A formalization of a concurrent object calculus up to α-conversion

Guillaume Gillard

INRIA Sophia Antipolis

Abstract. We experiment a method for representing a concurrent object calculus in the Calculus of Inductive Constructions. Terms are first defined in de Bruijn style, then names are re-introduced in binders. The terms of the calculus are formalized in the mechanized logic by suitable subsets of the de Bruijn terms; namely those whose de Bruijn indices are relayed beyond the scene. The α-equivalence relation is the Leibnitz equality and the substitution functions can de defined as sets of partial rewriting rules on these terms. We prove induction schemes for both the terms and some properties of the calculus which internalize the renaming of bound variables . We show that, despite that the terms which formalize the calculus are not generated by a last fixed point relation, we can prove the desire inversion lemmas. We formalize the computational part of the semantic and a simple type system of the calculus. At least, we prove a subject reduction theorem and see that the specifications and proofs have the nice feature of not mixing de Bruijn technical manipulations with *real* proofs.

1 Introduction

Providing a satisfactory method to encode the binding operators of a programming language when we want to formalize it in a Logical Framework is still a challenge. Although many different methods have been proposed so far, none seems completely satisfactory. From de Bruijn codes to higher order encoding each method has is advantages and disadvantages, its supporters and its detractors. A major problem raise by all these methods is that theorems and theirs proofs become highly linked with chosen encoding. In other words, if it is sometimes possible to have specifications and theorems close to the unmechanized version (which is not the case for de Bruijn encoding), the proof structures are themselves very different from the informal ones. In this paper, we show that the method proposed in [Gor94] for representing binders in mechanized logic can successfully be extended to a large calculus with different kinds of binders. Beside, we show that, once some work has been done with de Bruijn indices, *real* proofs do not manipulate them moreover, they are similar to the unmechanized ones.

We have chosen to formalize the **concς-calculus** [GH98], a concurrent object calculus consisting of M. Abadi and L. Cardelli's imperative object calculus **impς** [AC96] extended with primitives from the π-calculus [MPW92]. This calculus was introduced as a possible formalism for modeling computations based on concurrent processes and objects. We think that formal proofs of properties of protocols can be realized within proof-assistant if we have good methods for encoding such calculus. Our choice of this calculus is motivated by its size, its different kinds of binder and its good expressiveness, thus giving an idea of the real problems that arise when we encode such formalisms in computational logics.

The COQ system we use for our implementation is a proof-assistant based on the calculus of inductive constructions [Wer94], a higher order logic with dependent types and inductive definitions. All the proofs have been done with the user-interface CtCoq [BBC$^+$97]. 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 we only show significant parts of large COQ encodings. Please refer to [Gil00] for a full presentation of our techniqueal results.

Organization of the paper: In section 2 we formalize the concς-calculus in COQ. In section 3 we prove a more powerful induction theorem for the terms of the calculus which internalizes the renaming of bound names. In section 4 we formalize the semantics of the concς-calculus in COQ and we produce an efficient induction principle for the semantic relation. In section 5 we give a simple type system for the calculus and discuss the problems raised by the inversion lemmas generated by the COQ system on this example. Section 6 presents the statement and the proof of the subject reduction theorem. Section 7 is a short discussion about the formalizing technique we used. Finally, section 8 draws some conclusions.

2 The concς-calculus

This calculus was first introduced by A. Gordon and P. Hankin. It is the imperative object calculus **imp**ς of M. Abadi and L. 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 *references, variables* and *labels.* We distinguish references, representing addresses of stored object (channels in the π-calculus) from variables, representing intermediate values (variables in the λ-calculus). Let us call both notions *names.* The expressions of the language are defined as follows:

In the method $\xi(x)b$, and in the expression *let* $x = a$ *in* b the variable x is bound in b. In a restriction, $(\nu p).a$ the reference p is bound in a. The notation $a = b$ means that the terms a and b are equal up to bound names renaming and reordering of the labeled components of objects.

Table 1. Syntax of the informal **concς-calculus**

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
$a \upharpoonright b$	parallel composition
$\nu p.a$	restriction
d:=	**denotations**
$[l_i = \xi(x_i)a_i\ ^{i \in 1..n}]$	object

2.2 de Bruijn specification

We define a de Bruijn syntax (table 2) in which free names n, m are encoded by *named* names (reference?). Variables **x,y,z** are either *free variables x,y,x* or de Bruijn variables *(dvar i), (dvar j)*, labels l,l_i are named and references **p,q** are either *free references p,q* or de Bruijn references *(derf i),(dref j)*. We assume there are infinite disjoint sets of *references, variables* and *labels* and that equality is decidable on each of these sets.

Table 2. de Bruijn formalism

$$DB \overset{def}{=} \mathbf{x} \mid \mathbf{p} \mid \mathbf{p} \mapsto ODB \mid DB.l \mid$$

$$DB.l \Leftarrow_{db} DB \mid clone(DB) \mid let_{db}\ DB\ in\ DB \mid DB \upharpoonright DB \mid \nu_{db}.DB$$

$$ODB \overset{def}{=} [] \mid [l : DB :: ODB]_{db}$$

Except otherwise stated in the sequel of the paper we no longer refer to *references, variables, labels, results* of the informal calculus described in 2.1. De Bruijn terms *a,b,c* are call *dbterm* and for readability reasons, *var* and *ref* will be use for free variables and free references respectively in all our formal definitions.

The binding constructors here are $\Leftarrow_{db}$, let_{db} (in their second argument) and the *object-constructor* ($[\ \ ]_{db}$) for de Bruijn variables. The ν_{db} 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].

Our syntax is a bit more general than the one proposed by A. Gordon for his concς-calculus in the sense that we allow cloning, method calling, and method updating not only for results but for all terms of our syntax. This choice was motivated because *dbterm* are less nested than they would have been with a de Bruijn result type. Since the concς-calculus terms will eventually be identified by an inductively defined subset of *dbterm* this will not have any consequences on its formalization in COQ.

Thanks to de Bruijn indices we do not need an *alpha-equivalence* notion and $a = b$ means that the *dbterm* a and b are equal in the sense of the Leibnitz equality. De Bruijn formalization for binders takes off the syntax its intuitive meaning. We shall show how to recover it later on (see 2.5).

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.

2.3 Function as binders

Abstraction and instantiation functions. We define a variable abstraction function $Abst_v$. For a given *dbterm* a, $Abst_v(a\ x\ i)$ is computed by substituting in a all the occurrences of the variable x by the de Bruijn variable $(dvar\ i)$. The substitution is defined recursively on the *dbterms* such that the Bruijn indices substituted is increased by one each time a binder is met. In a dual way, we define an instantiation function $Inst_v$. $Inst_v(a\ i\ x)$ is computed by substituting all the occurrences of $(dvar\ i)$ in a by the variable x. Similarly, we define $Abst_r$ and $Inst_r$ on *references.*

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.

Table 3. Functions as constructor

$$(res\ p\ a) \overset{def}{=} \nu_{db}.Abst_r(a\ 0\ p) \qquad (eta\ x\ a) \overset{def}{=} Abst_v(a\ 0\ x)$$
$$(let\ x\ a\ b) \overset{def}{=} let_{db}\ a\ in\ Abst_v(a\ 0\ x)$$

[1] Using the Scheme tactic

In the following, we shall write $let\ x := a$ in b for $(let\ x\ a\ b)$, $\nu p.a$ for $(res\ p\ a)$ and $\xi(x).a$ for $(eta\ x\ a)$. We will also drop the $_{db}$ mark in a constructor when it can be guessed from the context.

2.4 Substitution and α-equivalence

To relegate de Bruijn indices of the underlying terms behind the scene we also need to define two new substitution functions $Subst_v$ and $Subst_r$. Intuitively these functions are defined to rename free *names* in *dbterms*. Their definitions (see definition table 4) use de Bruijn indices in their bodies but, with some work, we shall manipulate them without referring to de Bruijn indices (see 3.2). We write $a[x/y]_v$ and $a[p/q]_r$ for $Subst_v(a\ y\ x)$ and $Subst_r(a\ q\ p)$ respectively.

Table 4. Substitution functions

$Subst_v(a\ x\ y) \overset{def}{=} Inst_v(Abst_v(a\ 0\ x)\ 0\ y)$	$Subst_r(a\ q\ p) \overset{def}{=} Inst_r(Abst_r(a\ 0\ q)\ 0\ p)$

If we think of the ν, *let* and ξ function as constructors and $Subst_v$ and $Subst_r$ as renaming functions, we prove that α-equivalent *dbterms* are encoded in our formalism by a unique *dbterm*.

2.5 Formalization of the syntax

The result type $(u,v,)$ is defined as the disjoint union of our free names:

$$\mathsf{result} \overset{def}{=} \ var \mid ref$$

The inductive predicate Term (table 5) defines the subset of *dbterm* which formalizes the concς-calculus. The proof of correctness of this encoding is straightforward (omitted) if one thinks of *let*, *res* and *eta* as constructors.

From now on, we shall call *Term*, a *dbterm* having the *Term* property and write $\forall a : Term.(P\ a)$ as a short hand for $\forall a : dbterm.(Term\ a) \Rightarrow (P\ a)$ in the translation of our COQ notations. *Terms* are based on a de Bruijn formalism but de Bruijn indices are hidden in the syntax by the *let*, *res* and *eta* functions.

3 An induction principle for the concς-calculus

In the sequel, for readability reasons, we only show one (significant) case of the theorems. A more detailed presentation is available in [Gil00].

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

$Term : dbterm \rightarrow Prop :=$

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:	$\forall l : label.\forall a : dbterm.\forall obj : denotation.\forall x : var.$ $(Term\ a) \Rightarrow (OTerm\ obj) \Rightarrow (OTerm\ (l : \xi(x).a :: obj))$

3.1 The induction scheme generated by the COQ system

It appears than the induction scheme generated by COQ (table 6) for the predicate Term is not powerful enough for our purpose[2] .

Table 6. Induction scheme generated by COQ

Term_ind:=
$\forall P : dbterm \rightarrow Prop.$
...
$\langle \forall x : var.\forall a, b : Term.(P\ a) \Rightarrow (P\ b) \Rightarrow (P\ \ let\ x := a\ in\ b)\rangle \Rightarrow$
...
$\forall a, b : Term.(P\ a\ b).$

For example, it is not clear how one can derive the fact that *Terms* are closed under the substitution $Subst_v$ and $Subst_r$ with it. We shall not be able to deduce $(Term\ (let\ x := a\ in\ b)[z/y]_v)$ from $(Term\ a[z/y]_v)$ and $(Term\ b[z/y]_v)$ because we have not the necessary informations on x, y and z to compute $(let\ x := a\ in\ b)[z/y]_v$. In this COQ formalization of the concς-calculus α-equivalence terms are equal. We want to integrate inside the induction scheme of Term the fact that bounded *names*, in *Terms*, can always be renamed .

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

3.2 An intermediate induction scheme for *Terms*

We define a new function *length* on *dbterms* which computes the numbers of constructors appearing in a term. The order $\leq_{length}$ induced on *dbterms* by this function is well founded and we show that renaming *names* in a *dbterm* does not change its length. Using the general induction theorem for well founded relation with respect to $\leq_{length}$ we prove a more powerful induction scheme than *Term_ind* for the Term relation (table 7).

Table 7. Intermediate induction scheme

Term_length_ind:=
$\forall P : dbterm \rightarrow Prop.$
$\quad \dots$
$\quad \forall x : var. \forall a, b : Term.(P\ a) \Rightarrow \langle \forall b' : Term.\ length(b) = length(b') \Rightarrow (P\ b') \rangle \Rightarrow$
$\qquad\qquad\qquad\qquad (P\ \ let\ x := a\ in\ b)) \Rightarrow$
$\quad \dots$
$\forall a, b : Term.(P\ a\ b).$

With this theorem, as the length of *dbterms* is invariant for the renaming functions, we prove that substitution can be propagated inside binders for *Terms* if the *side conditions* are satisfied (see table 8).

Table 8. substitution rewriting rules for the *let* binder

let_rw1: $\forall x, y, z : var. \forall a, b : Term. x \neq y \Rightarrow x \neq z \Rightarrow$
$\qquad (let\ x := a\ in\ b)[z/y] = let\ x := a[z/y]\ in\ b[z/y].$
let_rw2: $\forall x, y : var. \forall a, b : Term.(let\ x := a\ in\ b)[y/x] = let\ x := a[y/x]\ in\ b.$

3.3 The full induction scheme for *Terms*

The use of the *length* function inside *Term_length_ind* is not satisfactory because this is not natural. We prove a final induction scheme on *Term* (table 9) using the *Term_length_ind* theorem and the properties of the substitutions functions we have deduced from it.

This induction scheme internalizes the property that bounded *names* can always be chosen outside any set of *names* in the context.

Example: *Given a property P on $Term$, we prove that it holds for all $Terms$ using the term-induction theorem. In order to prove that $(P\ \ let\ x := a\ in\ b)$ holds, we select a finite set X and try to solve our goal under the assumptions $(P\ a)$, $(P\ b)$ and $x \notin X$. Giving the set X amounts to specify that x is a fresh variable.*

Table 9. Induction scheme for the *Term* predicate

Term_induction:=
$\forall P : dbterm \rightarrow Prop.$
$\quad \ldots$
$(\forall a : Term.(P\ a) \Rightarrow \langle \exists X : set.(Finite\ X) \wedge$
$\qquad \forall x : var.\forall b : Term.\ x \notin X \Rightarrow (P\ b) \Rightarrow (P\ \ let\ x := a\ in\ b)\rangle) \Rightarrow$
$\quad \ldots$
$\forall a, b : Term.(P\ a\ b).$

4 Semantics of the concς-calculus

The semantics of the calculus is given by a reduction relation and a structural congruence. The formalization of the reduction rules in COQ is natural and we can prove an induction scheme which internalizes α-renaming .

4.1 Rules for the semantics

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. As opposed to many concurrent calculi the parallel composition ($\Rsh$) is not commutative. The term $a \Rsh b$ is an expression in which a and b run in parallel. Its result is the result returned by b; any result returned by a is discarded. The structural congruence ($\equiv$), except from the unusual behavior for ($\Rsh$) is standard. The reduction relation ($\rightarrow_{red}$) (table 10) is the matching piece to the 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.

Table 10. Reduction relation: $a \rightarrow b$

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

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

The notations $fn(a)$ and $fv(a)$ denote respectively the sets of free names and free variables in the expression a. The expression $a[p/x]$ is the notation for the substitution of the reference p for each free occurrence of the variable x in the expression a.

Formalization in COQ. We use two inductive definitions to formalize the above relation in COQ. The first one (table 11) is a restriction to *effective* reductions in terms. The second one (table 12) is the complete formalization in the COQ system of the semantics (proof omitted here). We use this trick to prevent looping in the proofs. In the sequel of this paper we shall focus on the first definition.

The COQ formalizations of both relations are the natural translations of the rules in table 10 into inductive definitions ($\rightarrow_{red}$ and $\rightarrow_{eval}$). The $\rightarrow_{red}$ relation is defined for *Terms* and not *dbterms* so, for every *dbterm* a appearing in the COQ definition of $\rightarrow_{red}$ $(Term\ a)$ must hold.

Table 11. Formalization of the $\rightarrow_{red}$ relation in COQ

$\rightarrow_{red}$: $dbterms \rightarrow dbterms \rightarrow Prop :=$
$\quad\dots$
Let_red1: $\forall a : Term.\forall p : ref.(let\ x := p\ in\ a) \rightarrow_{red} a[p/x]_*$
$\quad\dots$

For any given term a, $Subst_*(a\ x\ p)$, written $a[p/x]_*$, is computed by substituting all the occurrences of the variable x by the reference p in a. In the COQ system, $Subst_*$ must be defined on *dbterms* and de Bruijn indices are used in the body of this function. With the help of technical lemmas, we show that $Subst_*$ restricted to *Terms* can be manipulated without dealing with de Bruijn indices.

Table 12. Formalization of the reduction relation $\rightarrow$ in COQ

$\rightarrow_{eval}$: $dbterm \rightarrow dbterm \rightarrow Prop :=$
Eval: $\forall a, a', b, b, : Term.(a \equiv a') \Rightarrow (b \equiv b') \Rightarrow (a' \rightarrow_{red} b') \Rightarrow (a \rightarrow_{eval} b)$

4.2 Induction scheme for the semantics

As before, for readability reasons, we only show one (significant) part of the theorems. Courageous readers could refer to [Gil00] for a more detailed presentation.

We can extend the induction scheme for the $\rightarrow_{red}$ relation as we did for the *Term* predicate. In the induction scheme generated by the COQ system (table 13) we do not have any informations for binded *names*.

Table 13. $\rightarrow_{red}$ induction scheme generated by COQ

Red_ind:=
$\forall P : dbterm \rightarrow dbterm \rightarrow Prop.$
$\quad \ldots$
$\quad (\forall p : ref. \forall a, b : Term. (a \rightarrow_{red} b) \Rightarrow (P\ a\ b) \Rightarrow (P\ \nu p.a\ \nu p.b)) \Rightarrow$
$\quad \ldots$
$\forall a, b : dbterm. (a \rightarrow_{red} b) \Rightarrow (P\ a\ b).$

Following the idea of the section 3 we can produce an extended induction scheme which internalizes the α-renaming of bound *names* in proofs. By using the general well founded induction theorem for a suitable order on pairs of *dbterm* we prove an intermediate theorem (table 14) in which the length of *dbterm* is introduced.

Table 14. $\rightarrow_{red}$ induction scheme with the *length* function

Red_length_ind:=
$\forall P : dbterm \rightarrow dbterm \rightarrow Prop.$
$\quad \ldots$
$(\forall p : ref. \forall a, b : Term.$
$\langle \forall a', b' : Term.$
$length(a) = length(a') \Rightarrow length(b) = length(b') \Rightarrow (a' \rightarrow_{red} b') \Rightarrow (P\ a'\ b') \rangle$
$\qquad \Rightarrow (a \rightarrow_{red} b) \Rightarrow (P\ \nu p.a\ \nu p.b)) \Rightarrow$
$\quad \ldots$
$\forall a, b : dbterm. (a \rightarrow_{red} b) \Rightarrow (P\ a\ b).$

At least, we prove the induction scheme (table 15) in which bounded *names* can be chosen *outside* the set of *names* in the context. To prove the theorem, we first need an intermediary lemma stating that $\rightarrow_{red}$ is closed for *names* renaming. More precisely, we show that $\rightarrow_{red}$ is closed for *names* renaming provided that *names* are renamed in new *names*.

Table 15. $\rightarrow_{red}$ induction scheme

Red_induction:=
$\forall P : dbterm \rightarrow dbterm \rightarrow Prop.$
$\quad \ldots$
$\langle \exists Q : set. (Finite\ Q) \wedge$
$\quad (\forall p : ref. \forall a, b : Term. p \notin Q \Rightarrow (a \rightarrow_{red} b) \Rightarrow (P\ a\ b) \Rightarrow (P\ \nu p.a\ \nu p.b)) \rangle \Rightarrow$
$\quad \ldots$
$\forall a, b : dbterm. (a \rightarrow_{red} b) \Rightarrow (P\ a\ b).$

5 Well-formed terms

The concς-calculus can be typed to distinguish expressions from processes. This very basic types system has only two types Exp and $Proc$ standing for expressions and processes respectively. Basically, this typing system only ensures that proper processes cannot appear in a context expecting an expression and that references are correctly handled in a term. A term a is defined as an *expression* or a *process* if $a : Exp$ and $a : Proc$, respectively.

5.1 Definition

The typing rules are define in table 16. T stands for either Exp or $Proc$. The *domain* of a term a, $dom(a)$ is the set of the free references representing the addresses of an object. Please refer to [GH98] for a general overview.

Table 16. The $well - formed$ relation

$$\text{(Well Result)}\ \frac{}{u : Exp} \qquad \text{(Well Clone)}\ \frac{}{clone(u) : Exp} \qquad \text{(Well Res)}\ \frac{a : T \quad p \in dom(a)}{\nu p.a : T}$$

$$\text{(Well Select)}\ \frac{}{u.l : Exp} \qquad \text{(Well Concur)}\ \frac{a : Exp}{a : Proc} \qquad \text{(Well Update)}\ \frac{b : Exp \quad dom(b) = \emptyset}{u.l \Leftarrow \varsigma(x)b : Exp}$$

$$\text{(Well Let)}\ \frac{a : Exp \quad b : Exp \quad dom(b) = \emptyset}{let\ x = a\ in\ b : Exp} \qquad \text{(Well Par)}\ \frac{a : Proc \quad b : T \quad dom(a) \cap dom(b) = \emptyset}{a \upharpoonright b : T}$$

$$\text{(Well Object)}\ \frac{b_i : Exp \quad dom(b_i) = \emptyset \quad \forall i \in 1..n}{p \mapsto [l_i = \varsigma(x_i)b_i^{i \in 1..n}] : Proc}$$

The COQ formalization of the *well-formed* relation is its natural translation as an inductive definition *well_formed*, given table 16. *well_formed* is defined for *Term* and not *dbterms*. We must insure than for every term a, $(Term\ a)$ holds in the COQ definition.

5.2 Inversion

In the activity of proofs, inversion theorems are as important as induction schemes. In the usual cases, inversion theorems automatically generated by the proof assistants are those expected because the syntax of the calculi are defined in terms of a least fixed point. In our formalism, binders are functions on top

Table 17. Formalization of $well\_formed$ in COQ

$$\mathsf{flag} \overset{def}{=} \quad Exp \mid Proc$$

$well\_formed : dbterms \rightarrow flag \rightarrow Prop :=$
$\quad \ldots$
Well_Res: $\forall a : Term.\forall p : ref.\forall T : flag.(well\_formed\ a\ T) \Rightarrow$
$\qquad\qquad p \in dom(a) \Rightarrow (well\_formed\ \nu p.a\ T).$
$\quad \ldots$

of a de Bruijn syntax thus from the equality $\nu p.a = \nu q.b$ it is not possible to deduce than $p = q$ and $a = b$. If the inversion theorems generated by COQ are used roughly they introduced news terms not directly related to anything in the proof. In table 18 we present the inversion theorem for the *Well_Res* constructor of the *well_formed* property generated by COQ [3].

Table 18. Inversion lemma for $well\_formed$ generated by COQ

$\forall P : dbterm \rightarrow flag \rightarrow Prop.$
$\forall a : Term.\forall p : ref.\forall T : flag.$
$\langle \forall q : names.\forall b : Term. \quad \nu p.a = \nu q.b \Rightarrow q \in dom(b) \Rightarrow (well\_formed\ b\ T) \Rightarrow (P\ a\ T)\rangle$
$\qquad \Rightarrow (well\_formed\ \nu p.a\ T) \Rightarrow (P\ a\ T).$

In a proof in which $(well\_formed\ \nu p.a\ T)$ is amongst the assumptions, using this theorem will not add $(well\_formed\ a\ T)$ in the hypothesis as expected. Similarly to induction schemes, the *right* inversion lemmas must be proved. Fortunately, it is sufficient to derive a specialized lemma for each constructors of the inductive definition. Then the COQ system provides tactics to use them properly[4]. Because we can produce one lemma for each constructor, their formulation remains simple (see table 19 as an example).

Table 19. Inversion lemma for the *Well_Res* constructor

Lemma *well_res_inv*: $\forall P : dbterm \rightarrow flag \rightarrow Prop.$
$\qquad\qquad\qquad \forall a : Term.\forall p : ref.\forall T : flag$
$(p \in dom(a) \Rightarrow (well\_formed\ a\ T) \Rightarrow (P\ a\ T)) \Rightarrow (well\_formed\ \nu p.a\ T) \Rightarrow (P\ a\ T).$

To prove *mres_inv* we use a property stating than if $\nu p.a = \nu p.b$ holds then $a = b$ holds (such properties has to be proved for each of our binders). To complete the proof, we must show that if $(well\_formed\ a\ T)$ holds for a term a

[3] The COQ system generates a general inversion theorem for *well_formed*; this is a specialized version

[4] inversion *hyp* using *lemma*

then $(well\_formed\ a[p/q]\ T)$ holds for any q and any p such that $p \notin Q$ for a finite set Q. In other words, we need to prove than *well_formed* is closed under reference (name in general) renaming. Again, this is not surprising. This property of the relation *well_formed* should also be checked when we are reasoning *up to α-conversion* during informal proofs, through this is most often omitted.

6 Subject reduction theorem

We show that *well_formed* terms are closed for the $\rightarrow_{red}$ relation. The formulation of this theorem (see table 20) is exactly the same as its unmechanized version appearing in [GH98]. The proof is done by induction on $\rightarrow_{red}$ using the extended induction theorem (see table 15). For each induction case, there is an hypothesis of the form $(well\_formed\ a\ T)$. We use our inversion lemmas to extract informations on sub-terms of a from it.

Table 20. Subject reduction theorem

Theorem srt:
$\forall a, b : dbterm. \forall T : flag.\ (well\_formed\ a\ T) \Rightarrow (a \rightarrow_{red} b) \Rightarrow$
$(well\_formed\ b\ T) \wedge dom(a) = dom(b).$

The proof of the theorem is very closed to the informal proof with implicit renamings of bound names. We do not manipulate de Bruijn indices neither are we doing α-renaming. All the lemmas used during the proof have a semantic contents.

7 Discussion and related work

The size of the different parts of the COQ code is summarize in the table below. In the column of Term we consider all the formalizations and proofs necessary for using the *Terms*. It includes the properties for the α-conversion and the renaming, the proofs of the extended induction principles the Term property most of the lemmas we have proved with the theorem *Term_induction*. We classify in the column of well_formed and $\rightarrow_{red}$ all the COQ codes which deal with the corresponding property (induction schemes, inversion lemmas, behavior of $Subst_*$ and *dom*). The srt column stand for the subject reduction theorem COQ codes part and the total column include all the previous ones plus some general lemmas (mainly set theory theorems) which do not use de Bruijn indices.

	Term	well_formed	$\rightarrow_{red}$	srt	**total**
lignes of COQ code	7 700	2 400	2 500	1000	14 600
% of de Bruijn code	65%	10%	25%	-	40%

The percentage of the de Bruijn code in proofs is high during the setting of this technique. In fact, large de Bruijn codes mainly concern the $Abst$ and $Inst$ functions. But, once we have completely mastered the behaviors of *Terms* we do not use de Bruijn indices.

Given a property $P : Term \rightarrow \ldots \rightarrow Term \rightarrow Prop$, we must show that there exists a finite set X such that $m \notin X \Rightarrow (P\ a_1 \ldots a_n) \Rightarrow (P\ a_1[m/n] \ldots a_n[m/n])$ to get an induction principle which internalizes name renaming and the expected inversion lemmas. Checking that P is closed by renaming of names can be laborious in COQ whereas this is assumed for *on paper* proofs. Moreover, as we have experimented during this development, it clears the way for further proofs on the P property.

Related work. Among all the works formalizing the variable-binding operators in calculi none, as far as we know, uses the technique we have used here. Daniel Hirschkoff has encoded a polyadic π-calculus with de Bruijn numbers and proved many bisimulation results [Hir97]. Bruno Barras [Bar95] formalizes COQ in COQ with de Bruijn indices. In both approaches de Bruijn indices appear in almost all theorems and specifications. We think this is not natural. L. Henry-Gréard [Hen98] uses R. Pollack and J. McKinna technique [MP93] to formalize the π-calculus and prove a subject reduction theorem for it. In this technique, closer to the *on paper* formalism, there are two kinds of names, one for free ones and another for bound one. We think this is not completely natural. 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 [Des]. See [DH94], for a general approach of this technique in COQ. F. Honsel, M. Miculan and I. Scagnetto [FI98] have encoded the π-calculus in COQ following a higher order approach. They use *Co inductive* types in their encoding of bisimulation. Although second order techniques are very efficient, we think that proofs using these techniques are very different from proofs on paper.

8 Conclusion and future work

We have formalized a concurrent object calculus in the COQ system with names in binders using a technique proposed by A. Gordon [Gor94]. We have shown that defining properties on *Terms*, namely those who formalize the concς-calculus in the COQ system, is very natural and easy because we just need to rewrite them using the COQ syntax. Under the assumption that a given property P is invariant under the renaming of names, the induction theorem generated by COQ for P can be strengthened to internalize α-renaming of bound variables. In spite of our syntax is not generated by a last fixed point we have inversion lemmas for P but they must be proved. The proofs of these theorems as the proof of the subject reduction theorem, are de Bruijn indices free. Moreover, the proofs dealing with *real* property of the calculus follow the general guideline of their *on paper* matching piece.

The main drawback of this approach is that each time we have to define functions on *Terms* we have to define them on *dbterms* first, then prove that they behave as expected on *Terms*. We believe that with a good understanding of the behavior of a function on *Terms*, it is not hard to give its definition on *dbterms*. We claim that this weakness does not overcome the advantages of the method. In fact, new functions on our syntax will probably use functions we have already defined, allowing re-use of our COQ proofs (as it is done for the function $Subst_*$ which appears in $\rightarrow_{red}$).

For property P, the strengthened induction theorems could be a large term. It is interesting to develop tools for generating it automatically because this extended induction scheme is mechanically derivable (not provable) from P. Another reasonable development could be to include tactics for automating, on *Terms*, the computation steps of functions. We have done some preliminary work in this direction.

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$^+$97] 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.

[Des] Joëlle Despeyroux. A higher-order specification of the pi-calculus. Presented at the Modelisation and Verification seminar, Marseille, Dec 98. Submitted for publication, March.

[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).

[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.

[Gil00] Guillaume Gillard. A full formalization of a concurrent object calculus up to alpha-conversion. draft, January 2000. Available at `//ftp-sop.inria.fr/certilab/ps/conc_calculus.ps`.

[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-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 Amy Felty and Elsa Gunter, editors, *Proceedings of the International Conference on Theorem Proving in Higher Order Logics*, Murray Hill, New Jersey, August 1997.

[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.