# A higher-order specification of the $\pi$-calculus

Jolle Despeyroux

INRIA*

**Abstract.** We present a formalization of a typed $\pi$-calculus in the Calculus of Inductive Constructions. We give the rules for type-checking and for evaluation and formalize a proof of type preservation in the Coq system. The encoding of the $\pi$-calculus in Coq uses Coq functions to represent bindings of variables. This kind of encoding is called a higher-order specification. It provides a concise description of the calculus, leading to simple proofs. The specification we propose for the pi-calculus formalizes communication by means of function application.

## 1 Introduction

The $\pi$-calculus [MPW92,Mil91] is a model of concurrent computation based upon the notion of naming. Processes receive and emit names, which denote channels.

We propose here new formalizations of both evaluation and typing rules for a typed $\pi$-calculus, with the aim to enable machine-checked proofs of various properties of languages based on this calculus. As a simple, typical illustration of such a proof, we give a proof of preservation of types, also called the subject-reduction property, for the calculus.

For our experiment, we chose the Coq system [BBC$^+$97,HKPM97], based on the Calculus of Inductive Constructions [CH88,PM92]. In this system, proofs are performed by applying tactics, in a goal-directed manner. We like this system because of its rich meta-language (a higher-order functional language, allowing full dependent inductive types), and because it is based on Type Theory.

Thanks to the use of the *higher-order abstract syntax* technique, giving rise to so-called *higher-order specifications*, our formalization of the $\pi$-calculus does not require definitions for free or bound variables in a term. Nor does it require definitions of notions of substitutions, which are implemented using the meta-level application, i.e. application available in the Logical Framework used to implement our calculus (which in our case is the Calculus of Inductive Constructions).

Robin Milner [Mil91] introduced notions of *abstractions* and *concretions* leading to a presentation of the calculus in which input processes evaluate to functions and output processes evaluate to concretions, i.e. (intuitively) pairs of a value and a process. Our formalization starts from this idea, replacing concretions by higher-order functions, thus leading to a nicer, more uniform presentation, in which communication is formalized by function application.

Related works include formalizations of various $\pi$-calculi in the same system. These works were conducted with different goals in mind. Some of them had the same goals as ours. The others aimed at the study of bisimulation techniques. We shall discuss both kinds of related works in detail at the end of the paper.

* INRIA, 2004 Route des Lucioles, B.P. 93, F-06902 Sophia-Antipolis Cedex, France. `Joelle.Despeyroux@inria.fr` – Unpublished draft. Revised version presented at TCS'00.

**Note on the Coq syntax.** Terms in Coq are terms of a typed $\lambda$-calculus, where the abstraction $\lambda x : t.a$ is written [x:t]a, the dependent product type $\forall x : t.a$ is written (x:t)a, while the non-dependent product type $A \to B$ is simply written $A \to B$. The application of a function $f$ to $x$ is written $(f\ x)$. The expression $\exists x : t.a$ is written (Ex [x:t]a). The type of sets is denoted as Set while the type of propositions in denoted as Prop. As usual in Type Theory, the type $A_1 \to \cdots \to A_n \to B$ where $B$ is a proposition represents (through the Curry-Howard isomorphism) the inference rule with premisses $A_1, \cdots, A_n$ and conclusion $B$. Inductive types are defined by a list of constructors together with their types, as for example for the naturals:

```
Inductive nat : Set := 0 : nat | S : nat -> nat.
```

An inductive definition automatically generates both an inductive principle for the type being defined and elimination principles which enable the definition of recursive functions on this type. In the following, Coq text will be given as above in a typewriter font.

## 2 Syntax

The terms of the $\pi$-calculus are processes, which receive and emit names denoting channels. For the sake of simplicity, we do not consider here the operators of choice and matching. It will be straightforward to add them.

### 2.1 The monadic $\pi$-calculus

Processes are defined by the following rule:

$$\text{Processes}: \ P, Q ::= \bar{x}y.P \mid x(y:t).P \mid \nu x : t.P \mid (P \mid Q) \mid !\, P \mid 0$$

in which $x$ and $y$ belong to an enumerable set of names and $t$ denotes the type of a name as described in section 5.

The expression $\bar{x}y.P$ denotes a process which sends a name $y$ along the channel $x$ (*output* operation), then behaves as $P$. The term $x(y).P$ denotes a process which receives a name $z$ along the channel $x$ (*input* operation), then behaving like $P[z/y]$ ($P$ in which $z$ replaces $y$). The expression $(P \mid Q)$ denotes the *parallel* composition of $P$ and $Q$ where $P$ and $Q$ are concurrently active. 0 denotes the *empty* process. $!\, P$ denotes the infinite process $P \mid P \mid \cdots$; the ! operator (pronounce "bang") is called the *replication* operator. Finally, $\nu x : t.P$ declares $x$ as a private channel with type $t$ in $P$. The $\nu$ operator is called a *restriction* as it restricts the use of the name $x$ to $P$.

In the term $\nu x : t.P$, the name $x$ is bound in the term $P$. Similarly, in the term $x(y).P$, the name $y$ is bound in the term $P$, waiting to be substituted by another name before $P$ can be executed.

Processes evaluate to other processes while performing *actions*. In the standard approach, there are four kinds of actions: input actions $xy$ (input a value for variable $y$ on a channel $x$), output actions $\bar{x}y$ (output value $y$ on a channel $x$), bounded output action $\bar{x}(y:t)$, and silent actions $\tau$. Communications are always silent.

$$\text{Actions}: \ A ::= xy \mid \bar{x}y \mid \bar{x}(y:t) \mid \tau$$

An original, and central notion of the $\pi$-calculus is the notion of *scope extrusion*. A name is said to be extruded when it is a private name sent to a process. In such a case, the scope of the name is extended to the new process which receives it. This phenomenon, which may make the notion of binding in the $\pi$-calculus unclear at first glance, will be illustrated and discussed later on.

The informal account of terms requires the definition of the set of variables in a term, that we denote as $Var(t)$. It also requires the definition of free variables ($F(t)$) and bound variables ($B(t)$) in a term. In the informal presentation of the calculus, we will need a notion of substitution of a variable for another variable in a term, written as usual as $P\{y/x\}$.

### 2.2 Formalization of the syntax

We encode the terms of the calculus in Coq as follows. First we introduce a new parameter *name*.

```
Parameter name: Set.
```

Then we introduce the set of types as a parameter too, stating some properties on this set later on.

```
Parameter typ : Set.
```

The processes are then defined as follows:

```
Inductive proc : Set :=
    Nil  : proc
  | In   : name -> typ  -> (name -> proc) -> proc
  | Out  : name -> name -> proc -> proc
  | Par  : proc -> proc -> proc
  | Bang : proc -> proc
  | Res  : typ  -> (name -> proc) -> proc.
```

The representation of processes requires some explanations. There are two binding operators in the $\pi$-calculus: the input and the restriction operator. We use here Coq functions to represent bindings of variables; hence the type ($name \rightarrow proc$) in the types of input processes and restrictions. This kind of encoding is called a *higher-order specification*. This method of formalization, called the higher-order abstract syntax technique, provides an elegant way to formalize the binding nature of the constructors. In particular, it provides for free a definition of terms up to $\alpha$-*equivalence*. This method also provides an elegant way to formalize *substitutions*, as we shall see later on, when formalizing the rules for evaluation.

Note that the type name is declared as a parameter, not as a variable. This prevents name to be instantiated by an inductive set.

A restricted name never gets instantiated. Nevertheless, using functions (i.e. binding operations) to represent the restriction yields a simple and elegant formalization of the notion of private names.

In our description of the $\pi$-calculus, the only visible argument of an action will be the name of a channel. In other words, actions only need to carry a name. As a consequence, there is no need for bound output actions in our development:

```
Inductive action : Set :=
    InA : name -> action | OutA : name -> action | Tau : action.
```

## 3 Simple examples of evaluation

We give here some simple examples of communication. The simplest case of communication is the following one, where a name $y$ is sent to $Q$, along a channel $x$:

$$\bar{x}y.P \mid x(z:t).Q \xrightarrow{\tau} P \mid Q\{y/z\}$$

Then let us illustrate the phenomenon of scope extrusion. This occurs in a communication where a process sends a private name to an external process, as in:

$$\nu y : t.(\bar{x}y.P) \mid x(z:t').Q \quad \text{which reduces to} \quad \nu y : t.(P \mid Q[y/z]).$$

The private name $y$ has been passed to the external process $Q$. We say that $P$ has extruded the scope of the private channel $y$. We have a similar phenomenon in imperative languages when a local name $y$ is sent to a procedure Q taking its parameters by reference.

## 4 Evaluation rules

The evaluation rules define a judgement $P \xrightarrow{a} Q$ meaning that $P$ evaluates to $Q$ producing an action $a$. The chosen semantics is the early transition semantics.

### 4.1 Informal evaluation rules

We need here the definition of all variables in a term ($Var(t)$), together with the definitions of fresh and bound variables ($F(t)$ and $B(t)$). Moreover, usual informal presentations of the calculus use the open and close rules, which *open and close the scope of a name*, in order to describe the phenomenon of scope extrusion that we illustrated above (section 3). The informal description of the evaluation rules is the following one.

$$\textit{input}: \qquad x(y:t).P \xrightarrow{xz} P\{z/y\}$$

$$\textit{output}: \qquad \bar{x}y.P \xrightarrow{\bar{x}y} P$$

$$\textit{com}: \qquad \frac{P \xrightarrow{xy} P' \qquad Q \xrightarrow{\bar{x}y} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \qquad \frac{P \xrightarrow{xy} P' \qquad Q \xrightarrow{\bar{x}y} Q'}{Q \mid P \xrightarrow{\tau} Q' \mid P'}$$

$$\textit{open}: \qquad \frac{P \xrightarrow{\bar{x}y} P'}{\nu y : t.P \xrightarrow{\bar{x}(y:t)} P'} \; x \neq y$$

$$\textit{close}: \quad \frac{P \xrightarrow{xy} P' \qquad Q \xrightarrow{\bar{x}(y:t)} Q'}{P \mid Q \xrightarrow{\tau} \nu y : t.(P' \mid Q')} \; y \notin F(P) \qquad \frac{P \xrightarrow{xy} P' \qquad Q \xrightarrow{\bar{x}(y:t)} Q'}{Q \mid P \xrightarrow{\tau} \nu y : t.(Q' \mid P')} \; y \notin F(P)$$

$$res: \quad \frac{P \xrightarrow{a} P'}{\nu x : t.P \xrightarrow{a} \nu x : t.P'} \quad x \notin Var(a)$$

$$par: \quad \frac{P \xrightarrow{a} P'}{P \mid Q \xrightarrow{a} P' \mid Q} \quad B(a) \cap F(Q) = \emptyset \qquad \frac{P \xrightarrow{a} P'}{Q \mid P \xrightarrow{a} Q \mid P'} \quad B(a) \cap F(Q) = \emptyset$$

$$bang: \quad \frac{!P \mid P \xrightarrow{a} P'}{!\ P \xrightarrow{a} P'}$$

In the above set of rules, rules for *input* and *output* describe the basic cases of input and output processes. The *com* and *close* rules describe communication, while the rest of the rules describe the propagation of evaluation through a restriction, a parallel composition, or a replication.

The replication rule could be simpler. The advantage of the chosen rule is that it enables the extension of the rules with a choice operator.

### 4.2 Formalization of the evaluation rules

We shall take full advantage here from our choice to use functions in the formalization of the syntax.

First, as input processes are described as functions, we just say that input processes evaluate to themselves. This follows Robin Milner approach [Mil91] where he introduced a notion of *abstraction* to denote the result of the evaluation of input processes. We will define an inductive predicate *evali* for processes performing an input action:

```
Inductive evali : proc -> action -> (name -> proc) -> Prop
```

with as first and simplest case the case for input processes:

```
evali_in: (x:name)(t:typ)(p:name->proc) (evali (In x t p) (InA x) p)
```

Robin Milner also introduced a notion of *concretion* [Mil91], which (intuitively) was a pair of a value and a process, to denote the result of the evaluation of output processes. Then he (informally) described the communication as the "application" of an abstraction to a concretion. We thought it would be nice, in order to formalize the communication by a real application, to describe the evaluation of an output process as yielding a *higher-order function* which will then be applied to the result of the evaluation of an input function in order to describe a communication, as we shall see later on.

Let us say that the evaluation of output processes yields higher-order functions taking functions of type ($name \rightarrow proc$) as arguments. We define an inductive predicate *evalo* for processes performing an output action:

```
Inductive evalo : proc -> action -> ((name->proc) -> proc) -> Prop
```

with as first and simplest case the case for output processes:

```
evalo_out : (x,y:name)(p:proc)
   (evalo (Out x y p) (OutA x) [f:name->proc](Par (f y) p))
```

**Example** We are now in a position to see how we can describe communication on a basic example:

$$\frac{P \xrightarrow{\bar{x}y} P' \qquad Q \xrightarrow{xy} Q'}{\bar{a}b.P \mid a(y:t).Q \xrightarrow{\tau} P' \mid Q'\{b/y\}}$$

The evaluation of the input process $a(y).Q$ gives the function $Q$ itself. Then the result of the evaluation of the output $\bar{a}b.P$ is a function $\lambda f : name \rightarrow proc.((f\ \ b) \mid P)$. Applying this function to $Q$ yields the expected result: $(Q\ \ b) \mid P$.

```
(Out a b P)  --(OutA a)--> [f:name->proc]((f b) | P) = P'
   (In a t Q)   --(In a)-->  Q
-------------------------------------------------------
   (Out a b P) | (In a Q) --Tau--> (P' Q) = (Q b) | P
```

Thus the evaluation of both input and output processes yields functions. Communication is formalized by applying the result of the evaluation of the output process to the result of the evaluation of the input process. Our formalization has replaced concretions by higher-order functions, thus leading to a nicer, more uniform presentation, in which communication is formalized by a real application of a function. On a technical level, the higher-order abstract syntax method enables us to describe *substitution* in evaluation rules by simply using the *application* of a Coq function to terms -names in this case.

It is staightforward to prove (on paper) the consistency of this presentation of the semantics of the $\pi$-calculus with functions with respect to the standard approach.

The rule for example for left communication will be formalized as follows, in a third set of inductive rules describing the evaluation of processes that perform silent actions:

```
Inductive eval :  proc -> action -> proc -> Prop :=
  | eval_coml:
      (p:proc)(x:name)(p':(name->proc)->proc)
      (evalo p (OutA x) p') ->
      (q:proc)(q':name->proc) (evali q (InA x) q') ->
      (eval (Par p q) Tau (p' q'))
  [...].
```

Note that communication is always simply formalized by an application, no matter whether names are extruded or not during the communication. This is the reason why we do not need rules for opening and closing the scope of names in our formalization.

The other rules -for the parallel, replication and restriction operators- need to be duplicated, one for each type of action involved (input, output or silent). Note however that all proofs, either formal or informal, proceed by a case analysis on the type of the action involved anyway. Thus this apparent defect should not be considered as a drawback of our formalization.

Thanks to the use of the higher-order abstract syntax technique, when evaluating a process which declares a private name, we do not need to check that this private name is a fresh name. The freshness of the name naturally comes from the use of a universal quantification on names in the premiss of the rule.

```
eval_res :
      (u:typ)(p:name->proc)(a:action)(q:name->proc)
      ((x:name) (eval (p x) a (q x))) ->
      (eval (Res u p) a (Res u q)).
```

The rules for parallel composition do not require any side condition regarding bound variables either. This is because the actions do not carry values here, but only the name of a channel.

```
eval_parl : (p:proc)(a:action)(p':proc)
      (eval p a p') -> (q:proc) (eval (Par p q) a (Par p' q))
```

The rule for replication (omitted here) is straightforward.

## 5 Type checking rules

The type system that we chosed uses a notion of *directionality* of names [PS95], which is their ability to be used as emitters, receivers or both. This name usage discipline is implemented by type-checking rules. This type system, while being simple, is at the same time widely used as witnessed by the experiment of Pict [PT97] and non-trivial because of the sub-typing that it captures. For our purpose, it is more interesting than the type system for the polyadic $\pi$-calculus [Mil91], whose typing rules are totally straightforward.

### 5.1 Types

A type $t$ of a name of a channel is the union of its *capacity*, which represents how the channel can be used (read, write or both) and the *types* of the names that can transit in it:

$$\begin{array}{ll} \text{Capacities :} & c ::= \mathit{read} \mid \mathit{write} \mid \mathit{both} \\ \text{Types :} & t ::= (c, t) \end{array}$$

We formalize the capacities of names as follows:

```
Inductive cap : Set :=
   read : cap | write : cap | both : cap. % | top : cap.
```

Remember we only defined types as:

```
Parameter typ : Set.
```

This means that we do not impose a particular implementation of types. Instead we state some properties this implementation must enjoy. First we must be able to extract both the capacity $\langle t\rangle$ and the type $[t]$ parts of a type $t$:

```
Parameter get_cap : typ -> cap.
Parameter get_typ : typ -> typ.
```

Then we have a subtyping relation on capacities, denoted as $\leq$ and defined as follows:

$$c \leq c \mid both \leq c$$

The subtyping relation on capacities induces a subtyping relation on types, that we still denote as $\leq$ and define as follows:

$$\begin{array}{c}\langle s\rangle = both\ \wedge \langle t\rangle = both\ \wedge [s] \leq [t] \wedge [t] \leq [s] \wedge s \leq t \\ \langle s\rangle \leq read\ \wedge \langle t\rangle = read\ \wedge [s] \leq [t] \Rightarrow s \leq t \\ \langle s\rangle \leq write \wedge \langle t\rangle = write \wedge [t] \leq [s] \Rightarrow s \leq t\end{array}$$

The $\leq$ predicates are formalized by the following inductive types, the obvious rules of which we omit here:

```
Inductive stype_cap : cap -> cap -> Prop
Inductive stype : typ -> typ -> Prop
```

The covariance of the read capability and the contravariance of the write capability introduce a subtyping in depth, as the typing rules will show.

A process is said to be well *typed in a particular environment* which gives types to each of its free name. In contrary to the usual case in typed programming languages, the type-checking rules do not give types to terms in the $\pi$-calculus.

### 5.2 Examples

The process $\bar{x}y \mid x(u:t).\bar{u}v \mid \bar{x}z$ is well typed in an environment where $x$ has the capabilities *both* (both read and write). $\bar{x}y$ alone is well typed in an environment where $x$ has a capabilities $c \leq write$, which means that $c$ is either *both* or *write*.

The following example shows how certain interferences between processes can be eliminated. Consider the process $P = \nu y : (both, t).(\bar{x}y.y(z:t'))$, where $x$ has type $(write, (write, t))$. The process $P$ gives the writing capacity to the outside, while keeping the reading capacity for himself on the name $y$. After the transmission of $y$, a process sending a value to $P$ on channel $x$ is sure that this value cannot be captured by another process.

The previous example also illustrates the use of subtyping in depth.

### 5.3 Informal typing rules

The informal typing rules, defining a judgement $env \vdash proc$, are self-explanatory. (We omit the obvious rules for par and bang.) Note the need for the $Var$ predicate in the input and res rules.

$$typ\_nil: \quad \Gamma \vdash 0$$

$$typ\_input: \quad \frac{\Gamma(x) \leq s^r \quad \langle \Gamma(x)\rangle \leq read \quad [\Gamma(x)] \leq s \qquad \forall z \notin Var(P)\ \Gamma.(z:[\Gamma(x)]) \vdash P\{z/y\}}{\Gamma \vdash x(y:s).P}$$

$$typ\_output: \quad \frac{\langle \Gamma(x)\rangle \leq write \qquad \Gamma(y) \leq [\Gamma(x)] \qquad \Gamma \vdash P}{\Gamma \vdash \bar{x}y.P}$$

$$typ\_res : \frac{\forall y \notin Var(P)\ \Gamma.(y:t) \vdash P\{y/x\}}{\Gamma \vdash \nu x : t.P}$$

### 5.4 Formalization of the typing rules

As usual when following the LF methodology (i.e. using an higher order method of encoding), the environment is encoded by a list of hypothesis. In our case, the typing judgement $\vdash p$ is represented by a simple judgment ($type\ p$), typing a process $p$ in a environment where $x$ as type $t$ being eventually represented as $(type\_n\ x\ t) \rightarrow (type\ p)$.

We first introduce the type of names:

```
Parameter type_n : name -> typ.
```

Then the type-checking rules are a straightforward translation from informal rules, where the only interesting point to note is that we do not need to look for fresh names in the rules for input and restriction. The freshness condition is implemented by an universal quantification on variables in the premisses involved.

The complete set of rules (where we omit the obvious rules for par and bang, as before) is as follows:

```
Inductive type : proc -> Prop :=
    type_nil : (type Nil)
  | type_in  :
       (x:name) (stype_cap (get_cap (type_n x)) read) ->
       (s:typ) (stype (get_typ (type_n x)) s) ->
       (r:cap) (stype (type_n x) (typ_c s r)) ->
       (p:name->proc)
       ((y:name) (type_n y)=(get_typ (type_n x)) -> (type (p y))) ->
       (type (In x s p))
  | type_out :
       (x,y:name)(p:proc)
       (stype_cap (get_cap (type_n x)) write) ->
       (stype (type_n y) (get_typ (type_n x))) -> (type p) ->
       (type (Out x y p))
  | type_res :
       (t:typ)(p:name->proc)
       ((y:name) ((type_n y)=t) -> (type (p y))) ->
       (type (Res t p)).
```

## 6 Preservation of types

### 6.1 Main lemmas

We have to prove three theorems, one for each kind of actions. The theorem for the silent actions simply states the following:

```
Theorem sr_tau:
   (p,q:proc) (eval p Tau q) -> (type p) -> (type q).
```

The proof of this theorem uses the following natural property for communications. The property says that a communication between two processes resulting from the evaluation of well-typed processes is well-typed:

```
Theorem sr_com:
  (p:proc)(x:name)(p':(name->proc)->proc)
  (evalo p (OutA x) p') -> (type p) ->
  (q:proc)(q':name->proc) (evali q (InA x) q') -> (type q) ->
  (type (p' q')).
```

The proof of the above theorem needs the theorems stating the preservation of types properties for input and output actions:

```
Theorem sr_in:
   (p:proc)(a:action)(q:name->proc)
   (evali p a q) -> (x:name)(a=(InA x)) ->
   (type p) ->
   (y:name) (stype (type_n y) (get_typ (type_n x))) -> (type (q y)).

Theorem sr_out:
   (p:proc)(a:action)(q:(name->proc)->proc) (evalo p a q) ->
   (x:name)(a=(OutA x)) -> (type p) -> (f:name->proc)
   ((y:name) (stype (type_n y) (get_typ (type_n x))) -> (type (f y))) ->
   (type (q f)).
```

Note in the above theorems how we deal with the typing of functions. We say that a function $f$ from name to processes is well-typed if the process $(f\ y)$ is well-typed for every name $y$ of the appropriate type. Similarly, a function $F$ from (name to processes) to processes will be said to be well-typed if the process $(F\ f)$ is well-typed for every well-typed function $f$.

The above theorems need one lemma on the sub-typing relation:

```
Lemma narrowing:
  (p:name->proc) (x:name)(s:typ)
  (((type_n x) = s) -> (type (p x))) ->
  (t:typ)(stype t s) ->
  ((type_n x) = t) -> (type (p x)).
```

As usual when using an higher-order method, there is no need here for a substitution lemma. There is no need either for tedious proofs of lemmas about renaming, freshness of variables and the like, contrary to almost all the other developments we have seen so far.

### 6.2 Induction on functions

As usual when following the LF methodology, we need induction, and inversion, principles on terms of functional type. This is not provided in the current systems based on Logical Frameworks [1]. In previous work [DPS97,DL98], we proposed various induction principles that we used in the development presented here. As pointed out in these works, the induction principles used here are not valid in the Calculus of Inductives Constructions. However, there are all valid in the Calculus of Constructions, as in the Edinburgh Logical Framework. Thus we took the liberty to state them as axioms here, in order to demonstrate how proofs become clear and easy when we work at a more abstract level than usual; when we work with closed functions instead of open terms.

The induction principle associated with the definition of processes is defined as follows:

$proc\_ind$ :
$\forall P : (proc \rightarrow Prop).$
$(P\ Nil)$
$\rightarrow (\forall n : name, t : typ, p : (name \rightarrow proc).$
$\quad (\forall n : name.\ (P\ (p\ n))) \rightarrow (P\ (In\ n\ t\ p)))$
$\rightarrow (\forall n, n0 : name, p : proc.\ (P\ p) \rightarrow$
$\quad (P\ (Out\ n\ n0\ p)))$
$\rightarrow (\forall p : proc.\ (P\ p) \rightarrow \forall q : proc.\ (P\ q) \rightarrow$
$\quad (P\ (Par\ p\ \ q)))$
$\rightarrow (\forall p : proc.\ (P\ p) \rightarrow (P\ (Bang\ p)))$
$\rightarrow (\forall t : typ, p : (name \rightarrow proc).$
$\quad (\forall n : name.\ (P\ (p\ n))) \rightarrow (P\ (Res\ t\ p)))$
$\rightarrow \forall p : proc.\ (P\ p)$

The induction principle associated with the definition of processes abstracted over a name can be defined as:

$proc_1\_ind$ :
$\forall P : (name \rightarrow proc \rightarrow Prop).$
$(P\ \lambda x : name.\ Nil)$
$\rightarrow (\forall n : name, t : typ, p : (name \rightarrow proc).$
$\quad (P\ p) \rightarrow (P\ \lambda x : name.\ (In\ n\ t\ p)))$
$\rightarrow (\forall n, n0 : name, p : name \rightarrow proc.\ (P\ p) \rightarrow$
$\quad (P\ \lambda x : name.\ (Out\ n\ n0\ (p\ x))))$
$\rightarrow (\forall p : name \rightarrow proc.\ (P\ p) \rightarrow \forall q : proc.\ (P\ q) \rightarrow$
$\quad (P\ \lambda x : name.\ (Par\ (p\ x)\ \ (q\ x))))$
$\rightarrow (\forall p : name \rightarrow proc.\ (P\ p) \rightarrow$
$\quad (P\ \lambda x : name.\ (Bang\ (p\ x))))$
$\rightarrow (\forall t : typ, p : (name \rightarrow name \rightarrow proc).$
$\quad (\forall n : name.\ (P\ \lambda x : name.\ (p\ x\ n))) \rightarrow$
$\quad (P\ (\lambda x : name.\ Res\ t\ (p\ x))))$
$\rightarrow \forall p : name \rightarrow proc.\ (P\ p).$

[1] The Elf system provides tools for performing induction, but without a clear idea of the induction principles used.

Similarly, Coq generates various inversion principles associated with an inductive definition:

```
Derive Inversion_clear type_inv_in with
  (x:name)(s:typ)(p:name->proc)(type (In x s p)) Sort Prop.
```

$$
\begin{array}{l}
\mathit{type\_inv\_in} : \\
\forall P : (\mathit{name} \to \mathit{typ} \to (\mathit{name} \to \mathit{proc}) \to \mathit{Prop}), \\
\quad x : \mathit{name},\ s : \mathit{typ},\ p : (\mathit{name} \to \mathit{proc}). \\
(\forall r : \mathit{cap}. \\
\quad (\mathit{stype\_cap}\ (\mathit{get\_cap}\ (\mathit{type\_n}\ x))\ \mathit{read}) \\
\quad \to (\mathit{stype}\ (\mathit{get\_typ}\ (\mathit{type\_n}\ x))\ s) \\
\quad \to (\mathit{stype}\ (\mathit{type\_n}\ x)\ (\mathit{typ\_c}\ s\ r)) \\
\quad \to (\forall y : \mathit{name}. \\
\qquad (\mathit{type\_n}\ y) = (\mathit{get\_typ}\ (\mathit{type\_n}\ x)) \to \\
\qquad (\mathit{type}\ (p\ y))) \\
\quad \to (P\ x\ s\ p)) \\
\to (\mathit{type}\ (\mathit{In}\ x\ s\ p)) \\
\to (P\ x\ s\ p).
\end{array}
$$

For terms of functional types, we can define these principles as follows, for example for a term of type $((\mathit{type\_n}\ z) = t) \to (\mathit{type}\ (\mathit{In}\ x\ s\ p))$.

$$
\begin{array}{l}
\mathit{type\_inv\_in\_zt} : \\
\forall P : (\mathit{name} \to \mathit{name} \to \mathit{typ} \to \mathit{typ} \to (\mathit{name} \to \mathit{proc}) \\
\quad \to \mathit{Prop}), \\
\quad x, z : \mathit{name},\ s, t : \mathit{typ},\ p : (\mathit{name} \to \mathit{proc}). \\
(\forall r : \mathit{cap}.\ ((\mathit{type\_n}\ z) = t) \to \\
\quad (\mathit{stype\_cap}\ (\mathit{get\_cap}\ (\mathit{type\_n}\ x))\ \mathit{read}) \\
\quad \to (\mathit{stype}\ (\mathit{get\_typ}\ (\mathit{type\_n}\ x))\ s) \\
\quad \to (\mathit{stype}\ (\mathit{type\_n}\ x)\ (\mathit{typ\_c}\ s\ r)) \\
\quad \to (\forall y : \mathit{name}. \\
\qquad (\mathit{type\_n}\ y) = (\mathit{get\_typ}\ (\mathit{type\_n}\ x)) \to \\
\qquad (\mathit{type}\ (p\ y))) \\
\quad \to (P\ x\ z\ s\ t\ p)) \\
\to (((\mathit{type\_n}\ z) = t) \to (\mathit{type}\ (\mathit{In}\ x\ s\ p))) \\
\to (P\ x\ z\ s\ t\ p).
\end{array}
$$

## 7 Related Work

Related works include several recent formalizations of various $\pi$-calculi, all done in the Coq system. As we mentioned in the introduction of the paper, some of these works had the same goals as ours, while others aimed at the study of bisimulation techniques. To be fair with the experiments in the latter case, we must say that the use of functions in our specification of the $\pi$-calculus, while providing excellent results for our goals, might make the proofs of correctness of bisimulation techniques a bit harder than usual.

An extensive implementation of the $\pi$-calculus, including proofs of correctness of bisimulation techniques, has been realized by Daniel Hirschkoff [Hir97,Hir99] using the de Bruijn method. Our contribution is much more modest in the sense that it only provides a new basis for such a study. However, the de Bruijn technique quickly leads to very obscure semantic descriptions and makes proofs long and tedious. 75 percent of the development by Daniel Hirschkoff concern manipulation of de Bruijn codes, which is not the subject of interest.

A different method has recently been investigated by Guillaume Gillard [Gil00], on a concurrent object calculus recently proposed by Andrew Gordon and Paul Hankin. Guillaume Gillard provides a proof of preservation of types for this calculus. The method used is due to Andrew Gordon [Gor94]. It consists in describing terms modulo alpha-conversion, over a predefined set of $\lambda$-terms implemented with de Bruijn codes. The part of the development devoted to the manipulation of de Bruijn terms, while still sizeable, is definitely more reasonable than in the standard approach.

A recent development has been realized by Loic Henry-Greard [Hen98]. The proof done is the same as ours: the proof of preservation of types for the monadic $\pi$-calculus. The proof uses a (slight extension of a) method due to Randy Pollack and James Mc Kinna, extensively used to formalize Lego in Lego. The basic idea of the method is to distinguish between bound variables (represented by variables) and free variables (represented by parameters). This gives a nice solution to the problem of capture-avoiding substitution. However, there is still an overhead in both the descriptions and the proofs, to manipulate the variables and the parameters. It is interesting to note that our proof, while being considerably shorter, has exactly the same structure as Loic Henry-Greard's proof.

Finally, Furio Honsell, Marino Miculan and Ivan Scagnetto have proposed a higher-order description of the $\pi$-calculus (with the match operator) [HMS99], different from ours, which they have used in the proofs of correctness of various bisimilarity laws. Their formalisation uses meta-level (Coq) functions to encode the input and restriction operators like us; However, the rest of the presentation greatly differs from ours. They need to duplicate the evaluation rules in a non-natural way: some of the rules yield processes while the others yield functions from name to processes. Communication is formalized using open and close rules, which, despite the use of higher-order syntax, require a freshness predicate. Meanwhile, they provide a formal study of some bisimilarity principles, which, as it uses rules of evaluation (partly) described by means of functions, should a priori inspire a similar study using our specification.

## 8 Conclusion and Future Work

We have presented in this paper a higher-order formalization of a monadic $\pi$-calculus in the Calculus of Inductive Constructions, and provided a proof of type preservation for this calculus in the Coq system. The specification we proposed formalizes communication by a function application. Our proof is very short: it is only three pages long. We believe that the same proof performed on alternative descriptions of the same calculus would give much longer and more tedious proofs. Extending our development to the polyadic $\pi$-calculus, while requiring longer proofs, should be straightforward.

It might be worth noting that the technique of higher-order abstract syntax has been rarely used to describe imperative languages up to now. It seems that this method, while being of great benefit in the descriptions of functional programming languages, is a bit less suitable for imperative languages. However, the $\pi$-calculus is considered as an imperative language, as communication is described as a side-effect in the meta-theoretical studies of the calculus. We may hope that the specification proposed here might give new ideas for the description of imperative languages.

Another interesting point to note is that only a limited form of higher-order abstract syntax is required to describe first-order, usual $\pi$-calculi. The point is that we only need functions from name to processes to describe binders, while functions from processes to processes would be required in the description of the higher-order $\pi$-calculus, where processes themselves can be passed on a channel. The problem is that such types are not allowed as types of constructors of an inductive type for processes, at least in the standard way as the one provided by almost all the current systems providing induction. Consequently, we are not able to use the technique of higher-order abstract syntax to describe the higher-order $\pi$-calculus today. However, there are already propositions for various meta-logics in the literature [DPS97,DL98,MM97,Hof99] which could be used as a basis for the implementation of the system we need.

Concerning future work, our main goal is to provide the basis for a machine-checked study of languages based on the $\pi$-calculus. Particularly relevant here are proofs of properties such as the preservation of types for languages for which the rich type system make those proofs challenging, even in their hand-written presentations. We think here of languages involving polymorphism as the Pict [PT97] or the Join [FGL+96] languages, or to type-checking rules to prevent dead-lock as proposed in [Kob98]. Other interesting languages will be those arising from the various current propositions for concurrent and object calculi [Bou97,GH98].

As we already mentioned, future work of a different nature is the study of the theory of the $\pi$-calculus: bisimilarity techniques. It would be very interesting to see whether the higher-order description of the $\pi$-calculus we propose here can be of real benefit too in such formal studies.

In the long term, we hope such formal studies to be used to formalize proof of correctness of programs written in languages based on the $\pi$-calculus.

**Acknowledgements.** Thanks go to Davide Sangiorgi who introduced us to the $\pi$-calculus. Our description of the calculus comes out of many discussions we had, trying to understand and formalize the binding mechanism in the $\pi$-calculus. We also thank the anonymous reviewers for their constructive comments.

## References

[BBC+97] B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Fillitre, E. Giménez, H. Herbelin, G. Huet, C. Muoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA, August 1997.

[Bou97] Grard Boudol. The pi-calculus in direct style. In *proceedings of the POPL ACM Conference on Principles of Programming Languages*, pages 228–241, 1997.

[CH88] Thierry Coquand and Grard Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[DL98] Jolle Despeyroux and Pierre Leleu. A modal $\lambda$-calcul with iteration and case constructs. In *proceedings of the annual Types for Proofs and Programs seminar*, Springer-Verlag LNCS 1657, March 1998. An extended and revised version will appear in the forthcoming special issue of MSCS on Intuitionistic Modal Logics and Applications.

[DPS97] Jolle Despeyroux, Frank Pfenning, and Carsten Schrmann. Primitive recursion for higher-order abstract syntax. In Philippe de Groote and J. Roger Hindley, editors, *proceedings of the TLCA 97 Int. Conference on Typed Lambda Calculi and Applications, Nancy, France, April 2–4*, pages 147–163. Springer-Verlag LNCS 1210, April 1997. An extended version is available as CMU Technical Report CMU-CS-96-172. An extended and revised version will appear in TCS (Theoretical Computer Science).

[FGL+96] Cdric Fournet, Georges Gonthier, Jean-Jacques Lvy, Luc Maranget, and Didier Rmy. A calculus of mobile agents. In *Proc. of the CONCUR conference*, 1996.

[GH98] A. Gordon and P. Hankin. A concurrent object calculus: reduction and typing. In *proceedings of the HLCL Conference*, 1998.

[Gil00] Guillaume Gillard. A formalization of a concurrent object calculus up to alpha-conversion. In *Proceedings of the 17th International Conference on Automated Deduction, CADE-17*, June 2000.

[Gor94] A. Gordon. A mechanisation of name-carrying syntax up to alpha-conversion. In *proceedings of the int. workshop on Higher Order Logic Theorem Proving and its Applications, Vancouver*, Springer-Verlag LNCS 780, pages 414–426, 1994.

[Hen98] Loc Henry_Greard. A proof of type preservation for the pi-calculus in Coq. Research Report RR-3698, Inria, December 1998. Also available in the Coq Contrib library.

[Hir97] Daniel Hirschkoff. A full formalization of pi-calculus theory in the Calculus of Constructions. In Elsa Gunter, editor, *proceedings of the International Conference on Theorem Proving in Higher Order Logics*, Murray Hill, New Jersey, August 1997.

[Hir99] Daniel Hirschkoff. *Mise en oeuvre de preuves de bisimulation*. Phd thesis, cole Nationale des Ponts et Chausses (ENPC), January 1999. In French.

[HKPM97] Grard Huet, Gilles Kahn, and Christine Paulin-Mohring. The coq proof assistant: a tutorial, version 6.1. Technical Report RT 0204, Inria, Rocquencourt, France, August 1997. page html: http://www.inria.fr/RRRT/RT-0204.html.

[HMS99] Furio Honsell, Marino Miculan, and Ivan Scagnetto. Pi-calculus in (co)Inductive Type Theory. *to appear in TCS*, 1999.

[Hof99] Martin Hofmann. Semantical analysis of higher-order abstract syntax. In IEEE, editor, *proceedings of the International Conference on Logic In Computer Sciences, LICS*, 1999.

[Kob98] Naoki Kobayashi. A partially deadlock-free typed process calculus. *ACM Transactions on Programming Languages and Systems*, 20-2, December 1998. A preliminary summary appeared in LICS'97.

[Mil91] Robin Milner. The polyadic pi-calculus: a tutorial. Technical Report ECS-LFCS-91-180, LFCS, Edinburgh University, October 1991.

[MM97] Raymond McDowell and Dale Miller. A logic for reasoning with higher-order abstract syntax: An extended abstract. In Glynn Winskel, editor, *Proceedings of the Twelfth Annual Symposium on Logic in Computer Science*, Warsaw, Poland, June 1997. To appear.

[MPW92] R. Milner, R. Parrow, and J. Walker. A calculus of mobile processes, (part I and II). *Information and Computation*, 100:1–77, 1992.

[PM92] Christine Paulin-Mohring. Inductive definitions in the system coq. rules and properties. In J.F. Groote M. Bezem, editor, *Proceedings of the International Conference on Typed Lambda Calculi and Applications, TLCA'93*, Springer-Verlag LNCS 664, pages 328–345, 1992. also available as a Research Report RR-92-49, Dec. 1992, ENS Lyon, France.

[PS95] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 11, 1995.

[PT97] Benjamin Pierce and David Turner. Pict: A programming language based on the pi-calculus. Technical Report ., IU, 1997.