HOπ in Coq

Sergueï Lenglet, Alan Schmitt

To cite this version:

Sergueï Lenglet, Alan Schmitt. HOπ in Coq. CPP 2018 - The 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, Jan 2018, Los Angeles, United States. pp.14, 2018, <10.1145/3167083>. <hal-01614987v3>

HAL Id: hal-01614987

https://hal.inria.fr/hal-01614987v3

Submitted on 15 Dec 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HOπ in Coq

Sergueï Lenglet
Université de Lorraine
France

Alan Schmitt
Inria
France

Abstract

We propose 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. 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.

CCS Concepts • **Theory of computation** → **Process calculi**;

Keywords Higher-order process calculus, Howe's method

ACM Reference Format:
Sergueï Lenglet and Alan Schmitt. 2018. HOπ in Coq. In *Proceedings of 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP'18).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3167083

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 [22], 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 [9] and Perera and Cheney [18] list some of them), only a few recent works propose a formalization of a higher-order calculus [15, 17]. The semantics of the calculus of Parrow et al. [17] is based on triggers and clauses to enable the execution of transmitted processes. Maksimović and Schmitt [15] 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 present a formalization in Coq of the Higher-Order π-calculus HOπ [21], 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 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.

Besides, 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; in addition, 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.

Because of this discrepancy, we use representations that we believe are suitable for each construct, namely de Bruijn indices as a nested datatype [4] for process variables, and locally nameless [5] for channel names. We then formalize the HOπ *context bisimilarity* [21] and prove it is *compatible* (i.e., preserved by the operators of the language) using *Howe's method* [12], a systematic proof technique to prove compatibility in a higher-order setting. Our proofs follow a previous paper [13] 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 pinpoint the issues related to binders that do not occur in pen-and-paper proofs relying on α-conversion,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CPP'18, January 8–9, 2018, Los Angeles, CA, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-5586-5/18/01…$15.00
https://doi.org/10.1145/3167083

such as how to handle scope extrusion or the renaming lemma for context bisimilarity.

- We demonstrate that our formalization allows for complex proofs such as Howe's method, a technique that has so far been formalized only for functional languages [1, 2, 16, 23].

We present HOπ in Section 2. We show how we formalize binders in Section 3, and how we define context bisimilarity in Section 4. We detail Howe's method and its formalization in Section 5, and we discuss related work and our choices of binder representations in Section 6. The formal developments, available at http://passivation.gforge.inria.fr/hopi/html/toc.html, consists of 5000 lines of Coq code (about 1000 lines of definitions and 4000 lines of proofs). A symbol in the paper indicates a link to the online proofs scripts. We use the TLC library [6], which provides tools for classical reasoning in Coq, predefined datatypes, and tools for automation. In particular, we use the conditional `If` (note the capital `I`) which 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.

2 The Higher-Order π-Calculus

We recall the syntax, semantics, and bisimilarity of HOπ. This section is meant to be introductory and thus, the syntax and notations we define here do not correspond to the Coq formalization. We adapt the syntax to be closer to the formalization in Section 4.3.

Syntax. The syntax and semantics of the process-passing fragment of HOπ [21] 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 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 sub-processes 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 $\nu a_1 . \dots \nu a_n.P$ as $\nu\widetilde{a}.P$.

Semantics. We define 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

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

$$(P \parallel Q) \parallel R \equiv P \parallel (Q \parallel R) \qquad P \parallel Q \equiv Q \parallel P$$

$$P \parallel \oslash \equiv P \qquad \nu a.\nu b.P \equiv \nu b.\nu a.P$$

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

Extension of operators to all agents

$$(X)Q \parallel P \triangleq (X)(Q \parallel P)$$

$$P \parallel (X)Q \triangleq (X)(P \parallel Q)$$

$$\nu a.(X)P \triangleq (X)\nu a.P$$

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

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

$$\nu a.(\nu\widetilde{b}.\langle Q\rangle R) \triangleq \nu a,\widetilde{b}.\langle Q\rangle R \text{ if } a \in \mathsf{fn}(\nu\widetilde{b}.Q)$$

$$\nu a.(\nu\widetilde{b}.\langle Q\rangle R) \triangleq \nu\widetilde{b}.\langle Q\rangle \nu a.R \text{ if } a \notin \mathsf{fn}(\nu\widetilde{b}.Q)$$

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

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

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

$$\frac{P \xrightarrow{\alpha} A}{P \parallel Q \xrightarrow{\alpha} A \parallel Q}\ \textsc{Par} \qquad \frac{P \xrightarrow{\alpha} A \quad \alpha \notin \{a, \overline{a}\}}{\nu a.P \xrightarrow{\alpha} \nu a.A}\ \textsc{Restr}$$

$$\frac{P \xrightarrow{a} F \quad Q \xrightarrow{\overline{a}} C}{P \parallel Q \xrightarrow{\tau} F \bullet C}\ \text{HO}$$

Figure 1. Contextual LTS for HOπ

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 $(X)Q \parallel P \triangleq (X)(Q \parallel P)$, 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 2.1. 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 $a \notin \mathrm{fn}(P\{Q/X\})$, but it is not true in all calculi; for instance, it does not hold in calculi with passivation [14], where restriction does not commute with localities. We use lazy scope extrusion because it appears to be the most commonly used in process calculi [7].

Bisimilarity. We relate processes with the same behavior using strong *context bisimilarity* [21], shortened as bisimilarity, defined as follows.

Definition 2.2. A relation $\mathcal{R}$ on closed processes is a *simulation* if $P \mathrel{\mathcal{R}} Q$ implies:

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

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 2.3. Given a relation $\mathcal{R}$ on closed processes and two processes P and Q, $P \mathrel{\mathcal{R}^\circ} Q$ holds if $P\sigma \mathrel{\mathcal{R}} Q\sigma$ 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 2.4. A relation $\mathcal{R}$ is a simulation up to $\equiv$ if $P \mathrel{\mathcal{R}} Q$ implies the clauses of Definition 2.2, replacing $\mathcal{R}$ with $\equiv\mathcal{R}\equiv$.

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

Lemma 2.5. *If $\mathcal{R}$ is a bisimulation up to $\equiv$, then $\mathcal{R} \subseteq \sim$.*

3 Formalizing Binders

3.1 Formalization of Process Variables

We represent process variables with de Bruijn indices, so that we write $a?.\mathbb{0}$ for $a?X.X$ and $a?.b?.(\mathbb{1} \parallel \mathbb{0})$ for $a?X.b?Y.(X \parallel Y)$. We encode the indices using a nested datatype, as originally introduced by Bird and Patterson [4]. 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 : proc V → proc V.            (* 'nu' P *)
```

We discuss the datatype `name` representing channel names and name restriction `nu P` in Section 3.2. The comments contain the Coq notation we use for each construct. 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 $a?.b?.(\mathbb{1} \parallel \mathbb{0})$ is thus written `a? b? (pr_var (VS VZ) // (pr_var VZ))`.

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 4.1.

Substitution is defined in terms of shifting and monadic operations. To prevent capture, 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`, thus allowing us to shift variables.

```
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 name capture with restriction; we discuss this case in Section 3.2.

Finally, we define substitution `subst P Q` which replaces the occurrences of `VZ` in `P` with `Q`. We never use a more general operation that would replace any given variable (not only `VZ`) with a process.

```
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 usually follow by straightforward structural inductions on processes.

3.2 Formalization of Channel Names

We use the *locally nameless* paradigm [5] for channel names, where free channels are represented by names and bound ones by de Bruijn indices. We believe this representation is well adapted to formalize the sequence of bound names of a concretion (see Section 3.3); we discuss our choices further in Section 6. The datatype `name` is defined as follows.

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

We use the TLC type `var` to represent free names. Among other features, TLC provides tactics to generate fresh names and reason about them. In the process `nu P`, the name restriction construct binds the occurrences of the bound name `0` in `P`. The process $\nu a.(\overline{a}!(\oslash).\oslash \parallel \overline{b}!(\oslash).\oslash)$ is thus written as `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.

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

Given a type `A`, TLC provides a type `fset A` of finite sets of elements of type `A`, and operations on these sets, like `\notin` in the above definition. 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` gives a more tractable induction principle on fully bound processes, as we have some leeway on the set `L` from which `a` should be fresh.

As explained by Charguéraud [5, 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 usually consequences of more general lemmas showing that a property is preserved by substitution, since renaming is just a particular case of substitution. It is not the case here, as free names are not substituted in our language, 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)`.

[1] In the code, they are coercions from respectively `nat` and `var` to `name`.
[2] Charguéraud [5] 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.

```
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.
```

Figure 2. Name restriction for concretions

In proofs, we may substitute processes that are not fully bound (see Remark 4.7), 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 3.1, 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`.

3.3 Formalization of HOπ Semantics

So far, the presented definitions and results are typical for the nested datatype and locally nameless representations of binders. Here, we discuss binder issues more specific to HOπ which arise when we formalize the 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 3.1).

```
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 type-checking: `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.
```

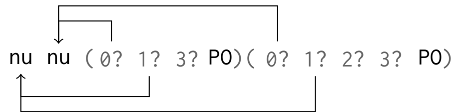
When extending parallel composition to concretions, we need to shift names if the process `P` we put in parallel is not fully bound (similarly for `conc_parr` (🐓)).

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

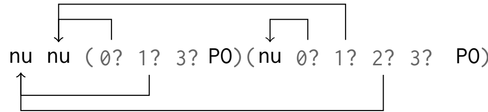
Defining name restriction for concretions implements lazy scope extrusion. If we add a restriction to the concretion `conc_def n P Q` such that `P` contains the bound name `n`, then `n` must be extruded, and the result is `conc_def (S n) P Q`. Otherwise, the added restriction needs to encompass `Q` only, but we must reorganize the names in `P` and `Q` accordingly. Indeed, consider `conc_def 2 (0? 1? 3? P0)(0? 1? 2? 3? P0)`; the binding structure is originally

```
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)).
```

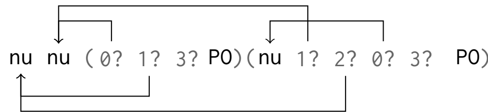
Figure 3. Formalization of the LTS



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 correspond to 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 :=                 Inductive label: Set :=
| ag_proc: proc0 → agent           | tau: label
| ag_abs : abs   → agent           | inp: var → label
| ag_conc: conc  → agent.          | 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 that perform parallel composition and name restriction on agents (🐓, 🐓, 🐓), and `appl` and `appr` the functions which compute the pseudo-applications $F \bullet C$ and $C \bullet F$ (🐓, 🐓).

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 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
```

The operator `\u` is the union of finite sets. 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 $P \triangleq 0?.\mathbb{0} \parallel \overline{a}!(\oslash).\oslash$, then $\{0 \longrightarrow a\}P$ can perform a communication, but $\{0 \longrightarrow b\}P$ cannot for $b \neq a$.

Remark 3.1. 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 Equivalences on Processes

We discuss here the formalization of the equivalences we need on processes, namely bisimilarity and structural congruence. We represent relations using the TLC type `binary`, and we write $\circ$ for the composition of two relations.

4.1 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 2.2, but it is also more tractable in Coq [15]. In the following, `test_proc`, `test_abs`, and `test_conc` are notations representing the testing conditions of Definition 2.2 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 2.2 (🐓, 🐓). 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.

4.2 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, but most cases are straightforward. The most difficult cases are the ones manipulating name restrictions.

```
Inductive struct_congr {V:Set}: binary (proc V) :=
(*...*)
(*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)
(*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 outputs, as we have to distinguish cases based on whether the name restrictions bind a name in the message.

Once we prove `sc0` is a bisimulation (🐓), we can define bisimulation up to `sc0` as in Definition 2.4 (🐓), and prove it is a sound up-to technique in the sense of Lemma 2.5 (🐓).

4.3 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 $\mathbb{0}, \mathbb{1}, \ldots$, we use k to range over indices representing bound names, written $0, 1, \ldots$, 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 $\{k \longrightarrow a\}A$ and $\{k \longleftarrow a\}A$ 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 (decomp_agent_gen,).

Lemma 4.1. *For all k, x, and A such that $k \notin bn(A)$, there exists A' such that $A = \{k \to x\}A'$ and $x \notin fn(A')$.*

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 4.2. *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 4.3. 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 $a \notin \mathsf{fn}(P) \cup \mathsf{fn}(Q)$, $\{k \to a\}P \mathrel{\mathcal{R}} \{k \to a\}Q$ implies that for all b such that $b \notin \mathsf{fn}(P) \cup \mathsf{fn}(Q)$, $\{k \to b\}P \mathrel{\mathcal{R}} \{k \to b\}Q$.

This renaming notion is stricter than for the LTS (Lemma lts_rename in Section 3.3), as it requires b to be fresh from P and Q. This condition is necessary for $\sim$ to be stable by renaming, as we illustrate with an example 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 ().

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

Lemma 4.4. *If $\mathcal{R}$ is a simulation, then so is $\mathcal{R}^{\to}$ ().*

Let P, Q, a, and b such that $a \neq b$, $\{a,b\} \mathrel{\#} \mathsf{fn}(P) \cup \mathsf{fn}(Q)$, $\{k \to a\}P \mathrel{\mathcal{R}^{\to}} \{k \to a\}Q$, and $\{k \to b\}P \mathrel{\mathcal{R}^{\to}} \{k \to b\}Q$. 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 $\{k \to a\}P \mathrel{\mathcal{R}^{\to}} \{k \to a\}Q$, we show that they hold for $\{k \to b\}P \mathrel{\mathcal{R}^{\to}} \{k \to b\}Q$ 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 $F_1 \bullet C \mathrel{\mathcal{R}^{\to}} F_2 \bullet C$. The main idea is as follows. By Lemma 4.1, 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 $\{k \to a\}P \mathrel{\mathcal{R}^{\to}} \{k \to a\}Q$ to get F_2' such that $\{k \to a\}Q \xrightarrow{c\{a/b\}} \{k \to a\}F_2'$ and $\{k \to a\}F_1' \bullet C \mathrel{\mathcal{R}^{\to}} \{k \to a\}F_2' \bullet C$. We want to write this as $\{k \to a\}(F_1' \bullet C) \mathrel{\mathcal{R}^{\to}} \{k \to a\}(F_2' \bullet C)$, to then use $\mathcal{R}^{\to}$ to rename a into b, but we need $\{a,b\} \mathrel{\#} \mathsf{fn}(F_1' \bullet C) \cup \mathsf{fn}(F_2' \bullet C)$, 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 4.1 twice, we can decompose C as $C = \{k \to b\}\{k+1 \to a\}C'$ for some C' such that $\{a,b\} \mathrel{\#} \mathsf{fn}(C')$. We then apply the induction hypothesis on $\{k \to a\}P \mathrel{\mathcal{R}^{\to}} \{k \to a\}Q$ 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 $\{k \to a\}F_1' \bullet C'_{a,d} \mathrel{\mathcal{R}^{\to}} F_2'' \bullet C'_{a,d}$. By Lemma 4.1, $F_2'' = \{k \to a\}F_2'$ 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

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

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

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

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

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

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

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

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 wished.

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 $c\{a/b\}\{b/a\} = c$ ().

This concludes the proof of Lemma 4.4, from which we can deduce the following result ().

Theorem 4.5. $\sim$ is stable by renaming.

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

Lemma 4.6. *If $\mathcal{R}$ is stable by renaming, then so is $\mathcal{R}^{\circ}$.*

The proof requires several renamings, as in that of Lemma 4.4. Indeed, let P, Q, a, and b such that $a \neq b$, $\{a,b\} \mathrel{\#} \mathsf{fn}(P) \cup \mathsf{fn}(Q)$, 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,

```
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 ∅ x P) (open ∅ x Q)) → howe Rel (nu P) (nu Q).

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).
```

Figure 4. Howe's closure: formalization and pseudo-simulation lemma

$(\{k \to b\}P)\sigma\ \mathcal{R}\ (\{k \to b\}Q)\sigma$. To conclude, we would like to use the hypothesis that $\mathcal{R}$ is stable by renaming on $(\{k \to a\}P)\sigma\ \mathcal{R}\ (\{k \to a\}Q)\sigma$, which we would like to rewrite into $\{k \to a\}(P\sigma)\ \mathcal{R}\ \{k \to a\}(Q\sigma)$. But σ is arbitrary and may contain a or b, so we modify σ the same way we modify C in the Lemma 4.4 proof.

Remark 4.7. When we write, e.g., $\{k \to b\}(F_1' \bullet C')$ in the proof of Lemma 4.4, 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.

5 Howe's Method

5.1 Sketch of the Method

Howe's method [10, 12] is a systematic proof technique to show that a bisimilarity $\mathcal{B}$ (and its open extension $\mathcal{B}^\circ$) is compatible. The method can be divided in three steps: first, prove some basic properties on the *Howe's closure* $\mathcal{B}^\bullet$ of the relation. By construction, $\mathcal{B}^\bullet$ contains $\mathcal{B}^\circ$ and is compatible. Second, prove a simulation-like property for $\mathcal{B}^\bullet$. Finally, prove that $\mathcal{B}$ and $\mathcal{B}^\bullet$ coincide on closed processes. Since $\mathcal{B}^\bullet$ is a compatible, then so is $\mathcal{B}$. This is applied to HOπ's bisimilarity at the end of this section, which we state again here before diving into the definitions and proofs.

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

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

Definition 5.1. 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.

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

The base cases for this definition are the base cases of the syntax of processes, namely $\oslash\ \mathcal{R}^\bullet\ \oslash$ and $X\ \mathcal{R}^\bullet\ X$. The second rule of the definition ensures that $\mathcal{R}^\bullet$ is compatible.

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

Lemma 5.2. *If $\mathcal{R}$ is symmetric, then $(\mathcal{R}^\bullet)^+$ is symmetric.*

If $\mathcal{B}^\bullet$ is a simulation, then $(\mathcal{B}^\bullet)^+$ (restricted to closed terms) is also a simulation. By Lemma 5.2, $(\mathcal{B}^\bullet)^+$ is in fact a bisimulation. Consequently, we have $\mathcal{B} \subseteq \mathcal{B}^\bullet \subseteq (\mathcal{B}^\bullet)^+ \subseteq \mathcal{B}$ on closed terms, and we conclude that $\mathcal{B}$ is compatible.

The main challenge is to state and prove a simulation-like property for the Howe's closure $\mathcal{B}^\bullet$. For higher-order process calculi equipped with a context bisimilarity such as $\sim$, the difficulty is in the communication case. We propose in our previous paper [13] 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 5.3 (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 5.3 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 5.3 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 [13] for more details.

5.2 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 5.2 (🐓).

Lemma 5.4. *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 4.1, 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 4.6 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 state Lemma 5.2 using the TLC transitive closure `tclosure`, defined as follows.

$$\frac{P \; \mathcal{R} \; Q}{P \; \mathcal{R}^+ \; Q} \qquad \frac{P \; \mathcal{R}^+ \; R \qquad R \; \mathcal{R}^+ \; Q}{P \; \mathcal{R}^+ \; Q}$$

The proof of Lemma 5.2 (🐓) is 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 $Q \; (\mathcal{R}^\bullet)^+ \; P$ 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 $\widetilde{P' \; \mathcal{R}^\bullet \; Q'}$, then we have $\widetilde{Q' \; (\mathcal{R}^\bullet)^+ \; P'}$ using (IH_2). We must show that $\widetilde{Q' \; (\mathcal{R}^\bullet)^+ \; P'}$ 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 5.4 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').
```

5.3 Pseudo-Simulation Lemma

Figure 4 contains the formalization of Lemma 5.3, 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 5.3 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 does not necessarily imply R closed. We then have to show that R can be closed without changing the size of the derivation, so the proofs are done by induction on the size of

the derivation of $P\ \mathcal{R}^\bullet\ Q$ on not the derivation itself [13]. 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 [13]. 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 // P0) (Q // P0) → 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 [13] 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 $(P_i\ \mathcal{R}^\bullet\ Q_i)_{1\le i\le n}$ as in [13], 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 processes.

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.
```

6 Related Work and Conclusion

Related work. To our knowledge, only two prior works [15, 17] propose formalizations of higher-order process calculi. The calculus studied by Maksimović and Schmitt [15] is a sub-calculus of HOπ as it does not feature name restriction. Parrow et al. [17] extend an existing formalization of the psi-calculus to accommodate for higher-order communication. The calculus is based on *triggers* [21]: 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 [14], where the capture of running processes would require the dynamic generation of new clauses. Besides, the bisimulation of Parrow et al. [17] 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 [21]. 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.

Several works [1, 2, 16, 23] formalize Howe's method in functional languages, and some of them [2, 23] 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`).

Discussion. We discuss here our representation choices for binders. Since process input is similar to λ-abstraction, any representation that can handle the latter—such as locally nameless [5], nominal theory [20], or higher-order abstract syntax (HOAS) [19]—should also handle the former. We mentioned before that 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 [16], except in systems with dedicated support for simultaneous substitutions, such as Beluga [23]. We also expect that the locally nameless or nominal representations of open extension would not be as trivial as the nested datatypes approach.

Representing name restriction is more complicated because of scope extrusion. Locally nameless is convenient to represent a concretion $\nu\widetilde{b}.\langle P\rangle Q$, because the set of bound names $\widetilde{b}$ is just represented by its size k. A pure de Bruijn representation would also use the size k, and a nominal representation would use k named binders. The nested datatype representation does not seem practical enough to work with concretions: to express the fact that P should have at least k free names, we need to write a dependent type of the form `proc (inc k 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 k. Turning to HOAS, its main idea is to encode the binding structure of the object language (here, HOπ) into the binding constructs of the meta-language (here, Coq) to benefit from the infrastructure of the meta-language, in particular substitution. This approach is not well-suited to represent names that do not need to be substituted but that may be extruded, except for systems, such as the Abella theorem prover [3], that provide support for fresh name generation. In such systems, scope extrusion is modeled using the substitution of freshly generated names.

The main difficulty with the locally nameless representation is proving the renaming lemmas, especially for the bisimilarity. We believe we would have the same issue with a nominal or pure de Bruijn representation. Indeed, the nominal representation relies heavily on *swapping* to avoid name captures, an operation that exchanges names, even bound ones, so that swapping a and b in $\nu a.(\overline{a}!(\oslash).\oslash \parallel b?X.X)$ produces $\nu b.(\overline{b}!(\oslash).\oslash \parallel a?X.X)$. We expect that with a nominal representation, we would have to prove that Howe's closure and bisimilarity are stable by swapping, and the proofs should be similar in complexity to our renaming proofs. With a pure de Bruijn representation, to prove that $P\ \mathcal{R}^\bullet\ Q$ and $R\ \mathcal{R}^\bullet\ S$ implies $P\{R/\mathbb{0}\}\ \mathcal{R}^\bullet\ Q\{S/\mathbb{0}\}$, we need to show that Howe's closure is preserved by shifting, as we have to shift R and S if we substitute under a name restriction in P and Q. As a result, we also have to prove that open extension and the bisimilarity are also preserved by shifting, and we conjecture that these proofs are as difficult as our renaming lemmas.

Conclusion and future work. In this paper, we propose a formalization of HOπ in Coq, tractable enough to handle complex proofs such as Howe's method. Outside of the already discussed binders representations issues, we find the formalized proofs to be similar in complexity with the pen-and-paper ones [13]. The only significant difference is in the induction principle of the simultaneous induction proof, as we cannot generate the ad hoc one we use on paper. A quality of life improvement would be to have better tactics than the ones we wrote to manipulate sets, e.g., to decide membership or equality, as our proofs work quite heavily with finite sets of free and dangling names.

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 [13]. 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 [14].

A long-term goal is to develop support tools (tactics, proof libraries, ...) to write proofs with higher-order process calculi in Coq. In particular, we aim to make Howe's method more easily available in these calculi, as a library. A first step would be to find a representation of binders better suited for name restriction, i.e., a binding structure without substitution but with scope extrusion. An idea could be to investigate a mix between the nominal and locally nameless representations, as proposed by de Vries and Koutavas [8], to benefit from cofinite quantification in a nominal framework.

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 output 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 $\{a \rightarrow b\}P$), P can do a communication on a, a $\xrightarrow{\tau}$-transition that Q cannot match.

Erratum The conference version of this article then define two HOπ processes P and Q which mimics 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 [11], we conjecture that we cannot find two HOπ processes that are strongly bisimilar but are no longer bisimilar after a renaming.

However, there exists 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 4.3 is still the right property to establish in general.

Acknowledgments

We thank Jasmin Blanchette, Damien Pous, and the anonymous reviewers for their insightful comments.

Sergueï Lenglet carried out this work at the IRISA lab, Université de Rennes 1, thanks to a CNRS funding.

References

[1] Simon Ambler and Roy L. Crole. 1999. Mechanized Operational Semantics via (Co)Induction. In *TPHOLs'99 (Lecture Notes in Computer Science)*, Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin-Mohring, and Laurent Théry (Eds.), Vol. 1690. Springer, Nice, France, 221–238.

[2] Abhishek Anand and Vincent Rahli. 2014. Towards a Formally Verified Proof Assistant. In *ITP 2014 (Lecture Notes in Computer Science)*, Gerwin Klein and Ruben Gamboa (Eds.), Vol. 8558. Springer, Vienna, Austria, 27–44.

[3] David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. 2014. Abella: A System for Reasoning about Relational Specifications. *J. Formalized Reasoning* 7, 2 (2014), 1–89. https://doi.org/10.6092/issn.1972-5787/4650

[4] Richard S. Bird and Ross Paterson. 1999. De Bruijn Notation as a Nested Datatype. *J. Funct. Program.* 9, 1 (1999), 77–91.

[5] Arthur Charguéraud. 2012. The Locally Nameless Representation. *Journal of Automated Reasoning* 49, 3 (2012), 363–408.

[6] Arthur Charguéraud. 2017. TLC: A non-constructive library for Coq. (2017). http://www.chargueraud.org/softs/tlc/

[7] Silvano Dal Zilio. 2001. Mobile Processes: a Commented Bibliography. In *MOVEP'2K – 4th Summer school on Modelling and Verification of Parallel processes (Lecture Notes in Computer Science)*, Vol. 2067. Springer-Verlag, 206–222.

[8] Edkso de Vries and Vasileios Koutavas. 2012. Locally Nameless Permutation Types. (2012). available at http://edsko.net/pubs/lnpt.pdf.

[9] Simon J. Gay. 2001. A Framework for the Formalisation of Pi Calculus Type Systems in Isabelle/HOL. In *TPHOLs 2001*, Richard J. Boulton and Paul B. Jackson (Eds.), Vol. 2152. Springer, Edinburgh, Scotland, UK, 217–232.

[10] Andrew D. Gordon. 1995. Bisimilarity as a theory of functional programming. *Electronic Notes in Theoretical Computer Science* 1 (1995), 232–252.

[11] Daniel Hirschkoff and Damien Pous. 2007. A Distribution Law for CCS and a New Congruence Result for the π-Calculus. In *FOSSACS 2007 (Lecture Notes in Computer Science)*, Helmut Seidl (Eds.), Vol. 4423. Springer, Braga, Portugal, 228–242.

[12] Douglas J. Howe. 1996. Proving Congruence of Bisimulation in Functional Programming Languages. *Information and Computation* 124, 2 (1996), 103–112.

[13] Sergueï Lenglet and Alan Schmitt. 2015. Howe's Method for Contextual Semantics. In *26th International Conference on Concurrency Theory, CONCUR 2015 (LIPIcs)*, Luca Aceto and David de Frutos-Escrig (Eds.), Vol. 42. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Madrid, Spain, 212–225.

[14] Sergueï Lenglet, Alan Schmitt, and Jean-Bernard Stefani. 2011. Characterizing contextual equivalence in calculi with passivation. *Information and Computation* 209, 11 (2011), 1390–1433.

[15] Petar Maksimovic and Alan Schmitt. 2015. HOCore in Coq. In *ITP 2015 (Lecture Notes in Computer Science)*, Christian Urban and Xingyuan Zhang (Eds.), Vol. 9236. Springer, Nanjing, China, 278–293.

[16] Alberto Momigliano. 2012. A supposedly fun thing I may have to do again: a HOAS encoding of Howe's method. In *LFMTP 12*. ACM, Copenhagen, Denmark, 33–42.

[17] Joachim Parrow, Johannes Borgström, Palle Raabjerg, and Johannes Åman Pohjola. 2014. Higher-order psi-calculi. *Mathematical Structures in Computer Science* FirstView (3 2014), 1–37.

[18] Roly Perera and James Cheney. 2016. Proof-relevant π-calculus: a constructive account of concurrency and causality. (2016). arXiv:cs.LO/1604.04575v2

[19] Frank Pfenning and Conal Elliott. 1988. Higher-Order Abstract Syntax. In *PLDI 88*. ACM, Atlanta, Georgia, USA, 199–208.

[20] Andrew M. Pitts. 2003. Nominal logic, a first order theory of names and binding. *Inf. Comput.* 186, 2 (2003), 165–193.

[21] Davide Sangiorgi. 1996. Bisimulation for Higher-Order Process Calculi. *Information and Computation* 131, 2 (1996), 141–178.

[22] Davide Sangiorgi and David Walker. 2001. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press.

[23] David Thibodeau, Alberto Momigliano, and Brigitte Pientka. 2016. A Case-Study in Programming Coinductive Proofs: Howe's Method. (2016). available at momigliano.di.unimi.it/papers/bhowe.pdf.