Formalization of a concurrent object calculus up to α-conversion

Gillard Guillaume

INRIA Sophia Antipolis

Abstract. We present a formalization of a concurrent object calculus in the Calculus of Inductive Constructions. We use de Bruijn technique in an intermediate syntax, but de Bruijn indices do not appear in the final formalization of the terms of the calculus, which are still defined up to α-conversion. We derive substitution rewriting rules and an inductive principle on the subset of the terms which formalize the calculus. Once a certain amount of preliminary work has been done on the intermediate syntax this induction theorem makes possible natural proofs which do not deal with de Bruijn number.

1 Introduction

Specifying languages together with their semantics in theorem provers has become quite a big issue these days. Among the difficulties encountered is the well known problem of formalizing α-equivalence. Though *α-equivalence* is a well understood notion on terms and reasoning *up to α-equivalence* is usual for works on paper, things are different in the world of proof assistants. Among all the known methods to formalize calculi, languages, programs... none seems to have reach a consensus. In the case of calculi, we believe this is highly dependent on what people want to do once it is formalized.

Our goal is to define a concurrent object calculus up to α-equivalent terms as closed as possible from its presentation **on paper** with equivalence relation for bound names renaming. We use the de Bruijn encoding for bound variables, but we hide de Bruijn indices by simulating binding constructors with functions on expressions as [Gor94].

We formalize the **concξ-calculus** [GH98], a concurrent object calculus consisting of Abadi and Cardelli's imperative object calculus **impξ** [AC96] extended with primitives from the π-calculus [MPW92]. This calculus has a good expressiveness, objects are primitive and there is no need for store, configuration or labeled transition in its semantics; given in terms of a reduction relation on expressions.

The COQ system [BBC$^+$97a] [HKPM97] we use for our implementation is a proof-assistant based on the calculus of inductive constructions [Wer94], a higher order logic with inductive definitions. All the proofs have been done with the user-interface CtCoq [BBC$^+$97b]. This paper has been written so as to be understood by people not familiar with the COQ system. We use mathematical notations instead of the COQ syntax and all technical difficulties specific to the system have been avoided.

Organization of the paper: In section 2 we present the **concξ-calculus** as it has been defined in [GH98] . In section 3 we give an intermediate syntax and the substitution functions. In section 4 we show how we identify a subset of our formalism with the **concξ-calculus**. We present the formalization of the **concξ-calculus** in which α-equivalent terms are in fact equal. We derive a powerful induction theorem which internalizes α-equivalence. In section 5 we show that we can describe the semantics of the **concξ-calculus** in a very natural way using your formalization. Section 6 is a short discussion about the formalizing technique we used and section 7 draws some conclusions.

2 The concξ-calculus

This calculus was first introduced by A.Gordon and P.Hankin. It is the imperative object calculus **imp**ξ of Abadi and Cardelli in which objects are located at addresses, extended with a parallel composition and the name restriction operator from the π-calculus. The reader interested in a more detailed presentation of this calculus should refer to [GH98].

2.1 Informal syntax of the calculus

We assume that there are infinite disjoint sets of *names, variables* and *labels.* The expressions of the language are defined as follows:

l	labels
u,v	**result**
x,y,z	variables
p,q	references
a,b,c	**terms**
u,v	result
$p \mapsto d$	denomination
$p.l$	method select
$p.l \Leftarrow \xi(x).b$	method update
$clone(p)$	cloning
$let\ x = a\ in\ b$	let
$a \Rsh b$	parallel composition
$(\nu p).a$	restriction
d:=	**denotations**
$[l_i = \xi(x_i)a_i\ ^{i \in 1..n}]$	object

Table 1. Syntax of the informal **conc**ξ**-calculus**

In method $\xi(x)b$, the variable x is bound in b. In the expression $let\ x = a\ in\ b$ the variable x is bound in$b$. In a restriction, $(\nu p).a$ the name p is bound in a. The notations $fn(a)$ and $fv(a)$ denote respectively the sets of free names and variables in the expression a. The expression $a[x \leftarrow v]$ is the notation for the substitution of the result v for each free occurrence of the variable x in the expression a. The notation $a = b$ means that the terms a and b are equal up to bound names or bound variables renaming, and the reordering of the labeled components of objects.

2.2 Informal semantics

Terms of the calculus are interpreted either as *processes* or as *expressions. Expressions* and *processes* are concurrent computations but an expression is expected to return a result while a process is not. The term $a \Rsh b$ is interpreted as being an expression in which a and b run in parallel such that its result is the result returned by b; any result returned by a is discarded. Thus as opposed to many concurrent calculus the parallel composition ($\Rsh$) is not commutative. The operational semantics is based on a structural congruence ($\equiv$) and a reduction relation ($\rightarrow$). The reduction relation is the equivalent of β-reduction for the λ-calculus. The structural congruence relation allows the rearrangement inside a term so that reduction may be applied. Please refer to [GH98] for the motivations and more details on this semantics.

The structural congruence is the least congruence on expressions which satisfies:

$$
\begin{array}{ll}
(a \Rsh b) \Rsh c \equiv a \Rsh (b \Rsh c) & \\
(a \Rsh b) \Rsh c \equiv (b \Rsh a) \Rsh c & \\
\nu p.(\nu q.a) \equiv \nu q.(\nu p.a) & \\
\nu p.(a \Rsh b) \equiv a \Rsh (\nu p.b) & p \notin fn(a) \\
\nu p.(a \Rsh b) \equiv (\nu p.a) \Rsh b & p \notin fn(b) \\
let\ x = (let\ y = a\ in\ b)\ in\ c) \equiv let\ y = a\ in\ (let\ x := b\ in\ c) & y \notin fv(c) \\
\nu p.(let\ x = a\ in\ b) \equiv let\ x = (\nu p.a)\ in\ b & p \notin fn(b) \\
a \Rsh (let\ x = b\ in\ c) \equiv let\ x = (a \Rsh b)\ in\ c &
\end{array}
$$

Table 2. Definition of $\equiv$

The reduction relation $\rightarrow$ is the least relation on expressions which satisfy:

For the first three rules, let $d = [l_i = \xi(x)b_i^{i \in 1..n}]$.

$$
\begin{array}{ll}
(p \mapsto d) \Rsh p.l_j \rightarrow (p \mapsto d) \Rsh b_j[p/x_j] & \text{if } j \in 1..n \\
(p \mapsto d) \Rsh (p.l_j \Leftarrow \xi(x).b) \rightarrow (p \mapsto d') \Rsh p & \text{if } j \in 1..n \\
 & d' = [l_j : \xi(x).b, l_i = \xi(x_i)b_i\ ^{i \in (1..n)-j}] \\
(p \mapsto d) \Rsh (clone\ p) \rightarrow (p \mapsto d) \Rsh \nu q.((q \mapsto d) \Rsh q) & \text{if } q \notin fn(d) \\
let\ x := p\ in\ a \rightarrow a[p/x] & \\
\nu p.a \rightarrow \nu p.a' & \text{if } a \rightarrow a' \\
(a \Rsh b) \rightarrow (a' \Rsh b) & \text{if } a \rightarrow a' \\
(b \Rsh a) \rightarrow (b \Rsh a') & \text{if } a \rightarrow a' \\
let\ x := a\ in\ b \rightarrow let\ x := a'\ in\ b & \text{if } a \rightarrow a' \\
a \rightarrow b\ a \equiv a', b \equiv b', a' \rightarrow b' &
\end{array}
$$

Table 3. Reduction relation: $a \rightarrow b$

3 An intermediate syntax

Except otherwise stated in the sequel of the paper we no longer refer to *names, variables, labels, results* ... of the informal calculus described above.

3.1 The formal syntax

We distinguish references, representing addresses of stored object (channels in π-calculus) from variables, representing intermediate values (variables in λ-calculus). Let us call both notions *names*. We can encode free *names* by *real* names and bounded *names* by de Bruijn indices. We assume there are disjoint infinite sets of references, variables and labels and equality is decidable on each of these sets.

		variable
x,y,z		free variable
(dvar i)	$i \in N$	bounded variable
		reference
p,q		free reference
(dref i)	$i \in N$	bounded reference
		label
l		label

Except otherwise stated, p,q range over free references, x,y,z over free variables, **p,q** over references, **x,y,z** over variables (either free or bounded), i,j over naturals and l over labels. For readability reasons *var* and *ref* will be use for free variables and free references respectively in all our formal definition. The definition of the terms of the calculus consists of two mutually defined sorts, *dbterm* and *denotation*:

a,b,c	**dbterms**	
x,y,z	variable	
p,q	reference	
$\mathbf{p} \mapsto d$	denomination	
$a.l$	method select	
$a.l \Leftarrow b$	method update	*dbupdt*
$clone(a)$	cloning	
$let\ a\ in\ b$	let	*dblet*
$a \upharpoonright b$	parallel composition	
$\nu.a$	restriction	*dbres*
d:=	**denotation**	
[]	empty object	
$[l : a :: obj]$	object constructor	

Table 4. Syntax of the concurrent object calculus

For variables the binding constructors are *dbupdt*, *dblet* in their second argument and the *object-constructor* for de Bruijn variables. The *dbres* operator is the only binding constructor for de Bruijn references. Objects are represented by lists. Although sets seem closer to the idea of objects (a collection of attributes and methods), we cannot define object as sets because sets in COQ are specified as predicates, and predicates cannot be used in the type of a constructor of an inductive set. Moreover, COQ provides an efficient tool for generating induction scheme for mutual inductive definitions when some type is a list of another [1].

Example: *in* $\nu.((dref\ 0) \mapsto [l : (dvar\ 0) \upharpoonright (dvar\ 1) :: []])$*, the term* $(dref\ 0)$ *is bound by the dbres constructor and the term* $(dvar\ 0)$ *by the object-constructor while the term* $(dvar\ 1)$ *is unbounded.*

Our syntax is a bit more general than the one proposed by A Gordon for his **conc**ξ**-calculus** in the sense that we allow cloning, method calling, and method updating not only for results but for all terms of our syntax. We will recover the **conc**ξ**-calculus** by restricting our syntax later on. Since the formalized syntax of the **conc**ξ**-calculus** will eventually be an inductively defined subset of *dbterm* this is not a real problem. Of course, the technical

[1] Using the Scheme tactic

proofs regarding de Bruijn indices are longer than the one expected but, not in an significant way. What we gain here is a more general framework in which similar concurrent calculus can be easily encoded.

Thanks to De Bruijn indices we do not need an *alpha-equivalence* notion. The drawback is that we have temporary lost the convivial form of syntax because of de Bruijn notations for binder constructors.

We define functions that compute the set of free references and free variables, respectively fn and fv. As we have de Bruijn indices for both references and variables, we define two degree functions (computing the usual notion of degree for a term with de Bruijn indices), one for each kind. A dbterm is said to be *closed* when both degrees are zero.

3.2 Abstraction and instantiation functions

We define a variable abstraction function $\_[\_/\_]^a_v : \mathit{dbterm} \rightarrow \mathit{var} \rightarrow \mathit{nat} \rightarrow \mathit{dbterm}$. for a given $\mathit{dbterm}\ a$, $a[(\mathit{dvar}\ i)/x]^a_v$ is obtained by substituting in a all the occurrences of the variable x by the de Bruijn indice i. The substitution is defined recursively on the $\mathit{dbterms}$ such that the Bruijn indice substituted is increase by one each time a variable binder is met (i.e *dblet*, *dbupdt* and *object-constructor* but not dbres).

In a similar way, we define an instantiation function $\_[\_/\_]^i_v : \mathit{dbterm} \rightarrow \mathit{nat} \rightarrow \mathit{var} \rightarrow \mathit{dbterm}$ which replaces a bound variable (dvar i) by a free variable x.

We also define an abstraction $\_[\_/\_]^a_p$: $\mathit{dbterm} \rightarrow \mathit{ref} \rightarrow \mathit{nat} \rightarrow \mathit{dbterm}$ and an instantiation funrecursively tion $\_[\_/\_]^i_p$: $\mathit{dbterm} \rightarrow \mathit{nat} \rightarrow \mathit{ref} \rightarrow \mathit{dbterm}$ for references. The behaviors of these functions are standard for de Bruijn specifications. Note that for references only the dbres constructor is a binder.

For closed $\mathit{dbterms}$ (i.e with no unbound de Bruijn indices) we can prove the following properties:

$$\vdash \forall a : \mathit{dbterm}.\forall x : \mathit{var}.\forall i : N. \quad a = (a[i/x]^a_v)[x/i]^i_v.$$
$$\vdash \forall a : \mathit{dbterm}.\forall p : \mathit{ref}.\forall i : N. \quad a = (a[i/p]^a_v)[p/i]^i_v.$$

4 Formalization of the conc$\boldsymbol{\xi}$-calculus

Our goal now is to define a subset of our de Bruijn syntax without referring directly to bound *names*. As oppose to [Gor94],we do not provide a general way for modeling α-equivalence in theorem provers. Instead of defining a set of terms up to α-conversion used for the underlying logic of the system we use his technique to formalize the **concξ-calculus**.

4.1 Functions as constructors

We define new functions on dbterms let, res, and eta behaving like the de Bruijn binding operators except that they use names in their arguments.

- $\mathit{let} : \mathit{var} \rightarrow \mathit{dbterm} \rightarrow \mathit{dbterm} \rightarrow \mathit{dbterm} \ := \ \lambda x : \mathit{var}.\lambda a, b : \mathit{dbterm}.(\mathit{dblet}\ a\ b[0/x]^a_v)$
- res: $\mathit{ref} \rightarrow \mathit{dbterm} \rightarrow \mathit{dbterm} \ := \lambda p : \mathit{ref}.\lambda a : \mathit{dbterm}.(\mathit{dbres}\ a[0/p]^a_p)$
- eta: $\mathit{var} \rightarrow \mathit{dbterm} \rightarrow \mathit{dbterm} \ := \ \lambda x : \mathit{var}.\lambda a : \mathit{dbterm}.a[0/x]^a_v$

Instead of using the usual de Bruijn binding constructors, the well formed terms of our restricted syntax, will be builded with these functions as *binding constructors*.

In the following, we will write $\mathit{let}\ x := a$ in b for ($\mathit{let}\ x\ a\ b$), $\nu p.a$ for ($\mathit{res}\ p\ a$) and $\xi(x).a$ for ($\mathit{eta}\ x\ a$).

Note. *If x occurs free in a, then $(eta\ x\ a)$ has unbound occurrences of the bound variable $(dvar\ 0)$. These occurrences become bound in the denotation $[l : \xi(x).a :: obj]$ and thus in the dbterm $p \mapsto [l : \xi(x).a :: obj]$ which is what we want.*

For relegating de Bruijn indices of the underlying terms behind the scene we also need two new substitution functions which substitute a free *name* for another of the same kind. Although we use bound *names* in the body of these functions, bound names do not appear in the type of these functions.

- a variable substitution:
  $\_[\_/\_]_v : dbterm \to var \to var \to dbterm \ := \lambda u : dbterm.\lambda x, y : var.(u[0/x]^a_v)[y/0]^i_v$
- a reference substitution:
  $\_[\_/\_]_n : dbterm \to ref \to ref \to dbterm \ := \lambda u : dbterm.\lambda p, q : ref.(u[0/p]^a_n)[q/0]^i_n$

4.2 Formalization of the expressions

First, we define the type result as the disjoint union of our free names:

u,v	**result**
x,y,z	free variable
p,q	free reference

Then the inductive predicate **Term** (table 5) defines the subset of dbterms we are interested in. It is easy to see that these terms are exactly those of the **concξ-calculus** if one thinks of *let*, *res* and *eta* as constructors. We omit the obvious proof of this here.

Mutual Inductive
$Term : dbterm \Rightarrow Prop :=$

Resu: $\forall r : result.(Term\ r)$
| Deno: $\forall p : ref.\forall obj : denotation.\ (OTerm\ obj) \Rightarrow (Term\ p \mapsto obj)$
| Msel: $\forall l : labels.\forall u : result.\ (Term\ u.l)$
| Mupd: $\forall u : result.\forall a : dbterm.\forall l : labels.\forall x : var.$
$(Term\ a) \Rightarrow (Term\ u.l \Leftarrow \xi(x).a)$
| Clone: $\forall u : result.(Term\ (clone\ u))$
| Let: $\forall a, b : dbterm.\forall x : var.(Term\ a) \Rightarrow (Term\ b) \Rightarrow (Term\ let\ x := a\ in\ b)$
| Par: $\forall a, b : dbterm.(Term\ a) \Rightarrow\ (Term\ b) \Rightarrow (Term\ a \upharpoonright b)$
| Res: $\forall a : dbterm.\forall p : ref.\ (Term\ a) \Rightarrow (Term\ \nu p.a)$

with
$OTerm : denotation \Rightarrow Prop :=$

Mnul: $(OTerm\ [])$
| Mocons: $\forall l : label.\forall a : dbterm.\forall obj : denotation.\forall x : var.$
$(Term\ a) \Rightarrow (OTerm\ obj) \Rightarrow (OTerm\ (l : \xi(x).a :: obj))$

Table 5. formalization of the **concξ-calculus** in COQ

From now on, we shall call **term**, a dbterm having the Term property.
Terms are based on a de Bruijn formalism and we prove that they are closed. Moreover bounded *names* are hidden in the syntax by the *let*, *res* and *eta* functions. To convince ourselves that **term** is an admissible encoding of the **concξ-calculus** we must prove that α-equivalent informal terms are encoded by a unique **term**. Moreover, we shall prove that the substitution functions $\_[\_/\_]_v$ and $\_[\_/\_]_p$ we have defined on dbterms have the expected behavior on our **terms**.

4.3 Alpha conversion for dbterms

Using the *let*, *res* and *eta* definitions together with $\_[\_/\_]_v$ and $\_[\_/\_]_p$ we can prove the lemmas in table 6.

$$\begin{array}{ll}
\vdash & \forall x, y : var.\forall a, b : dbterm \\
 & y \notin (fv\ b) \Rightarrow (let\ x := a\ in\ b) = (let\ y := a\ in\ b[y/x]_v) \\
\vdash & \forall p, q : ref.\forall a : dbterm\ q \notin (fn\ a) \Rightarrow \nu p.a = \nu q.a[q/p]_p \\
\vdash & \forall x, y : var.\forall a : dbterm\ y \notin (fv\ a) \Rightarrow \xi(x).a = \xi(y).a[y/x]_v
\end{array}$$

Table 6. α-equivalence equalities for the *let*, *res* and *eta* functions:

This means that there is a subset of dbterms whose α-equivalent terms can be proved equals. In particular the equations table 6 hold for **terms**. What we have shown is that α-equivalent terms of the informal calculus are encoded in our formalism by a unique **term**. To be complete, we need to show that our substitution functions have the right behaviors on our new binders. It appear that we must introduce new tools before being able to prove that they behave as expected (see section 4.5).

4.4 An intermediate induction principle for terms

It appears than the induction scheme generated by COQ for the predicate Term is not powerful enough for our purpose[2] . As an example it is not clear how one can derive the fact that **terms** are closed under the substitution $\_[\_/\_]_v$ and $\_[\_/\_]_p$ with it. Somehow, we want to integrate that α-equivalence terms are equal for COQ inside a new induction scheme.
To this end, we introduce a new function *length* on dbterms witch computes the numbers of constructors appearing in a term. We prove that the relation define by comparing the length of two terms is a well founded order on *dbterms*. Moreover, we show that renaming *names* in a term doesn't change its length (table 7).

$$\begin{array}{l}
\vdash \forall a : dbterm.\forall x, y : var.\ (length\ a) = (length\ a[y/x]_v) \\
\vdash \forall a : dbterm.\forall p, q : ref.\ (length\ a) = (length\ a[q/p]_p)
\end{array}$$

Table 7. length is invariant under renaming

A new more powerful induction scheme, based on the length measure, which integrates that the name of a *soon to be* bound variable shouldn't be significant for proof purposes can then be derive. As an example, if we use the COQ induction scheme to prove that a property $P : dbterm \to Prop$ holds for all *terms* the subgoal generates by COQ for the Let case is:

$$\frac{a, b : dbterm \quad x : var \quad (Term\ a) \quad (Term\ b) \quad (P\ a) \quad (P\ b)}{(P\ let\ x := a\ in\ b)}$$

The Let subgoal generated by our new principle is:
a, b and $b0$ range over *dbterms* and x over free variables.

[2] Actually we need to use the Scheme tactic to generate an efficient principle

$$\frac{(Term\ a) \quad (P\ a) \quad (Term\ b) \quad \dfrac{(length\ b) = (length\ b0) \quad (Term\ b0)}{(P\ b0)}}{(P\ let\ x := a\ in\ b)}$$

The use of the length function, is rather unusual and produces an heavy induction principle. Fortunately, once we have check than our substitutions compute as expected on **terms**, we can get ride of the *length* function and produce a more conventional induction theorem (see section 4.6).

4.5 Substitutions

In this section we show that the $\_[\_/\_]_v$ and the $\_[\_/\_]_p$ functions have the expected behavior on **terms**.

Rewriting rules for substitutions Problems arise because $\_[\_/\_]_v$, $\_[\_/\_]_p$, are not define on terms but on $dbterms$. Their behavior for our let, eta and res function are not defined by case analysis of the terms constructors but must be proved. Indeed, our functions have been defined such that $\_[\_/\_]_v$, $\_[\_/\_]_p$ behave has expected. We prove that:

Rules for ξ
$\vdash \forall r : result.\forall v : Term.\forall x, y : var\ (r.l \Leftarrow \xi(x).v)[y/x]_v = r[y/x]_v. \Leftarrow \xi(x).v$
$\vdash \forall r : result.\forall v : Term.\forall x, y, z : var\ x \neq y \Rightarrow z \neq x \Rightarrow$
$\quad (r.l \Leftarrow \xi(x).v)[z/y]_v = r[z/y]_v. \Leftarrow \xi(x).v[z/y]_v$
Rules for let
$\vdash \forall u, v : Term.\forall x, y : var\ (let\ x := u\ in\ v)[y/x]_v = let\ x := u[y/x]_v\ in\ v$
$\vdash \forall u, v : Term.\forall x, y, z : var\ x \neq y \Rightarrow x \neq z \Rightarrow$
$\quad (let\ x := u\ in\ v)[z/y]_v = let\ x := u[z/y]_v\ in\ v[z/y]_v$
Rules for res
$\vdash \forall a : Term.\forall p, q : ref.(\nu p.a)[q/p]_p = (\nu p.a)$
$\vdash \forall a : Term.\forall p, q, r : ref.p \neq q \Rightarrow p \neq r \Rightarrow (\nu p.a)[r/q]_p = \nu p.(a[r/q]_p)$

Table 8. One step rewriting rules for substitutions

The rewriting rules for all others cases are directly given by the substitution definitions on $dbterms$.

Terms are closed under substitutions . If one thinks of the way substitutions $\_[\_/\_]_v$ and $\_[\_/\_]_p$ behave on **terms**, it's easy to understand why this subset of dbterm is closed under these two operators. Indeed, using the induction principle *length_induction* together with the rewriting rules above, the two following theorems are easy to prove:

$$\vdash \forall x, y : var.\forall u : dbterm\ (Term\ u) \Rightarrow (Term\ u[y/x]_v)$$
$$\vdash \forall p, q : ref.\forall u : dbterm\ (Term\ u) \Rightarrow (Term\ u[q/p]_p)$$

terms are closed under $\_[\_/\_]_v$ and $\_[\_/\_]_p$ operators

4.6 An efficient induction principle for terms

Following Gordon's method we derive a new induction principle for **terms**: *term_induction*. As expected, *term_induction* doesn't mention the length function.

In the following, X ranges over sets of free variables, Q ranges over free references and *Finite* is a predicate over sets stating that a set is finite.

$$
\begin{array}{l}
\forall P : dbterm \rightarrow Prop \\
(\forall r : results.\ (P\ r)) \Rightarrow \\
(\forall p : ref.\ (P\ p \mapsto [])) \Rightarrow \\
(\forall p : ref.\forall l : labels.\forall obj : denotation. \\
\exists X \quad \langle (Finite\ X) \ \wedge \quad \forall d : dbterm.\forall x : var.\ x \notin X \Rightarrow \\
\qquad (Term\ d) \Rightarrow (P\ d) \Rightarrow (Term\ p \mapsto obj) \Rightarrow (P\ p \mapsto obj) \Rightarrow (P\ p \mapsto [l : \xi(x).d :: obj])\rangle) \Rightarrow \\
(\forall r : result.\forall l : label.\ (P\ r) \Rightarrow (P\ r.l)) \Rightarrow \\
(\forall r : result.\forall l : label.\ (P\ r) \Rightarrow \\
\exists X \quad \langle (Finite\ X) \ \wedge \quad \forall a : dbterm.\forall x : var.\ x \notin X \Rightarrow \\
\qquad (Term\ a) \Rightarrow (P\ a) \Rightarrow (P\ r.l \Leftarrow \xi(x)d0)\rangle) \Rightarrow \\
(\forall r : result.\ (P\ r) \Rightarrow (P\ (clone\ r))) \Rightarrow \\
(\forall d : dbterm.\ (Term\ d) \Rightarrow (P\ d) \Rightarrow \\
\exists X \quad \langle (Finite\ X) \ \wedge \quad \forall d0 : dbterm.\forall x : var.\ x \notin X \Rightarrow \\
\qquad (Term\ d0) \Rightarrow (P\ let\ x := d\ in\ d0)\rangle) \Rightarrow \\
(\forall d : dbterm.\ (Term\ d) \Rightarrow (P\ d) \Rightarrow \forall d0 : dbterm.\ (Term\ d0) \Rightarrow (P\ d0) \Rightarrow (P\ d \vec{\ }\ d0)) \Rightarrow \\
(\exists Q \quad \langle (Finite\ Q) \ \wedge \quad \forall d : dbterm.\forall p : ref.\ p \notin Q \Rightarrow \\
\qquad (Term\ d) \Rightarrow (P\ d) \Rightarrow (P\ \nu p.d)\rangle) \Rightarrow \\
\forall d : dbterm.\ (Term\ d) \Rightarrow (P\ d).
\end{array}
$$

The above induction theorem shares exactly the same properties as the third induction theorem proposed in[Gor94], for an encoded λ-calculus in HOL. As example, to prove the case for *res*-term, first select a set Q of references, and then prove that $(P\ \nu p.u)$ holds from the assumption than $(Term\ u)$ & $(P\ u)$ hold and that $p \notin Q$. Giving the set Q is a way to specify that p is a fresh free reference.

5 Semantics of the conc$\boldsymbol{\xi}$-calculus

In this section we present our formalized semantics in COQ and see it is not very hard to derive it from the informal one.

5.1 A new substitution

To write the COQ code for the semantics we need a function which substitutes variables by names. Unfortunately, it is not possible to define this function directly on **terms**. we have to come back to de Bruijn indices in the body of this function:

$\_[\_/\_]_* : dbterm \rightarrow var \rightarrow ref \rightarrow dbterm := \lambda u : dbterm.\lambda x : var.\lambda p : ref\ (u[0/x]^a_v)[p/0]^i_p$.

For any given term a, $a[p/x]_*$ is computed by substituting all the occurrences of the variable x by the reference p in a. Fortunately, because we use already defined functions in the body of $\_[\_/\_]_*$, we can re-use many technical lemmas in the proofs that this substitution is the good one on our terms. We prove, using the induction theorem *term_ induction*, that **terms** are closed under $\_[\_/\_]_*$.

5.2 The semantic rules

Our **term** subset of dbterm together with the $\_[\_/\_]_*$ function are the only things we need to formalize the semantics of our calculus. What we have to do is just to translate the proposed semantics (see table 2 and 3) in the syntax of our system. The structural rules are the same, except than they hold for terms, the reduction relation is defined table 9.

In the following a b and c range over terms,obj over denotation, x over variables and p and q over references.
Inductive $\rightarrow$: $dbterm \rightarrow dbterm \rightarrow Prop :=$

$$\text{Call\_me: } \frac{(Term\ p \mapsto obj) \quad (Term\ a) \quad (Access\ obj\ l) = (Some\ \xi(x).a) \quad (p \mapsto obj) \upharpoonright p.l}{(p \mapsto obj) \upharpoonright a[p/x]_*}$$

$$\mid \text{Updt\_me: } \frac{(Term\ p \mapsto obj)) \quad (Term\ a)(p \mapsto obj) \upharpoonright (p.l \Leftarrow \xi(x).a)}{(p \mapsto [l : \xi(x).a :: obj]) \upharpoonright p}$$

$$\mid \text{Cloning: } \frac{(Term\ (p \mapsto obj)) \quad q \notin (Ofn\ obj) \quad (p \mapsto obj) \upharpoonright (clone\ p)}{(p \mapsto obj) \upharpoonright \nu q.(q \mapsto obj) \upharpoonright q}$$

$$\mid \text{Let\_eval1: } \frac{(Term\ a)}{let\ x := p\ in\ a \rightarrow a[p/x]_*} \qquad \mid \text{Res\_eval: } \frac{(Term\ a) \quad a \rightarrow c}{\nu p.a \rightarrow \nu p.c}$$

$$\mid \text{Par\_eval1: } \frac{(Term\ a) \quad (Term\ b) \quad a \rightarrow c}{(a \upharpoonright b) \rightarrow (c \upharpoonright b)} \mid \text{Par\_eval2: } \frac{(Term\ a) \quad (Term\ b) \quad a \rightarrow c}{(b \upharpoonright a) \rightarrow (b \upharpoonright c)}$$

$$\mid \text{Let\_eval2: } \frac{(Term\ a) \quad (Term\ b) \quad a \rightarrow c}{let\ x := a\ in\ b \rightarrow let\ x := c\ in\ b}$$

Table 9. Reduction relation: $a \rightarrow b$

We use an *Access* function of type *denotation* $\rightarrow$ *label* $\rightarrow$ *dbterm obtion* which return the first method of name l encountered in the object or a $\perp$ otherwise (*dbterm obtion* is the *option type* associated to dbterm).

The evaluation is given by the relation bellow:

In the following a, a', b and b' range over terms.
Inductive $eval$: $dbterm \rightarrow dbterm \rightarrow Prop :=$

$$\text{Red: } \frac{a \rightarrow b}{a \rightarrow_{eval} b} \qquad \mid \text{Equive: } \frac{a \equiv a' \quad b \equiv b' \quad a' \rightarrow_{eval} b'}{a \rightarrow_{eval} b}$$

Table 10. evaluation $\rightarrow_{eval}$ relation on terms

We split the evaluation definition in two because we conjecture than by keeping the structural equivalence away from *effective* reduction in terms we can avoid looping in proofs. More precisely, we can use *inversion* tactics efficiently when dealing with the $\rightarrow$ and the $\equiv$ relations. Properties of the *eval* relation will follows from their corresponding ones on the $\rightarrow$ and $\equiv$ predicates.

5.3 Some properties of the semantics

Using the induction principle generated by COQ for the above inductive definition, we are able to prove that $\equiv$ is in fact a relation defined on **terms**. At last, we show that $\rightarrow_{eval}$ (table 10) is defined on terms only.

6 Discussion and related work

We have presented a name carrying syntax in which α-equivalent terms are represented by a unique term. Although the properties we want for our substitutions on **terms** seems natural it was not easy to derive all of them. This is mostly due to the presence of de Bruijn indices in the body of the definitions of these substitutions. The main drawback of this approach is that each time we have to define new functions on **terms** we shall have to define them on *dbterm* first, then prove that they behave as expected on **terms**. We believe that if one think of what a function should do on **terms** it is not hard to give its definition on *dbterms*. We claim that this weakness doesn't overcome the advantages of the formalism. In facts, new functions on our syntax will probably use functions we have already defined, allowing re-use of our COQ proofs (like our substitution for the evaluation). One the other and, defining inductive properties on **terms** is done without references to de Bruijn indices. To get the full advantages of this method, we still have to derive strengthened induction principles for our inductively defined predicates (like *term_induction)*.

Related work. Among all the works regarding formalizing the variable-binding operators in calculi none, as far as we know uses the technique we used. Daniel Hirschkoff has encoded a polyadic π-calculus with de Bruijn numbers and proved many bisimulation results [Hir97], Bruno Barras [Bar95], formalize COQ in COQ with de Bruijn indices and bootstrap it, L.Henry-Gréard [Hen98], uses R.Pollack and J.McKinna technique [MP93], to formalize the π-calculus and prove a subject reduction theorem for his calculus. All these techniques are first order in the sense that binders in the calculi are distinct from binders in the logic. J.Despeyroux has investigated a higher-order approach in which the lambda abstraction of the logic is used for binding free variables of the calculus. see [DH94], for a general approach and [DS99] for applications on the π-calculus with Davide Sangiorgi. F.Honsel, M.Miculan and I.Scagnetto [FI98] have encoded a π-calculus in COQ a higher order approach they use *Co inductive* types in their encoding of bisimulation.

7 Conclusion and future work

We have formalized an concurrent object calculus in the COQ system. The technique we used was first proposed as a way to build an underlying logic for proofs-assistants by A.Gordon [Gor94]. The key idea is, first to define a de Bruijn style syntax and then, to put back names in binders. To this end, we define functions on de Bruijn terms witch behave as binders but with names in their arguments. We have started with a calculus in witch terms are defined *up to α-conversion* and we build a COQ formalism very close to it. Beside α-equivalent terms are equal in our formalism. Thanks to this trick their is no apparent de Bruijn indices in our terms but, we have lost the possibility of directly define functions on our terms. For each function on our terms, we must dive back in the de Bruijn syntax and produce the proof that they do what we want. What can be seen a serious restriction can partially be overcome because we have already prove many results dealing with de Bruijn *lifting*.

As α-equivalent is in fact the Leibniz equality on our terms, we can strengthen our inductions theorem. Some how, we generate an induction scheme in witch all renaming that could be need in cases of capture of variable have already been done. As oppose to functions, there are no problems for defining inductive properties on our terms. But, as for the induction theorem, for each property defined on terms, it is be possible to produce a strengthen induction scheme in the same way than *term_induction*. We conjecture than the only condition is that the given property is invariant under renaming (implicit when one reason *up to* α-conversion). It is necessary to automate as much as possible the generation of such induction theorems. At least, our **concξ-calculus** working environment should have tools witch give us the theorems statement and the lemmas we need for proving them. Strengthen induction schemes produce strengthen inversion lemmas, we are also investigating this way. Some work should also be done to automate the application of our substitution functions on terms.

The next step is to formalize the type system proposed in [GH98], and prove the subject reduction theorem for this calculus. To encode the type system, we propose to use the same method for binders and, thanks to the low numbers of type constructors we think the proofs are going to be much easier than those needed for our implementation of the **concξ-calculus**.

Acknowledgments I specially thank Joëlle Despeyroux, my advisor, for enlightening discussions about this work.

References

[AC96] Martin Abadi and Luca Cardelli. *A Theory of Objects.* Monographs in Computer Science. Springer-Verlag, 1996.

[Bar95] B. Barras. Coq en Coq. Mémoire du DEA informatique, mathématiques et applications, École Polytechnique, 1995. INRIA research report RR-3026, October 1996.

[BBC+97a] B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Mu noz, 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.

[BBC+97b] Janet Bertot, Yves Bertot, Yann Coscoy, Healfdene Goguen, and Francis Montagnac. *User guide to the CtCoq proof environment*, October 1997. Inria technical report, RT-0210.

[DH94] Joëlle Despeyroux and André Hirschowitz. Higher-order syntax and induction in Coq. In F. Pfenning, editor, *Proceedings of the fifth Int. Conf. on Logic Programming and Automated Reasoning (LPAR 94)*, volume 822, pages 159–173. Springer-Verlag LNAI, July 1994. Also appears as INRIA Research Report RR-2292 (June 1994).

[DS99] Joëlle Despeyroux and Davide Sangiorgi. A higher-order specification of the pi-calculus. Draft, presented at the Modelisation and Verification seminar, Marseille, Dec 98, 1999.

[FI98] M.Miculan F.Honsell and I.Scagnetto. Pi calculus in (co)inductive type theories. Technical report, Universita' di Udine, September 1998.

[GH98] A. Gordon and P. Hankin. A concurrent object calculus: reduction and typing. In *Proceedings of the HLCL'98 Conference.* Elsevier ENTCS, 1998.

[Gor94] A. Gordon. A mechanisation of name-carrying syntax up to alpha-conversion. In *Proceedings of the 6th int. workshop on Higher Order Logic Theorem Proving and its Applications, Vancouver*, Springer-Verlag LNCS 780, pages 413–425, 1994.

[Hen98] Loïc Henry_Greard. A proof of type preservation for the pi-calculus in Coq. Research Report RR-XXX, Inria, December 1998. To appear. Available in the Coq Contrib library.

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

[HKPM97] Gérard 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.

[MP93] James McKinna and Robert Pollack. Pure Type Sytems formalized. In M. Bezem and J. F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 289–305. Springer-Verlag LNCS 664, March 1993.

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

[Wer94] B. Werner. *Une Théorie des Constructions Inductives.* PhD thesis, Université Paris VII, Mai. 1994.