



HAL
open science

HO π in Coq

Sergueï Lenglet, Alan Schmitt

► **To cite this version:**

| Sergueï Lenglet, Alan Schmitt. HO π in Coq. 2017. hal-01614987v1

HAL Id: hal-01614987

<https://inria.hal.science/hal-01614987v1>

Preprint submitted on 11 Oct 2017 (v1), last revised 15 Dec 2017 (v3)

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.

HO π in Coq

Sergueï Lenglet
Université de Lorraine

Alan Schmitt
Inria

Abstract

We propose a formalization in Coq of HO π , 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 using Howe's method, based on several proof schemes we developed in a previous paper.

Keywords Higher-order process calculus, Howe's method

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 $\bar{a}!(M).Q$, 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 [19], 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 [8] lists some of them), only a few recent works propose a formalization of a higher-order calculus [13, 15]. The semantics of the calculus of Parrow et al. [15] is not truly higher-order, as processes only exchange some data, that can then be used to trigger the execution of a process. Maksimović and Schmitt [13] 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 HO π [18], 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 HO π , 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 HO π . 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 $\nu b.\bar{a}!(Q).R$, the scope of b does not change if b does not occur in Q :

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

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

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

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 [3] for process variables, and locally nameless [5] for channel names. We then formalize the HO π *context bisimilarity* [18] and prove it is a *compatible* (i.e., preserved by the operators of the language) using *Howe's method* [10], a systematic proof technique to prove compatibility in a higher-order setting. Our proofs follow a previous paper [11] 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 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, 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, 14, 20].

We present HO π 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 [4], which provides tools for classical

CPP'18, ,

2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...\$15.00

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

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 $\text{HO}\pi$. 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, after we present how we encode binders in Section 3.

Syntax. The syntax and semantics of the process-passing fragment of $\text{HO}\pi$ [18] are given in Figure 1. We use a, b to range over channel names, \bar{a}, \bar{b} to range over conames, and X, Y to range over process variables. Multisets $\{x_1 \dots x_n\}$ are written \bar{x} .

We write \emptyset 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 , $\bar{a}!(P).Q$ for a sending process which emits P on a before continuing as Q , and $va.P$ 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 $va.P$ binds a in P . We write $\text{fv}(P)$ for the free variables of a process P and $\text{fn}(P)$ 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 $va_1 \dots va_n.P$ as $v\bar{a}.P$.

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 $v\bar{b}.\langle R \rangle S$. Like for processes, the ordering of restrictions does not matter for a concretion; therefore we write them using a set of names \bar{b} , except if $\bar{b} = \emptyset$, 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* $P \xrightarrow{\bar{a}} C$. 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 $P \xrightarrow{\bar{a}} v\bar{b}.\langle R \rangle S$ means that P may send the process R on a and then continue as S , and the scope of the names \bar{b} must be expanded to encompass the recipient of R . We expect \bar{b} to contain names that are indeed in R , so that we only extend the scope of names for which it is necessary.

Syntax

Processes	$P ::= \emptyset \mid X \mid P \parallel P \mid a?X.P \mid \bar{a}!(P).P \mid va.P$
Agents	$A ::= P \mid F \mid C$
Abstractions	$F ::= (X)P$
Concretions	$C ::= \langle P \rangle Q \mid va.C$

Structural congruence

$(P \parallel Q) \parallel R \equiv P \parallel (Q \parallel R)$	$P \parallel Q \equiv Q \parallel P$
$P \parallel \emptyset \equiv P$	$va.vb.P \equiv vb.va.P$
$va.(P \parallel Q) \equiv (va.P) \parallel Q$ if $a \notin \text{fn}(Q)$	$va.\emptyset \equiv \emptyset$

Extension of operators to all agents

$(X)Q \parallel P \stackrel{\Delta}{=} (X)(Q \parallel P)$
$P \parallel (X)Q \stackrel{\Delta}{=} (X)(P \parallel Q)$
$va.(X)P \stackrel{\Delta}{=} (X)va.P$
$(v\bar{b}.\langle Q \rangle R) \parallel P \stackrel{\Delta}{=} v\bar{b}.\langle Q \rangle (R \parallel P)$ if $\bar{b} \cap \text{fn}(P) = \emptyset$
$P \parallel (v\bar{b}.\langle Q \rangle R) \stackrel{\Delta}{=} v\bar{b}.\langle Q \rangle (P \parallel R)$ if $\bar{b} \cap \text{fn}(P) = \emptyset$
$va.(v\bar{b}.\langle Q \rangle R) \stackrel{\Delta}{=} va, \bar{b}.\langle Q \rangle R$ if $a \in \text{fn}(v\bar{b}.\langle Q \rangle R)$
$va.(v\bar{b}.\langle Q \rangle R) \stackrel{\Delta}{=} v\bar{b}.\langle Q \rangle va.R$ if $a \notin \text{fn}(v\bar{b}.\langle Q \rangle R)$

Pseudo-application

$(X)P \bullet v\bar{b}.\langle R \rangle Q \stackrel{\Delta}{=} v\bar{b}.\langle P\{R/X\} \parallel Q \rangle$ if $\bar{b} \cap \text{fn}(P) = \emptyset$

LTS rules

$\alpha ::= \tau \mid a \mid \bar{a}$	
$a?X.P \xrightarrow{a} (X)P$ IN	$\bar{a}!(Q).P \xrightarrow{\bar{a}} \langle Q \rangle P$ OUT
$\frac{P \xrightarrow{\alpha} A}{P \parallel Q \xrightarrow{\alpha} A \parallel Q}$ PAR	$\frac{P \xrightarrow{\alpha} A \quad \alpha \notin \{a, \bar{a}\}}{va.P \xrightarrow{\alpha} va.A}$ RESTR
$\frac{P \xrightarrow{a} F \quad Q \xrightarrow{\bar{a}} C}{P \parallel Q \xrightarrow{\tau} F \bullet C}$ HO	

Figure 1. Contextual LTS for $\text{HO}\pi$

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 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 extend the scope of a restriction, meaning that $va.\langle(Q)R\rangle \stackrel{\Delta}{=} va.\langle Q\rangle R$ in all cases. In HO π , the two are equivalent, since $va.(P\{Q/X\} \parallel R) \equiv P\{Q/X\} \parallel va.R$ if $a \notin \text{fn}(P\{Q/X\})$, but it is not true in all calculi; for instance, it does not hold in calculi with passivation [12]. We use lazy scope extrusion because it appears to be the most commonly used in process calculi [6], and because it poses more formalization challenges.

Bisimilarity. We relate processes with the same behavior using strong *context bisimilarity* [18], shortened as bisimilarity, defined as follows.

Definition 2.2. 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 $P \xrightarrow{\bar{a}} C$, for all F , there exists C' such that $Q \xrightarrow{\bar{a}} C'$ 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 2.3. 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 2.4. A relation \mathcal{R} is a simulation up to \equiv if $P \mathcal{R} Q$ 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 $a?.0$ for $a?X.X$ and $a?.b?.(1 \parallel 0)$ for $a?X.b?Y.(X \parallel Y)$. We encode the indices using a nested datatype, as originally introduced by Bird and Patterson [3]. This representation enforces the set of free variables a process can be build

on at the level of types: given a set V , $\text{proc } V$ is the set of processes that can be built with variables taken from V .

Inductive $\text{incV } (V:\text{Set}): \text{Set} :=$

```
| VZ: incV V
| VS: V  $\rightarrow$  incV V.
```

Inductive $\text{proc } (V:\text{Set}): \text{Set} :=$

```
| pr_nil: proc V
| pr_var: V  $\rightarrow$  proc V
| pr_par: proc V  $\rightarrow$  proc V  $\rightarrow$  proc V (* P // Q *)
| pr_inp: name  $\rightarrow$  proc (incV V)  $\rightarrow$  proc V (* a? P *)
| pr_snd: name  $\rightarrow$  proc V  $\rightarrow$  proc V  $\rightarrow$  proc V
(* a!(P) Q *)
| pr_nu : proc V  $\rightarrow$  proc V.
(* 'nu' P *)
```

We discuss the datatype name representing channel names and name restriction $\text{nu } P$ 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 $\text{proc } (\text{incV } V)$, where $\text{incV } V$ 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 $a?.b?.(1 \parallel 0)$ is thus written $a? b? (\text{pr_var } (VS VZ) // (\text{pr_var } VZ))$.

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 $\text{proc0} := (\text{proc } \text{Empty_set})$.

Notation $\text{proc1} := (\text{proc } (\text{incV } \text{Empty_set}))$.

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 (mapV) transforming the free variables of type V into variables of type W , thus allowing us to shift variables.

Notation $\text{shiftV} := (\text{mapV } (@VS _))$.

Next, we define lift and bind operations (bind), to replace the variables of a process $\text{proc } V$ with processes $\text{proc } W$.

Definition $\text{liftV } \{V W:\text{Set}\} (f:V \rightarrow \text{proc } W) (x:\text{incV } V):$
 $\text{proc } (\text{incV } W) := \text{match } x \text{ with}$

```
| VZ  $\Rightarrow$  pr_var VZ
```

```
| VS y  $\Rightarrow$  shiftV (f y)
```

```
end.
```

Fixpoint $\text{bind } \{V W:\text{Set}\} (f:V \rightarrow \text{proc } W) (P:\text{proc } V):$
 $\text{proc } W := \text{match } P \text{ with}$

```
| pr_var x  $\Rightarrow$  f x
```

```
| a ? P  $\Rightarrow$  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 $\text{subst } P Q$ which replaces the occurrences of VZ in P with Q . We do not need a more

general operation that would replace any given variable (not only VZ) with a process.

Definition $\text{subst_func} \{V:\text{Set}\} (Q:\text{proc } V) (x:\text{incV } V):$
 $\text{proc } V := \text{match } x \text{ with}$

| $VZ \implies Q$
 | $VS \ y \implies \text{pr_var } y$
end.

Notation $\text{subst } P \ Q := (@\text{bind } _ _ (@\text{subst_func } _ \ Q)) \ P.$

In $\text{subst } P \ Q$, P is of type $\text{proc } (\text{incV } V)$ while Q is of type $\text{proc } V$ for some V .

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

Lemma $\text{bind_bind} \{V1 \ V2 \ V3:\text{Set}\}: \forall (P:\text{proc } V1)$
 $(f:V1 \rightarrow \text{proc } V2) (g:V2 \rightarrow \text{proc } V3),$
 $\text{bind } g (\text{bind } f \ P) = \text{bind } (\text{fun } x \implies \text{bind } g (f \ x)) \ P.$

The proofs of these results are usually 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 $\text{name} : \text{Set} :=$
 | $\text{b_name} : \text{nat} \rightarrow \text{name}$
 | $\text{f_name} : \text{var} \rightarrow \text{name}.$

We use the TLC type var to represent free names. Among other features, TLC provides a tactic to generate fresh names. In the process $\text{nu } P$, the name restriction construct binds the occurrences of the bound name \emptyset in P . The process $\text{va}.\overline{a}!(\emptyset).\emptyset \parallel \overline{b}!(\emptyset).\emptyset$ is thus written as

$\text{nu } ((\text{b_name } \emptyset)!(P0) \ P0 // (\text{f_name } b)!(P0) \ P0)$

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 $\text{nu } (\text{b_name } 1)!(P0) \ P0$, 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 (\spadesuit). The converse *closing* operation, written $\{k \leftarrow a\}P$, replaces a free name a by a dangling name k (\spadesuit). The fully bound predicate is_proc is defined as follows.

¹In the code, they are coercions from respectively nat and var to name .

²Chargueraud [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.

Inductive $\text{is_proc} \{V:\text{Set}\}: \text{proc } V \rightarrow \text{Prop} :=$
 | $\text{proc_nil} : \text{is_proc } P0$
 | $\text{proc_var} : \forall x, \text{is_proc } (\text{pr_var } x)$
 | $\text{proc_par} : \forall P \ Q, \text{is_proc } P \rightarrow \text{is_proc } Q \rightarrow$
 $\text{is_proc } (P//Q)$
 | $\text{proc_inp} : \forall (a:\text{var}) \ P, @\text{is_proc } (\text{incV } V) \ P \rightarrow$
 $\text{is_proc } (a? \ P)$
 | $\text{proc_out} : \forall (a:\text{var}) \ P \ Q, \text{is_proc } P \rightarrow \text{is_proc } Q \rightarrow$
 $\text{is_proc } (a!(P) \ Q)$
 | $\text{proc_nu} : \forall (L:\text{fset } \text{var}) \ P, (\forall a, a \notin L \rightarrow$
 $\text{is_proc } (\text{open } \emptyset \ a \ P)) \rightarrow \text{is_proc } (\text{nu } P).$

Given a type A , TLC provides a type $\text{fset } A$ of finite sets of elements of type A , and operations on these sets, like $\backslash \text{notin}$ 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 $\text{nu } P$ 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 (\spadesuit).

Lemma $\text{is_proc_rename} \{V:\text{Set}\}: \forall (P:\text{proc } V) \ k \ a,$
 $\text{is_proc } (\text{open } k \ a \ P) \rightarrow \forall b, \text{is_proc } (\text{open } k \ b \ P).$

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

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 (\spadesuit).

Fixpoint $\text{mapN} \{V:\text{Set}\} (f:\text{nat} \rightarrow \text{nat})(P:\text{proc } V): \text{proc } V.$

Notation $\text{shiftN } n := (\text{mapN } (\text{fun } k \implies n+k)).$

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

| $\text{nu } P \implies \text{nu } (\text{bind } (\text{fun } x \implies \text{shiftN } 1 (f \ x)) \ P)$

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

```

441 Definition down n k := If (k <= n) then k else k-1.
442 Definition permut n k := If (k < n) then (k+1) else If (k=n) then 0 else k.
443 Definition conc_new (C:conc) := match C with
444 | conc_def n P Q  $\implies$  If n \in bn P then conc_def (S n) P Q
445 else conc_def n (mapN (down n) P) (nu (mapN (permut n) Q))
446 end.

```

Figure 2. Name restriction for concretions

```

450 Lemma close_open {V:Set}:  $\forall$  (P:proc V) k a,
451   a \notin fn P  $\rightarrow$  close k a (open k a P) = P.
452 Lemma open_close {V:Set}:  $\forall$  (P:proc V),
453   is_proc P  $\rightarrow$   $\forall$  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` (\clubsuit), and they distribute over `bind` (\clubsuit , \clubsuit). As a result, we have the following relationship between `open` and `subst`

```

459 Lemma open_subst {V:Set}:  $\forall$  P (Q:proc V) k a,
460   open k a (subst P Q)=subst (open k a P) (open k a Q).

```

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

3.3 Formalization of HO π Semantics

So far, the presented definitions and results are typical for the nested datatype and locally nameless representations of binders. Here, we discuss binders issues more specific to HO π 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).

```

476 Inductive abs: Set := | abs_def: proc1  $\rightarrow$  abs.

```

We define parallel composition for abstractions as follows

```

479 Definition abs_par1 F (P:proc0) := match F with
480 | abs_def Q  $\implies$  abs_def (Q // shiftV P)
481 end.

```

with a symmetric function `abs_parr` (\clubsuit). 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 (\clubsuit).

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

```

493 Inductive conc: Set :=
494 | conc_def: nat  $\rightarrow$  proc0  $\rightarrow$  proc0  $\rightarrow$  conc.

```

A concretion $\nu \tilde{b}. \langle P \rangle Q$ 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 \tilde{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` (\clubsuit), and we define a well-formedness predicate on concretions as follows.

```

513 Definition conc_wf C := match C with
514 | conc_def n P Q  $\implies$   $\forall$  k, k < n  $\rightarrow$  k \in bn P
515 end.

```

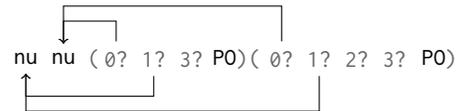
When extending parallel composition to concretions, we need to shift names if the process `P` we put in parallel is not fully bound (similarly for `conc_parr` (\clubsuit)).

```

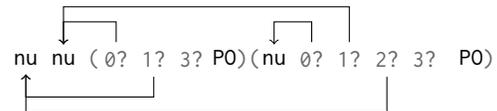
520 Definition conc_par1 C (P:proc0) := match C with
521 | conc_def n P' Q  $\implies$ 
522   conc_def n P' (Q // (shiftN n P))
523 end.

```

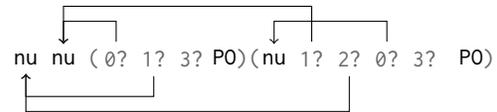
Defining name restriction for concretions implements lazy scope extrusion. If we add a restriction to the concretion `conc_def n P Q` such that `P` contains the bound name `n`, then `n` must be extruded, and the result is `conc_def (S n) P Q`. Otherwise, the added restriction needs to encompass `Q` only, 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



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.



```

551 Inductive lts: proc0 → label → agent → Prop :=
552 | lts_out : ∀ (a:var) P Q,          lts (a !(P) Q) (out a) (ag_conc (conc_def 0 P Q))
553 | lts_inp : ∀ (a:var) P,           lts (a ? P) (inp a) (ag_abs (abs_def P))
554 | lts_parl : ∀ P Q l A, lts P l A → lts (P // Q) l (parl A Q)
555 | lts_parr : ∀ P Q l A, lts P l A → lts (Q // P) l (parr Q A)
556 | 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))
557                                     → lts (nu P) l (new A)
558 | lts_tau1 : ∀ P Q a F C, lts P (out a) (ag_conc C) → lts Q (inp a) (ag_abs F)
559                                     → lts (P // Q) tau (ag_proc (appl C F))
560 | lts_tau2 : ∀ P Q a F C, lts P (out a) (ag_conc C) → lts Q (inp a) (ag_abs F)
561                                     → lts (Q // P) tau (ag_proc (appr F C)).

```

Figure 3. Formalization of the LTS

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 $\text{conc_def } 2 \ (0? \ 1? \ 2? \ P0)(\text{nu } 1? \ 2? \ 0? \ 3? \ P0)$.

To summarize, if we add to a concretion $\text{conc_def } n \ P \ Q$ 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 $\text{HO}\pi$. We define agents and labels as follows.

```

580 Inductive agent :=
581 | ag_proc: proc0 → agent
582 | ag_abs : abs → agent
583 | ag_conc: conc → agent.
584
585 Inductive label: Set :=
586 | tau: label
587 | inp: var → label
588 | 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 $\text{nu } P$, we open the process and instantiate the bound name 0 with a fresh name a , using cofinite quantification; $\text{open } 0 \ a \ P$ should transition to an agent of the form $\text{open_agent } 0 \ a \ A$, 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 $\text{abs_def } P$, we just check that P is fully bound.

For a concretion $\text{conc_def } n \ P \ Q$, 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 $\setminus u$ is the union of finite sets. If $n = 0$, then $\text{bn } P$ and $\text{bn } Q$ are empty, i.e., P and Q are fully bound. Note that in a well-formed fully bound concretion $\text{conc_def } n \ P \ Q$, 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 $\text{subst_lab } l \ a \ b$ 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 $a \notin \text{fn}(P)$ is necessary: if $P \stackrel{\Delta}{=} 0?.0 \parallel \bar{a}!(0).0$, then $\{0 \rightarrow a\}P$ can perform a communication, but $\{0 \rightarrow b\}P$ cannot for $b \neq a$.

Remark 3.1. We manipulate the bound names of a concretion $\text{conc_def } n \ P \ Q$ 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 , \dots (?)). For example, it is easy to show that P is fully bound iff $\text{bn}(P)$ is empty (? , ?). We can also prove a more general version of the lemma open_close using bn (?).

661 **Lemma** open_close_gen $\{V:\mathbf{Set}\}: \forall (P:\text{proc } V) k a,$
 662 $k \notin \text{bn } P \rightarrow \text{open } k a (\text{close } k a P) = P.$

664 4 Equivalences on Processes

665 We discuss here the formalization of the equivalences we
 666 need on processes, namely bisimilarity and structural con-
 667 gruence. We represent relations using the TLC type binary,
 668 and we write \circ for the composition of two relations.
 669

670 4.1 Bisimilarity

671 Even though bisimilarity can be defined using a coinductive
 672 datatype, we prefer to use the set theoretic approach, where
 673 two terms are bisimilar if there exists a bisimulation relat-
 674 ing them. Not only it corresponds to Definition 2.2, but it
 675 is also more tractable in Coq. In the following, test_proc ,
 676 test_abs , and test_conc are notations representing the test-
 677 ing conditions of Definition 2.2 for each kind of agent (♣).
 678 In test_abs , the concretion we use to compare abstractions
 679 is fully bound and well-formed, and similarly in test_conc ,
 680 the testing abstraction is fully bound.
 681

682 **Definition** simulation (Rel: binary proc \emptyset) :=
 683 $\text{is_proc_Rel } \text{Rel} \wedge$
 684 $\forall P Q, \text{Rel } P Q \rightarrow \text{test_proc } \text{Rel } P Q \wedge$
 685 $\text{test_abs } \text{Rel } P Q \wedge \text{test_conc } \text{Rel } P Q.$

686 We restrict the notion of simulation to relations on fully
 687 bound processes thanks to the predicate is_proc_Rel (♣). We
 688 then define bisimulation and bisimilarity as in Definition 2.2
 689 (♣, ♣). The definition of open extension is simple thanks to
 690 the chosen representation of process variables.

691 **Definition** open_extension $\{V:\mathbf{Set}\} (\text{Rel}:\text{binary proc}\emptyset)$
 692 $(P Q:\text{proc } V) :=$
 693 $\forall (f: V \rightarrow \text{proc}\emptyset), (\forall v, \text{is_proc } (f v)) \rightarrow$
 694 $\text{Rel } (\text{bind } f P) (\text{bind } f Q).$

695 Since the free variables of P and Q are in V , a closing substitu-
 696 tion is simply a function from V to $\text{proc}\emptyset$. The condition on f
 697 ensures that f maps variables to fully bound processes.
 698

699 4.2 Structural Congruence

700 Structural congruence is denoted as struct_congr (♣) in the
 701 development. We prove that its restriction to fully bound
 702 closed processes, written $\text{sc}\emptyset$, is a bisimulation. The simula-
 703 tion proof is by induction on the derivation of $P \equiv Q$, and
 704 then by case analysis on the transition performed by P . The
 705 proof is quite lengthy because of the number of cases in the
 706 definition of \equiv and the number of possible transitions, but
 707 most cases are straightforward. The most difficult cases are
 708 the ones manipulating name restrictions.

709 **Inductive** struct_congr $\{V:\mathbf{Set}\}: \text{binary } (\text{proc } V) :=$
 710 $(*...*)$
 711 $(*\text{nu } a (P // Q) = (\text{nu } a P) // Q \text{ if } a \notin \text{fn } Q*)$
 712 $| \text{sc_scope}: \forall P Q, \text{struct_congr}$
 713 $(\text{nu } (P // (\text{shiftN } 1 Q))) ((\text{nu } P) // Q)$
 714 $(*\text{nu } a \text{ nu } b P = \text{nu } b \text{ nu } a P*)$
 715

716 $| \text{sc_nu_nu}: \forall P, \text{struct_congr}$
 717 $(\text{nu } \text{nu } P) (\text{nu } \text{nu } (\text{mapN } (\text{permut } 1) P))$

718 As we can see from the definition of permut in Figure 2,
 719 $\text{permut } 1$ exchanges 0 and 1 and leaves the other bound
 720 names unchanged. The difficulty with these two structural
 721 rules is when checking outputs, as we have to distinguish
 722 cases based on whether the name restrictions bind a name
 723 in the message.
 724

725 Once we prove $\text{sc}\emptyset$ is a bisimulation (♣), we can define
 726 bisimulation up to $\text{sc}\emptyset$ as in Definition 2.4 (♣), and prove it
 727 is a sound up-to technique in the sense of Lemma 2.5 (♣).
 728

729 4.3 Renaming Lemmas

730 The most intricate proof in the formalization outside of the
 731 proof of the main result is the renaming lemma for the bisim-
 732 ilarity; we sketch the proof here to see why. We use mathe-
 733 matical notations for readability, but we modify the syntax
 734 of Figure 1 to stay faithful to the formalization. We use X, Y
 735 to range over indices representing process variables, written
 736 $0, 1, \dots, k$ to range over indices representing bound names,
 737 written $0, 1, \dots$, and a, b, c to range over free names.
 738

739 $P ::= \emptyset \mid X \mid P \parallel P \mid N?.P \mid \overline{N}!(P).P \mid \nu.P \quad N ::= k \mid a$

740 For example, the process $\nu a.(\overline{a}!(\emptyset).\emptyset \parallel b?.X)$ is now writ-
 741 ten $\nu.(\overline{0}!(\emptyset).\emptyset \parallel b?.0)$. Abstractions F are just processes P
 742 such that $\text{fv}(P) \subseteq \{\emptyset\}$, and concretions C are now written
 743 $\nu^n \langle P \rangle Q$, where n is the number of name restrictions. We
 744 write $\text{fn}(A)$ and $\text{bn}(A)$ for the free names and dangling bound
 745 names of an agent A , and we write $\{k \rightarrow a\}A$ and $\{k \leftarrow a\}A$
 746 for respectively the opening and closing operations, extended
 747 to all agents. We write $l\{b/a\}$ for the renaming operation
 748 on labels $\text{subst_lab } l a b$. Given two sets S_1, S_2 , we write
 749 $S_1 \# S_2$ if $S_1 \cap S_2 = \emptyset$.
 750

751 In this section, we use extensively the following decom-
 752 position result (♣).
 753

754 **Lemma 4.1.** *For all k, x , and A such that $k \notin \text{bn}(A)$, there
 755 exists A' such that $A = \{k \rightarrow x\}A'$ and $x \notin \text{fn}(A')$.*
 756

757 Indeed, take $A' = \{k \leftarrow x\}A$, and then conclude with lemma
 758 open_close_gen . We also use this commuting property of
 759 opening (♣).
 760

761 **Lemma 4.2.** *For all P, k_1, k_2, a , and $b, k_1 \neq k_2$ implies
 762 $\{k_1 \rightarrow a\}\{k_2 \rightarrow b\}P = \{k_2 \rightarrow b\}\{k_1 \rightarrow a\}P$.*
 763

764 To prove a renaming lemma for the bisimilarity, we define
 765 a general renaming criterion on relations (♣).
 766

767 **Definition 4.3.** A relation \mathcal{R} is stable by renaming if for all
 768 k, a, P , and Q such that $\{k \rightarrow a\}P, \{k \rightarrow a\}Q$ are fully bound
 769 and $a \notin \text{fn}(P) \cup \text{fn}(Q)$, $\{k \rightarrow a\}P \mathcal{R} \{k \rightarrow a\}Q$ implies that
 770 for all b such that $b \notin \text{fn}(P) \cup \text{fn}(Q)$, $\{k \rightarrow b\}P \mathcal{R} \{k \rightarrow b\}Q$.
 771

772 This renaming notion is stricter than for the LTS (Lemma
 773 1ts_rename in Section 3.3), as it requires b to be fresh from P
 774

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 (♣).

$$\frac{P \mathcal{R} Q}{P \mathcal{R}^\rightarrow Q} \quad \frac{\{a, b\} \# \text{fn}(P) \cup \text{fn}(Q) \quad \{k \rightarrow a\}P \mathcal{R}^\rightarrow \{k \rightarrow a\}Q}{\{k \rightarrow b\}P \mathcal{R}^\rightarrow \{k \rightarrow b\}Q}$$

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

Let P, Q, a , and b such that $a \neq b$, $\{a, b\} \# \text{fn}(P) \cup \text{fn}(Q)$, $\{k \rightarrow a\}P \mathcal{R}^\rightarrow \{k \rightarrow a\}Q$, and $\{k \rightarrow b\}P \mathcal{R}^\rightarrow \{k \rightarrow b\}Q$. We show that the transitions from $\{k \rightarrow b\}P$ are matched by $\{k \rightarrow b\}Q$. The proof is by induction on the derivation of \mathcal{R}^\rightarrow : assuming that the simulation tests hold for $\{k \rightarrow a\}P \mathcal{R}^\rightarrow \{k \rightarrow a\}Q$, we show that they hold for $\{k \rightarrow b\}P \mathcal{R}^\rightarrow \{k \rightarrow b\}Q$ 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 $\{k \rightarrow b\}P \xrightarrow{c} F_1$ for some F_1 . Let C be a well-formed fully bound concretion. We want to find F_2 such that $\{k \rightarrow b\}Q \xrightarrow{c} F_2$ and $F_1 \bullet C \mathcal{R}^\rightarrow F_2 \bullet C$. The main idea is as follows. By Lemma 4.1, we have in fact $\{k \rightarrow b\}P \xrightarrow{c} \{k \rightarrow b\}F'_1$ for some F'_1 , so with `lts_rename`, we have $\{k \rightarrow a\}P \xrightarrow{c\{a/b\}} \{k \rightarrow a\}F'_1$. At this point, we want to use the induction hypothesis on $\{k \rightarrow a\}P \mathcal{R}^\rightarrow \{k \rightarrow a\}Q$ to get F'_2 such that $\{k \rightarrow a\}Q \xrightarrow{c\{a/b\}} \{k \rightarrow a\}F'_2$ and $\{k \rightarrow a\}F'_1 \bullet C \mathcal{R}^\rightarrow \{k \rightarrow a\}F'_2 \bullet C$. We want to write this as $\{k \rightarrow a\}(F'_1 \bullet C) \mathcal{R}^\rightarrow \{k \rightarrow a\}(F'_2 \bullet C)$, to then use \mathcal{R}^\rightarrow to rename a into b , but we need $\{a, b\} \# \text{fn}(F'_1 \bullet C) \cup \text{fn}(F'_2 \bullet C)$, 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 $C = \{k \rightarrow b\}\{k+1 \rightarrow a\}C'$ for some C' such that $\{a, b\} \# \text{fn}(C')$. We then apply the induction hypothesis on $\{k \rightarrow a\}P \mathcal{R}^\rightarrow \{k \rightarrow a\}Q$ not with C , but with the concretion $C'_{a,d} \triangleq \{k \rightarrow a\}\{k+1 \rightarrow d\}C'$, where d is a fresh name. We get F''_2 such that $\{k \rightarrow a\}Q \xrightarrow{c\{a/b\}} F''_2$ and $\{k \rightarrow a\}F'_1 \bullet C'_{a,d} \mathcal{R}^\rightarrow F''_2 \bullet C'_{a,d}$. By Lemma 4.1, $F''_2 = \{k \rightarrow a\}F'_2$ for some F'_2 such that $a \notin \text{fn}(F'_2)$, so we have in fact $\{k \rightarrow a\}F'_1 \bullet C'_{a,d} \mathcal{R}^\rightarrow \{k \rightarrow a\}F'_2 \bullet C'_{a,d}$. We write this as

$$\{k \rightarrow a\}(F'_1 \bullet \{k+1 \rightarrow d\}C') \mathcal{R}^\rightarrow \{k \rightarrow a\}(F'_2 \bullet \{k+1 \rightarrow d\}C'). \quad (1)$$

We can prove that a and b do not occur in F'_1, F'_2, d , and C' , either by construction or because $\{a, b\} \# \text{fn}(P) \cup \text{fn}(Q)$; we

can thus use \mathcal{R}^\rightarrow to rename a into b in (1):

$$\{k \rightarrow b\}(F'_1 \bullet \{k+1 \rightarrow d\}C') \mathcal{R}^\rightarrow \{k \rightarrow b\}(F'_2 \bullet \{k+1 \rightarrow d\}C'). \quad (2)$$

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

$$\{k+1 \rightarrow d\}\{k \rightarrow b\}(F'_1 \bullet C') \mathcal{R}^\rightarrow \{k+1 \rightarrow d\}\{k \rightarrow b\}(F'_2 \bullet C'). \quad (3)$$

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}^\rightarrow , and if we distribute back the opening operations, we get

$$\{k \rightarrow b\}F'_1 \bullet \{k+1 \rightarrow a\}\{k \rightarrow b\}C' \mathcal{R}^\rightarrow \{k \rightarrow b\}F'_2 \bullet \{k+1 \rightarrow a\}\{k \rightarrow b\}C'. \quad (4)$$

But $\{k+1 \rightarrow a\}\{k \rightarrow b\}C' = C$, so we have $\{k \rightarrow b\}F'_1 \bullet C \mathcal{R}^\rightarrow \{k \rightarrow b\}F'_2 \bullet C$, as wished.

What is left to prove is $\{k \rightarrow b\}Q \xrightarrow{c} \{k \rightarrow b\}F'_2$; but we know that $\{k \rightarrow a\}Q \xrightarrow{c\{a/b\}} \{k \rightarrow a\}F'_2$, so by `lts_rename`, we have $\{k \rightarrow b\}Q \xrightarrow{c\{a/b\}\{b/a\}} \{k \rightarrow b\}F'_2$; 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}° .*

The proof requires several renamings, as in that of Lemma 4.4. Indeed, let P, Q, a , and b such that $a \neq b$, $\{a, b\} \# \text{fn}(P) \cup \text{fn}(Q)$, and $\{k \rightarrow a\}P \mathcal{R}^\circ \{k \rightarrow a\}Q$. We want to prove that $\{k \rightarrow b\}P \mathcal{R}^\circ \{k \rightarrow b\}Q$, i.e., for all closing substitution σ , $(\{k \rightarrow b\}P)\sigma \mathcal{R} (\{k \rightarrow b\}Q)\sigma$. To conclude, we would like to use the hypothesis that \mathcal{R} is stable by renaming on $(\{k \rightarrow a\}P)\sigma \mathcal{R} (\{k \rightarrow a\}Q)\sigma$, which we would like to rewrite into $\{k \rightarrow a\}(P\sigma) \mathcal{R} \{k \rightarrow a\}(Q\sigma)$. 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., $\{k \rightarrow b\}(F'_1 \bullet C')$ in the proof of Lemma 4.4, the agents F'_1 and C' are not fully bound. This justifies why process substitution (performed in ♣) should be defined on processes with dangling names.

5 Howe's Method

5.1 Sketch of the Method

Howe's method [9, 10] is a systematic proof technique to show that a bisimilarity \mathcal{B} (and its open extension \mathcal{B}°) is compatible. The method can be divided in three steps: first,

```

881 Inductive howe {V:Set}: binary proc0  $\rightarrow$  proc V  $\rightarrow$  proc V  $\rightarrow$  Prop :=
882 | howe_comp:  $\forall$  Rel P Q R, is_proc Q  $\rightarrow$  howe Rel P R  $\rightarrow$  open_extension Rel R Q  $\rightarrow$  howe Rel P Q
883 | howe_nil :  $\forall$  Rel, howe Rel PO PO
884 | howe_var :  $\forall$  Rel x, howe Rel (pr_var x) (pr_var x)
885 | howe_par :  $\forall$  Rel P Q P' Q', howe Rel P Q  $\rightarrow$  howe Rel P' Q'  $\rightarrow$  howe Rel (P // P') (Q // Q')
886 | howe_inp :  $\forall$  Rel (a:var) P Q, @howe (incV V) Rel P Q  $\rightarrow$  howe Rel (a ? P) (a ? Q)
887 | howe_out :  $\forall$  Rel (a:var) P Q P' Q', howe Rel P Q  $\rightarrow$  howe Rel P' Q'  $\rightarrow$  howe Rel (a!(P) P') (a !(Q) Q')
888 | howe_nu :  $\forall$  L Rel P Q, ( $\forall$  x, x \notin L  $\rightarrow$  howe Rel (open  $\emptyset$  x P) (open  $\emptyset$  x Q))  $\rightarrow$  howe Rel (nu P) (nu Q).
889
889 Notation pseudo_sim :=  $\forall$  {V W:Set} Rel (P1 Q1:proc V) (P2 Q2:proc W) a F1 C1 (f:V  $\rightarrow$  proc0) (g: W  $\rightarrow$  proc0),
890 simulation Rel  $\rightarrow$  refl0 Rel  $\rightarrow$  rename_compatible Rel  $\rightarrow$  (Rincl sc0 Rel)  $\rightarrow$  trans Rel  $\rightarrow$ 
891 howe Rel P2 Q2  $\rightarrow$  howe Rel P1 Q1  $\rightarrow$  ( $\forall$  v, is_proc (f v))  $\rightarrow$  ( $\forall$  v, is_proc (g v))  $\rightarrow$ 
892 lts (bind f P1) (inp a) (ag_abs F1)  $\rightarrow$  lts (bind g P2) (out a) (ag_conc C1)  $\rightarrow$ 
893  $\exists$  F2 C2, lts (bind f Q1) (inp a) (ag_abs F2)
894  $\wedge$  lts (bind g Q2) (out a) (ag_conc C2)  $\wedge$  (sc0  $\circ$  Rel  $\circ$  sc0) (appr F1 C1) (appr F2 C2).

```

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

prove some basic properties on the *Howe's closure* \mathcal{B}^\bullet of the relation. By construction, \mathcal{B}^\bullet contains \mathcal{B}° 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} .

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

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.

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

The base cases for this definition are the base cases of the syntax of processes, namely $\emptyset \mathcal{R}^\bullet \emptyset$ and $X \mathcal{R}^\bullet X$.

Instantiating \mathcal{R} as \mathcal{B} , \mathcal{B}^\bullet is compatible by definition. The composition with \mathcal{B}° 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 [11] 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 $P_1 \xrightarrow{\bar{a}} C_1$ and $P_2 \xrightarrow{a} F_1$, then there exist C_2, F_2 such that $Q_1 \xrightarrow{\bar{a}} C_2, Q_2 \xrightarrow{a} F_2$, and $F_1 \bullet C_1 \equiv \sim_c^\bullet F_2 \bullet C_2$.*

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 HO π , 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 [11] 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 (\clubsuit).

Lemma `howe_implies_proc` {V:Set}: \forall Rel (P Q:proc V),
`howe Rel P Q \rightarrow is_proc P \wedge is_proc Q.`

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

We now prove a renaming lemma for Howe's closure, as we need it in the proof of Lemma 5.2 (\clubsuit).

Lemma 5.4. *If \mathcal{R} is stable by renaming, then so is \mathcal{R}^\bullet .*

Let P, Q, a , and b such that $a \neq b$, $a \notin \text{fn}(P) \cup \text{fn}(Q)$, $b \notin \text{fn}(P) \cup \text{fn}(Q)$, and $\{k \rightarrow a\}P \mathcal{R}^\bullet \{k \rightarrow a\}Q$. We prove that $\{k \rightarrow b\}P \mathcal{R}^\bullet \{k \rightarrow b\}Q$ by induction on the size of the derivation of $\{k \rightarrow a\}P \mathcal{R}^\bullet \{k \rightarrow a\}Q$. 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 $\{k \rightarrow a\}P \mathcal{R}^\bullet \{k \rightarrow a\}Q$. In the formalization, we define a predicate $\text{howe}' \text{ Rel } P \ Q \ n$ 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 $\{k \rightarrow a\}P \mathcal{R}^\bullet R \mathcal{R}^\circ \{k \rightarrow a\}Q$ for some R . We would like to apply the induction hypothesis on $\{k \rightarrow a\}P \mathcal{R}^\bullet R$, to rename a into b . Even though $R = \{k \rightarrow a\}R'$ for some R' such that $a \notin \text{fn}(R')$ with Lemma 4.1, we still cannot apply the induction hypothesis to rename a into b , as we may have $b \in \text{fn}(R')$. Instead, we first apply the induction hypothesis to $\{k \rightarrow a\}P \mathcal{R}^\bullet R$ to rename b into a fresh name c . This leaves $\{k \rightarrow a\}P$ unchanged, since $b \notin \text{fn}(\{k \rightarrow a\}P)$, so we get $\{k \rightarrow a\}P \mathcal{R}^\bullet R_c$ for some R_c . We then apply the induction hypothesis again to rename a into b to obtain $\{k \rightarrow b\}P \mathcal{R}^\bullet R'_c$ for some R'_c . We can do the same reasoning on $R \mathcal{R}^\circ \{k \rightarrow a\}Q$ using Lemma 4.6 to get $R'_c \mathcal{R}^\circ \{k \rightarrow b\}Q$. As a result, we have $\{k \rightarrow b\}P \mathcal{R}^\bullet R'_c \mathcal{R}^\circ \{k \rightarrow b\}Q$, i.e., $\{k \rightarrow b\}P \mathcal{R}^\bullet \{k \rightarrow b\}Q$, as wished.

We state Lemma 5.2 using the TLC transitive closure tclosure , defined as follows.

$$\frac{P \mathcal{R} Q}{P \mathcal{R}^+ Q} \quad \frac{P \mathcal{R}^+ R \quad R \mathcal{R}^+ Q}{P \mathcal{R}^+ Q}$$

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 $P (\mathcal{R}^\bullet)^+ Q$ (IH₁). 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 $P \mathcal{R}^\bullet Q$ (IH₂). First, suppose $P \mathcal{R}^\bullet R \mathcal{R}^\circ Q$ for some R . Using (IH₂), 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 $P = \text{op}(\widetilde{P}')$, $Q = \text{op}(\widetilde{Q}')$, and $\widetilde{P}' \mathcal{R}^\bullet \widetilde{Q}'$, then we have $\widetilde{Q}' (\mathcal{R}^\bullet)^+ \widetilde{P}'$ using (IH₂). We must show that $\widetilde{Q}' (\mathcal{R}^\bullet)^+ \widetilde{P}'$ implies $\text{op}(\widetilde{Q}') (\mathcal{R}^\bullet)^+ \text{op}(\widetilde{P}')$, which is direct for all the operators, except name restriction. In that case we have $\forall a, a \notin L \Rightarrow \{0 \rightarrow a\}Q (\mathcal{R}^\bullet)^+ \{0 \rightarrow a\}P$ for some L , and we must prove $\nu.Q (\mathcal{R}^\bullet)^+ \nu.P$. We want to do an induction on $\{0 \rightarrow a\}Q (\mathcal{R}^\bullet)^+ \{0 \rightarrow a\}P$ (IH₃), but we have to choose a fresh a first. As a result, in the base case we have $\{0 \rightarrow a\}Q \mathcal{R}^\bullet \{0 \rightarrow a\}P$ only for a given $a \notin L$, but to apply howe_nu , we want $\forall b, b \notin L' \Rightarrow \{0 \rightarrow b\}Q \mathcal{R}^\bullet \{0 \rightarrow b\}P$ for some L' . We need Lemma 5.4 to rename a into

$b \notin L \cup \text{fn}(Q) \cup \text{fn}(P)$ 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 $\{V \ \text{W} : \text{Set}\}$:

$$\begin{aligned} & \forall \text{Rel } (P \ Q : \text{proc } V) \ (f \ g : V \rightarrow \text{proc } W), \\ & (\forall x, \text{howe } \text{Rel } (f \ x) \ (g \ x)) \rightarrow \text{howe } \text{Rel } P \ Q \rightarrow \\ & \text{howe } \text{Rel } (\text{bind } f \ P) (\text{bind } g \ Q). \end{aligned}$$

Lemma howe_subst $\{V : \text{Set}\}$: $\forall \text{Rel } (P' \ Q' : \text{proc } V) \ P \ Q$,

$$\begin{aligned} & \text{howe } \text{Rel } P \ Q \rightarrow \text{howe } \text{Rel } P' \ Q' \rightarrow \\ & \text{howe } \text{Rel } (\text{subst } P \ P') (\text{subst } Q \ Q'). \end{aligned}$$

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 [11]. 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 [11]. 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 $\{V : \text{Set}\}$: $\forall \text{Rel } (P_1 \ Q_1 : \text{proc } V)$

$$\begin{aligned} & (P' \ Q' : \text{proc} \emptyset) \ a \ F_1 \ (f : V \rightarrow \text{proc} \emptyset), \\ & \text{simulation } \text{Rel} \rightarrow \text{refl} \emptyset \ \text{Rel} \rightarrow \text{rename_compatible } \text{Rel} \\ & \rightarrow (\forall P \ Q, \ \text{Rel } (P \ // \ P_0) \ (Q \ // \ P_0) \rightarrow \text{Rel } P \ Q) \rightarrow \\ & \text{howe } \text{Rel } P_1 \ Q_1 \rightarrow \text{howe } \text{Rel } P' \ Q' \rightarrow \\ & (\forall v, \ \text{is_proc } (f \ v)) \rightarrow \\ & \text{Its } (\text{bind } f \ P_1) \ (\text{inp } a) \ (\text{ag_abs } F_1) \rightarrow \\ & \exists F_2, \ \text{Its } (\text{bind } f \ Q_1) \ (\text{inp } a) \ (\text{ag_abs } F_2) \wedge \\ & \text{howe } \text{Rel } (\text{asubst } F_1 \ P') \ (\text{asubst } F_2 \ Q'). \end{aligned}$$

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 $Q_1 \sigma_g \xrightarrow{a} F_2$ and $F_1 \bullet v^0 \langle P' \rangle \otimes \mathcal{R}^\bullet F_2 \bullet v^0 \langle Q' \rangle \otimes$. The condition $P \ // \ \mathcal{R} \ Q \ // \ \otimes \Rightarrow P \ \mathcal{R} \ Q$ —which is weaker than containing structural

congruence—then allows to remove any \emptyset 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 $X\sigma_g \xrightarrow{\bar{a}} C$ for some C , and `howe_out`, where $P_2 = \bar{a}!(P_2^1).P_2^2$, $Q_2 = \bar{a}!(Q_2^1).Q_2^2$, $P_2^1 \mathcal{R}^\bullet Q_2^1$, 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}:  $\forall$  Rel (P2 Q2:proc V)
  (P' Q':proc1) a C1 (g:V  $\rightarrow$  proc0),
simulation Rel  $\rightarrow$  refl0 Rel  $\rightarrow$  trans Rel  $\rightarrow$ 
rename_compatible Rel  $\rightarrow$  (Rincl sc0 Rel)  $\rightarrow$ 
howe Rel P2 Q2  $\rightarrow$  howe Rel P' Q'  $\rightarrow$ 
( $\forall$  v, is_proc (g v))  $\rightarrow$ 
lts (bind g P2) (out a) (ag_conc C1)  $\rightarrow$ 
 $\exists$  C2, lts (bind g Q1) (out a) (ag_conc C2)  $\wedge$ 
(sc0  $\circ$  howe Rel  $\circ$  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 [11] 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. 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 [11], 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:  $\forall$  Rel,
simulation Rel  $\rightarrow$  refl0 Rel  $\rightarrow$  trans Rel  $\rightarrow$ 
rename_compatible Rel  $\rightarrow$  (Rincl sc0 Rel)  $\rightarrow$ 
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 [13, 15] propose formalizations of higher-order process calculi. The calculus studied by Maksimović and Schmitt [13] is a sub-calculus of HO π as it does not feature name restriction. Parrow et al. [15] extend an existing formalization of the psi-calculus to accommodate for higher-order communication. The calculus is not truly higher-order as it uses *triggers* [18]: instead of exchanging executable processes, psi-terms exchange private names that are then used to trigger the execution of a process. As a result, substitution is defined on names, as in the π -calculus. However, a triggered semantics cannot be defined in some higher-order calculi, such as calculi with passivation [12]. Besides, the bisimulation of Parrow et al. [15] 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 HO π , as it distinguishes processes that should be considered equivalent [18]. 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, 14, 20] formalize Howe's method in functional languages, and some of them [2, 20] 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 [17], or higher-order abstract syntax (HOAS) [16]—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 its set of free variables. Open extension is not as easily defined in HOAS [14], except in systems with dedicated support for simultaneous substitutions, such as Beluga [20]. 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 $v\tilde{b}. \langle P \rangle Q$, because the set of bound names \tilde{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 . Similarly, we do not think HOAS is well suited to represent channel names and concretions; the main idea behind HOAS is to encode the bindings structure of the object language (here, $\text{HO}\pi$) into the binding constructs of the meta-language (here, Coq) to benefit from the infrastructure of the meta-language, in particular substitution. But we do not substitute names, so it is not clear how HOAS could help here.

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 $va.(\bar{a}!(\odot).\odot \parallel b?X.X)$ produces $vb.(\bar{b}!(\odot).\odot \parallel a?X.X)$. 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 $P \mathcal{R}^* Q$ and $R \mathcal{R}^* S$ implies $P\{R/\emptyset\} \mathcal{R}^* Q\{S/\emptyset\}$, 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 $\text{HO}\pi$ 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 [11]. 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 [11]. 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 [12].

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 [7], to benefit from cofinite quantification in a nominal framework.

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] Richard S. Bird and Ross Paterson. 1999. De Bruijn Notation as a Nested Datatype. *J. Funct. Program.* 9, 1 (1999), 77–91.
- [4] Arthur Charguéraud. [n. d.]. TLC: A non-constructive library for Coq. ([n. d.]). <http://www.chargueraud.org/softs/tlc/>
- [5] Arthur Charguéraud. 2012. The Locally Nameless Representation. *Journal of Automated Reasoning* 49, 3 (2012), 363–408.
- [6] 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.
- [7] Edsko de Vries and Vasileios Koutavas. 2012. Locally Nameless Permutation Types. (2012). available at <http://edsko.net/pubs/lnpt.pdf>.
- [8] 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.
- [9] Andrew D. Gordon. 1995. Bisimilarity as a theory of functional programming. *Electronic Notes in Theoretical Computer Science* 1 (1995), 232–252.
- [10] Douglas J. Howe. 1996. Proving Congruence of Bisimulation in Functional Programming Languages. *Information and Computation* 124, 2 (1996), 103–112.
- [11] 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.
- [12] Sergueï Lenglet, Alan Schmitt, and Jean-Bernard Stefani. 2011. Characterizing contextual equivalence in calculi with passivation. *Information and Computation* 209, 11 (2011), 1390–1433.
- [13] 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.
- [14] 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.
- [15] 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.
- [16] Frank Pfenning and Conal Elliott. 1988. Higher-Order Abstract Syntax. In *PLDI 88*. ACM, Atlanta, Georgia, USA, 199–208.
- [17] Andrew M. Pitts. 2003. Nominal logic, a first order theory of names and binding. *Inf. Comput.* 186, 2 (2003), 165–193.
- [18] Davide Sangiorgi. 1996. Bisimulation for Higher-Order Process Calculi. *Information and Computation* 131, 2 (1996), 141–178.
- [19] Davide Sangiorgi and David Walker. 2001. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press.
- [20] 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.

A Appendix

We define two processes P and Q such that $b \in \text{fn}(P) \cup \text{fn}(Q)$, $P \sim Q$, 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

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

the example would be $P \triangleq \bar{a}!(\emptyset).\emptyset \parallel b?X.\emptyset$ and $Q \triangleq \bar{a}!(\emptyset).b?X.\emptyset + b?X.\bar{a}!(\emptyset).\emptyset$. 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.

We do not have $+$ in HO π , but we can mimick this example. We first define a process P_τ such that $P_\tau \xrightarrow{\tau} P_\tau$, without any other possible transition.

$$P_{rec} \triangleq e?X.(X \parallel \bar{e}!(X).\emptyset) \quad P_\tau \triangleq ve.(P_{rec} \parallel \bar{e}!(P_{rec}).\emptyset)$$

Let $R \triangleq d?Y.a?X.\bar{b}!(\emptyset).c?Z.\emptyset \parallel d?Y.\bar{b}!(\emptyset).(a?X.\emptyset \parallel c?Z.\emptyset)$, and

$$P \triangleq P_\tau \parallel vd.(\bar{d}!(\emptyset).\emptyset \parallel R)$$

$$Q \triangleq P_\tau \parallel vd.(\bar{d}!(\emptyset).\emptyset \parallel R \parallel d?Y.(a?X.\emptyset \parallel \bar{b}!(\emptyset).c?Z.\emptyset))$$

In P , the communication on the bound name d makes a choice between the two sub-processes of R , while we choose between three processes in Q . Any transition from P is clearly matched by the same transition in Q . Conversely, the only transition which is not obvious to check is when Q chooses the third process, i.e.,

$$Q \xrightarrow{\tau} P_\tau \parallel vd.(\emptyset \parallel R \parallel a?X.\emptyset \parallel \bar{b}!(\emptyset).c?Z.\emptyset).$$

Let Q''' be the resulting process of this transition; then P matches with $P \xrightarrow{\tau} P$, by a τ -step from P_τ . The process Q' can then either do an input on a and then an output on b , or do the output on b before the input on a . In the former case, P matches by choosing the process on the left in R , and in the latter case, it chooses the process on the right.

In contrast, we have $\{a \rightarrow b\}P \not\sim \{a \rightarrow b\}Q$, as $\{a \rightarrow b\}Q \xrightarrow{\tau} \{a \rightarrow b\}Q'$, and $\{a \rightarrow b\}Q'$ can then do an internal communication on b , making an input on c possible. It is not possible to make the input on c available in $\{a \rightarrow b\}P$ with only $\xrightarrow{\tau}$ -transitions.