate the binder on a_1 with a fresh name a to inductively apply `pull`, to get a function of the form $a_n \mapsto \nu a_2 \dots a_{n-1}.\langle \mathbb{R}\ a\rangle \mathbb{Q}\ a$. We then restore the name restriction on a_1 in the final result.

If applying `pull` to C produces $\mathbb{C}$, we show that applying `conc_new` to $\mathbb{C}$ results in C, as wanted (🦕). We also prove that any instantiation of $\mathbb{C}$ is smaller than C (🦕) and well-formed (🦕), meaning that we can apply the induction hypothesis to $\mathbb{C}$ in the proof of `conc_to_proc`. Finally, we have to show that `pull` is total on concretions with at least one binder (🦕), a result which itself requires a renaming lemma (🦕). In the end, we need around 200 lines to prove this simple result.

7.5 Conclusion

In a context where renaming is widespread, being able to express it just with function application simplifies the development a lot: we do not have to define a specific renaming operator (such as `open` and `close`, `mapN` or `swap`) and prove its properties, like commutativity or distributivity. Performing β-expansion on goals or hypotheses to allow for the application of renaming lemmas is easier with the `change` or `pattern` tactics. Manipulating functions is also simpler than computing with de Bruijn indices, but not as straightforward as using plain names, as we can see with the definition of `conc_new`.

However, Coq does not provide the best support for reasoning with functions, especially compared to HOAS-based provers [43,5,44]. We cannot reason by structural induction, but we rather have to rely on induction on sizes, to be able to use renamings in the name restriction case. Cofinite quantification reduces the need for renamings, by giving the possibility of choosing an already fresh enough name, and thus allowing for induction on predicates other than size. Discriminating on

functions is also problematic, as inverting an inductive definition defined by case analysis on functions does not do proper unification. Instead, it produces equalities which are either absurd or can be decomposed further, as experienced when inverting a `conc_new` hypothesis.

Having `name` as a parameter with only axioms about it is also not convenient, as we cannot compute with it or use `simpl` to simplify goals or hypotheses. E.g., we cannot compute the set of free names with finite sets libraries, so we define (non-)membership predicates. Manipulating the extensionality and β-expansion axioms is commonplace in proofs and requires specific tactics to automate their use to make any simple proof tractable.

Most operators defined inductively on syntax cannot be defined using `Fixpoint`, but with an `Inductive` relation predicate instead. We then have to prove totality for that predicate, a proof which itself relies on a renaming lemma to handle the name restriction case. The definition of `pull` is an example of that problem: the main lemma is neither hard nor long to prove (), but the supporting results it requires can grow quite large (, , ,). Having to prove a renaming lemma for each operator on syntax we introduce makes the HOAS formalization close to the nominal one with its equivariance lemmas.

Another problem reminiscent of nominal is the formalization of concretions, as its binding construct requires its own set of axioms, related tactics, and constructs. For example, we have to redefine a non-membership predicate to state that a name is fresh w.r.t. a concretion, and prove its properties. Unlike in the other formalizations, the definition of concretions is an inductive datatype, meaning that destructing a concretion does not directly give access to its message. This complicates the definitions and proofs of operations relying on that message (`conc_new` and `conc_wf`).

Finally, cofinite quantification raises the same issues as in locally nameless when used in simulation proofs, as any entity existentially introduced depends on the chosen name, which then should be renamed. However, thanks to the better support for renaming, the proofs are shorter in HOAS than in locally nameless, as we can see when comparing the name restriction cases when doing Howe's proof (e.g., compare the restriction cases of () and ()). Similarly, the renaming proofs follows the same scheme in HOAS than in locally nameless but are much shorter, as witnessed by the bisimilarity renaming lemma (() vs ()).

In the end, the HOAS formalization shares many issues with the other representations, but its better support for renaming and some dedicated tactics to handle names make the proofs quite tractable. With 3600 lines of code, it is shorter than the locally nameless or nominal formalizations, but still longer than the one relying on plain de Bruijn indices.

8 Assessment

We compare the different techniques we use to represent names. We start by recalling the perks inherent to each method before moving to issues more specific to HOπ. We also discuss the representation of process variables. Table 2 gives the size of each development in lines of code (loc), detailing per item of the formalized theory.

	Locally nameless	de Bruijn	Nominal	Weak HOAS
Tools	250	220	240	0
Syntax	930	240	700	570
Semantics	710	450	1070	820
Bisimulation	470	560	290	310
Structural congruence	1100	670	670	1020
Howe's method	1460	810	1150	880
Total	4920	2950	4120	3600

Table 2 Size of each formalization, in lines of code (specifications and proofs)

The "Tools" line regroups basic results about finite sets, de Bruijn indices, or swapping. The HOAS representation does not rely on any of these, and uses predicates—which depend on the syntax of the calculus—instead of finite sets to handle free names. The "Tools" line is therefore at 0 for HOAS, while its "Syntax" line includes results about names that are proved in "Tools" in the other formalizations.

8.1 Intrinsic Differences

The techniques have their own particularities which are independent of the formalized language. Locally nameless requires to move between de Bruijn indices to names using `open` and `close`, and relies on a predicate to rule out ill-formed terms. Locally nameless and plain de Bruijn indices formalizations require arithmetic operations on indices, while nominal uses swapping and α-conversion. HOAS uses Coq functions to delegate to the theorem prover the bookkeeping of bound names. The HOAS formalization is parametric in the (non-inductive) datatype `name`, whose properties must be stated through axioms.

The formalization of the syntax illustrates these differences. We see in Table 2 that the syntax in locally nameless exceeds the de Bruijn one by around 700 loc, which are all the properties we need to relate `open`, `close`, `is_proc`, and the sets `fn` and `bn`. The syntax is also larger in nominal than in de Bruijn, in part because of the definition of α-equivalence, but also because of the extra work induced by the nominal representation of names when handling process variables (cf Section 6.2), an issue specific to nominal. With 570 loc, the HOAS representation is fairly compact, considering that it should be compared to the sum of the "Tools" and "Syntax" numbers of the the other formalizations. While we do not have to define any renaming operator in HOAS similar to `open`/`close`, `swap`, or `mapN`, we still have to define the membership predicates `isin` and `notin` and prove their properties.

Writing definitions and lemmas is easier in nominal, as the formalized definitions remain faithful to the pen-and-paper ones. Doing systematic α-conversions becomes quickly tedious however, as this operation is usually left implicit on paper. The function-based HOAS representation is more contrasted: it may lead to very simple definitions without any freshness side-conditions, like for `conc_parl` (🐓), but also to definitions that are much less intuitive and do not correspond to the pen-and-paper ones, such as `conc_new` (🐓), or the formalization of the scope extrusion congruence rule `sc_scope` (🐓). In any case, functions and names are usually easier to manipulate than de Bruijn indices.

Each representation comes with proof schemes which may not be well handled by Coq or may conflict with the result being proved. As explained at the end of Section 4.7, cofinite quantification entails many renamings when used with existentially quantified properties, such as simulation. The problem is visible in locally nameless for Howe's method, but also for structural congruence, for which the development is twice as big in locally nameless than in de Bruijn or nominal (cf Table 2). Structural congruence manipulates processes under binders, which are instantiated using cofinite names, and we are proving that structural congruence is a simulation. As a result, we get existentially quantified terms which depend on a cofinite name, which then should be renamed. This is the exact same situation as with Howe's pseudo simulation lemma.

The problem is less acute with HOAS, which also relies on cofinite quantification and where the length of Howe's proof is on par with the de Bruijn one, simply because it handles renamings better than locally nameless. The structural congruence simulation proof is longer in HOAS than in de Bruijn or nominal not so much because of cofinite quantification, but more because of the `sc_scope` case. The HOAS representation suffers more from the lack of induction principle on functions in Coq, or the poor support for case analysis on functions in general (cf Sections 7.1 and 7.2). Another issue specific to HOAS is that any operation on the syntax of processes should be defined as an inductive predicate and not as a recursive function, with again less tactic support for the former than for the latter.

8.2 Representing Concretions

Using de Bruijn indices for bound names simplifies the representation of a concretion $\nu\widetilde{b}.\langle P\rangle Q$, because the set of bound names $\widetilde{b}$ is just represented by its size n. Manipulating concretions then relies mostly on the same operations as for processes—e.g., shifting, but by any n and not only by 1—, except when defining `conc_new`, which needs a specific permutation of indices `permut` because of lazy scope extrusion. Again, the locally nameless representation of the semantics is bigger than in de Bruijn, because of the generalization of the predicate `is_proc` to all agents, and the corresponding proofs of its properties.

We do not use for channel names the nested datatype representation of de Bruijn indices that we use for process variables. Indeed, such a representation does not seem practical enough to work with concretions: to express the fact that P should have at least n free names, we need to write a dependent type of the form `proc (inc n N)` (where `inc N` is similar to the `incV V` type we use for process variables), and such dependent types are hard to reason about since we often do arithmetic on n.

Representing concretions is more difficult in nominal, as we need to bind a list of names. As a result, we have to extend the results about swapping to lists. More importantly, since we define a new binding structure, we need to define its corresponding α-conversion and prove its properties: we cannot simply reuse the one for processes. Extending name restriction to concretions is however simpler than with de Bruijn indices, as it simply mirrors the pen-and-paper definition. The formalization of the semantics ends up to be significantly bigger in nominal than in plain de Bruijn or with locally nameless (cf Table 2).

HOAS suffers from the same issue as nominal, in that we cannot reuse for concretions the axioms about names for processes. We need to rewrite the axioms as well as the non-membership predicate `conc_notin` and proves its properties. In contrast with the other formalizations, the definition of concretions is recursive in HOAS, which implies that we do not have directly access to the emitted process when we have a concretion of the form $\nu.\mathbb{C}$, which in turn complicates the definitions of `conc_new` and `conc_wf`. The formalization of the semantics ends up to be bigger in HOAS than in de Bruijn or locally nameless, but still smaller than nominal, as the meta-theory we have to redevelop is more lightweight (e.g., no α-equivalence).

8.3 Renaming Lemmas

The representations rely on renaming lemmas for different uses. In locally nameless or HOAS, renaming lemmas are expected to be used when opening binders and in conjunction with cofinite quantification. With plain de Bruijn indices, they enable the proof that a given property is preserved by shifting or lifting. Being equivariant, i.e., to be preserved by swapping, a more general form of renaming, is core to the nominal theory, so it is no surprise that equivariance proofs are for the most part straightforward.

The renaming proofs are much more difficult with de Bruijn indices, and even more in locally nameless than with plain de Bruijn. In locally nameless, renaming lemmas are stated in terms of the opening operation $\{k \to a\}P$, with side conditions on a (e.g., fresh w.r.t. P). In plain de Bruijn, renamings are functions from indices to indices, which must be injective to prevent collisions between channel names. Expressing renaming is easy in HOAS, as it consists in applying a function to different names, but we still need side conditions on these names, usually the same as in locally nameless. In nominal, one of the benefits of using swapping over renaming is to not have any side condition on the swapped names. As a result, proving that bisimilarity is preserved by swapping is much easier than proving it is preserved by renaming, as we do not have any side conditions to check.

Another benefit of swapping in nominal is that the composition of two swappings is itself a swapping. It is also the case in de Bruijn, as the composition of two injective functions on indices is an injective function on indices. The problem does not arise in HOAS, as we do not have a renaming operation to compose. It is more complicated in locally nameless, as an opening $\{k \to a\}P$ can only be composed with a reverse closing $\{k \leftarrow a\}P$. As a result, to chain several openings on the same index, we end up with sequences of several open and close operations which are not so easy to manipulate.

As an example, the proofs of the renaming lemmas themselves for the bisimilarity and open extension represent 20 loc in nominal, 80 loc in de Bruijn, 150 loc in HOAS, and 300 loc in locally nameless. The proofs in de Bruijn however rely on many boilerplate results about injective functions and sets of free variables (for the open extension), which makes the overall proofs longer in de Bruijn than in locally nameless (cf. the "Bisimulation" line in Table 2).

While nominal allows for shorter proofs of the renaming (swapping) lemmas, applying these lemmas is easier in HOAS, because of Coq tactics like `change` or `pattern`. For example, turning $([a \leftrightarrow b]P)\{[a \leftrightarrow b]Q/X\} = ([a \leftrightarrow b]P')\{[a \leftrightarrow b]Q'/X\}$

into $[a \leftrightarrow b](P\{Q/X\}) = [a \leftrightarrow b](P'\{Q'/X\})$ requires two uses of `rewrite` and a proof that `swap` (or `open/close` in locally nameless, or `mapN` in de Bruijn) commutes with substitution. In HOAS, turning $(\mathbb{P}\ a)\{\mathbb{Q}\ a/X\} = (\mathbb{P}'\ a)\{\mathbb{Q}'\ a/X\}$ into $(x \mapsto (\mathbb{P}\ x)\{\mathbb{Q}\ x/X\})\ a = (x \mapsto (\mathbb{P}'\ x)\{\mathbb{Q}'\ x/X\})\ a$ is done in a `change` command without any other requirement. However, the gains we have in either proving or applying renaming lemmas in nominal or HOAS are mitigated by the fact that we need to prove and use many more of such lemmas with these representations than with de Bruijn or locally nameless.

8.4 Process variables

Since process input is similar to λ-abstraction, any representation that can handle the latter should also handle the former. As explained in Section 4.3, the nested datatype representation is however well suited to define closing substitutions and therefore open extension, since the type of a process contains (an over-approximation of) the set of free variables. Open extension is not as easily defined in HOAS [39], except in systems with dedicated support for simultaneous substitutions, such as Beluga [52]. We expect that a locally nameless or nominal representations of open extension to not be as trivial as with the nested datatypes approach.

8.5 Conclusion

Locally nameless appears to be the less tractable of the techniques we experiment with, mostly because it does not support renaming very well. Indeed, it requires more boilerplate results about its functions `open` and `close`. These functions do not compose well when successive renamings are performed, which makes the already difficult proofs of renaming lemmas quite lengthy. The nominal formalization loses the benefits of its straightforward swapping lemmas proofs in the extra development needed for the binding construct for concretions. It also interacts not so smoothly with the nested datatype formalization of process variables.

The HOAS formalization handles renaming better in general, but as in nominal, it requires duplicated results about the binding constructs in processes and concretions. It also suffers from the fact that most operations defined for processes or concretions are defined as relation predicates for which we have to prove renaming lemmas. In the end, the de Bruijn formalization of names is the most concise, at the price of non-trivial manipulations of de Bruijn indices and complex renaming lemmas proofs.

A nominal representation would fare better in Isabelle/HOL which proposes a dedicated support for such a representation [53]. It would save us from proving the basic results about swapping and would provide tactics for equivariance proofs. Nominal 2 [54] also allows for bindings of sets of names, as we need for concretions. It would however not help with the problems caused by the interaction with the nested datatype formalization of process variables. The issue would be avoided by also using a nominal representation for process variables, but the resulting more complex formalization of open extension may outweigh the benefits in the end. A

HOAS formalization would also be simpler in provers relying on this representation [43,5,44]. Their respective logics, however, may not be as expressive as the Coq logic, so expressing advanced techniques such as Howe's method may require some tricks [39].

9 Related Work

Several works [1,39,52,2] formalize Howe's method in functional languages, and some of them [2,52] point out substitutivity of Howe's closure (Lemma `howe_subst`) as a difficult part of the proof. We do not have this issue thanks to the nested datatype representation, which allows for a very simple representation of closing substitutions (see the definition of `open_extension`).

To our knowledge, only two prior works [33,40] propose formalizations of higher-order process calculi. The calculus studied by Maksimović and Schmitt [33] is a sub-calculus of HOπ as it does not feature name restriction. Parrow et al. [40] extend an existing formalization of the psi-calculus to accommodate for higher-order communication. The calculus is based on *triggers* [48]: instead of exchanging executable processes, psi-terms exchange data, which may be processes, that are then used to trigger the execution of a process using the invocation of a clause. Such clause-based semantics cannot be used to encode some higher-order calculi, such as calculi with passivation [32], where the capture of running processes would require the dynamic generation of new clauses. Besides, the bisimulation of Parrow et al. [40] is *higher-order*: two emitting processes are bisimilar if the messages (in fact, the triggered processes) and continuations are pairwise bisimilar when considered separately. Unlike the context bisimilarity we formalize, higher-order bisimilarity is not *complete* for HOπ, as it distinguishes processes that should be considered equivalent [48]. Higher-order bisimilarity is also easier to formalize, as it does not quantify over abstractions F in the output case; proving that higher-order bisimilarity is stable by renaming does not require several renamings (in F) as we have to do with context bisimilarity.

In contrast with higher-order calculi, the first-order π-calculus has been formalized many times in different provers with several techniques to represent names. We remind that in a first-order language, message input and name restriction both bind names, so the representation of names should fit the two constructs. The works we list below (except for two [22,19]) are interested in the *monadic* π-calculus, where only one name is exchanged during a communication, which implies that scope extrusion takes place for at most one name. Similarly, most formalized semantics rely on the original presentation of the π-calculus [37] where a distinction is made in the LTS between the free and bound outputs—scope extrusion occurring only in the latter. An exception is Despeyroux's work [17] which uses concretions. Because of its inductive nature, a LTS is easier to reason about than a reduction semantics combined with a structural congruence, but some works do manage to represent the reduction semantics [19,10]. No previous work on the first-order π-calculus combines concretions and scope extrusion with several names, like ours. We prefer the concretion approach, because as pointed out, e.g., by Honsell et al. [25], the original presentation of the π-calculus [37] is not preserved by α-equivalence.

We classify the π-calculus formalizations depending on their representations of channel names. The first formalizations [35,38] use ad-hoc representations of names as strings, reasoning then up to α-equivalence. Henry-Gréard [21] uses a primitive locally nameless representation [34], where the distinction between free and bound names is present, but bound names are not de Bruijn indices, and cofinite quantification is not yet used. The distinction ensures that a free name cannot be captured when substituted, but scope extrusion still requires side conditions in the formalization of the LTS to be handled properly.

Formalizations using de Bruijn indices have been written in Coq [22,23], Isabelle/HOL [19], and Agda [41]. Hirschkoff's work [22,23] is the closest to ours, as he formalizes a polyadic variant of the π-calculus. In particular, he discusses a canonical representation of restricted outputs, for which he needs a permutation on indices when adding an extra name restriction. While the definition of this permutation is not in the paper, it should be similar to the `permut` function we use (cf Section 4.2). The formalization of Perera and Cheney [41] follows a similar path in a monadic setting; their function only needs to exchange 0 and 1. Gay [19] uses a completely different strategy, as he relies on an intermediate language similar to the λ-calculus (formalized with de Bruijn indices), in which he encodes the π-calculus constructs. Such two-steps formalization allows him to encode the semantics as a reduction relation and not a LTS.

Nominal formalizations of the π-calculus have been proposed so far only in variants of Isabelle [46,28,6]. We only discuss the most recent one [6], which mainly differs from our work in that it relies on tailored induction principles in the cases involving bound names, allowing to pick these bound names fresh from the context. First, such tailored principles would not help when reasoning about `bind`, as generating fresh names is part of its definition, and therefore cannot be avoided in proofs. It turns out that in our development, we need to pick fresh names only in the structural congruence simulation proof, when applying the lemmas mimicking `sc_scope`. As discussed at the end of the Section 6.5, these lemmas have a freshness side-condition (like `sc_scope`) which are not necessarily met by the bound names introduced in the name restriction case. A tailored induction principle for structural congruence would be useful here, but as it concerns only one proof, we decided to do the swappings by hand instead.

Our HOAS formalization is partly inspired by Honsell et al. [25], in particular the use of cofinite quantification. Despeyroux [17] proposes an alternative representation, where concretions are functions from abstractions to processes, and pseudo-application $\bullet$ becomes application of a concretion to an abstraction. For example, the output is written $\overline{a}!(R).Q \xrightarrow{\overline{a}} (F \mapsto (F\ R) \parallel Q)$; the resulting function applied to an abstraction $(X)P$ becomes $P\{R/X\} \parallel Q$, as wished. Such a representation allows for eager scope extrusion (cf. Remark 1) only, as a function $F \mapsto P$ does not give access to the message or the continuation, while we are interested in lazy scope extrusion.

Cervesato et al. [10] represent π-calculus terms into CLF, an extension of linear LF [9] with concurrency constructs; the π-calculus reduction semantics is then encoded with interactions between linear logic connectives. Röckl and Hirschkoff [47] relate shallow and deep embeddings of the π-calculus syntax in Isabelle/HOL. They rule out exotic terms in HOAS by using a well-formedness predicate, and prove the axioms of the Theory of Contexts in classical logic. Baelde et al. formal-

ization [5] relies on the Abella nabla (∇) quantifier [36] to generate fresh names. Roughly, opening a term $\nabla x.P$ is done by replacing x with a constant n such that n is distinct from the names of P and from any other generated constant. As a result, $\nabla x.\nabla y.\ x \neq y$ is a true statement, in contrast with $\forall x.\forall y.\ x \neq y$. We are not aware of any implementation of the nabla quantifier in Coq, a work which may require a preliminary study of an extension of Coq's logic with nabla.

10 Conclusion and Future Work

In this paper, we formalize HOπ in Coq and compare several formalizations of name restriction, using Howe's method as a test bed. Our study pinpoints the main issues with each technique. W.r.t. concretions, de Bruijn based formalizations require unusual permutations on indices, while nominal and HOAS need specific binding constructs. The de Bruijn and HOAS formalizations rely on difficult to prove renaming lemmas, in contrast with the straightforward equivariance proofs in nominal, but they interact much better than in nominal with the nested datatype representation of process variables. Finally, cofinite quantification in simulation proofs entails many renamings, which are much better handled in HOAS than in locally nameless.

In the end, the de Bruijn formalization is the most concise, while the nominal one remains closer to the pen-and-paper definitions. HOAS lies between the two, with a function-based representation easier to manipulate than de Bruijn indices and resulting in a short enough development, but with definitions more complex than in nominal (e.g., `conc_new`). Locally nameless does not seem well-suited to represent process calculi, as we do as much indices manipulation than in plain de Bruijn and in addition pay the price of turning indices into names when opening a binder.

This work is only a first step in the formalization of higher-order process calculi. The bisimilarity of this paper is *strong* as one transition $\xrightarrow{l}$ from a process is matched by exactly one transition $\xrightarrow{l}$ from the other. *Weak* bisimilarities are more flexible, as they allow several silent $\xrightarrow{\tau}$-transitions before and after $\xrightarrow{l}$. Extending our developments to a weak bisimilarity should be straightforward but cumbersome, as it would require extra inductions on the closure $(\xrightarrow{\tau})^+$.

Another possible extension is to add passivation or join patterns to the language, as in our previous work [30]. The semantics of a language with join patterns is more difficult to formalize as an input expects several messages on possibly different channels; a concretion should thus map channels to messages. Passivation brings different issues; e.g., the bisimilarity of a calculus with passivation features capturing evaluation contexts in its definition [32].

A long-term goal is to develop support tools (tactics, proof libraries, ...) to write proofs with higher-order process calculi in Coq. Parts of our development are language-independent and could be singled-out as libraries, like the results and tactics about swapping in nominal, or those about `permut` or injective functions in general in plain de Bruijn. Given the syntax of a language, generating the definitions, boilerplate lemmas and proofs for a given representation and theorem prover is an active line of research with, e.g., LNGen [4] for locally nameless, the Isabelle nominal package [53,54], Needle and Knot [29] or Autosubst 2 [50]

for de Bruijn indices. It would be natural to extend these frameworks so that they provide support for scope extrusion. However, it is not clear how much of our proofs of boilerplate results could be automatically generated, as the proofs tend be become quickly complex as soon as name restriction is involved. Finally, mechanizing simulation has not been studied has much as, e.g., type systems [3], so we believe advances are possible on that front. In particular, we aim to make Howe's method more easily available in both sequential and process calculi, as a library.

A Appendix

We define two processes P and Q such that $b \in \mathsf{fn}(P) \cup \mathsf{fn}(Q)$, $P \sim Q$, but renaming a into b in P and Q breaks the bisimilarity. The example has not been formalized in Coq, so we use the notations of Section 2 for readability.

In a calculus with a choice operator, so that

$$\frac{P_i \xrightarrow{\alpha} A \qquad i \in \{1,2\}}{P_1 + P_2 \xrightarrow{\alpha} A}$$

the example would be $P \triangleq \overline{a}!(\oslash).\oslash \parallel b?X.\oslash$ and $Q \triangleq \overline{a}!(\oslash).b?X.\oslash + b?X.\overline{a}!(\oslash).\oslash$. The process P can either do an output on a and then an input on b, which corresponds to the first branch in Q, or do the opposite, which corresponds to the second branch. If we rename a into b in P and Q (written $\{a \to b\}P$), P can do a communication on a, a $\xrightarrow{\tau}$-transition that Q cannot match.

Erratum The conference version of this article then defines two HOπ processes P and Q which mimic the above behavior without using $+$. The example is incorrect: the P and Q given in the conference version are not strong bisimilar, some τ-actions are not matched. In the light of previous works by Hirschkoff and Pous [24], we conjecture that we cannot find two HOπ processes that are strongly bisimilar but are no longer bisimilar after a renaming.

However, there exist such processes if we consider weak bisimilarity instead of strong bisimilarity, and if we consider more expressive calculi (as shown with $+$ above). Therefore we believe Definition 5 is still the right property to establish in general.

References

1. Simon Ambler and Roy L. Crole. Mechanized operational semantics via (co)induction. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin-Mohring, and Laurent Théry, editors, *TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*, pages 221–238, Nice, France, 1999. Springer.
2. Abhishek Anand and Vincent Rahli. Towards a formally verified proof assistant. In Gerwin Klein and Ruben Gamboa, editors, *ITP 2014*, volume 8558 of *Lecture Notes in Computer Science*, pages 27–44, Vienna, Austria, 2014. Springer.
3. Brian Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The PoplMark challenge. In *TPHOLs*, pages 50–65, 2005.
4. Brian E. Aydemir and Stephanie Weirich. LNgen: Tool support for locally nameless representations. Technical report, University of Pennsylvania, 2010.
5. David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. Abella: A system for reasoning about relational specifications. *J. Formalized Reasoning*, 7(2):1–89, 2014.
6. Jesper Bengtson and Joachim Parrow. Formalising the pi-calculus using nominal logic. *Logical Methods in Computer Science*, 5(2), 2009.
7. Richard S. Bird and Ross Paterson. De Bruijn notation as a nested datatype. *J. Funct. Program.*, 9(1):77–91, 1999.

8. Anna Bucalo, Furio Honsell, Marino Miculan, Ivan Scagnetto, and Martin Hofmann. Consistency of the theory of contexts. *J. Funct. Program.*, 16(3):327–372, 2006.
9. Iliano Cervesato and Frank Pfenning. A linear logical framework. *Inf. Comput.*, 179(1):19–75, 2002.
10. Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework ii: Examples and applications. Technical Report CMU-CS-02-102, Carnegie Mellon University, 2002.
11. Arthur Charguéraud. LN:locally nameless representation with cofinite quantification. Available at `http://www.chargueraud.org/softs/ln/`.
12. Arthur Charguéraud. TLC: A non-constructive library for Coq. Available at `http://www.chargueraud.org/softs/tlc/`.
13. Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, 49(3):363–408, 2012.
14. Alberto Ciaffaglione and Ivan Scagnetto. Mechanizing type environments in weak HOAS. *Theor. Comput. Sci.*, 606:57–78, 2015.
15. Silvano Dal Zilio. Mobile Processes: a Commented Bibliography. In *MOVEP'2K – 4th Summer school on Modelling and Verification of Parallel processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 206–222. Springer-Verlag, June 2001.
16. Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies: A tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, 75(5):381–392, 1972.
17. Joëlle Despeyroux. A higher-order specification of the pi-calculus. In Jan van Leeuwen, Osamu Watanabe, Masami Hagiya, Peter D. Mosses, and Takayasu Ito, editors, *IFIP TCS 2000*, volume 1872 of *Lecture Notes in Computer Science*, pages 425–439. Springer, 2000.
18. Joëlle Despeyroux, Amy P. Felty, and André Hirschowitz. Higher-order abstract syntax in coq. In Mariangiola Dezani-Ciancaglini and Gordon D. Plotkin, editors, *TLCA '95*, volume 902 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 1995.
19. 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, *TPHOLs 2001*, volume 2152, pages 217–232, Edinburgh, Scotland, UK, 2001. Springer.
20. Andrew D. Gordon. Bisimilarity as a theory of functional programming. *Electronic Notes in Theoretical Computer Science*, 1:232–252, 1995.
21. Loïc Henry-Gréard. Proof of the subject reduction property for a pi-calculus in COQ. Technical Report RR-3698, INRIA, 1999.
22. Daniel Hirschkoff. A full formalisation of pi-calculus theory in the calculus of constructions. In *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics*, volume 1275, pages 153–169. Springer, August 1997.
23. Daniel Hirschkoff. Up to context proofs for the π-calculus in the Coq system. Technical Report 97-82, CERMICS, 1997.
24. Daniel Hirschkoff and Damien Pous. A distribution law for CCS and a new congruence result for the *pi*-calculus. In *Proc. of FoSSaCS'07*, volume 4423 of *LNCS*, pages 228–242. Springer, 2007.
25. Furio Honsell, Marino Miculan, and Ivan Scagnetto. pi-calculus in (co)inductive-type theory. *Theoretical Computer Science*, 253(2):239–285, February 2000.
26. Furio Honsell, Marino Miculan, and Ivan Scagnetto. The theory of contexts for first order and higher order abstract syntax. *Electr. Notes Theor. Comput. Sci.*, 62:116–135, 2001.
27. Douglas J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996.
28. Murdoch J. Gabbay. The pi-calculus in fm. *Thirty Five Years of Automating Mathematics*, 28:247–269, 01 2003.
29. Steven Keuchel, Stephanie Weirich, and Tom Schrijvers. Needle & knot: Binder boilerplate tied up. In *ESOP 16*, volume 9632 of *Lecture Notes in Computer Science*, pages 419–445. Springer, 2016.
30. Sergueï Lenglet and Alan Schmitt. Howe's method for contextual semantics. In Luca Aceto and David de Frutos-Escrig, editors, *26th International Conference on Concurrency Theory, CONCUR 2015*, volume 42 of *LIPIcs*, pages 212–225, Madrid, Spain, 2015. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.
31. Sergueï Lenglet and Alan Schmitt. HOπ in Coq. In June Andronick and Amy P. Felty, editors, *CPP 2018*, pages 252–265. ACM, 2018.
32. Sergueï Lenglet, Alan Schmitt, and Jean-Bernard Stefani. Characterizing contextual equivalence in calculi with passivation. *Information and Computation*, 209(11):1390–1433, 2011.

33. Petar Maksimovic and Alan Schmitt. Hocore in Coq. In Christian Urban and Xingyuan Zhang, editors, *ITP 2015*, volume 9236 of *Lecture Notes in Computer Science*, pages 278–293, Nanjing, China, 2015. Springer.
34. James McKinna and Robert Pollack. Pure type systems formalized. In Marc Bezem and Jan Friso Groote, editors, *TLCA '93*, volume 664 of *Lecture Notes in Computer Science*, pages 289–305. Springer, 1993.
35. Thomas F. Melham. A mechanized theory of the pi-calculus in HOL. *Nord. J. Comput.*, 1(1):50–76, 1994.
36. Dale Miller and Alwen Tiu. A proof theory for generic judgments. *ACM Trans. Comput. Log.*, 6(4):749–783, 2005.
37. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Information and Computation*, 100(1):1–40, 1992.
38. Otmane Aït Mohamed. Mechanizing a pi-calculus equivalence in hol. In *TPHOL 95*, pages 1–16. Springer-Verlag, 1995.
39. Alberto Momigliano. A supposedly fun thing I may have to do again: a HOAS encoding of Howe's method. In *LFMTP 12*, pages 33–42, Copenhagen, Denmark, 2012. ACM.
40. Joachim Parrow, Johannes Borgström, Palle Raabjerg, and Johannes Åman Pohjola. Higher-order psi-calculi. *Mathematical Structures in Computer Science*, FirstView:1–37, 3 2014.
41. Roly Perera and James Cheney. Proof-relevant π-calculus: a constructive account of concurrency and causality. *Mathematical Structures in Computer Science*, 28(9):1541–1577, 2018.
42. Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *PLDI 88*, pages 199–208, Atlanta, Georgia, USA, 1988. ACM.
43. Frank Pfenning and Carsten Schürmann. System description: Twelf - A meta-logical framework for deductive systems. In Harald Ganzinger, editor, *CADE 99*, volume 1632 of *Lecture Notes in Computer Science*, pages 202–206. Springer, 1999.
44. Brigitte Pientka and Joshua Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In Jürgen Giesl and Reiner Hähnle, editors, *IJCAR 2010*, volume 6173 of *Lecture Notes in Computer Science*, pages 15–21, Edinburgh, UK, 2010. Springer.
45. Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Inf. Comput.*, 186(2):165–193, 2003.
46. Christine Röckl. A first-order syntax for the pi-calculus in isabelle/hol using permutations. *Electr. Notes Theor. Comput. Sci.*, 58(1):1–17, 2001.
47. Christine Röckl and Daniel Hirschkoff. A fully adequate shallow embedding of the [pi]-calculus in isabelle/hol with mechanized syntax analysis. *J. Funct. Program.*, 13(2):415–451, 2003.
48. Davide Sangiorgi. Bisimulation for higher-order process calculi. *Information and Computation*, 131(2):141–178, 1996.
49. Davide Sangiorgi and David Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.
50. 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 19*, pages 166–180. ACM, 2019.
51. The Penn PL Club. The Penn locally nameless metatheory library. Available at `https://github.com/plclub/metalib`.
52. David Thibodeau, Alberto Momigliano, and Brigitte Pientka. A case-study in programming coinductive proofs: Howe's method. available at `momigliano.di.unimi.it/papers/bhowe.pdf`, 2016.
53. Christian Urban. Nominal techniques in Isabelle/HOL. *J. Autom. Reasoning*, 40(4):327–356, 2008.
54. Christian Urban, Stefan Berghofer, and Cezary Kaliszyk. Nominal 2. Archive of Formal Proofs, February 2013. `http://isa-afp.org/entries/Nominal2.html`, Formal proof development.