Bisimulation proofs for the π-calculus in the Calculus of Constructions

Daniel Hirschkoff

avril 1996

N^o 96-62

Bisimulation proofs for the π-calculus in the Calculus of Constructions

Daniel Hirschkoff

Résumé

Nous présentons une implémentation dans le système Coq des techniques de preuves de bisimulation exposées par Davide Sangiorgi dans [San94]. Coq [CCF+96] est un logiciel d'aide à la preuve développé à l'INRIA et à l'École Normale Supérieure de Lyon. Nous décrivons la théorie des progressions de relations sur un ensemble de processus quelconque, que nous appliquons ensuite à l'implantation d'un mini π-calcul polyadique fini; nous nous intéressons en particulier à la clôture d'une relation pour une famille de contextes. Les techniques implantées permettent de simplifier les preuves de bisimulation entre termes.

Abstract

We present an implementation of the bisimulation proof techniques described by Davide Sangiorgi in [San94]. The system we use is the Coq Proof Assistant [CCF+96], a theorem prover developped at INRIA and at the École Normale Supérieure de Lyon. We firstly implement the theory about progressions of relations on a set of processes, and then specialise it with our implementation of a finite polyadic π-calculus; we consider in full detail the particular case of the closure under contexts of a relation. This gives a toolkit to make bisimulation proofs much shorter than the usual ones.

Introduction

We describe here what could be considered as a "mechanisation" of the theory exposed in [San94]. This work is in relation with [Hue93], that has been a source of inspiration, and an example of a previous experiment implementing in the Coq system an aspect of theoretical computer science.

Such a process often gives an opportunity to understand accurately the constructions that are implemented, as well as a good test for the target system, in terms of expressiveness and efficiency. With regards to formalisation of abstract models for computation or concurrency such as the λ-calculus or the π-calculus, it can be considered as a dual approach to the actual building of a system based on these calculi (as ML is for the λ-calculus, as Pict [Pie95] is for the π-calculus, ... and as Coq itself is for the Calculus of Constructions!), not using the theory as a 'meta' paradigm, but rather as a material to work on.

The subject matter of the implementation is a part of the theory developped in [San94], that we call here the *theory of progressions*. It defines a new kind of methods for proving bisimulation relations; the actual formalisation is quite general, and as such it was interesting to see how an implementation of general constructions can be specialised in order to focus onto one of the applications that are described in the paper, namely π-calculus.

This paper should be self-contained; nevertheless, some familiarity with the π-calculus and with traditional functional-style notations (in order to read Coq text) would be preferable. In the first section, we describe briefly the Coq proof system and the notions relative to Sangiorgi's theory and to the π-calculus that we shall use. We then discuss the theory of progressions and its implementation more precisely, as well as the definition of a closure under contexts function in the general case. The third section is devoted to the encoding of the syntax and the semantics for the specific π-calculus we use, and we then describe applications of Sangiorgi's techniques to π-terms: after adapting our definition of the closure under contexts function to the π-calculus in the fourth section, we use it in the fifth section to prove the uniqueness of solutions for equations in the π-calculus. We finally conclude and discuss future work.

1 Preliminaries

1.1 The Coq system

The present work has been done using the Coq Proof Assistant [CCF+96], a proof checker based on the Calculus of Inductive Constructions (CIC). This system supports the definition of objects in the Calculus of Inductive Constructions (a typed λ-calculus with inductive definitions), and can interactively generate proofs about these objects through the use of tactics implementing backward reasoning. The excerpts of the implementation that are stated in this paper are actually all written in Gallina, the specification language of the Coq system. In order to make these statements understandable, let us briefly sketch the syntax of the most important Gallina constructions:

- *abstraction* is written with square brackets, application as usual with parentheses, implication is written with the symbol `->`, while dependent product, that can be interpreted as universal quantification, is written with brackets as well. Let us look at an example to illustrate that: the term `(n:nat)(([x:nat]x) n) = n` denotes the proposition $\forall n : \texttt{nat}\ (\lambda x.x\ n) = n$ (`nat` is the type for natural numbers).

- An important feature of Gallina is the opportunity to make inductive definitions, such as:

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

  This definition reads: "`nat` *is a new inductive type of sort* `Set` *whose constructors are* `O` *of type* `nat` *(a constant) and* `S` *of type* `nat -> nat` *(the successor function, taking a natural number and returning as a result a natural number)*".

When necessary, a brief comment on the form of a proof in Coq will be given, but no quotation will be made of the actual proof scripts. In the statement of Coq objects, we will distinguish between statements that can actually be found in Sangiorgi's paper and lemmas that were introduced in order to carry the proof: objects from the original text will be marked by a * and something like **(2.10)**, indicating their numbering in the source text.

The reader interested in a more detailed approach to the Coq Proof Assistant should refer to [HKPM95].

We now turn to the topics that make the subject of the implementation.

1.2 Bisimulation proof techniques

A common notion for the formulation of equivalence between processes is the ability for two equivalent processes to simulate eachother actions, called *bisimulation.* In [San94], Sangiorgi presents a nice generalisation of the techniques that are traditionally used to achieve bisimulation proofs. In most of the cases, one has to explicitely exhibit an infinite relation containing the two processes to be proven equivalent, and then prove that it is a bisimilarity. The aim of Sangiorgi's work is to reduce drastically the size of the invoked relations, in order to consider finite sets.

To understand intuitively how to achieve this, let us have a look at the definition of a bisimulation: *we say that P and Q are bisimilar, written $P \sim Q$, if whenever P performs an action α to become P', there exists a Q' such that Q can perform α to become Q', and $P' \sim Q'$.* The idea of Sangiorgi is that if we look at this statement from a formal (nearly syntactic) point of view, we can see that the $\sim$ symbol occurs twice, once in the thesis, and once in the conclusion. Intuitively, the non-finiteness comes from this circularity, that has thus to be broken if we want to work with finite sets. This is what is done in [San94], by considering *progressions* from relations to relations, as well as functions taking a relation as an argument and returning a relation. Some properties of such functions with respect to progressions can provide bisimulation proof techniques, confining the "non-finiteness" of the actual bisimulation relation in these functions, that progressively build an infinite set (the bisimilarity) starting from a finite one.

This is of course only a vague sketch of the ideas presented in the original paper, as the exact formulation of the theory will be presented, along with its implementation in Coq, in the next section.

1.3 The π-calculus, syntax and semantics

Syntax We consider in this paper a finite, polyadic, mini π-calculus. We take as a reference for the "mini" π-calculus the calculus that is used by Milner in [Mil92]. This is a calculus with no sum (+) construction, which is not really a problem since this operator can be encoded into the mini π-calculus. There are other calculi inspired by the original full π-calculus that, with a simpler syntax, still have its expressivity, thus candidating for being a "minimal" representation of concurrency: we can cite Boudol's asynchronous mini π-calculus[Bou92], and, more recently, the join calculus [FG].

In our setting, we get rid of the replication operator (commonly written !) as well, mostly for the sake of simplicity. However, the results presented here would probably have been preserved if we had considered a "non-finite" π-calculus, since we don't really "execute" the processes in the proofs, thus preventing the infinite behaviour of a replicated term to have impact on the success of our proofs.

Our calculus is polyadic, in the sense that a communication on a given channel involves a list of names rather than only one. Polyadic π-calculus is described in [Mil91], which is a strong inspiration for this work, but we do not implement the three process "species" defined in Milner's paper (normal processes, processes, and agents), for two reasons essentially: from the technical point of view the definition in Coq of three mutually recursive types would have been very tedious to handle, and from the theoretical point of view this distinction relies heavily on the task of making a clean axiomatisation of the calculus (what is called in the paper *Strong Ground Equivalence*), thus taking the sum as an essential operator to build normal forms for processes; since we renounce to this operator here, the distinction is not really relevant anymore.

Processes of our calculus can thus be described as:

$$P \;\equiv\; 0 \mid (\nu x)\, P \mid P \,|\, Q \mid x.(\lambda \vec{y})\, P \mid \bar{x}.[\vec{y}]\, P$$

For a process P, we define the sets of its bound names $bn(P)$ by saying that constructors ν and λ act as binders, free names $fn(P)$ and names $n(P) = bn(P) \cup fn(P)$.

Semantics There are many ways to formulate the semantics for the π-calculus in the litterature, giving different flavours to it, such as *late, early* [MPW92], *barbed* [MS], and *open* [San93]. In [San94], to apply the theory of progressions to the π-calculus, Sangiorgi works with an early bisimulation; we have chosen to stick to this choice in order to keep the Coq proofs as close as possible to the original ones. The difference here comes from the fact that we work in a polyadic setting, so the actions contain more informations than in the monadic case.

Actions Actions have the form:

$$\begin{array}{lll} Act & \equiv & x(\vec{y}) \mid (\nu \vec{z})\bar{x}[\vec{y}]\;^{*} \mid \tau \\ & & ^{*}\textit{side condition}: \;\vec{z} \subseteq \vec{y} \end{array}$$

`INP`	$x.\,(\lambda\vec{y})\,P \xrightarrow{x(\vec{z})} P\,\{\vec{y} := \vec{z}\}$	
`OUT`	$\bar{x}.\,[\vec{y}]\,P \xrightarrow{\bar{x}[\vec{y}]} P$	
`OPEN`	$\dfrac{P \xrightarrow{(\nu\vec{z})\bar{x}.[\vec{y}]} P'}{(\nu t)P \xrightarrow{(\nu\, t::\vec{z})\bar{x}.[\vec{y}]} P'}$	$t \in \vec{y},\, t \neq x$
`CLOSE`	$\dfrac{X \xrightarrow{a(\vec{l})} P \qquad Y \xrightarrow{(\nu\vec{k})\bar{a}[\vec{l}]} Q}{X\,\|\,Y \xrightarrow{\tau} (\nu\vec{k})(P\,\|\,Q)}$	$\vec{k} \cap fn(X) = \emptyset$
`RES`	$\dfrac{P \xrightarrow{\mu} P'}{(\nu x)P \xrightarrow{\mu} (\nu x)P'}$	$x \notin n(\mu)$
`PAR`	$\dfrac{P \xrightarrow{\mu} P'}{P\|Q \xrightarrow{\mu} P'\|Q}$	$bn(\mu) \cap fn(Q) = \emptyset$

Table 1: Transition rules

This definition reads: *"an action is either an input, or a bound output, or the silent action τ"*. Note that an output action can be unbound if the list of transmitted new names (here $\vec{z}$) is empty. The meaning of the $(\nu\vec{z})\bar{x}[\vec{y}]$ action is an output along channel x of the list $\vec{y}$ of names, among which those that occur in $\vec{z}$ are freshly created (in technical terms, we say that the names in $\vec{z}$ are involved in *name extrusions*).

We define free and bound names (respectively $fn(\mu)$ and $bn(\mu)$), as well as the set of names $n(\mu)$ of an action μ.

Transitions The transition relation on processes we use is defined by the rules of table 1; they correspond to the classical formulation of *early* transition semantics [MPW92]. The notation $P\,\{\vec{y} := \vec{z}\}$ is used to represent the substitution of $\vec{z}$ for $\vec{y}$ in process P; note also that the symmetrical versions for `PAR` and `CLOSE` rules are omitted.

We can now proceed to examine the actual implementation of the notions exposed so far, and the results that have been proved with them.

2 Theory of progressions

The first section of [San94], entitled *"Progressions and respectful functions"*, introduces the basic notions in order to reason about bisimilarity in a general setting.

2.1 Progressions and bisimilarity

Sangiorgi considers transition systems of the form $(\mathcal{P}r, Act, \longrightarrow)$, where $\mathcal{P}r$ is a domain (the set of processes), Act a set of *actions* and $\longrightarrow$ a transition relation included in $\mathcal{P}r \times Act \times \mathcal{P}r$ (we note for example $P \xrightarrow{\mu} Q$). This translates in Coq as:

```
Variable Pr : Set.
Variable Act : Set.
Variable trans : Pr -> Act -> Pr -> Prop.
```

In the following, we are reasoning about relations, defined as follows:

```
Definition relation : Type := Pr -> Pr -> Prop.
```

We then proceed to define the first notions, namely progressions, bisimulations and bisimilarity:

* **Definition 2.1 (progression (2.1))** *Given two relations $\mathcal{R}$ and $\mathcal{S}$, we say that $\mathcal{R}$ progresses to $\mathcal{S}$, written $\mathcal{R} \longrightarrow \mathcal{S}$, if $P\mathcal{R}Q$ implies:*

1. *whenever $P \xrightarrow{\mu} P'$, there is Q' s.t. $Q \xrightarrow{\mu} Q'$ and $P'\ \mathcal{S}\ Q'$;*
2. *the converse, i.e., whenever $Q \xrightarrow{\mu} Q'$, there is P' s.t. $P \xrightarrow{\mu} P'$ and $P'\ \mathcal{S}\ Q'$.*

* **Definition 2.2 (bisimulation, bisimilarity (2.2, 2.3))** *$\mathcal{R}$ is a bisimulation relation if $\mathcal{R}$ progresses to itself, i.e. $\mathcal{R} \longrightarrow \mathcal{R}$ holds; two processes P and Q are bisimilar, written $P \sim Q$ if $P\mathcal{R}Q$ holds for some bisimulation relation $\mathcal{R}$.*

These definitions translate in Coq as:

```
Definition progress : relation -> relation -> Prop :=
  [R,S:relation]
  (p,q:Pr)(R p q) ->
    (
      ((p':Pr)(mu:Act) (trans p mu p') ->
        (Ex [q':Pr] ((trans q mu q') /\ (S p' q'))))
      /\
      ((q':Pr)(mu:Act) (trans q mu q') ->
        (Ex [p':Pr] ((trans p mu p') /\ (S p' q'))))
   ).

Definition bisimulation : relation -> Prop :=
  [R:relation] (progress R R).

Inductive bisimilar : Pr -> Pr -> Prop :=
  bisim_pr : (p,q:Pr)(R:relation)(bisimulation R) -> (R p q)
    -> (bisimilar p q).
```

The latter object, namely the `bisimilar` relation, is defined inductively; this is the natural way to implement the *"there exists"* phrase in its mathematical definition: to build an object of type `(bisimlar p q)`, we must apply constructor `bisim_pr` to a specific relation `R`.

2.2 Functions

To build the machinery we need for proving bisimilarities between processes, we consider what Sangiorgi calls *first-order functions*, i.e. functions on relations, briefly called *functions*. We first define two properties on functions:

* **Definition 2.3 (soundness (2.4))** *A function $\mathcal{F}$ is sound if, for any $\mathcal{R}$, $\mathcal{R} \longrightarrow \mathcal{F}(\mathcal{R})$ implies $\mathcal{R} \subseteq \sim$.*

* **Definition 2.4 (respectfulness (2.5))** *A function $\mathcal{F}$ is respectful if whenever $\mathcal{R} \subseteq \mathcal{S}$ and $\mathcal{R} \longrightarrow \mathcal{S}$ holds, then $\mathcal{F}(\mathcal{R}) \subseteq \mathcal{F}(\mathcal{S})$ and $\mathcal{F}(\mathcal{R}) \longrightarrow \mathcal{F}(\mathcal{S})$ also holds.*

and in Coq:

```
Definition r_incl : relation -> relation -> Prop :=
  [R,S:relation](p,q:Pr) (R p q) -> (S p q).

Definition sound : (relation -> relation) -> Prop :=
  [F:relation->relation](R:relation) (progress R (F R)) ->
    (r_incl R bisimilar).

Definition respectful : (relation -> relation) -> Prop :=
  [F:relation->relation](R,S:relation)
  (r_incl R S) -> (progress R S) ->
  ((r_incl (F R) (F S)) /\ (progress (F R) (F S))).
```

`r_incl` represents of course the inclusion relation for elements of type `relation`. We can now state our first result, namely that respectfulness implies soundness; this gives a first proof technique for proving bisimilarities. To achieve this, we need two lemmas (from the original text by Sangiorgi):

* **Lemma 2.5 (2.8)** *Let $\mathcal{R} = \bigcup_{i \in I} \mathcal{R}_i$ and suppose for all $i \in I$ there is $j \in I$ s.t. $\mathcal{R}_i \longrightarrow \mathcal{R}_j$ holds. Then $\mathcal{R}$ is a bisimulation relation.*

* **Lemma 2.6 (2.9)**

1. *If, for some $i \in I$, $\mathcal{S} \longrightarrow \mathcal{R}_i$, then also $\mathcal{S} \longrightarrow (\bigcup_{i \in I} \mathcal{R}_i)$;*
2. *If, for all $i \in I$, $\mathcal{R}_i \longrightarrow \mathcal{S}$, then also $(\bigcup_{i \in I} \mathcal{R}_i) \longrightarrow \mathcal{S}$.*

To implement these lemmas in Coq, we have to define the union of relations indexed by a countable set I; we have chosen to consider the most general case where we have a sequence of relations (that we call `suite` here), and we build the infinite union of the members of this sequence. The finite case would have required a more complicated machinery in Coq, if we had renounced to build an infinite sequence from a finite number of relations (taking for example for $i \geq N$ $\mathcal{R}_i = \mathcal{R}_N$ for a given N).

```
Variable suite : nat -> relation.

Definition union : (nat->relation) -> relation :=
  [seq:nat->relation][P,Q:Pr](Ex [n:nat] ((seq n) P Q)).
```

We can remark as above that a classical way to implement an infinite set in Coq is to make an inductive definition. We use here the `Ex` construction, to stay close to a mathematical text, but this hides actually the inductive type `ex`, so we recover in fact an inductive object.

We keep in the Coq implementation the lemma numbering from the original paper, in order to distinguish lemmas of this origin from technical lemmas introduced especially for the proof. The proof of the lemmas are quite trivial and involve only basic manipulations on hypotheses and existential quantifiers. Their statements in Coq are:

```
Lemma lemma_2_8 : ((i:nat) (Ex [j:nat] (progress (suite i) (suite j))))
  -> (bisimulation (union suite)).

Lemma lemma_2_9_1 : (S:relation)(i:nat) (progress S (suite i))
  -> (progress S (union suite)).

Lemma lemma_2_9_2 : (S:relation) ((i:nat) (progress (suite i) S))
  -> (progress (union suite) S).
```

To stick to the text of the paper, we prove as well a little corollary:

* **Corollary 2.7 (2.10)** *If for all $i \in I$ there is $j \in J$ s.t. $\mathcal{R}_i \longrightarrow \mathcal{S}_j$ holds, then also $(\bigcup_{i\in I} \mathcal{R}_i) \longrightarrow (\bigcup_{j\in J} \mathcal{S}_j)$.*

This lemma requires the declaration of another sequence of relations to represent $(\mathcal{S}_j)_{j\in J}$.

```
Variable suite2: nat -> relation.

Lemma corollary_2_10 :
  ((i:nat) (Ex [j:nat] (progress (suite i) (suite2 j))))
  -> (progress (union suite) (union suite2)).
```

We now have our first theorem:

* **Theorem 2.8 (soundness of respectful functions (2.11))** *If $\mathcal{F}$ is respectful, then $\mathcal{F}$ is sound.*

```
Theorem theorem_2_11 :
  (F:relation -> relation) (respectful F) -> (sound F).
```

Proof To prove this theorem, we consider a relation $\mathcal{R}$ such that $\mathcal{R} \longrightarrow \mathcal{F}(\mathcal{R})$ holds, and we show that $\mathcal{R} \subseteq \sim$; for this purpose, we define a relation sequence $\mathcal{R}_n$ the following way:

$$\begin{aligned} \mathcal{R}_0 &= \mathcal{R} \\ \mathcal{R}_{n+1} &= \mathcal{F}(\mathcal{R}_n) \cup \mathcal{R}_n \end{aligned}$$

This is implemented by the `F_suite` function in Coq.

```
Definition Union : relation -> relation -> relation :=
  [R,S:relation][P,Q:Pr](R P Q) \/ (S P Q).

Fixpoint F_suite [F:relation->relation;R:relation;n:nat] : relation :=
  <relation>Case n of
    R
    [n':nat](Union (F (F_suite F R n')) (F_suite F R n'))
  end.
```

We use a *fact* to achieve the proof of our theorem:

* **Fact 2.9** *For all $n \geq 0$, it holds that*

1. $\mathcal{R}_n \subseteq \mathcal{R}_{n+1}$

2. $\mathcal{R}_n \longrightarrow \mathcal{R}_{n+1}$

and in Coq:

```
Fact fact_1 : (F:relation -> relation)(R:relation)
  (n:nat) (r_incl (F_suite F R n) (F_suite F R (S n))).

Fact fact_2 : (F:relation->relation)(respectful F) ->
  (R:relation) (progress R (F R)) ->
  (n:nat) (progress (F_suite F R n) (F_suite F R (S n))).
```

The original proofs of these results easily translate in Coq; we then prove the theorem by showing that $\bigcup_n \mathcal{R}_n$ (denoted by the term `(union (F_suite F R))`) is a bisimulation relation.

This theorem gives us a general technique for proving bisimulation relations: if we want to prove bisimilarity between two processes p and q, it is sufficent to consider a "small" relation $\mathcal{R}$ containing (p, q), and a respectful function $\mathcal{F}$. By showing that $\mathcal{R}$ progresses to $\mathcal{F}(\mathcal{R})$, we get the bisimilarity between processes p and q, as expressed by the `tech` lemma below.

```
Lemma tech : (p,q:Pr) (R:relation) (R p q) ->
  (F:relation -> relation)
  (respectful F) -> (progress R (F R)) ->
    (bisimilar p q).
```

An example of the application of such a machinery will be given for the π-calculus.

2.3 Constructors

The next step is to exhibit some respectful functions, and even to *build* some of them. To achieve this, Sangiorgi introduces the notion of *constructors*, i.e. applications that take functions as argument and return a function (a "function" being here a function from relations to relations), or in other terms what could be considered *second-order functions.*

Definition 2.10 (Respectfulness of a constructor) *A constructor is respectful if whenever its first-order function arguments are respectful, then also the first-order function result is respectful.*

This translates straightforwardly in Coq for binary constructors:

```
Definition constructor : Type :=
  (relation -> relation) -> (relation -> relation)
  -> (relation -> relation).

Definition respectful_constr : constructor -> Prop :=
   [c:constructor]
  (f,g:relation -> relation) (respectful f)
     -> (respectful g) -> (respectful (c f g)).
```

We then define a few objects that can be viewed as a "toolkit" in order to build respectful functions; functions $\mathcal{I}$ and $\mathcal{U}$ are defined as:

$$\begin{aligned} \mathcal{I}(\mathcal{R}) &= \mathcal{R} \\ \mathcal{U}(\mathcal{R}) &= \sim \end{aligned}$$

* **Lemma 2.11 (identity and constant-to-$\sim$ functions (2.13))** *The identity function $\mathcal{I}$ and the constant-to-$\sim$ function $\mathcal{U}$ are respectful.*

This is still easy to implement in Coq:

```
Definition Ident : relation -> relation := [r:relation]r.

Definition U : relation -> relation := [_:relation]bisimilar.

Lemma lemma_2_13_1 : (respectful Ident).

Lemma lemma_2_13_2 : (respectful U).
```

To combine these functions, we define three constructors, namely *composition* ($\circ$), *union* ($\cup$) and *chaining* ($\frown$).

$$\begin{aligned} (\mathcal{G} \circ \mathcal{F})(\mathcal{R}) &= \mathcal{G}(\mathcal{F}(\mathcal{R})) \\ (\cup_{i \in I} \mathcal{F}_i)(\mathcal{R}) &= \cup_{i \in I}(\mathcal{F}_i(\mathcal{R})) \\ (\mathcal{G} \frown \mathcal{R})(\mathcal{R}) &= \mathcal{G}(\mathcal{R})\mathcal{F}(\mathcal{R}) \end{aligned}$$

We define $\mathcal{G}(\mathcal{R})\mathcal{F}(\mathcal{R}) = \{(P, P'); \text{for some } P'',\ (P, P'') \in \mathcal{G}(\mathcal{R}) \text{ and } (P'', P') \in \mathcal{F}(\mathcal{R})\}$.

* **Lemma 2.12 (composition, (2.14))** *Composition is a respectful constructor.*

* **Lemma 2.13 (union, (2.15))** *Union is a respectful constructor.*

* **Lemma 2.14 (chaining, (2.16))** *Chaining is a respectful constructor.*

We give two definitions in Coq for the union constructor, `union` (to define the union of two functions) and `union_inf` (to define the union of a sequence of functions). We prove respectfulness only in the finite case, since when in the following an infinite union of functions is considered, we prefer to build it "by hand" in Coq, instead of using the `union_inf` constructor, in order to simplify proofs. The `union_inf` constructor is actually never used in our implementation.

```
Definition composition : constructor :=
  [F,G:relation->relation]
  [R:relation](F (G R)).

Definition union : constructor :=
  [F,G:relation->relation]
  [R:relation][P,Q:Pr]((F R) P Q) \/ ((G R) P Q).

Definition union_inf
  : (nat -> (relation -> relation)) -> relation -> relation :=
  [Fi:nat -> (relation -> relation)][R:relation][P,Q:Pr]
  (Ex [i:nat] ((Fi i R) P Q)).

Definition chaining : constructor :=
  [F,G:relation->relation][R:relation]
  [P,Q:Pr](Ex [P':Pr] ((F R) P P') /\ ((G R) P' Q)).

Lemma lemma_2_14 : (respectful_constr composition).

Lemma lemma_2_15 : (respectful_constr union).

Lemma lemma_2_16 : (respectful_constr chaining).
```

The proofs of the lemmas follow the original text with no particular difficulty; we basically just "unfold" the definitions of the constructors and of the respectfulness property, and the results become straightforward.

We can now build some more complicated respectful functions:

$$\begin{aligned}
\mathcal{D}_n &= \mathcal{I}^\frown \ldots {}^\frown \mathcal{I}, n \text{ times (n} > 0) \\
\mathcal{B} &= \mathcal{U}^\frown \mathcal{I}^\frown \mathcal{U} \\
\mathcal{T} &= \cup_{n>0} \mathcal{D}_n
\end{aligned}$$

$\mathcal{B}$ is the classical *bisimulation up-to*$\sim$, as in Milner's book [Mil89][1], while $\mathcal{T}(\mathcal{R})$ is the transitive closure of a relation $\mathcal{R}$. As stressed above, we directly define $\mathcal{T}$ in Coq instead of using the union_inf constructor.

```
Fixpoint Dn [n:nat] : relation -> relation :=
  <relation -> relation>Case n of
    Ident
    [n':nat](chaining Ident (Dn n'))
  end.

Definition B : relation -> relation :=
  (chaining U (chaining Ident U)).

Definition T : relation -> relation :=
  [R:relation][P,Q:Pr] (Ex [n:nat] ((Dn n R) P Q)).
```

The fact that T represents the transitive closure of a relation is not obvious on the above definition; this notion would "classically" be implemented in Coq with an inductive definition, as:

```
Inductive T' [R:relation] : relation :=
  T'_R : (P,Q:Pr) (R P Q) -> (T' R P Q)
| T'_t : (P,Q,S:Pr) (T' R P Q) -> (T' R Q S) -> (T' R P S).
```

Constructor T'_R "says" that two processes that are in R are in (T' R), and constructor T'_t implements the transitivity: it "says" that if two processes P and S are in (T' R), and if S and Q are in (T' R) as well, then P and Q are in (T' R). Definitions for T and T' are of course equivalent:

```
Lemma T_T' : (R:relation) (P,Q:Pr) (T R P Q) -> (T' R P Q).

Lemma T'_T : (R:relation) (P,Q:Pr) (T' R P Q) -> (T R P Q).
```

In our proofs, the T' function will be easier to use, since the Coq system supplies a wide range of tactics to handle inductively defined types.

Since functions $\mathcal{D}_n$, $\mathcal{B}$ and $\mathcal{T}$ are defined by the application of respectful constructors to respectful functions, they are respectful. This is not difficult to check in Coq, except for the trasitive closure of a relation, which is not defined by a constructor (we thus need to check its respectfullnes "by hand"):

[1]Note that this does not imply that the argument of the $\mathcal{B}$ constructor is a bisimulation itself.

```
Lemma Dn_resp : (n:nat)(respectful (Dn n)).

Lemma B_resp : (respectful B).

Lemma T'_resp : (respectful T').
```

2.4 Closure of a relation under contexts

We now turn to the implementation of the subsection number 2.1 of Sangiorgi's paper, entitled *"Closure of a relation under contexts"*. To define a context, we must define more precisely the shape of elements of the set of processes $\mathcal{P}r$; in [San94], Sangiorgi considers a term algebra over a one-sorted signature Σ:

Definition 2.15 (term algebra over a one-sorted signature) *Let Σ be a set of operators, each operator having an arity $n \geq 0$. The term algebra over signature Σ, written $\mathcal{P}r_\Sigma$, is the least set of strings which satisfy:*

- *if f is an operator in Σ with arity 0, then f is in $\mathcal{P}r_\Sigma$*
- *if f is an operator in Σ with arity $n > 0$, and $t_1, \ldots, t_n$ are already in $\mathcal{P}r_\Sigma$, then $f(t1, \ldots, t_n)$ is in $\mathcal{P}r_\Sigma$.*

To implement this definition in Coq, we use two mutually inductive types, the type `Pr` for the actual processes (with constructors `const` for the first rule stated above, and `funct` for the second one), and the type `p_list` for non-empty lists of processes (in order to build the argument for an operator of arity $n > 0$).

```
Variable E : Set.

Variable arity : E -> nat.

Mutual Inductive Pr : Set :=
  const : (x:E) (arity x)=O -> Pr
| funct : (x:E) (p_list (arity x)) -> Pr

with p_list : nat -> Set :=
  one : Pr -> (p_list (S O))
| cons : (n:nat) Pr -> (p_list n) -> (p_list (S n)).
```

We can now define Σ-*contexts*:

Definition 2.16 (Σ-context) *We note by $[.]$ a symbol not in Σ, called hole. $\Sigma([.])$ is the signature which has all operators in Σ as before, and in addition* *symbol* $[.]$ *with arity 0. A* Σ*-context is an element of* $\mathcal{P}r_{\Sigma([.])}$ *(the term algebra over* $\Sigma([.])$*, also written* $\mathcal{P}r_{\Sigma}([.])$*) with at most one occurrence of the hole* $[.]$ *in it.*

In our Coq implementation, we define as above two mutually inductive types.

```
Mutual Inductive context : Set :=
  hole : context
| Cconst : (x:E) (arity x)=O -> context
| Cfunct : (x:E) (Cp_list (arity x)) -> context
with
Cp_list : nat -> Set :=
  Cone : context -> (Cp_list (S O))
| Ccons_y : (n:nat) context -> (p_list n) -> (Cp_list (S n))
| Ccons_n : (n:nat) Pr -> (Cp_list n) -> (Cp_list (S n)).
```

Type `context` is for contexts, and is defined by saying that a context is either the hole, or a constant process, or the application of an operator of arity $n > 0$ to a list of type `Cp_list`. The type `Cp_list` is for non-empty lists of contexts with at most one occurrence of the hole. This property is ensured by the two constructors `Ccons_y` and `Ccons_n`: we can either build a list from a context and list of processes (the `Ccons_y` case: if the hole occurs, it can only be found in the first element of the list), or take a process and add it to a list of type `Cp_list` (in this case, we have no extra hole since we add a process: it is the `Ccons_n` constructor).

We use C to range over Σ-contexts. If C is a Σ-context and $P \in \mathcal{P}r_{\Sigma}$, then $C[P] \in \mathcal{P}r_{\Sigma}$ is the process obtained from C by filling the hole $[.]$ with P. We define the corresponding Coq functions `C2Pr` and `C2Pr_l` by "destructuring" a context with the `Case` operator:

```
Fixpoint C2Pr [C:context] : Pr -> Pr :=
  [P:Pr]<Pr>Case C of
    P
    [x:E][H:(arity x)=O](const x H)
    [x:E][l:(Cp_list (arity x))](funct x (C2Pr_l (arity x) l P))
  end
with
C2Pr_l [n:nat;l:(Cp_list n)] : Pr -> (p_list n) :=
  [P:Pr]<[n:nat](p_list n)>Case l of
    [C:context](one (C2Pr C P))
    [n:nat][C:context][l':(p_list n)](cons n (C2Pr C P) l')
    [k:nat][Q:Pr][l':(Cp_list k)](cons k Q (C2Pr_l k l' P))
  end.
```

The definition of the closure of a relation under contexts requires the definition of a certain class of contexts, namely the *faithful* ones.

*** Definition 2.17 (faithfulness of context sets and of contexts, (2.17))** *A set $Cont$ of Σ-contexts is a* faithful context-set *if for all $C \in Cont$ and $P \in \mathcal{P}r_\Sigma$ whenever $C[P] \xrightarrow{\mu} R$, there exist $C' \in Cont$ s.t. either*

- *$R = C'[P]$ and, for all Q, it holds that $C[Q] \xrightarrow{\mu} C'[Q]$, or*
- *there are $P' \in \mathcal{P}r_\Sigma$ and $\lambda \in Act$ s.t. $P \xrightarrow{\lambda} P'$ and $R = C'[P']$ and, moreover, for all $Q, Q' \in \mathcal{P}r_\Sigma$ s.t. $Q \xrightarrow{\lambda} Q'$, it holds that $C[Q] \xrightarrow{\mu} C'[Q']$.*

  A Σ-context C is faithful if $C \in Cont$, for some faithful context-set $Cont$.

To make definitions more concise, we adopt a new notation: $P \xrightarrow{\hat{\lambda}} Q$ means "$P = Q$ or $P \xrightarrow{\lambda} Q$"; we can this way merge the two previous properties:

*** Remark 2.18 (2.18)** *With the previous notations, $Cont$ is faithful if there are $P' \in \mathcal{P}r_\Sigma$ and $\hat{\lambda}$ s.t. $P \xrightarrow{\hat{\lambda}} P'$ and $R = C'[P']$ and, moreover, for all $Q, Q' \in \mathcal{P}r_\Sigma$ s.t. $Q \xrightarrow{\hat{\lambda}} Q'$ it holds that $C[Q] \xrightarrow{\mu} C'[Q']$.*

We directly implement the $\xrightarrow{\hat{\lambda}}$ notation in Coq, by defining a new transition relation `hat`:

```
Variable Act : Set.
Variable trans : Pr -> Act -> Pr -> Prop.

Definition Act_eq : Set := (Act+{True}).

Inductive hat : Pr -> Act_eq -> Pr -> Prop :=
  hat_tr : (P,Q:Pr)(a:Act) (trans P a Q)
    -> (hat P (inleft Act True a) Q)
| hat_eq : (P:Pr) (hat P (inright Act True I) P).
```

The type `Act_eq` is the sum of types `Act` (the type for actions) and `{True}` (a type with only one element, to denote the case where we have equality). Constructors for `hat` are `hat_tr` if we have actually a transition (hence the hypothesis `(trans P a Q)`) and `hat_eq` if we have equality between processes.

We then define predicates `faithful_cont_set` (over a set of contexts, denoted itself as a predicate over contexts) and `faithful_cont` (over contexts):

```
Definition faithful_cont_set : (context -> Prop) -> Prop :=
  [Cont:context -> Prop]
  (C:context)(Cont C) ->
  (P:Pr)(mu:Act)(R:Pr) (trans (C2Pr C P) mu R) ->
  (Ex [C':context]
    ((Cont C') /\
     (Ex [P':Pr] ((Ex [lam:Act_eq] (
    (hat P lam P') /\ (R = (C2Pr C' P')) /\
    ((Q,Q':Pr) (hat Q lam Q') -> (trans (C2Pr C Q) mu (C2Pr C' Q')))
  )))))).

Inductive faithful_cont : context -> Prop :=
  f_cont : (C:context) (P:context -> Prop) (P C) ->
  (faithful_cont_set P) -> (faithful_cont C).
```

The closure under contexts function $\mathcal{C}_\Sigma$ can now be introduced:

Definition 2.19 (closure under contexts)

$$\mathcal{C}_\Sigma(\mathcal{R}) \quad = \quad \bigcup_{C \text{ faithful}} \{(C[P], C[Q]) \; (P, Q) \in \mathcal{R}\}$$

and in Coq:

```
Inductive context_closure [R:(relation Pr)] : (relation Pr) :=
  cont_clos : (C:context) (faithful_cont C) ->
    (P,Q:Pr) (R P Q) ->
    (context_closure R (C2Pr C P) (C2Pr C Q)).
```

After all this definition machinery, we can state our result:

* **Lemma 2.20 (closure under contexts, (2.23))** *The function $\mathcal{C}_\Sigma$ is respectful.*

To state this result in Coq, we have to use the definition of `respectful` given above, instanciating the variable `Pr` with our inductively defined type `Pr`. This is achieved using Coq's modularity: more precisely, we enclose the definitions of the previous subsections into a Coq `Section`; when we close this section, a universal closure upon the declared variables is built. We then use the `Require` command to recover the definitions included in a separate Coq file. In our case, the definition of the `respectful` property has been abstracted over the set `Pr` of processes, the actions `Act` and the transition relation `trans`; we thus need to instanciate them in the statement of the theorem, as follows:

```
Lemma lemma_2_23 : (respectful Pr Act trans context_closure).
```

Here again, the original proof in [San94] can be ported to the Coq system without modifications. The result comes actually from a straightforward application of the faithfulness property for a context; in Coq, the proof steps are sometimes a bit tedious, but what we do is basically unfold definitions and apply the related properties. To shorten the proof, we use two auxiliary lemmas, that state an intermediate result of Sangiorgi's proof: `aux_lemma` is the implementation of the property *if* $\mathcal{R} \longrightarrow \mathcal{S}$ *and* $\mathcal{R} \subseteq \mathcal{S}$*, for every* $(P, Q) \in \mathcal{R}$*, whenever* $P \xrightarrow{\hat{\lambda}} P'$*, the following diagram commutes:*

$$\begin{tikzcd} P \arrow[d, "\hat{\lambda}"'] & \mathcal{R} & Q \arrow[d, "\hat{\lambda}"'] \\ P' & \mathcal{S} & Q' \end{tikzcd}$$

`aux_lemma2` is the symmetrical version of this lemma, which is not mentioned in Sangiorgi's proof, but that we need in our implementation [2].

```
Lemma aux_lemma : (R,S:(relation Pr))
  (progress Pr Act trans R S) -> (r_incl Pr R S) ->
  (P,Q:Pr) (R P Q) ->
  (P':Pr) (lam:Act_eq) (hat P lam P') ->
  (Ex [Q':Pr] ((S P' Q') /\ (hat Q lam Q'))).

Lemma aux_lemma2 : (R,S:(relation Pr))
  (progress Pr Act trans R S) -> (r_incl Pr R S) ->
  (P,Q:Pr) (R P Q) ->
  (Q':Pr) (lam:Act_eq) (hat Q lam Q') ->
  (Ex [P':Pr] ((S P' Q') /\ (hat P lam P'))).
```

In [San94], a sufficient condition for faithfulness of a set of contexts is given, using the notion of *transition rules in unary De Simone format.* We have not implemented this part of the paper, since it would have required complicated definitions in order to represent transition rules and the unary De Simone format; the given results require as well technical proofs, and are not of special interest for us, since we need to redefine completely the closure under contexts function in our application for the π-calculus. Furthermore, we can say that in a way they do not belong directly to the theory of progressions, so we can skip them without significant loss of meaning.

[2] Actually, Sangiorgi treats only one of the cases for progression (when the process on the left makes a transition); in Coq, of course, both cases, even if symmetrical, have to be treated.

See the conclusion for further considerations about the contrast between the implementation of general results about an abstract theory and the particularisation to a specific case.

3 The π-calculus and its semantics

We describe here the implementation of the π-calculus terms and of the early transition relation.

Processes are implemented in the Coq system through inductive definitions for types `name`, `l_name` (for name lists) and `pi` (for π-calculus terms). Following the implementation style of [Hue93], we adopt a de Bruijn representation for names. In this framework (see [dB72]), bound names are represented by the depth of their binding inside a term, and free names are considered to belong to a list (that can be viewed as an environment) coming with the process (i.e. their binding depth goes "above" the term). The advantage of de Bruijn indexes, traditionally, is that they supply α-conversion for free, since two α convertible terms share the same representation. Furthermore, in the specific case of the π-calculus, they allow to discard many side conditions in the definition of the semantics, as stressed in [Amb91], since terms are safer in this representation with respect to name clash problems. Of course, this improvement does not come for free, and we will see in the following that such an implementation requires many technical definitions in order to manipulate de Bruijn indexes; in addition to that, the proofs of Sangiorgi's results will show that in some way, *"what is gained on bound names is lost on free names"* (see next section).

Many implementations of the π-calculus using de Bruijn notation exist. In [Amb91], Ambler defines a de Bruijn notation for processes and proves the correspondence between a transition relation defined with this notation and a transition relation of the π-calculus; this is not our approach, since we directly implement π-terms with de Bruijn indexes, and in our case such results come as "meta" theorems that are implicitly admitted. The Mobility WorkBench [VM94] is another example of π-calculus implementation using de Bruijn indexes; it is a tool for checking *open* bisimulations equivalences on a polyadic version of the π-calculus that is used to describe mobile concurrent systems. We can cite as well Pict [Pie95], a programming language built on a mini asynchronous π-calculus where names are represented by De Bruijn's indexes.

Regarding formalisation of π-calculus into logical frameworks, an important amount of work has been done in HOL [Mel94, Ait94]; this implementation is a definition of automatic methods to check bisimilarity between processes that are represented in a "deep" way in the HOL system (see [Mel94] for more details); for this application, the de Bruijn notation has not been chosen, and α conversion has to be "manually" implemented.

We shall see in the following that the de Bruijn notation strongly influences the implementation style of the constructions we are manipulating, and that sometimes even the shape of some proofs is dictated by index managing considerations.

3.1 Syntax

In the de Bruijn notation, a variable is represented by a natural number; we thus define types `name` and `l_name` the following way:

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

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

The π-calculus has two binding constructions, namely *restriction* and *abstraction*. In the polyadic case, both notions deal with lists of names instead of one name at a time. However, we have not treated ν and λ the same way in our implementation; more precisely we keep a monadic meaning for the restriction operator ν, whereas the abstraction is truly polyadic. This distinction comes from the fact that manipulating lists of names is rather tedious in Coq, and we thus prefer to stick as much as possible to a monadic approach. We believe that while reception is strongly polyadic in our π-calculus, because a communication can involve many names *at the same time*, the definition of a new list of names can be split into several "monadic" definitions with no significant loss of meaning[3].

These choices motivate to the following definition of the inductive type `pi` for processes in Coq:

```
Inductive pi : Set :=
  Skip : pi
| Res : pi -> pi
| Par : pi -> pi -> pi
| Inp : name -> nat -> pi -> pi
| Out : name -> l_name -> pi -> pi.
```

[3]We will see in the following how we can still define in a simple way a bound output involving a list of names, thus achieving name extrusion with a polyadic flavour.

The Res constructor takes no argument more than the subject process (and thus acts only as a "monadic" binder), while the Inp constructor requires a name (where the communication occurs), an arity (of type nat) and a continuation, to define an abstraction: in other words, in (Inp x k P), k names are bound by the abstraction.

Other constructors are self-explanatory. As an example, the process $P \equiv (\nu x)\, \bar{y}.[x](\, \bar{z} \mid y.(\lambda(a,b))\bar{a})$ can be represented in Coq by the definition:

```
Definition P : pi :=
  (Res (Out (Ref (S O)) (Cons (Ref O) Nil)
          (Par
            (Out (Ref (S (S O))) Nil Skip)
            (Inp (Ref (S O)) (S (S O))
               (Out (Ref (S O)) Nil Skip))))).
```

Note that the representation of free variables y and z is in no way "canonical" (here y is denoted by index 1 and z by 2, but we could have actually chosen any two different integers greater than 0); we will discuss this point later on. Another interesting fact is that we do not implement sorts in this work: on our example, the y channel is first used to emit a single name, then to receive a list of two names; the definition of the process P would normally generate a sorting error for the y channel.

We proceed now to the definition of semantics, by implementing the transition relation given in the first section. The forecoming paragraphs involve considerations about the implementation of de Bruijn indexes that are pretty technical; the reader interested in a not too detailed approach should directly refer to the subsection entitled *"Implementing the semantics"*, skipping the technical stuff.

3.2 Managing de Bruijn indexes in a communication

Before defining the transition relation in the Coq system, let us have a look at the transformations on de Bruijn indexes that are involved in a communication. Consider the two processes of Figure 1 that are put in parallel: on the left, an abstraction of the form (Inp x n P) is about to receive a list of names l from the term (Res .. (Res (Out x l Q))) on the right; the *concretion* (i.e. the emitting process) has k restrictions on top that correspond to name extrusions (i.e. communications of private names); we also note that the length of list l has to be equal to n, because we want to preserve arity. In the receiving process P, the n first names are bound by abstraction; we consider two occurences of such names, represented by arrows 1 and 2

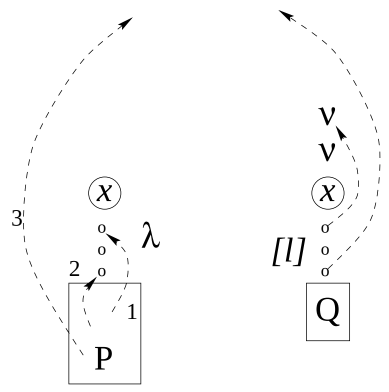

Figure 1: Before communication: $x.(\lambda\vec{y})P \mid (\nu\vec{z})\,\bar{x}.[\vec{l}]Q$

(we will see in the following how they differ); arrow 3 represents a name free in `(Inp x n P)`, that can be considered as being bound "above" the term `(Par (Inp x n P) (Res .. (Res (Out x l Q))))`. In the emitted list `l`, we consider two references, one pointing to a name that is free in the emitting process, the other one pointing to a name that is transmitted *via* name extrusion.

Figure 2 represents both terms after the communication occured; let us see how the diffferent entities are modified.

Both terms have now the `k` restrictions in common, because of name extrusion. References in the process `Q` have not changed, which is quite natural with respect to semantics: intuitively, a concretion emits an information and then goes on its own way.

For the abstraction, things are much more complicated, since many events have to be considered: first of all the `n` "unknown" names that were bound by a λ now have a meaning (they are instanciated), and among them some (`k` of them to be precise) are new; furthermore, with all these changes, references to "free" names in the continuation process `P` have to be kept coherent. The possible cases for a name in `P` are illustrated by the behaviour of arrows 1, 2 and 3 of figure 1:

- Arrow 1 corresponds to a variable, bound by abstraction, that is instanciated with a newly created name (name extrusion), and becomes arrow $1'$ on figure 2. From the point of view of de Bruijn references, this variable is viewed at the same depth in the term coming from the

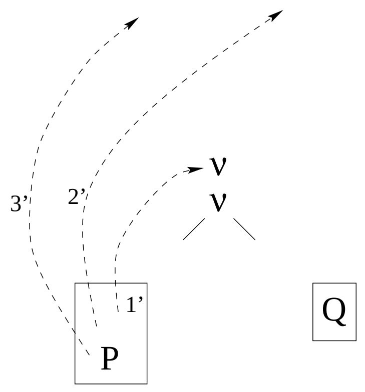

Figure 2: After communication: $(\nu\vec{z})(P_{\{\vec{y}:=\vec{l}\}} \mid Q)$

abstraction as in the list that is emitted by the concretion: the instanciation thus suffices to get the index right in this case, with no extra manipulation.

- This also holds for the variable represented by arrow 2 on figure 1, that gets instanciated with a name free in `(Res .. (Res (Out x l Q)))` and becomes arrow $2'$: since its depth is the same after the communication as it was in the list `l`, instanciation is the only operation to perform.

- Arrow 3, that pointed to a reference above the `Par` construct, say `i`, now must point to reference `i-n+k` to indicate the same binder: it has not to pass over the `n` names that were bound by the `Inp` construct anymore (hence the `-n`), but over the `k` newly transmitted names (hence the `+k`).

Let us sum up these considerations, and state precisely the general algorithm for updating a name of the form `(Ref i)` that occurs in the process `P` at depth `d`; we consider three cases:

- If `0`≤`i`<`d`, the name is not freee in `P`, and remains unchanged.

- If `d`≤`i`<`d+n` (recall that `n` is the length of list `l`), the name was bound by abstraction, and has to be instanciated. This is done by lifting all references in list `l` by `d` (this way we "translate" them and get their meaning at depth `d`), and taking the (`i-d`)th coordinate of this list. As seen above, this is enough to get references right in this case.

- If `d+n`$\leq$`i`, `i` represents a name that was free in `(Inp x n P)`; `i` becomes thus `i+k-n`.

3.3 Coq functions on de Bruijn indexes

We now consider the Coq functions that are defined in order to implement the de Bruijn index manipulations shown above. Each operation is usually implemented in three steps, namely with a function that takes a name as argument, a function on name lists, and finally the function on processes; although the function that really implements the machinery on indexes is the one for names (where de Bruijn notation actually appears), most of the time we will only invoke the function on processes, which itself calls the other two functions.

Lifting a term When working with de Bruijn indexes, we always look at a term considering that we are at a given binding depth `n`. This means that each reference to an index smaller than `n` will correspond to a bound name, while each reference to an index greater than `n` will go "outside the bounds of the term", and thus will represent a free name.

The first basic operation on de Bruijn indexes that we need to perform is *name lifting*, i.e. adding an integer `k` to all free names of a term. Reasoning at a given depth `n`, we make a test on the integer `i` that represents a name: if `n`$\leq$`i`, then we have a free name, `i` is replaced by `i+k`, while if `n>i`, `i`, as a bound name, remains unchanged. This simple algorithm translates in Coq as:

```
Definition lift_na : name -> nat -> nat -> name :=
  [na:name][n,k:nat]<name>Case na of
    [i:nat](Ref <nat>Case (test n i) of
      [_:(le n i)](plus k i)
      [_:(gt n i)]i
    end)
  end.
```

The `test` object is a lemma stating the disjunction we use, namely that for any two natural numbers `n` and `m`, we have `n`$\leq$`m` or `n>m`.

We then propagate this definition to treat name lists and finally processes, using the `Fixpoint` construction for recursively defined functions, associated to the `Case` operator that destructs a term:

```
Fixpoint lift_ln [l:l_name] : nat -> nat -> l_name :=
  [n,k:nat]<l_name>Case l of
    l
    [na:name][l':l_name](Cons (lift_na na n k) (lift_ln l' n k))
  end.

Fixpoint lift_pi [p:pi] : nat -> nat -> pi :=
  [n,k:nat]<pi>Case p of
    Skip
    [q:pi](Res (lift_pi q (S n) k))
    [p1,p2:pi](Par (lift_pi p1 n k) (lift_pi p2 n k))
    [na:name][i:nat][q:pi]
      (Inp (lift_na na n k) i (lift_pi q (plus n i) k))
    [na:name][l:l_name][q:pi]
      (Out (lift_na na n k) (lift_ln l n k) (lift_pi q n k))
  end.
```

As expected, in the definition of `lift_pi`, whenever we cross a binder, the depth increases: it becomes `(S n)` if we cross a `Res` construct, it becomes `(plus n i)` if we cross an `Inp` construct with arity `i`.

We will see in the following that this `lift_pi` function is often useful.

Substitution We just give here the definition of functions `project` and `subst_na` that implement substitution for names, the definition of the corresponding functions for name lists and processes being a straightforward "propagation", following the model for the lifting functions.

In our setting, if we want to substitute a list of variables in a term with a given list of names `l` of length `n`, we must replace the `i`th bound name of this term, for `i<n`, by the `i`th coordinate of `l`. We thus must have a function `project`, that takes a list and a position in the list and returns the corresponding element:

```
Fixpoint project [l:l_name] : nat -> name :=
  [n:nat]<name>Case l of
    (Ref O)
    [na:name][l':l_name]<name>Case n of na
      [n':nat](project l' n') end
  end.
```

To write the function `subst_na`, that implements substitution for a name, we must take binding depth in account, thus considering three cases for an index `k` at depth `depth`:

1. if the corresponding name is bound, but is not among the `n` first names that are bound in the term, that means that the index does not point to one of the names of the list: it is thus left unchanged (we are in the case where `depth`>`k`)

2. if the name is bound and has to be instanciated with a name of the list, i.e. if `depth`≤`k`<`depth+n`, we use the `project` function to get the (`k-depth`)th coordinate of list `l`

3. if the name is free (i.e. `i`≥`depth+n`), it is left unchanged

The Coq implementation uses the `test2` lemma to distinguish these three cases:

```
Definition subst_na : name -> l_name -> nat -> name :=
  [na:name][l:l_name][depth:nat]<name>Case na of
    [k:nat]<name>Case (test2 k depth (plus depth (l_length l))) of
        [C:{(gt depth k)}+{(le depth k) /\ (gt (plus depth (l_length l)) k)}]
        <name>Case C of
          [_:(gt depth k)]na
          [_:(le depth k)/\ (gt (plus depth (l_length l)) k)]
            (project l (minus k depth))
          end
        [_:(le (plus depth (l_length l)) k)]na
      end
  end.
```

We also define a function `low_subst`, to implement the algorithm described in the previous section for the implementation of an input action. `low_subst_na` behaves as `subst_na`, except for free names: a free name (`Ref k`) is replaced by (`Ref (k-n)`); we will see in the following how this function is used to implement the algorithm stated above.

```
Definition low_subst_na : name -> l_name -> nat -> nat -> name :=
  [na:name][l:l_name][depth,n:nat]<name>Case na of
  [k:nat]<name>Case (test2 k n (plus n (l_length l))) of
    [C:{(gt n k)}+{(le n k) /\ (gt (plus n (l_length l)) k)}]
    <name>Case C of
      [_:(gt n k)]na
      [_:(le n k) /\ (gt (plus n (l_length l)) k)](project l (minus n k))
    end
    [_:(le (plus n (l_length l)) k)](Ref (minus k n))
    end
  end.
```

3.4 Implementing the semantics

Actions

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

A term of the form (Aou k na l) represents the output action of the list l along channel na, with k name extrusions (i.e. implicitly the application of the Res constructor k times).

To implement an input action, we also need to know how many new names are communicated to the receiving process; as we will see in the next paragraph, this is due to the "early" choice for the transition relation, that leads to de Bruijn index manipulation, in order to "create" new names in the abstraction (see below).

We also implement some functions about actions, namely lower_action, that "lowers" all names occuring in an action with step one, bound_action, that returns the number of names bound in the action (i.e. for non-τ actions, the first argument of the constructor), and occ_act, a function that checks if the name (Ref O) occurs in an action. We omit the definitions of these objects here, since they are straightforward and not very interesting.

Transition relation Let us state the inductive definition of the type commit, implementing the transition relation over processes, and then discuss each of its constructors (that can be interpred as rules).

We have in Coq:

```
Inductive commit : pi -> action -> pi -> Prop :=
  comm_in : (na:name)(n:nat)(p:pi)
    (k:nat)(l:l_name)
    (commit (Inp na k p) (Ain n na l)
      (low_subst_pi (lift_pi p n k) l O n))
| comm_ou : (na:name)(l:l_name)(p:pi)
    (commit (Out na l p) (Aou O na l) p)
| comm_op : (na:name)(k:nat)(l:l_name)(x,p:pi)
    (commit x (Aou k na l) p)
    -> ~(na = (Ref (S k))) -> (occ_n_ln l (S k))
    -> (commit (Res x) (Aou (S k) na l) p)
| comm_c1 : (x,y:pi)(na:name)(k:nat)(l:l_name)(p,q:pi)
    (commit x (Ain k na l) p) -> (commit y (Aou k na l) q)
```

```
   -> (commit (Par x y) Tau (add_nus k (Par p q)))
| comm_c2 : (x,y:pi)(na:name)(k:nat)(l:l_name)(p,q:pi)
   (commit x (Ain k na l) p) -> (commit y (Aou k na l) q)
   -> (commit (Par y x) Tau (add_nus k (Par q p)))
| 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))
| comm_pr : (x,p:pi)(a:action) (commit x a p) ->
   (y:pi) (commit (Par x y) a
                 (Par p (lift_pi y 0 (bound_action a))))
| comm_re : (x,p:pi) (a:action) (commit x a p) -> ~(occ_act a)
   -> (commit (Res x) (low_action a) (Res p)).
```

- `comm_in`: this is an atomic transition rule, defining the committment for an abstraction process; it says that the process `(Inp na k p)` is susceptible to receive a list `l` at channel `na`, with `n` new names transmitted, and then become `(low_subst_pi (lift_pi p n k) l 0 n)`. To understand how the latter process is built, the reader should refer to the explanations in the previous section: we first lift all free names in `(Inp na k p)` by `n` (or equivalently lift all free names in `p` at depth `k` by `n`, thus the `(lift_pi p n k)`), and then apply the `low_subst_pi` function stated above.

  What is important to notice here is that the transmission of `n` new names results in a lifting of the term `p`, to actually "make room" for these new names. In other words, the de Bruijn setting gives an operational meaning to the ν constructor, that actually *builds* new names in a term. In fact, each time the `lift` function is invoked, it corresponds to a creation of new names, which is the most natural way of avoiding clashes that one can encounter in a non-de Bruijn approach; this gives a hint on how the side conditions of table 1 are dismissed with this framework.

- `comm_ou` and `comm_op`: these two rules define the transitions that can be performed by a concretion; the `comm_ou` rule defines *unbound* output (hence the `0` as a first argument to the action constructor), while the `comm_op` defines how to "add" a restriction to an output to perform name extrusion. There are premisses to this rule, to check that the name being restricted is among the transmitted names (in order to transmit a *private* name), and that the communication channel is not the subject of this restriction, since communication cannot occur on a private channel. `comm_ou` and `comm_op` are direct translations of respectively rules `OUT` and `OPEN` of table 1.

- `comm_c1` and `comm_c2`: these rules define communication between processes; they represent the two symmetrical versions of rule `CLOSE` from table 1. If a communication occurs with `k` name extrusions, the resulting process will be made of the two actors in parallel with `k` restrictions on top of them. This is achieved with the `add_nus` function, that adds a given number of restrictions on top of a term (thus partly recovering a polyadic flavour for restriction, as hinted in the paragraph about π-terms syntax).

- `comm_pl` and `comm_pr`: these are the two symmetrical rules implementing `PAR`: here again, the side condition to avoid name clash translates into a lifting of one of the terms, in order to make room for the new names that are transmitted (recall that the function `bound_action` returns the number of new names involved in an action).

- `comm_re`: this is the translation of the `RES` rule; here the side condition remains, since we still have to check that the restricted name (referenced by `0`, since a `Res` constructor binds the first free name in a term) is not involved in the action. Note that this rule has to be related with `comm_op`, since in the latter case we can add a restriction and perform an action even if the name `(Ref 0)` occurs in the action, provided it is not the location of this action.

Having implemented in the Coq system our π-calculus and its semantics, we can now use our previous results on progression theory to prove results about π-terms.

4 Closure under contexts in the π-calculus

4.1 Applying the theory to the π-calculus

While our implementation of Sangiorgi's paper has been kept very close to the original text so far, our work will now begin to differ from Sangiorgi's presentation, basically for technical reasons depending on our implementation choices.

As stressed in [San94], the general theory about progression of relations has to be a little updated in order be used for π-calculus processes. This is mainly due to the fact that our machinery has to be defined more precisely, in order to tackle questions related to name substitutions, a crucial problem in the π-calculus. Our treatment of this difficulty differs from Sangiorgi's one, because of the de Bruijn representation we adopt for names; however, it is important to notice that although technical problems are not treated the same way in this implementation and in the original paper, they basically have the same source.

Let us first have a look at how Sangiorgi reformulates his theory in order to take in account the problem of name instanciation, before discussing our own presentation.

The original definitions. First of all, the classical problem of α-conversion, that naturally arises in a calculus with binders for variables, is not precisely treated in [San94]; it is just mentioned that *"we shall identify processes which only differ on the choice of the bound names"*. This means that from the point of view of relations between processes, if for two processes P and Q we have a relation $\mathcal{R}$ such that $P\mathcal{R}Q$, we have $P'\mathcal{R}Q'$ for any P' and Q' obtained by changing bound names in respectively P and Q. We will see below how this question, which is naturally tackled with our de Bruijn implementation for names, in some way arises elsewhere, namely for free names.

Moreover, since bound and free names share the same representation in Sangiorgi's setting, one has to avoid clashes between them whenever an action involving instanciation of names is performed by a process. This forces to redefine progression between relations with a side condition, to ensure the "freshness" of bound names of an action, as follows:

* **Definition 4.1 (4.1)** *A progression $\mathcal{R}\rightarrowtail\mathcal{S}$, between two relations $\mathcal{R}$ and $\mathcal{S}$ on π-calculus processes, holds if for all $P\mathcal{R}Q$*

- *whenever $P \xrightarrow{\mu} P'$ with $bn(\mu) \cap fn(Q) = \emptyset$, there is Q' s.t. $Q \xrightarrow{\mu} Q'$ and $P'\mathcal{S}Q'$,*

and the symmetric clause, on the actions by Q.

Our implementation. Within the framework of de Bruijn indexes for names, two α-convertible terms share the same representation, and thus cannot be distinguished. Consequently, a relation on terms of type `pi` in Coq naturally solves the α-conversion problem that arises in a classical setting.

Furthermore, we do not need to add an extra side condition in the definition of progression, since there is a real distinction between bound and free names in the de Bruijn notation, which ensures automatically the "freshness" of names coming from an action.

Unfortunately, as we get rid of those problems, a difficulty arises, that could be stated as: "with the de Bruijn notation, what is gained on bound names is lost on free names". Let us try to explain what a relation between processes should be in our setting, in order to understand how free names are represented. As we saw in the implementation of the transition relation (type `commit`), when a process performs an input and receives a list `l` of names, `k` of them being new (name extrusion), "room" is made for these `k` names by lifting the de Bruijn indexes in the term[4]. In this process, a free name represented by index say i at depth 0 is represented after the lifting by index $i+k$; however, what we say is that indexes i and $i+k$ actually represent the same name at different moments of its "history". Therefore, relations involved in progressions (i.e. intuitively, relations where names have an "history") should be preserved by operations like lifting of names; in other words, for two given processes P and Q, we are interested in relations $\mathcal{R}$ such that whenever $P\mathcal{R}Q$, for all k, if P' and Q' are obtained by lifting processes P and Q with step k, we have $P'\mathcal{R}Q'$ (remind that lifting involves only free names, and bound names are left unchanged). An important remark is that lifting is not the only operation on terms that should preserve a relation for us: for example, in the definition of the `comm_in` constructor above, what we do, after performing a lift, is actually to lower free names in the receiving process, in parallel with the name instanciations. However, we will see below that for our proofs, a condition on relations involving only lifting of terms is enough. We leave for future work the precise study of admitted operators for which a relation should be preserved in our setting[5].

We thus consider in Coq a specific class of relations over processes (i.e. objects of type `pi -> pi -> Prop`), namely those who satisfy the predicate `liftable` stated below, asserting that a "liftable" relation is preserved by lifting any two related processes:

```
Definition liftable : (pi -> pi -> Prop) -> Prop :=
  [R:pi -> pi -> Prop]
```

[4]As mentioned before, this is the "de Bruijn translation" of a side condition: instead of checking that newly received names are not already known in the process, we create *dynamically* `k` names for the process as it receives them.

[5]Actually, we think that the basic operations consisting in lifting free names in a term of a given integer k and from a given depth, and the converse, i.e. lowering, are enough to represent any substitution on the free names of a term. Of course, the lower operator has to be handled with care, since we cannot lower by k at depth $d < k$: this would transform a free name into a bound one, or even return a negative index for a name! ... Because of the complications arising with this operator, and since the proofs can be carried without considering it in the definition of "good" relations, we have not implemented this kind of results.

```
  (p,q:pi) (R p q)
  -> (k,depth:nat) (R (lift_pi p depth k) (lift_pi q depth k)).
```

Since we do not consider *any* relation on objects of type pi, but only a class of such relations, the general theory that we have implemented cannot be used; nevertheless, what we do is just to rewrite all the definitions and proofs stated in the section about the theory of progressions, adding a variable good of type ((Pr -> Pr -> Prop) -> Prop) (a predicate over relations), and adding to each quantification over a relation the condition stating that the quantified relation satisfies predicate good. This is in no way difficult, and more importantly does not affect at all our proofs; let us just state as an example our new formulation of the soundness property:

```
Variable good : (Pr -> Pr -> Prop) -> Prop.

Definition sound : (relation -> relation) -> Prop :=
  [F:relation->relation](R:relation) (good R) ->
    (progress R (F R)) -> (r_incl R bisimilar).
```

We use in the following this new version of the general theory, in order to reason about liftable relations (instanciating the good variable with the liftable function).

4.2 Contexts for π-calculus terms

Contexts Because we have to take in account the substitutions on names that are involved in the transition relation on π-calculus processes, the general results about the closure under contexts function have to be revisited. The gap between the actual complexity of the implementation in the general case and in the specific case of the π-calculus is quite impressive, as the following will show. We first define monadic contexts for processes of type pi:

```
Inductive context : Set :=
  Hole : context
| cRes : context -> context
| cP_l : context -> pi -> context
| cP_r : pi -> context -> context
| cInp : name -> nat -> context -> context
| cOut : name -> l_name -> context -> context.
```

This definition is quite easy to understand: a context has globally the shape of a π-term, and if we encounter a Par branch, we must decide in which branch the hole is located. We then define the c2pi function, that given a context $C[.]$ and a process P, returns the process $C[P]$, obtained by replacing the hole in C by the process P:

```
Fixpoint c2pi [c:context] : pi -> pi :=
  [p:pi]<pi>Case c of
    p
    [d:context](Res (c2pi d p))
    [d:context][q:pi](Par (c2pi d p) q)
    [q:pi][d:context](Par q (c2pi d p))
    [na:name][n:nat][d:context](Inp na n (c2pi d p))
    [na:name][l:l_name][d:context](Out na l (c2pi d p))
  end.
```

We also implement the usual machinery about de Bruijn indexes for terms of type context (lifting a context, checking occurences of free names in a context, ...).

4.3 The closure under contexts function

Because of substitutions coming from communications, in a process $C[P]$, the context C can "modify" the process P; thus, to say things intuitively, it is not sound to prove only $P \sim Q$ if we have to prove $C[P] \sim C[Q]$, for $P \sim Q$ might not imply $C[P] \sim C[Q]$. Therefore, we introduce the *guardness* property for a context, saying that a process C is guarded if the hole occurs in C under a prefix construction. The definition of the closure under contexts function $\mathcal{C}_\pi$ is the following:

*** Definition 4.2 (Closure under contexts function)**

$$\mathcal{C}_\pi(\mathcal{R}) \quad = \quad \cup_{C\ non-guarded}\{(C[P], C[Q]) : (P, Q) \in \mathcal{R}\} \ \bigcup$$

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

We easily implement in Coq the two predicates guarded and unguarded about contexts:

```
Fixpoint guarded [c:context] : Prop :=
  <Prop>Case c of
    False
    [d:context](guarded d)
    [d:context][_:pi](guarded d)
    [_:pi][d:context](guarded d)
    [_:name][_:nat][_:context]True
    [_:name][_:l_name][_:context]True
  end.

Fixpoint unguarded [c:context] : Prop :=
  <Prop>Case c of
    True
    [d:context](unguarded d)
    [d:context][_:pi](unguarded d)
    [_:pi][d:context](unguarded d)
    [_:name][_:nat][_:context]False
    [_:name][_:l_name][_:context]False
  end.
```

The closure function is defined in two steps: we first implement a function `close_cons`, taking a context `c` and a relation `R` over processes, and returning the condition on `R` corresponding to the guardness of the context `c`, from the above definition of $\mathcal{C}_\pi$ (`dec_guarded` is a lemma stating that a context satisfies either predicate `guarded` or predicate `unguarded`); we then define `Close`, the closure under contexts function, in an inductive way:

```
Definition close_cons :
  context -> (pi -> pi -> Prop) -> pi -> pi -> Prop
  :=
  [c:context]<(pi -> pi -> Prop) -> pi -> pi -> Prop>Case (dec_guarded c) of
    [_:(guarded c)]
      ([R:pi -> pi -> Prop][p,q:pi]
        (l:l_name) (R (pi_subst p l) (pi_subst q l)))
    [_:(unguarded c)]
      ([R:pi -> pi -> Prop][p,q:pi](R p q))
  end.

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)).
```

4.4 Respectfulness theorem

We now come to our result:

***Proposition 4.3 (5.2)** *Function C_π is respectful.*

As seen before, since we are taking advantage of Coq's modularity, we have to instanciate the definition of `respectful` with the sets of processes and actions (`pi` and `action`), the transition relation (`commit`), the predicate over relations (`liftable`) and the actual argument of the `respectful` predicate, the closure function (`Close`). We thus write in Coq:

`Lemma Close_respectful : (respectful pi action commit liftable Close).`

Proof Our proof is close to the one in [San94] in its global form; the technical parts require of course much more attention in our work and generally do not follow the hints given by the original text, mainly for implementation reasons.

We consider for this proof two liftable relations on π-terms $\mathcal{R}$ and $\mathcal{S}$; we suppose $\mathcal{R} \subseteq \mathcal{S}$ and $\mathcal{R} \longrightarrow \mathcal{S}$. The proof of $\mathcal{C}_\pi(\mathcal{R}) \subseteq \mathcal{C}_\pi(\mathcal{S})$ is straightforward; we also have to check that $\mathcal{C}_\pi(\mathcal{R}) \longrightarrow \mathcal{C}_\pi(\mathcal{S})$ holds. We thus consider $(C[P], C[Q]) \in \mathcal{C}_\pi(\mathcal{R})$, an action μ and a process R such that $C[P] \xrightarrow{\mu} R$. We must exhibit C', P' and Q' such that

$$R = C'[P'], \quad C[Q] \xrightarrow{\mu} C'[Q'], \quad \text{and} \quad (C'[P'], C'[Q']) \in \mathcal{C}_\pi(\mathcal{S})$$

As in the original proof, we proceed by induction on the structure of the context C; we will focus here on the technical difficulties that appear in our proof.

4.4.1 `C = Hole`

This case is easy, since we have for any process P, $C[P] = P$, so the property comes from the hypothesis $\mathcal{R} \longrightarrow \mathcal{S}$.

4.4.2 `C = (cRes C1)`

In this case, the hypothesis $C[P] \xrightarrow{\mu} R$ translates in Coq as `(commit (c2pi (cRes C1) P) mu R)`, and, by unfolding the definition of function `c2pi`, as `(commit (Res (c2pi C1 P)) mu R)`. We apply the `Inversion` tactic to this hypothesis, which generates two subgoals coresponding to the two possible constructors used to derive this hypothesis, namely `comm_op` and `comm_re`[6]. The proofs are then quite easy and involve basic application of the definitions we use.

4.4.3 `C = (cP_l C1 T)` or `C = (cP_r T C1)`

The proofs of those two cases are very similar to eachother; although they have to be given explicitely in the implementation, we will consider just the first case here, namely where a context is built by adding a process `T` to

[6]To learn more about the `Inversion` tactic, the reader should refer to [CCF+96].

the right of a context `C1` with the parallel constructor. As in the previous case, we apply the `Inversion` tactic to the hypothesis `(commit (Par (c2pi C1 P) T) mu R)`, stating the transition relation between the two processes. Let us examine the four subgoals generated by this tactic (again, the rôle of the `Inversion` tactic is, by looking at the structure of an inductive term, to derive for each possible constructor of its type the necessary conditions that should hold):

1. The first subcase corresponds to the constructor `comm_c1`: a communication occurs between the processes, `(c2pi C1 P)` is the receiving process and `T` is emitting. The induction hypothesis allows us to build easily the context C' and processes P' and Q' in order to prove our result.

2. This subcase is symmetrical with respect to the latter: process `(c2pi C1 P)` emits and `T` is the receiving process; here again, the proof is not difficult.

3. The third subcase corresponds to the case where `T` performs an action and `(c2pi C1 P)` does not; this is actually the tricky situation where our hypothesis on relations $\mathcal{R}$ and $\mathcal{S}$ (the so-called "liftability") has to be used.

   If we look at the type of `comm_pl` (the one we are considering in this subcase, actually), we see that if the action performed by `T` has bound names, then the term `(c2pi C1 P)` must be lifted in order to "make room" for these names. To write things more formaly, we are in the case where $C_1[P]\ \mathcal{C}_\pi(\mathcal{R})\ C_1[Q]$, and $T \xrightarrow{\mu} T'$, hence $(\text{lift}^k(C_1[P])|T) \xrightarrow{\mu} (\text{lift}^k(C_1[P])|T')$ (k is the number of names bound in the action μ) [7]. We replace $\text{lift}^k(C_1[P])$ by $(\text{lift}^k(C_1))[\text{lift}'^k(P)]$, where lift'^k is the function that lifts all free names of k at depth n, if n is the depth of the hole in C_1. This is implemented in Coq by the following lemma (`ct_depth` is the function returning the depth of the hole in a context):

   ```
   Lemma subst_c_p : (c:context)(p:pi)(l:l_name)(n:nat)
     (pi_subst_n (c2pi c p) l n) =
     (c2pi (ct_subst c l n)
       (pi_subst_n p (lift_ln l O (ct_depth c))
                     (plus n (ct_depth c)))).
   ```

[7] We "mix" here a mathematical notation and names referring to objects from our Coq implementation, such as lift^k. We believe that this presentation is easier to read than the actual Coq text, although less rigorous. We will use it whenever we need to describe technical details without stating the original Coq syntax.

It is then quite natural to make the following proposition for C', P' and Q':

$$C' = (\text{lift}^k(C_1)|T'), \quad P' = \text{lift}'^k(P), \quad Q' = \text{lift}'^k(Q)$$

The proof of $C[Q] \xrightarrow{\mu} C'[Q']$ is easy (we just defined C' and Q' to this purpose). We also have to prove $C'[P']\ \mathcal{C}_\pi(\mathcal{S})\ C'[Q']$: this actually reduces to prove $\text{lift}'^k(P)\ \mathcal{S}\ \text{lift}'^k(Q)$, and since the relation $\mathcal{S}$ is liftable, to prove $P\mathcal{S}Q$; the latter result is easily derived from $\mathcal{R} \subseteq \mathcal{S}$ and $C[P]\ \mathcal{C}_\pi(\mathcal{R})\ C[Q]$.

4. This subcase is much easier to treat than the previous one: this time, process `(c2pi C1 P)` is performing an action, while `T` does not. We avoid here all the tedious work to prove the relation between processes, since `T`, which is the lifted process, is not really involved in the relation (we have to deduce $(C_1[P']\|\text{lift}^k(T))\ \mathcal{C}_\pi(\mathcal{S})\ (C_1[Q']\|\text{lift}^k(T))$ from $C_1[P']\ \mathcal{C}_\pi(\mathcal{R})\ C_1[Q']$, which is easy).

4.4.4 `C = (cInp na n C1)`

This is the more technical part of our proof, since a reception involves names substitutions, an issue where tedious problems often arise, as it is the case here.

We first remark that in this case, the context `C` is guarded, so the induction hypothesis "says" `(R (pi_subst_n P l 0) (pi_subst_n Q l 0))` for any list of names `l`: this will be a key result for us. If we look at constructor `comm_in`, we see that the committing process has a complicated shape after the reception, namely something like `(low_subst_pi (lift_pi p0 n k) l 0 n)`, for a given process `p0`, if `k` is the number of bound names in the input action. In the original proof, we just say that if the context is $C = na.(\lambda\vec{y})C_1$, the term becomes $C_1[P]\{\vec{y} := \vec{l}\}$: this contrast between a simple statement and our intricate terms is a good example of how objects involving technical definitions become quite complicated as soon as they are implemented in the Coq system.

It is important to notice that the really significant modification we apply to process `p0` is the substitution. In our implementation, this substitution comes along with a lifting followed by a lowering of free names; these two operations should actually preserve the relations over processes we are considering, since they are just needed to manage free names, as already discussed above. In our proof, though, we only consider "liftable" relations, and we do not allow to “lower” related terms; we will see in the following how we tackle this difficulty by using the strong hypothesis we have about `R`.

Let us turn to the actual proof: the type of constructor `comm_in` indicates the shape of terms C', P' and Q' that we chose, in order to establish the transition relation. These objects then appear in the proof of the $\mathcal{C}_\pi(\mathcal{S})$ relation, as we have to prove:

$$\text{low\_subst}(\text{lift}^k(C[P]), l, n) \;\; \mathcal{C}_\pi(\mathcal{S}) \;\; \text{low\_subst}(\text{lift}^k(C[Q]), l, n)$$

We use here an auxiliary lemma, in order to establish the closure under contexts of a relation:

Lemma 4.4 *Let $\mathcal{R}$ be a relation over π-calculus processes. If for two processes P and Q we have $\forall\sigma\ (P\sigma)\mathcal{R}(Q\sigma)$, then for any context C,*
$C[P]\ \mathcal{C}_\pi(\mathcal{R})\ C[Q]$.

The proof of this lemma is trivial, and, in conjunction with $\mathcal{R} \subseteq \mathcal{S}$ and the `subst_c_p` lemma stated above, allows us to reduce our goal to:

$$\forall\sigma \;\; (\text{low\_subst}(\text{lift}'^k(P),\, l, n))\sigma \;\; \mathcal{R} \;\; (\text{low\_subst}(\text{lift}'^k(Q),\, l, n))\sigma$$

Let us quote the Coq text corresponding to this proposition, the quantified substitution σ being here represented by a list of names `l'`:

```
(l':l_name)
 (R  (pi_subst_n
       (low_subst_pi (lift_pi P (plus n (ct_depth C)) k)
         (lift_ln l O (ct_depth C)) (ct_depth C) n)
       l' O)
     (pi_subst_n
       (low_subst_pi (lift_pi Q (plus n (ct_depth C)) k)
         (lift_ln l O (ct_depth C)) (ct_depth C) n)
       l' O))
```

We are now in front of the aforementioned difficulty: intuitively, the hypothesis `(l:l_name) (R (pi_subst_n P l O) (pi_subst_n Q l O))` allows us to prove that the relation `R` between `P` and `Q` is preserved by substitution with name lists `l` and `l'`; the lifting of free names achieved by function `lift_pi`, as well as the lowering achieved by `low_subst_pi`, also preserve this relation, but “for another reason”, namely because our relations over processes should be preserved by lifting and lowering of free names in related terms. Apparently, we are therefore not able to prove this result, since we just supposed that `R` is *liftable*, and we do not have an hypothesis about the lowering of terms.

We now use a trick which allows us to conclude the proof in a quite direct way (even if a little brutal). The idea is to represent each of the manipulations on de Bruijn indexes that are performed on the processes by a substitution, and this way take advantage of the aforementioned hypothesis on R.

We define a function build_lift, that given a depth depth, an integer k and a length max, returns a list representing the lifting of max+1 terms by k at depth depth. The lifting operator is actually an infinite substitution that relates an index i with index i+k; we represent in our setting the truncation of such a substitution, defining it for all indexes lower than the maximum free index in a process (the latter notion being implemented by function max_free, that returns the maximum free index in a term of type pi at a given depth). For a process p, we call the build_lift function with max:=(max_depth p depth), and we get in return a list of names that represents the lifting operator *for p*; this manipulation is summarized in the following lemma:

```
Lemma subst_lift : (p:pi)(n,k:nat)
  (lift_pi p n k) = (pi_subst_n p (build_lift (max_free p n) n k) n).
```

Similarly, we represent the low_subst_pi operator by a list built with a function named low_subst, and we have:

```
Lemma low_subst_subst : (p:pi)(depth,k:nat)(l:l_name)
  (low_subst_pi p l depth k)
  = (pi_subst_n p
      (low_subst (max_free p depth) depth k l) depth).
```

We can now use these lemmas to rewrite the conclusion of our goal, in order to get a term where three substitutions are applied to processes P and Q, namely the one corresponding to the lift_pi function, followed by the one corresponding to low_subst_pi (with list l), and finally the one corresponding to list l'. What we do now is to represent the succession of three substitutions by only one substitution, which will allow us to use the hypothesis about R. We therefore need to represent the composition of two substitutions defined by lists l1 and l2 at a given depth (function comp_subst), and we also have to replace a substitution at depth depth by a substitution at depth O, in order to apply our composition lemma (function depth_subst). We just give here the lemmas corresponding to these constructions:

```
Lemma comp_subst_pi : (p:pi)(depth:nat)(l1,l2:l_name)
  (pi_subst_n (pi_subst_n p l1 depth) l2 depth) =
  (pi_subst_n p (comp_subst (max_nat (l_length l1) (l_length l2))
    l1 l2 depth) depth).

Lemma depth_subst_pi : (p:pi)(depth:nat)(l:l_name)
  (pi_subst_n p l depth) = (pi_subst_n p (depth_subst l depth) 0).
```

After a big amount of technical reasoning on de Bruijn indexes and substitutions, we can replace all substitutions by only one (no need to say that the list involved in this substitution has a really ugly aspect in Coq!), and conclude the proof using the hypothesis about `R`. We stick to this quite informal description of the Coq proof we are writing, in order to avoid too technical details.

As hinted above, we somehow cheated in our proof, since we "misused" the hypothesis `(R (pi_subst_n P l 0) (pi_subst_n Q l 0))` in order to tackle the difficulty about the lowering of terms related by `R`. However, this has allowed to considerably simplify our proofs (replacing the *liftable* predicate by a *"lowerable" and liftable* property would have been very tedious to represent, especially since lowering free names in a term is not always safe). Moreover, this tedious work on substitutions is also useful for some proofs of the next section.

4.4.5 `C = (cOut na l C')`

This case is much easier to treat than the previous one, since by definition of the transition relation, a concretion just emits a list and continues along its continuation. We thus do not have to handle substitutions, and the proof is straightforward.

5 Unique solution of equations

An application of the latter proposition (respectfulness of the closure under contexts function), is the proof of uniqueness of equations for the π-calculus. This result, coming from Milner's book [Mil89], says that under certain conditions on a given context C, all processes P such that $C[P] \sim P$ are bisimilar.

The proof of this result in [San94] is actually given for CCS, a simpler calculus where communications do not carry values. Sangiorgi only mentions that it translates into π-calculus with no significant modification, except that we have to consider a smaller relation, written $\sim^c$, instead of "plain" bisimilarity; the definition of $\sim^c$ (which is actually the congruence induced by $\sim$) is the following:

*** Definition 5.1 (congruence induced by $\sim$, (4.2))** *We set $P \sim^c Q$, pronounced "P and Q are congruent", if $P\sigma \sim Q\sigma$, for all substitutions σ.*

And in Coq:

```
Definition bisim_c : pi -> pi -> Prop :=
  [p,q:pi](l:l_name)
  (bisimilar pi action commit liftable
          (pi_subst_n p l O) (pi_subst_n q l O)).
```

In the following, we shall describe our adaptation of the original proof for CCS to the π-calculus. We will keep references to original results from Sangiorgi's paper, since the various steps of the proof are exactly the same, although they are more technical in our case (both because of an enrichment of the calculus and of implementation issues).

For simplicity reasons, we consider monadic contexts in our result; this means that we only have one hole in a term of type `context`. In [San94], Sangiorgi considers an arbitrary number of different holes $[.]_1, \ldots, [.]_n$, each of them possibly having many occurrences. The implementation of such a general result would have involved some tedious implementing work in Coq, and in particular it would have not allowed us to use the results from the previous section about the closure under contexts function; therefore, we keep a simpler presentation, with the belief that we still have the essence of the theory. Our Coq proofs for this part of the implementation basically involve the same reasoning as for those of the previous section, so we just briefly sketch their shape in a "mathematical" style without entering the details.

5.1 Auxiliary lemmas

We first prove a few lemmas from Sangiorgi's paper, that are useful for our final proof.

Following [San94], we write $\mathcal{C}$ to denote the closure under contexts function $\mathcal{C}_\pi$, and if $\mathcal{R}$ is a relation over processes, we write $\mathcal{R}^{\mathcal{C}}$ for $\mathcal{C}(\mathcal{R})$ and $\mathcal{R}^{\mathcal{T}}$ for $\mathcal{T}(\mathcal{R})$.

*** Lemma 5.2 (3.2)** *If $\forall\sigma\ (P\sigma, Q\sigma) \in \mathcal{R}$, then $(C[P], C[Q]) \in (\mathcal{R}^{\mathcal{C}})^{\mathcal{T}}$.*

In the original paper, no substitution was mentioned; because of the specific definition we have for the closure under contexts function in the case of the π-calculus, we had to replace the hypothesis $(P, Q) \in \mathcal{R}$ by a much stronger one. While in [San94] the proof of this lemma is done by induction on the structure of context C, it is much easier in our setting, because of the hypothesis we have on $\mathcal{R}$, together with the definition of the closure under contexts function $\mathcal{C}$. We use for the Coq proof our lemma 4.4 stated above, and we could actually get rid of the transitive function $\mathcal{T}$, since we do not need it in our proof. We keep it anyway to stay close to Sangiorgi's original text (it is useful for a proof by induction on C to apply the induction hyptohesis in some cases). We need of course to define the function $(-^{\mathcal{C}})^{\mathcal{T}}$ in Coq:

```
Definition R_C_T : (pi -> pi -> Prop) -> pi -> pi -> Prop :=
  [R:pi -> pi -> Prop](T' (Close R)).
```

* **Lemma 5.3 ((3.3), lemma 4.13 in [Mil89])** *If C is guarded and $C[P] \xrightarrow{\mu} P'$, there exist a context C' and a substitution σ_0 s.t. $P' = C'[P\sigma_0]$, and moreover, for any Q, $C[Q] \xrightarrow{\mu} C'[Q'\sigma_0]$.*

Here again, we had to adapt the statement of this lemma to the case of the π-calculus, since whenever μ is an input action, the context can modify its "content" P: we thus add the substitution σ_0. The proof of this result comes from an induction on the structure of C; although it involves some tedious managing of de Bruijn indexes, it is not difficult.

We now turn to a couple of lemmas that are related to our definition of bisimilarity, in the general setting where we have a set of processes $\mathcal{P}r$, a set of actions Act, and a transition relation $trans$; from the point of view of the implementation, this means that we work within the Coq files where the variables `Pr`, `Act` and `trans` have not been instanciated. These lemmas are needed for our Coq proofs, but are not part of Sangiorgi's paper.

Lemma 5.4 *For any processes P and Q such that $P \sim Q$, whenever $P \xrightarrow{\mu} P'$, there exists a process Q' s.t. $Q \xrightarrow{\mu} Q'$ and $P' \sim Q'$, and the symmetrical property for an action performed by Q.*

This result, which is the classical definition of a bisimulation relation, is in our case a consequence of our definition for $\sim$ (and hence has to be proved in Coq). We prove another small result as well:

Lemma 5.5 (symmetry of $\sim$) *If $P \sim Q$, then $Q \sim P$.*

These lemmas easily translate in Coq, and their proofs are straightforward.

5.2 The uniqueness result

We can now turn to our main result:

***Proposition 5.6 (unique solution of equations, (3.5), proposition 4.14(2) in [Mil89])** *Suppose C is a guarded context, with $P \sim^c C[P]$ and $Q \sim^c C[Q]$. Then $P \sim^c Q$.*

Proof To prove $P \sim^c Q$, we consider a substitution σ and prove $P\sigma \sim Q\sigma$. This is done by using the proof technique given by lemma `tech` stated in the second section of this paper; let us remind here its formulation:

```
Lemma tech : (p,q:Pr) (R:relation) (good R) -> (R p q) ->
  (F:relation -> relation)
  (respectful pi action commit liftable F) -> (progress R (F R)) ->
    (bisimilar pi action commit liftable p q).
```

We therefore consider the relation:

$$\mathcal{R} = \{(P\rho, Q\rho), \rho \,\text{substitution}\},$$

and prove that $\mathcal{R} \longrightarrow \sim (\mathcal{R}^{\mathcal{C}})^{\mathcal{T}} \sim$ holds. The lemma `tech` will then allow us to conclude $P\sigma \sim Q\sigma$, since $(P\sigma, Q\sigma) \in \mathcal{R}$.

To implement $\mathcal{R}$, we define the function `subst_rel` the following way:

```
Inductive subst_rel [p,q:pi] : pi -> pi -> Prop :=
  sr_cons : (l:l_name)
    (subst_rel p q (pi_subst_n p l O) (pi_subst_n q l O)).
```

We also define the $\sim (-^{\mathcal{C}})^{\mathcal{T}} \sim$ function, and prove its respectfulness (hence its soundness):

```
Definition R_C_T_B : (pi -> pi -> Prop) -> pi -> pi -> Prop :=
  [R:pi -> pi -> Prop] (B (T' (Close R))).

Lemma resp_RCTB : (respectful pi action commit liftable R_C_T_B).
```

We now turn to the proof of the progression. We have:

$$\forall \rho \; P\rho \sim (C[P])\rho \quad (1)$$

$$\forall \rho \; Q\rho \sim (C[Q])\rho. \quad (2)$$

Suppose $P \xrightarrow{\mu} P'$; applying (1) to our substitution σ, and using lemma 5.4, we get a process R s.t. $(C[P])\sigma \xrightarrow{\mu} R$ and $P' \sim R$. We then rewrite $(C[P])\sigma$

into $C\sigma[P\sigma]$, in order to use lemma 5.3. This gives us a new context C' and a substitution σ_0 s.t. $R = C'[P\sigma\sigma_0]$ and

$$\forall T\ C\sigma[T] \xrightarrow{\mu} C'[T\sigma_0]. \quad (3)$$

For the proof of this result, and in the forecoming steps of our proof as well, we frequently use the constructions about substitutions from the previous section; since we just apply the corresponding lemmas without proving any new result about these constructs, we will not enter the technical details of these proofs. This will of course help keeping the following as clear and concise as possible; nevertheless, the reader should still have in mind that these mathematical results sometimes need complicated manipulation of de Bruijn indexes in order to be implemented.

We apply (3) with $Q\sigma$; this gives $C\sigma[Q\sigma] \xrightarrow{\mu} C'[Q\sigma\sigma_0]$, and since we have (2), using 5.4 and the symmetry of $\sim$, we can exhibit a process Q' s.t. the two following diagrams commute:

$$\begin{tikzcd} P\sigma \arrow[r, phantom, "\sim"] \arrow[d, "\mu"'] & C\sigma[P\sigma] \arrow[d, "\mu"'] \\ P' \arrow[r, phantom, "\sim"] & C'[P\sigma\sigma_0] \end{tikzcd} \qquad \begin{tikzcd} C\sigma[Q\sigma] \arrow[r, phantom, "\sim"] \arrow[d, "\mu"'] & Q\sigma \arrow[d, "\mu"'] \\ C'[Q\sigma\sigma_0] \arrow[r, phantom, "\sim"] & Q' \end{tikzcd}$$

Using lemma 5.2, we show that $(C'[P\sigma\sigma_0], C'[Q\sigma\sigma_0]) \in (\mathcal{R}^{\mathcal{C}})^{\mathcal{T}}$, and thus $(P', Q') \in \sim (\mathcal{R}^{\mathcal{C}})^{\mathcal{T}} \sim$. This finally proves $\mathcal{R} \longrightarrow \sim (\mathcal{R}^{\mathcal{C}})^{\mathcal{T}} \sim$, hence $P\sigma \sim Q\sigma$.

Conclusions and future work

We have implemented the general theory of progressions of relations on a set $\mathcal{P}r$ of processes, and then applied it to the case of the π-calculus. Quite significantly, the proofs of the second part of this work turned out to be considerably more tedious and complicated to handle than those on non-specified relations. We believe that this is not due to a coincidence: while general or abstract results translate quite straightforwardly into the Coq system, our experience is that technical applications often require further investigations. Moreover, it seems that in a way, "complexity grows faster in Coq than on a paper proof", or in other words, we need more and more tedious work in Coq as the mathematical notions become technical. Of course, the main point is to find good implementation paradigms in order to tackle as much as possible of the technical aspects, but we think that it is not entirely Coq's responsibility: it is often the case that mathematical texts do not enter very technical details. To exagerate our point of view, we can say that one could probably define a "technical depth" in mathematical results under which no exploration is made (or at least very rarely). The main reason for this is clarity, because intuition gets easily lost in a long and detailed proof; in a proof checker like Coq, we must get into all details, but still trying to follow as much as possible the original form of the proof. The "mathematical" proof (as opposed to the "mechanised" proof in Coq) plays the rôle of a guide for the user, who has to deal with all the technical parts needed by a rigorous approach.

We plan to implement one more result coming from Sangiorgi's paper, namely the proof of respectfulness for the closure under injective substitutions function; this work shall probably require some tedious manipulations, since injective substitutions do not really enjoy a natural representation in our implementation. Nevertheless, we believe that this tool can really be useful for proving bisimilarities, since it can be applied to very small relations, as showed by an example in [San94]. It is important to notice as well that this example is on two *specific* processes, while the application of the closure under contexts function, namely the proof of uniqueness of solutions for equations, is a more general and abstract result. Our aim is indeed to supply functions that can be used to prove bisimilarities between "real" processes, and not only to prove properties of our implementation of the π-calculus.

Obviously, we cannot hope to achieve an efficient and automatic machinery comparable to the HOL implementation of the π-calculus [Mel94, Ait94] in the near future; we still think that our Coq implementation can be useful for a reasonable class of bisimulation proofs, and we hope to use it in the field of concurrent language semantics.

References

[Ait94] Otmane Ait-Mohamed. Vérification de l'équivalence du π-calcul dans HOL. Rapport de recherche 2412, INRIA-Lorraine, November 1994. (In French).

[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.

[Bou92] Gérard Boudol. Asynchrony and the π-calculus (note). Rapports de Recherche 1702, INRIA Sofia-Antipolis, May 1992.

[CCF$^+$96] C. Cornes, J. Courant, JC. Filliâtre, E. Gimenez, G. Huet, P. Manoury, 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.

[dB72] N.G. de Bruijn. Lambda Calculus Notation with Nameless Dummies. In *Indagationes Mathematicae*, volume 34, pages 381–392. 1972.

[FG] C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *POPL 96.*

[HKPM95] G. Huet, G. Kahn, and C. Paulin-Mohring. *The Coq Proof Assistant, A Tutorial.* Projet Coq, INRIA Rocquencourt / CNRS - ENS Lyon, Février 1995.

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

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

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[Mil91] Robin Milner. The polyadic π-calculus: a tutorial. Technical report, LFCS, Dept. of Computer Science, University of Edinburgh, october 1991.

[Mil92] Robin Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Parts I and II. 100:1–77, September 1992.

[MS] Robin Milner and Davide Sangiorgi. Barbed bisimulation. pages 685–695.

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

[San93] D. Sangiorgi. A Theory of Bisimulation for the π-calculus. Technical report, LFCS, Department of Computer Science, University of Edinburgh, May 1993.

[San94] D. Sangiorgi. On the bisimulation proof method. Technical report, LFCS, Department of Computer Science, University of Edinburgh, 1994.

[VM94] B. Victor and F. Moller. The Mobility Workbench — a tool for the π-calculus. In D. Dill, editor, *Proceedings of CAV'94*, volume 818 of *Lecture Notes in Computer Science*, pages 428–440. Springer-Verlag, 1994.