A Full Formalisation of π-Calculus Theory in the Calculus of Constructions

Daniel Hirschkoff*

CERMICS - ENPC/INRIA and
Dipartimento di Scienze dell'Informazione - Università di Roma "La Sapienza"

Abstract. A formalisation of π-calculus in the Coq system is presented. Based on a de Bruijn notation for names, our implementation exploits the mechanisation of some proof techniques described by Sangiorgi in [San95b] to derive several results of classical π-calculus theory, including congruence, structural equivalence and the replication theorems. As the proofs are described, insight is given to the main implementational issues that arise in our study, without entering too much the technical details. Possible extensions of this work include the full verification for the "functions as processes" paradigm, as well as the design of a system to check bisimilarities for processes.

1 Introduction

This work is a continuation of [Hir97]; it presents a formalisation of some results of π-calculus theory in the Calculus of Inductive Constructions. π-calculus [MPW92, Mil91] has become a widely accepted theoretical model for concurrency, and aims at playing the rôle λ-calculus does play for (functional) sequential computation. Implementing the proofs for some classical laws of the algebraic theory of π-calculus in a logical framework like Coq raises many interesting questions, that help to understand both the results we prove and the techniques we use for these proofs. The main paradigm we focus on to compare π-calculus terms is *bisimulation*, which does not seem to be a handy notion to manage when it comes to proof mechanisation (see Section 3). To perform bisimulation proofs, we have implemented a theory described by Sangiorgi in [San95b], called in this paper *theory of progressions*, that turns out to be very tractable for our purposes, and allows us to prove many bisimilarity results of classical π-calculus theory in a simple and uniform way. Moreover, the generality of the results we derive within the theory of progressions gives a great expressiveness to our implementation, as the proof of the so-called *replication theorems* testifies.

In this paper, we will not describe precisely the whole Coq implementation we have built, but rather focus on the many implementational issues that arise as we formalise the theory of π-calculus. We believe that the present work at the same time shows the expressiveness of a logical framework like Coq and provides some insight on verification in the field of concurrency. The plan of the paper is the following: the remainder of this section is devoted to a very brief

* email: dh@cermics.enpc.fr

presentation of Coq. Section 2 presents our implementation of π-calculus syntax and semantics and describes the de Bruijn representation for names. The next Section presents Sangiorgi's theory of progressions, and the proof of the up to context bisimulation theorem, which is the main technique we use to establish bisimilarity relations. Section 4 presents some classical theorems of π-calculus theory that we prove this way, namely congruence and structural equivalence, and Section 5 is devoted to the proof of the replication theorems. We also discuss in Section 5 the encoding of λ-calculus into π-calculus, and the corresponding proof of operational correspondence, which has not been implemented yet. In the conclusion, we give some technical informations about our proofs, describe some related work and discuss future extensions of this implementation.

Coq [BBC+96a] is a proof-assistant based on the Calculus of Inductive Constructions [Wer94]. Let us briefly recall the syntax of Coq declarations (in the following, Coq text will be in `typewriter` style): product types can be written `(_)_` (dependent product) or `_->_` (non dependent product); abstraction is written `[_]_`, while application is as usually written `(_ _)`. A new inductive type is introduced by defining its constructors with their respective types; this automatically generates elimination lemmas, that allow one to perform induction over elements of these types. Among the primitive objects, let us mention inductive type `nat` for natural numbers, whose constructors are `O` and `S`.

2 A π-Calculus Implementation

In this Section, we first briefly describe the syntax and semantics for the π-calculus we have implemented. We then turn to our implementation, stressing the consequences of our choice of a de Bruijn notation for names.

2.1 Syntax and Semantics of Processes

Terms of the π-calculus, called *processes*, are built upon an infinite set of names $N = x, y, z, a, b, \ldots$ (also called *channels*). In the following, we range over (possibly empty) name lists with $\mathbf{a}, \mathbf{x}, \ldots$, and over processes with $P, Q, R, \ldots$. The language we have implemented can be described by the following syntax:

$$P \ \equiv \ \mathbf{0} \mid a(\mathbf{b}).P \mid \overline{a}[\mathbf{b}].P \mid (P|Q) \mid (\nu a)P \mid !P.$$

$\mathbf{0}$ is the inactive process; together with the two kinds of prefixes ($\overline{a}[\mathbf{b}].$ for output and $a(\mathbf{b}).$ for input), we have operators $|$ (parallel composition), ν (restriction), and ! (replication). Several consecutive restrictions $(\nu a_1)\,(\nu a_2)\,\ldots\,(\nu a_n)$ may be written as $(\nu \mathbf{a})$, and we further generalize this notation by saying that if $\mathbf{a}$ is the empty list, then $(\nu \mathbf{a})P$ is P.

The operational semantics of processes is given through the definition of a transition relation, stating that a process P can perform an *action* a to become process P'; we write this $P \xrightarrow{a} P'$. Actions are described by

$$A \ \equiv \ a(\mathbf{b}) \mid (\nu \mathbf{c})\overline{a}[\mathbf{b}]_{(\mathbf{c} \subseteq \mathbf{b})} \mid \tau,$$

where τ is a special action, called the silent action, and denoting an internal communication in a process. The side condition for output actions ($\mathbf{c} \subseteq \mathbf{b}$) will be explained below. Bound and free names both for processes and actions are defined as usual. The rules for early operational semantics are stated in table 1; symmetrical version of rules PAR_l and $CLOSE_1$ are omitted.

$$\texttt{INP} \quad x.(\mathbf{y})\,P \xrightarrow{x(\mathbf{z})} P\{\mathbf{y} := \mathbf{z}\}$$

$$\texttt{OUT} \quad \bar{x}.[\mathbf{y}]\,P \xrightarrow{\bar{x}[\mathbf{y}]} P$$

$$\texttt{OPEN} \quad \frac{P \xrightarrow{(\nu\,\mathbf{z})\bar{x}.[\mathbf{y}]} P'}{(\nu t)P \xrightarrow{(\nu\,t::\mathbf{z})\bar{x}.[\mathbf{y}]} P'} \qquad t \in \mathbf{y},\ t \neq x$$

$$\texttt{CLOSE}_1 \quad \frac{X \xrightarrow{a(\mathbf{l})} P \qquad Y \xrightarrow{(\nu\mathbf{k})\bar{a}[\mathbf{l}]} Q}{X \mid Y \xrightarrow{\tau} (\nu\mathbf{k})(P \mid Q)} \quad \mathbf{k} \cap fn(X) = \emptyset$$

$$\texttt{RES} \quad \frac{P \xrightarrow{\mu} P'}{(\nu x)P \xrightarrow{\mu} (\nu x)P'} \qquad x \notin n(\mu)$$

$$\texttt{PAR}_l \quad \frac{P \xrightarrow{\mu} P'}{Q|P \xrightarrow{\mu} Q|P'} \qquad bn(\mu) \cap fn(Q) = \emptyset$$

$$\texttt{BANG} \quad \frac{!P|P \xrightarrow{\mu} P'}{!P \xrightarrow{\mu} P'}$$

Table 1. Early operational semantics for π-calculus terms

Let us make a few remarks about the version of π-calculus we have chosen. Our processes belong to a polyadic (in the sense that many names can be transmitted in a single communication), and sum free π-calculus; both polyadic communications and the sum operator (also called *choice operator*) can be encoded into a monadic, sum free π-calculus (see [Mil91, NP96]). While sum can be useful from a theoretical point of view to build axiomatisations, it turns out that it can be discarded without much loss of convenience for implementation tasks. Polyadicity, on the other hand, is a very useful feature when taken as primitive, because it makes the treatment of many examples (like the study of λ-calculus encoding discussed in Section 5) much easier. The PICT programming language [Pie95] is another example of a π-calculus implementation where polyadicity is in some way "more primitive" than the choice construct (which is available, though, through the notion of *event channel*).

Terms possibly having an infinite behaviour are encoded using replication (!) instead of recursive definitions, mainly for the task of simplicity and clarity in our implementation (a single constructor is easier to introduce in Coq than a construction to represent recursive definitions). This is not a limitation, as

[Mil91] shows how to encode recursive definitions into replicated terms; as will be discussed below, this choice influences the shape of our proofs.

Finally, let us remark that our choice of an *early* transition semantics (as opposed to *late* or *open*, for example) was taken to stay as close as possible to the proofs given in [San95b], that are described in Section 3.

2.2 The de Bruijn Notation for Names

Processes For the task of our Coq implementation, we represent the basic notion of π-calculus syntax, that is names, with a de Bruijn notation (following the ideas of [Hue93]). In the de Bruijn representation (which is defined in [dB72]; a description of a de Bruijn notation for monadic π-calculus can be found in [Amb91]), a name is defined by a natural number indicating its *binding depth* inside the term where it occurs, i.e. the number of binding operators that have to be crossed to reach the operator it refers to. Let us give an example: in λ-calculus, there is one binding operator, abstraction, and the (closed) λ-term $\lambda x \lambda y.(y\ x)$ is represented with de Bruijn indexes as $\lambda\lambda.(\mathbf{0}\ \mathbf{1})$. If free variables occur, they are considered to be bound "above" the term, or alternatively one supposes that a binding list (the environment) is provided with the term. Such a way of writing the syntax of a term has an immediate advantage, in that it provides a canonical representation for bound names, and hence gives α-conversion for free. It turns out, however, that free names have to be handled with care in such a setting, and in some way one could say that *what is gained on bound names gets lost for free names*, as will be discussed in the next Section.

In the π-calculus, there are two binders for names, namely input prefix $a(\mathbf{b})$, that binds names of $\mathbf{b}$ in its continuation (i.e. the process being prefixed), and restriction (νx). This means that a de Bruijn index can either represent a name that is not yet known (bound by input prefix), or private (bound by restriction). Moreover, the input prefix is polyadic: this compells us to give an arity to such a construct, i.e. a number indicating how many names are bound in a receiving process (arity of agents also appears in [Mil91], for the task of the definition of sorting). For restriction, on the other hand, we have kept a monadic flavour, as suggested by the shape of rules *RES* and *OPEN* in Table 1.

The Coq inductive types for names, name lists and processes are called respectively `name`, `l_name` and `pi`, and are defined as follows:

```
Inductive name : Set := Ref : nat -> name.

Inductive l_name : Set := Nil : l_name | Cons : name -> l_name -> l_name.

Inductive pi : Set :=
  Skip : pi                          null process                  0
| Res : pi -> pi                     restriction (monadic)         ν
| Ban : pi -> pi                     replication                   !
| Par : pi -> pi -> pi               parallel composition          |
| Inp : name -> nat -> pi -> pi      abstraction: subject, arity,  ( )
                                     and continuation
```

```
| Out : name -> l_name -> pi -> pi.
```
concretion: subject, object, and continuation [ ]

Let us give an example to illustrate the de Bruijn representation for names; to represent process $P \equiv a(x,y).\overline{x}[y].\mathbf{0}$ in Coq, we make the following statement:

```
Definition P := (Inp (Ref 0) (S (S 0))
                  (Out (Ref (S 0)) (Cons (Ref 0) Nil) Skip)).
```

Notice how the choice of `(Ref 0)` to represent a is arbitrary, as a is free in P; this point will be further discussed in Section 3.

Actions Actions are built the same way as processes; some remarks can be done regarding their structure within the de Bruijn framework. The side condition for output actions ($\mathbf{c} \subseteq \mathbf{b}$), is explained by the shape of rules *OPEN* and *RES* of the semantics, that allow a communication to carry fresh names, i.e. names that before communication are not known by the receiving process (these names correspond to list $\mathbf{c}$). The great expressiveness of the π-calculus heavily relies on this phenomenon (called *name extrusion*), namely the communication of one or several private names. From the point of view of implementation, since actions may actually "carry restrictions", we have to keep track of the names that are bound inside an output action (this holds for input actions as well, because of our choice of early semantics, that forces to instantiate received names as soon as possible): since we work with de Bruijn indexes, a natural number is enough to indicate how many names among those in the transmitted name list are fresh (as for processes, names having an index less than this number are fresh, i.e. bound in the action). An input or output action thus consists of three parts: the *subject* of the action (the channel where the communication occurs), the *object* (the transmitted name list), and the number of fresh names. The type `action` is defined as follows:

```
Inductive action : Set :=   Ain : nat -> name -> l_name -> action
  | Aou : nat -> name -> l_name -> action  | Tau : action.
```

Transition relation The type `commit`, used to represent the transition relation, is inductively defined *à la Prolog*, each constructor expressing one of the rules of Table 1. Without stating the full definition of `commit`, which would require some pretty technical explanations, we will focus on a few examples, that we consider to be emblematic of the strong influence of the de Bruijn notation in our implementation.

A first immediate consequence of the use of a de Bruijn representation for names is that many side conditions in the transition rules can be discarded, by insuring their validity *within* the definitions of constructors. `comm_pl`, for example, implements rule PAR_l, and is defined as follows:

```
comm_pl : (x,p:pi)(a:action) (commit x a p) ->
    (y:pi) (commit (Par y x) a (Par (lift_pi y 0 (bound_action a)) p))
```

(recall that `(_)_` is the notation for dependent product; `bound_action` is the function that returns the number of bound names of an action). As can be seen on this definition, the inactive process `y` is in some way modified by the action, by the means of function `lift_pi`: the latter operator adds a number (here `(bound_action a)`) to all indexes of free names of `y`, hence avoiding possible name clashes with the fresh names brought by the performed action. The advantage of such a technique is that the renaming that has to be done in order to avoid conflicts between names is *built in*, and we do not have to check for such conflicts.

Rules *OPEN*, *RES* and $CLOSE_1$ define the behaviour of the restriction constructor in a communication. Intuitively, when a process is liable to send a list $\mathbf{l}$ of names on a given channel a, then if we add a restriction to this process, either we restrict a, in which case no communication is possible (because no other process can listen on a), or we apply one of the rules *RES* and *OPEN*. If the restricted name occurs in $\mathbf{l}$, then rule *OPEN* is applied (name extrusion); otherwise, rule *RES* is applied. Rule *RES* can also be used to add a restriction to a receiving process (if the restricted name is not among the names of the action) or to a process that performs an internal communication.

Within the de Bruijn framework, these rules imply some manipulation of the name that gets restricted, in order to "follow" its evolution inside the term. Indeed, as rule *RES* is applied, the restricted name is not involved in the action, and has to "pass over" those names that participate in the action. This is performed by exchanging some indexes inside the term, with an operator similar to function `lift_pi` discussed above, called `swap` (notice as well how action `a` becomes `(lower_action a)`, i.e. `a` with all indexes lowered by 1, because in some way the restricted name "crosses" the action):

```
comm_re : (x,p:pi) (a:action) (commit x a p)
  -> ~(occ_act a)                          (* side condition *)
  -> (commit (Res x) (lower_action a)
             (Res (swap p (bound_action a) 0 0)))
```

In the implementation of rule *OPEN*, some other technical manipulations have to be done on the list of transmitted names (and hence on the processes involved in the transition). Indeed, in classical π-calculus theory, processes $P_1 = (\nu x)(\nu y)\overline{a}[x,y].\mathbf{0}$ and $P_2 = (\nu y)(\nu x)\overline{a}[x,y].\mathbf{0}$ are considered to be equivalent (i.e. it should be provable from the rules of Table 1 that these processes are bisimilar). If we define these processes within the de Bruijn notation, we get (informally) $P'_1 = \nu\nu\overline{k}[1,0].\mathbf{0}$ and $P'_2 = \nu\nu\overline{k}[0,1].\mathbf{0}$, k being the index that represents free name a in the translation. With a naive definition of operational semantics, we cannot hope to prove the equivalence, since the actions they are liable to perform ($\nu^2\overline{k}[1,0]$ for P'_1, $\nu^2\overline{k}[0,1]$ for P'_2) are different. Actually, we have to provide a kind of α-conversion for actions, in order to relate these processes. The solution we adopt is to embed a canonical representation for actions *into the definition* of the transition relation: each time rule *OPEN* is applied, names in the transmitted list are exchanged in order to put the outermost restricted (fresh) name

as close as possible to the head of the list. Without entering the technical definition of such a manipulation, let us just stress the fact that this guarantees a unique representation for actions, allowing us in particular to prove processes P'_1 and P'_2 bisimilar (intuitively, as *OPEN* rule is applied for the second time to derive the transition of P'_2, indexes 0 and 1 are exchanged in order to get the outermost restricted one (1) at the head of the list). Such an embedding of a canonical form for actions is reminiscent of the "de Bruijn philosophy", in that the mechanism we provide to prove transitions gives the conversion between actions for free. We do not state here the definition of constructor `comm_op`, that implements rule *OPEN*, as its form should need explanations of some more technical details. However, we hope that some insight onto the main ideas related to the management of restrictions in the de Bruijn framework has been given. We have not proved that the full definition of `commit` (that can be found in [Hir97]) actually encodes the presentation given above; however, the many classical results we have been able to prove within our framework give confidence about its correctness.

3 Bisimulation and Related Proof Techniques

Having described how processes are constructed and how they behave, we may want to compare them; a popular notion to describe process equivalence is *bisimulation*. Let us state its usual definition:

Definition 3.1 (Bisimulation, bisimilarity) *A symmetrical relation $\mathcal{R}$ over processes is a bisimulation iff, whenever $P\mathcal{R}Q$ and $P \xrightarrow{a} P'$, there exists a process Q' s.t. $Q \xrightarrow{a} Q'$ and $P'\mathcal{R}Q'$. Bisimilarity (written $\sim$) is the greatest bisimulation.*

3.1 Sangiorgi's Theory of Progressions of Relations

Definition 3.1 contains in an essential way a certain form of circularity: two bisimilar processes perform the same action and stay "inside" the given bisimulation relation. When it comes to actually build bisimulation proofs, and *a fortiori* to mechanise these proofs, this turns out to be quite expensive in terms of the size of the relations one has to exhibit. Indeed, to prove two processes to be bisimilar, one has to exhibit a bisimulation relation that contains the whole "future" of the related processes.

Several techniques have been introduced to facilitate bisimulation proofs by reducing the size of the relation to be given. [San95b] presents a generalisation of these techniques, based on the notion of *progression of relations*. The clean mathematical presentation given in [San95b] allowed us to implement this theory quite straightforwardly in Coq, keeping the mechanised proofs remarkably close to the original ones. Therefore, we just state in a mathematical style the main results we prove, postponing the discussion of implementation issues to the proof of the main theorem of this Section, namely the up to context bisimulation Theorem 3.8.

We start by defining the notion of progression of relations:

Definition 3.2 (Progression of relations) *We say that a symmetrical relation $\mathcal{R}$ over processes progresses to a relation $\mathcal{S}$ (written $\mathcal{R} \rightarrow \mathcal{S}$) if, whenever $P\mathcal{R}Q$ and $P \stackrel{a}{\rightarrow} P'$, there exists a process Q' such that $Q \stackrel{a}{\rightarrow} Q'$ and $P'\mathcal{S}Q'$.*

Intuitively, $\mathcal{R} \rightarrow \mathcal{S}$ means that $\mathcal{S}$ contains the "future" of $\mathcal{R}$. From such a point of view, bisimulation can be given a new definition, in terms of progression:

Definition 3.3 (Bisimulation - new version) *A relation is a bisimulation if it progresses to itself.*

It is easy to prove that this definition is equivalent to Definition 3.1.

We now turn to the study of functions over relations (briefly called *functions*); we introduce two properties of such objects:

Definition 3.4 (Soundness, respectfulness)

- *A function $\mathcal{F}$ is* sound *if for any $\mathcal{R}$, $\mathcal{R} \rightarrow \mathcal{F}(\mathcal{R})$ implies $\mathcal{R} \subseteq \sim$.*
- *A function $\mathcal{F}$ is* respectful *if ($\mathcal{R} \subseteq \mathcal{S}$ and $\mathcal{R} \rightarrow \mathcal{S}$) implies ($\mathcal{F}(\mathcal{R}) \subseteq \mathcal{F}(\mathcal{S})$ and $\mathcal{F}(\mathcal{R}) \rightarrow \mathcal{F}(\mathcal{S})$).*

As proved below, respectfulness is a sufficient condition for soundness; moreover, it enjoys nice compositional properties, and will thus be convenient to use.

Theorem 3.5 (Soundness of respectful functions) *If a function $\mathcal{F}$ is respectful, then $\mathcal{F}$ is sound.*

The proof is easy, and basically reduces to build a bisimulation relation by successive iterations of $\mathcal{F}$ on a given relation $\mathcal{R}$. In the following, we shall work with respectful functions, that by Theorem 3.5 provide proof techniques to establish bisimilarities: given a respectful (hence sound) function $\mathcal{F}$, to prove that two processes P and Q are bisimilar, it is sufficient to exhibit a relation $\mathcal{R}$ such that $P\mathcal{R}Q$ and $\mathcal{R} \rightarrow \mathcal{F}(\mathcal{R})$. Intuitively, the rôle of $\mathcal{F}$ is to make $\mathcal{R}$ grow, by building its "future": if we find good respectful functions, we can hope to considerably reduce the size of $\mathcal{R}$, and as well to reduce the number of cases to consider in the proof of the progression relation (as will be shown in the following sections).

We have also implemented, following [San95b], some results that allow us to combine several respectful functions to obtain another respectful function. Due to lack of space, we do not describe this part of Sangiorgi's theory. Let us just mention a key result from it:

Lemma 3.6 (up to bisimilarity proof technique) *If $\mathcal{F}$ is respectful, then the function that associates $\sim \mathcal{F}(\mathcal{R}) \sim$ to $\mathcal{R}$ is also respectful (with $\sim \mathcal{S} \sim = \{(P, Q)$ s.t. $\exists P_1, Q_1.\ (P \sim P_1) \wedge (P_1 \mathcal{S} Q_1) \wedge (Q_1 \sim Q)\}$).*

3.2 Up to Context Proof Technique for the π-Calculus

We apply the above results to our implementation of π-calculus, in order to define the function that builds the closure under contexts of a relation, and prove its respectfulness. This will then give us a very useful technique to prove bisimilarity results, as shown in Section 4. Several questions arise when implementing this proof, that are related to the implementation choices described in Section 2. The main issue is the treatment of free names within the de Bruijn framework, as we attempt to compare processes: indeed, as seen above, transitions of processes may involve the application of operators on free names, like the *lift* function in rule PAR_l. These transformations, however, do not really have a strong significance, since they return another representation of the same process (in the non-de Bruijn version of PAR_l, process Q in the conclusion of the rule is the same as Q in the hypothesis), but rather do some kind of "administrative work" to keep the notation coherent. Therefore, if a pair (P, Q) of processes belongs to a given relation $\mathcal{R}$, there should be no reason for $(lift^k(P), lift^k(Q))$ not to be in $\mathcal{R}$. More generally, for any substitution f, injective on the free names of P and Q, $(f(P), f(Q))$ should belong to $\mathcal{R}$. We thus define a predicate over relations as follows:

```
Definition good : (pi->pi->Prop)->Prop :=
  [R: (relation pi)] (p,q:pi) (R p q) ->
  (l:l_name) (injective l) ->
  (le (max_nat (max_free_O p) (max_free_O q)) (l_length l)) ->
  (R (pi_subst_n p l O) (pi_subst_n q l O)).
```

The whole theory of progressions above is parametrised with the `good` predicate, since we work only with "good" relations over processes. Note that this predicate actually implements the respectfulness of the closure over injective substitutions (which is *proved* in [San95b]) as a built-in property of our implementation.

To describe function $_^{\mathcal{C}}$, that builds the closure under contexts of a relation (we shall write $\mathcal{R}^{\mathcal{C}}$), we must first define contexts. Here again, the syntax of our processes dictates the shape of the contexts we use: while in [San95b] Sangiorgi could restrict his study to *monadic* contexts (contexts with one occurrence of the hole), having recursive definitions in his calculus, the presence of the replication operator compells us to work with polyadic contexts, because of the shape of the *BANG* rule (see Table 1). We then define function $_^{\mathcal{C}}$ as follows:

Definition 3.7 (Closure under contexts function)
$\mathcal{C}(\mathcal{R}) = \cup_{C\ non-guarded}\{(C[P], C[Q]) : (P, Q) \in \mathcal{R}\} \bigcup$

$\cup_{C\ guarded}\{(C[P], C[Q]) : (P\sigma, Q\sigma) \in \mathcal{R}$, *for all substitutions* $\sigma\}$,
and in Coq:

```
Inductive Close [R:pi -> pi -> Prop] : pi -> pi -> Prop :=
  Clo_cons : (c:context)(p,q:pi)(close_cons c R p q) ->
    (Close R (c2pi c p) (c2pi c q)).
```

The guardness property for polyadic contexts is defined as follows: a context is *guarded* if some occurrence of the hole is under a prefix construct; if this condition does not hold, the context is said to be *unguarded*. Guarded contexts have a special rôle in π-calculus because they can modify the process that fills the hole, through the application of a substitution. In the Coq definition, `close_cons` is a function that returns the requirement on objects `R`, `p` and `q` corresponding to the guardness of its first argument (context `c`), according to Definition 3.7.

Actually, function $_^{\mathcal{C}}$ alone is not respectful in our setting. To obtain a respectful function, we must consider the transitive closure of $_^{\mathcal{C}}$, written $_^{\mathcal{C}^{\mathcal{T}}}$ (and implemented in Coq by a function named `T_Close`). Indeed, as we work with polyadic contexts, transitivity is needed for the parallel construct, in order to be able to use many times the inductive hypothesis. There is also a more tricky reason for which we have to consider the transitive closure of function $_^{\mathcal{C}}$; without being too technical, let us say that in the "input" case of the proof below, as the substitution propagates within the (possibly polyadic) context, we have to split in some way the polyadic context into several monadic contexts, i.e. perform an induction over the number of holes of our context, to establish the result we want to prove. This induction actually requires the transitivity property brought by the $_^{\mathcal{T}}$ function.

Theorem 3.8 (up to context bisimulation) *Function $_^{\mathcal{C}^{\mathcal{T}}}$ is respectful.*

Proof For the nontrivial part of this proof, we consider two relations $\mathcal{R}$ and $\mathcal{S}$ such that $\mathcal{R} \subseteq \mathcal{S}$ and $\mathcal{R} \rightarrow \mathcal{S}$, and prove $\mathcal{R}^{\mathcal{C}^{\mathcal{T}}} \rightarrow \mathcal{S}^{\mathcal{C}^{\mathcal{T}}}$. We therefore consider processes P, P' and Q and an action λ such that $P\,\mathcal{R}^{\mathcal{C}^{\mathcal{T}}}\,Q$ and $P \xrightarrow{\lambda} P'$. We first perform an elimination on the inductive function that builds the transitive closure. The transitivity case is straightforward; we thus suppose $P\,\mathcal{R}^{\mathcal{C}}\,Q$ (i.e. $C[P]\,\mathcal{R}\,C[Q]$ for some C), and we have to exhibit Q' such that $Q \xrightarrow{\lambda} Q'$ and $P'\,\mathcal{S}^{\mathcal{C}^{\mathcal{T}}}\,Q'$.

In [San95b], this proof is achieved by structural induction on the context C; we have checked that the original proof faithfully translates into the Coq system for the finite case (using monadic contexts). If we want to consider also non finite contexts, however, we have to eliminate the inductively defined proposition stating the transition relation $C[P] \xrightarrow{\lambda} P'$, otherwise an induction on C would generate circularity in the proof.

Due to lack of space, we just give as an example the shape of the Coq proof for the case where the transition relation is obtained by using the `comm_pl` rule above. The goal to prove is the following:

```
H : (good R)
H0 : (good S)
H1 : (r_incl pi R S)                          (R ⊆ S)
Hind : (q:pi)                                 (induction hypothesis)
        (close_cons c R p q)
         ->(Ex [q':pi]
               (commit (c2pi c q) lambda q')/\(T_Close S p' q'))
```

```
  Hclose : (close_cons (cPar c' c) R p q)   (induction hypothesis)
  H3 : (commit (c2pi c p) lambda p')        (induction hypothesis)
  ============================
   (Ex [q':pi]                              (goal to prove: "∃ q' ...")
        (commit (Par (c2pi c' q) (c2pi c q)) lambda q')
        /\(T_Close S
            (Par (lift_pi (c2pi c' p) 0 (bound_action lambda)) p') q'))
```

((c2pi c p) is the Coq translation of $C[P]$). The idea is to use the induction hypothesis Hind with process q the following way ((close_cons c R p q), which has to be proved in order to use Hind, is a direct consequence of Hclose):

```
Elim (Hind q).
Intros q0 Hfoo; Elim Hfoo; Clear Hfoo; Intros.
```

This generates the new hypotheses:

```
  H4 : (commit (c2pi c q) lambda q0)
  H5 : (T_Close S p' q0)
  ============================
```

We can now exhibit a process satisfying the conditions of the conclusion, with the Exists tactic:

```
Exists (Par (lift_pi (c2pi c' q) 0 (bound_action lambda)) q0); Split.
```

The left part of the conclusion is proved using the comm_pl rule:

```
Apply comm_pl; Trivial.
```

For the right part of the conclusion, we use the transitivity property, and some basic manipulations lead us to the following goal:

```
  H6 : (Close S (c2pi c' p) (c2pi c' q))
  ============================
   (Close S (lift_pi (c2pi c' p) 0 (bound_action lambda))
     (lift_pi (c2pi c' q) 0 (bound_action lambda)))
```

The proof of this subgoal is carried using hypothesis H0 : (good S), which allows us to remove the lift_pi operator from the conclusion of the goal, thanks to a lemma stating that this function can be represented by an injective substitution. We conclude this way the proof for this subcase, and work similarly for the other cases. We thus show respectfulness of the $_^{\mathcal{C}^T}$ function, which means correctness for the up to context bisimulation technique.

4 Classical Algebraic Theory of π-Calculus

Section 2 introduced the objects we study, and Section 3 was devoted to the definition of tools to manipulate these objects in order to state equivalence properties about them. This is actually all we need, at least from a conceptual point of view, to establish the main results of classical π-calculus theory.

Indeed, the bisimilarity results stated here are obtained in a uniform way: the method is to exhibit a relation $\mathcal{R}$, and to prove that $\mathcal{R} \rightarrow\sim (\mathcal{R}^{c^\tau}) \sim$. The definition of $\mathcal{R}$ follows the shape of the property we want to prove. The progression relation is proved with heavy use of Coq's `Inversion` tactic, that allows one to derive all possible hypotheses from an instance of a term belonging to an inductively defined type.

For some technical lemmas, we sometimes have to perform an induction over the term stating the transition of a process, even if it would be more natural to make a structural induction over the process itself, because of the shape of the *BANG* rule (as explained above). In some cases, a structural induction can be performed on the process *resulting from the transition*, a technique that also avoids the introduction of circularity in our proofs. It should be of interest to study how these restrictions on the shape of our proofs evolve when working with coinductive types [Gim96].

We first prove congruence, stated as follows:

Theorem 4.1 (Congruence) *$\sim$ is a congruence relation with respect to constructors $|$, ν and $!$. To obtain congruence for all operators of π-calculus, we must consider relation $\sim^c$, defined by: "$P \sim^c Q$ iff for all substitution σ, $P\sigma \sim Q\sigma$".*

We then turn to the bisimilarity properties called *structural equivalence* in [Mil91], or alternatively *π-calculus equivalence* in [AM94].

Theorem 4.2 (Structural equivalence) *We have, for all processes P, Q, R and names x, y:*

1. $P|\mathbf{0} \sim \mathbf{P}$;
2. $P|Q \sim Q|P$;
3. $P|(Q|R) \sim (P|Q)|R$;
4. $P|(\nu x)Q \sim (\nu x)(P|Q)$ *if* $x \notin fn(P)$ *(de Bruijn:* $P|\nu Q \sim \nu(lift^1(P)|Q)$*);*
5. $(\nu x)(\nu y)P \sim (\nu y)(\nu x)P$ *(de Bruijn:* $\nu\nu P \sim \nu\nu P_{\{0\leftrightarrow 1\}}$*.* $P_{\{0\leftrightarrow 1\}}$ *represents process P where indexes 0 and 1 are exchanged);*
6. $!P \sim !P|P$.

For lack of space, and because we always apply the same proof technique, we just give as an example the shape of the proof for relation 2 above. To prove $\forall P, Q.P|Q \sim Q|P$, without the up to context technique, one has to prove that relation

$$\mathcal{R} \;=\; \{(\,(\nu\mathbf{x})(P|Q), (\nu\mathbf{x})(Q|P)\,)\}$$

is a bisimulation (we have to add the restrictions for the case where a communication, which may involve name extrusion, occurs between P and Q). The rules we have to take in account when considering the transitions of processes related by $\mathcal{R}$ are thus PAR_l, PAR_r, $CLOSE_1$, $CLOSE_2$, RES and $OPEN$. With the technique we use, we can get rid of the restrictions (that in some way are brought along by the closure under contexts function), and we only have to consider rules PAR_l, PAR_r, $CLOSE_1$ and $CLOSE_2$. Notice that for this proof, work can be factorised by proving only a simulation instead of a bisimulation, because of symmetry (this also holds for relation 5, by remarking that $(P_{\{0\leftrightarrow 1\}})_{\{0\leftrightarrow 1\}} = P$).

We will not enter further the details of the proofs for Theorem 4.2. Let us just mention that their size and readability can be considerably different according to the shape of the bisimilarity property we want to prove. More precisely, it seems that the more the theorem involves names manipulations (especially properties about restriction), the bigger the amount of technical work concerned with the operators on de Bruijn indexes.

5 Towards Functions as Processes

The proof method that we used in the previous Section allows us to derive another important result of π-calculus theory, sometimes referred to as the "replication theorems". The "functions as processes" paradigm, namely the ability for π-calculus to encode (functional) sequential computation, is one of the many applications of these theorems. After proving the replication theorems, we discuss the verification of the operational correspondence for an encoding of lazy λ-calculus [Abr89] into π-calculus, as presented in [San95a].

5.1 The Replication Theorems

The results we prove here (still using, as a matter of fact, the method of Section 4) help to show the expressiveness of our implementation, or equivalently the power of Theorem 3.8. To state these results, we introduce a family of sets of processes, indexed by a name a, and defined as:

$$\mathcal{N}_a = \{P : a \text{ occurs in } P \text{ only in output subject position}\}$$

(the subject of a prefix is the name where the communication occurs). For the replication theorems, we will consider processes of the form $(\nu a)(\, P \mid !a(\mathbf{b}).Q\,)$, with $P, Q \in \mathcal{N}_a$. Roughly speaking, $\mathcal{N}_a$ is the set of processes that consider a as a name giving access to a resource (a request to the resource is performed by sending a message along channel a, with parameters $\mathbf{b}$, the resource being Q). We prove the following bisimilarity relations (taken as they appear in [San95b]; many other papers mention them, e.g. [San95a, Mil91]):

Theorem 5.1 (Replication theorems) *For all $R, P_1, P_2, \alpha.P \in \mathcal{N}_a$, we have:*

1. $(\nu a)(!a(\mathbf{b}).R \mid P \mid Q) \sim (\nu a)(!a(\mathbf{b}).R \mid P) \mid (\nu a)(!a(\mathbf{b}.R \mid Q)$;

2. $(\nu a)(!a(\mathbf{b}).R \mid !\alpha.P) \sim !(\nu a)(!a(\mathbf{b}).R \mid \alpha.P)$.

The up to context proof technique allows us to considerably simplify the proofs given in [Mil91], especially for the second result: while Milner had to consider relation

$$\mathcal{R} = \{(\,(\nu \mathbf{x})(\nu a)(!a(\mathbf{b}).R \mid !\alpha.P \mid Q), \\ (\nu \mathbf{x})(!(\nu a)(!a(\mathbf{b}).R \mid \alpha.P) \mid (\nu a)(!a(\mathbf{b}).R \mid Q))\,)\},$$

we can restrict ourselves to the relation suggested by the property, namely:

$$\mathcal{R}' \;=\; \{(\,(\nu a)(!a(\mathbf{b}).R \mid !\alpha.P), !(\nu a)(!a(\mathbf{b}).R \mid \alpha.P)\,)\}.$$

Thanks to a few lemmas that characterise the possible transitions of terms of the form $(P|!a(\mathbf{b}).Q)$, we get some quite simple and readable (although not small) proofs for these results.

5.2 λ-Calculus Encoding

At this point, an interesting subject we may want to consider is the study of the encoding of lazy λ-calculus into π-calculus. This can be seen as a benchmark for our π-calculus implementation, while being at the same time, of course, a step forward in the field of mechanisation of π-calculus theory (to our knowledge, no mechanisation of the proof of the functions as processes theorem exists by now). We have implemented the main lines of the proof of operational correspondence for the encoding, mostly following [San95a], but without giving a full proof for many technical results, and rather stating them as hypotheses. It has to be stressed that the replication theorems stated above are a key result to carry this proof, and have been fully verified. Actually, we think some work should be done to reformulate some parts of the proof in a way that would be more tractable for the task of mechanisation. We present below the main issues that arise as we try to implement the proof, stressing the notions that turned out to be awkward to handle in our Coq implementation.

To represent λ-terms, we start from Huet's work [Hue93], that also provides a de Bruijn notation for variables; the definition of the encoding supposes that the set of λ-calculus variables is a subset of π-calculus names. As we do not have a notion of subtyping in Coq, this cannot be directly formalised in our setting. Instead, the correspondence between variables in an encoded λ-term and their representation in the encoding has to be defined at meta-level, in order to guarantee the success of the proof of operational correspondence.

Another notion that is important in this proof is the *expansion* relation, a preorder over processes containing bisimilarity. Intuitively, a process P expands a process Q if it is liable to perform the same actions as Q modulo some extra τ-actions (in our case, we must actually prove that the encoding of a β-redex evolves to a process that *expands* the encoding of the reduced λ-term). The proof of [San95a] uses some quite technical reasoning to show that expansion is a congruence relation, taking advantage from the fact that processes resulting from

the encoding are asynchronous (i.e. processes for which there is no continuation after an output prefix). For asynchronous processes, we have a very strong result, namely that bisimulation (and expansion as well) is preserved by *any* kind of substitution (see for example [ACS96]). Instead of adapting the classical proof of this result (which would be rather tedious, as it turned out from a first attempt), we may want to see if the techniques presented in Section 3 can be adapted to expansion. They cannot be straightforwardly adapted, because some nice compositional properties of respectful functions would not be true; however, the task of providing a tractable method to prove expansion relations along the ideas of the theory of progressions seems to be quite interesting and promising.

6 Conclusion

We have presented an implementation of π-calculus, together with an account of classical results of π-calculus theory. Proofs for these results have been carried using the up to context technique, which has turned out to be efficient in performing clear and elegant proofs. Technical work, however, still represents the biggest part of our implementation, mainly due to the managing of De Bruijn indexes: indeed, as stressed above, the De Bruijn notation, while drastically simplifying work for bound names, requires accuracy in dealing with free names. Of our 800 proved lemmas, about 600 are concerned with operators on free names; the implementation consists of 70 files of Coq specifications and proofs in Coq V6.1 [BBC+96a]. Some proofs, especially those for the tedious results involving De Bruijn indexes, are somewhat "rough", and could probably be reformulated in a uniform way, instead of deriving *ad hoc* results along the way. A few files contain as well proof texts that were automatically generated by CtCoq [BBC+96b], which is an user interface that allows one to perform derivations with the *proof-by-pointing* technique [BKT94]. This technique has turned out to be helpful for general theorems, in that the user has facilities to build a proof by controlling its general shape without entering too much into the details at first. For technical lemmas, however, a direct interaction with the Coq system is often more efficient, mostly since the results one has to prove offer no conceptual difficulty, and the derivation is only some kind of an administrative work to do.

The work that can be considered as the closest to ours has been done within the HOL system. M. Nesi has defined CCS and a modal logic for this algebra in [Nes92]; regarding π-calculus, the original formalisation by T. Melham [Mel94] has been used as a starting point for the work of O. Aït-Mohamed, which consisted in proving equivalence [AM94], and building a system to prove bisimilarities for processes interactively [AM95]. Differences with our implementation arise at the level of technical issues, as well as at the level of the general presentation. Regarding the calculus itself and the way it is implemented, our choice of the De Bruijn notation is a key issue (while in [Nes92], assumptions are made about avoiding name clashes, in [Mel94] and [AM95], renaming has to be done in order to preserve freshness for bound names). About the formal representation of bound and free names, let us mention two alternative solutions to the

de Bruijn representation: [MG96] presents and axiomatisation of λ-terms up to alpha conversion, proves it sound with respect to de Bruijn terms, and suggests as one of the possible applications of this presentation the proof of semantical properties of π-terms. In [KP93], bound and free variables are distinguished syntactically, which allows one to simplify the definition of substitution, and hence the proofs. Another point of view on substitution is presented in [Sto88], where a definition by structural recursion is presented; in this work, bound variables are systematically renamed in a substitution, while in our de Bruijn setting we always rename free variables.

Miller has described in [Mil92] another interesting point of view onto π-calculus specification by the means of a translation into linear logic that allows one to define some testing equivalences for a subclass of processes.

Finally, let us mention as well Giménez work in Coq on coinductive types [Gim96], which includes an implementation of CBS; we only used coinductive types in a toy example, to prove that our notion of bisimulation coincides with a formulation in terms of a greatest fix point of a combinator (using the fact that a coinductive definition implicitly implements a greatest fixpoint).

Future work on this implementation has already been partially discussed in the previous section, regarding the full formalisation of the functions as processes theorem. Another interesting direction could be to take advantage of Coq's extraction facility in order to build a system for π-calculus verification, along the ideas of Aït-Mohamed's PIC system [AM95], but exploiting the greater generality of our theorems (especially for proofs involving the replication construct). Reflection [Bou96] could also be used to build certified tactics to manipulate our π-terms.

Acknowledgments We thank Davide Sangiorgi and René Lalement for enlightening discussions about this work, as well as the anonymous referees for some very useful comments.

References

[Abr89] S. Abramsky. The lazy lambda calculus. *Research Topics in Functional Programming*, pages 65–116, 1989.

[ACS96] R. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous π-calculus. In *CONCUR '96*, number 1119 in LNCS, 1996.

[AM94] O. Ait-Mohamed. Vérification de l'équivalence du π-calcul dans HOL. Rapport de recherche 2412, INRIA-Lorraine, November 1994. (In French).

[AM95] O. Ait-Mohamed. PIC: A proof checker for the π-calculus in Higher Order Logic. Technical report, INRIA-Lorraine, 1995.

[Amb91] Simon J. Ambler. A de Bruijn notation for the π-calculus. Technical Report 569, Dept. of Computer Science, Queen Mary and Westfield College, London, May 1991.

[BBC+96a] B. Barras, S. Boutin, C. Cornes, J. Courant, JC. Filliâtre, E. Gimenez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, and B. Werner. *The Coq Proof Assistant Reference Manual.* Projet Coq, INRIA Rocquencourt / CNRS - ENS Lyon, 1996.

[BBC+96b] J. Bertot, Y. Bertot, Y. Coscoy, H. Goguen, and F. Montagnac. *User Guide to the CtCoq Proof Environment*. INRIA, Sophia-Antipolis, February 1996.

[BKT94] Y. Bertot, G. Kahn, and L. Théry. Proof by pointing. In *Proceedings of STACS*, LNCS, Sendai (Japan), April 1994.

[Bou96] S. Boutin. Using reflection to build efficient and certified decision procedures. Unpublished - available at `http://pauillac.inria.fr/~boutin/publis.html`, 1996.

[dB72] N.G. de Bruijn. Lambda Calculus Notation with Nameless Dummies: a Tool for Automatic Formula Manipulation, with Application to the Curch-Rosser Theorem. In *Indagationes Mathematicae*, volume 34, pages 381–392. 1972.

[Gim96] E. Giménez. *Un calcul de constructions infinies et son application à la vérification de systèmes communicants*. PhD thesis, E.N.S. Lyon, 1996.

[Hir97] D. Hirschkoff. Up to context proofs for the π-calculus in the Coq system. Technical Report 97-82, CERMICS, Noisy-le-Grand, January 1997.

[Hue93] G. Huet. Residual theory in λ-calculus: A formal development. Technical Report 2009, INRIA, Rocquencourt - France, Août 1993.

[KP93] J. Mc Kinna and R. Pollack. Pure Type Systems Formalized. In *Proceedings of TLCA'93*, volume 664 of *LNCS*. Springer Verlag, 1993.

[Mel94] T. F. Melham. A mechanized theory of the π-calculus in HOL. *Nordic Journal of Computing*, 1(1):50–76, 1994.

[MG96] T. F. Melham and A. Gordon. Five Axioms of Alpha-Conversion. In *Proceedings of TPHOL'96*, volume 1125 of *LNCS*. Springer Verlag, 1996.

[Mil91] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS-91-180, LFCS, Dept. of Computer Science, Un. of Edinburgh, 1991.

[Mil92] D. Miller. The π-calculus as a theory in linear logic: Preliminary results. In E. Lamma and P. Mello, editors, *Proceedings of the Workshop on Extensions to Logic Programming*, volume 660 of *Lecture Notes in Computer Science*, pages 242–265. Springer-Verlag, 1992.

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

[Nes92] M. Nesi. A formalisation of the CCS process algebra in higher order logic. Technical Report 278, Computer Laboratory, University of Cambridge, December 1992.

[NP96] U. Nestmann and B. C. Pierce. Decoding choice encodings. In *Proceedings of CONCUR '96*, number 1119, August 1996.

[Pie95] B.C. Pierce. *Programming in the Pi-Calculus (Tutorial Notes)*. Computer Laboratory, Cambridge - UK, November 1995.

[San95a] D. Sangiorgi. Lazy functions and mobile processes. Technical Report RR-2515, INRIA - Sophia Antipolis, 1995.

[San95b] Davide Sangiorgi. On the bisimulation proof method. Revised version of Technical Report ECS–LFCS–94–299, University of Edinburgh, 1994. An extended abstract can be found in Proc. of MFCS'95, LNCS 969, 1995.

[Sto88] A. Stoughton. Substitution revisited. *Theoretical Computer Science*, 59:317–325, 1988.

[Wer94] B. Werner. *Une théorie des constructions inductives*. Thèse de doctorat, Université Paris 7, 1994.

This article was processed using the LaTeX macro package with LLNCS style