π-calculus in (Co)Inductive Type Theory*

Furio Honsell† Marino Miculan† Ivan Scagnetto†

Abstract

We present a large and we think also significant case-study in computer assisted formal reasoning. We start by giving a *higher order abstract syntax* encoding of π-calculus in the higher-order inductive/coinductive type theories CIC and $\mathrm{CC}^{(Co)Ind}$. This encoding gives rise to a full-fledged proof editor/proof assistant for the π-calculus, once we embed it in Coq, an interactive proof-development environment for $\mathrm{CC}^{(Co)Ind}$. Using this computerized assistant we prove *formally* a substantial chapter of the theory of *strong late bisimilarity*, which amounts essentially to Section 2 of *A calculus of mobile processes* by Milner, Parrow, and Walker. This task is greatly simplified by the use of higher order syntax. In fact, not only we can delegate conveniently to the metalanguage α-conversion and substitution, but, introducing a suitable axiomatization of the theory of contexts, we can accommodate also the machinery for generating new names. The axiomatization we introduce is quite general and should be easily portable to other formalizations based on higher order syntax. The use of coinductive types and corresponding tactics allows to give alternative, and possibly more natural, proofs of many properties of strong late bisimilarity, w.r.t. those originally given by Milner, Parrow, and Walker.

Introduction

The goal of this paper is to present an interactive proof assistant for Milner's π-calculus, based on a *higher-order abstract syntax* (HOAS) presentation of the calculus in the Intuitionistic Coinductive Type Theory $\mathrm{CC}^{(Co)Ind}$ [2]. The utility and flexibility of this proof assistant is illustrated by developing the formal theory of *strong late bisimilarity* as presented in Section 2 of *A calculus of mobile processes* by Milner, Parrow, and Walker [25].

The π-calculus is a process algebra which models communicating systems which can dynamically change the topology of the channels. It is widely accepted as a theoretical model for concurrency, and is intended to play the same rôle the λ-calculus plays for functional programming. Like many other processes algebras, however, the π-calculus is a rather intricate formal system in itself.

*Work partially supported by MURST-97 *Tecniche formali per la specifica, l'analisi, la verifica, la sintesi e la trasformazione di sistemi software* and Esprit WG-21900 *TYPES*.

†Dipartimento di Matematica e Informatica, Università degli Studi di Udine. Via delle Scienze, 206, 33100, Udine, Italy. `mailto:{honsell|miculan|scagnett}@dimi.uniud.it`

It is very easy to overlook some detail when carrying out by hand even an elementary derivation. Informal arguments for equivalence of processes are error prone, since exhaustive case analyses on possible transitions are elusive. Hence, this calculus, as all the field of formal methods for reasoning on concurrent systems, seems particularly in need of practical *computer assisted proof editors*.

Building an implementation of a particular version of the π-calculus, and a given notion of process equivalence, from scratch does not seem to be a tenable approach. The variability in the presentations and the wide range of equivalences under consideration, do not make it worth to put so much effort in an enterprise which is so particular. Hence, in order to achieve more generality, we have explored the possibility of tailoring a *generic* proof editor to the π-calculus using an HOAS encoding, which subsumes a number of different presentations, and which can serve as a paradigm for the plethora of π-calculi.

A *generic proof development environment* is a proof development environment for a particular system which can play the rôle of a logic specification language, *i.e.* of a *Logical Framework* [18, 12]. Since the '80's, higher order predicative, or impredicative, intuitionistic type theories have been successfully experimented as Logical Frameworks [12, 1, 6, 22]. In these theories one can represent (formalize) faithfully and uniformly all the relevant notions and aspects of the inference process in an arbitrary system: syntactic categories, terms, assertions, axiom schemata, rule schemata, tactics. The basic idea is the "judgements-as-types, λ-terms-as-proofs" paradigm.

Nowadays we have a very good choice of generic proof development environments based on type theory, *e.g.* NuPRL, Alf, Lego, Coq, Isabelle, which implement respectively Martin-Löf Type Theory, the Calculus of (Co)Inductive Constructions, and Higher Order Logic [31, 28, 2, 27].

Following this line of research, our interactive proof assistant for the π-calculus, is obtained directly from Coq [2], which is the full-fledged computerized proof environment for $\mathrm{CC}^{(Co)Ind}$, developed at INRIA. The crucial step is the specification in $\mathrm{CC}^{(Co)Ind}$ of an *adequate* encoding of the π-calculus. Our encoding builds upon a higher order abstract syntax (HOAS) presentation of the π-calculus, in the style of Church, whereby binding operators are represented by constructors of higher order type [12, 6, 22].

We think that we offer a substantial case study in computer assisted formal reasoning which is significant for various reasons.

First, we illustrate, building on ideas in [14], how even a complex formal system such as π-calculus can benefit from an HOAS presentation. This allows to delegate conveniently to the metalanguage the details of many syntactic manipulations, which, in the case of the π-calculus, include, *e.g.* α-conversion of binders, substitution of names and the machinery for generating fresh local names. Previous approaches in the literature, to the implementation of the π-calculus, had adopted either a direct first-order encoding, or had dropped names *tout court* in favour of de Brujin indexes [13, 20]. In both approaches one then needs to implement explicitly the machinery for dealing with names. The user is hence overwhelmed by technical details and lemmata about α-equivalence, free names operators, substitution functions and so on. In [13], 600 out of 800 proved lemmata concern the technical details of index handling. On the other hand, we delegate all these to the functional metalanguage. But moreover, HOAS allows us to construe many phrases of the system in terms of *new* binding operators. This is how we treat, for instance, bound transitions in the π-calculus. The system thus appears in a purified form. One can clearly single out the *essential* idiosyncrasies in the treatment of bound names, and explain away the many inessential side conditions which normally infest its rules.

Secondly, following the approach pioneered by Coquand and Gimenez [3, 11, 10], we investigate the possibility of reasoning on infinite and circular objects using proposition-as-coinductive types. Thus, coinduction proofs are rendered as infinite proof objects, and bisimulations need not be exhibited at the outset, but can be built incrementally using natural tactics. This approach turns out to be extraordinarily successful. However, in order to provide more elbow room in carrying out proofs, we represent process equivalence also in a more traditional axiomatic way, and we prove formally the equivalence between the two approaches.

The use of HOAS is critical when reflecting on those properties which HOAS delegates to the metalanguage, namely, substitution, α-conversion and freshness of names. This is the case of many of the Lemmata and propositions in [25, Section 2]. In order to prove such properties, we need to extend the encoding with new postulates which reify at the object level, some of the details that HOAS delegates at the metalanguage. These postulates are intended to capture, in a natural way, the basic theory of contexts, when using HOAS. They are quite general and hence easily portable to other encodings based on HOAS. They have been formulated with the attitude of committing ourselves in the least possible way, while still being able to carry out the operations normally carried out with "real life" contexts. This "minimalistic attitude" is the one we should normally have in applying the axiomatic method.

We have tried to isolate a restricted set of general properties capturing a general theory of contexts. These should be quite general and portable to other encodings based on HOAS.

This paper is part of an ongoing research programme at the Computer Science Department of the University of Udine on proof editors, started in 1992 [21, 15, 22], based on HOAS encodings in dependent typed λ-calculus for program logics[1]. Our experience, which is nicely confirmed also in dealing with π-calculus, is that Logical Frameworks allow to encode faithfully the formal systems under consideration, without imposing on the user of the proof editor the burden of cumbersome encodings. We can honestly say that we have a real, user-friendly interactive system for proving bisimilarities of processes.

Proof editors and Logical Frameworks are still under development however. We think that their construction will benefit from extensive case studies and applications, like the one presented here.

[1] Some of the material contained in this paper is the object of three Laurea thesis at the University of Udine [32, 8, 34].

Structure of this paper. In Section 1 we recall the theory of (monadic) π-calculus and the theory of strong late bisimilarity. In Section 2 we briefly discuss the basic ideas underlying (co)inductive type theory and logical Frameworks. We give also a short introduction to the Coq proof assistant. The encodings of the syntax of the π-calculus, of its operational semantics and of strong late bisimilarity appear in Section 3. In Section 4 the formal verification, carried out in Coq, of the theory of strong late bisimilarity presented in [25, Section 2] is discussed and compared to its "informal" counterpart. Conclusions, comparison with related work, and directions for future work appear in Section 5.

Complete Coq signatures, the lists of statements formally proved, and excerpts from verification sessions appear in the Appendix A.

Throughout the paper we use freely notions and definitions of type theory, Coq and π-calculus. The user can refer to [2, 12, 25] for more details.

1 The π-calculus

In this section we introduce briefly the π-calculus, see [25] for more details. In particular, we introduce the syntax of the language, the *late* operational semantics, and the relation of *strong late* bisimilarity.

In the π-calculus there are only two primitive entities: *names* and *processes* (or *agents*). Let $\mathcal{N}$ be a infinite set of names, ranged over by x, y. The set of processes $\mathcal{P}$, ranged over by P, Q, is defined by the following abstract syntax:

$$P ::= 0 \mid \bar{x}y.P \mid x(y).P \mid \tau.P \mid (\nu x)P \mid !P \mid P_1|P_2 \mid P_1 + P_2 \mid [x = y]P \mid [x \neq y]P$$

The operators are listed in decreasing order of precedence. The *input prefix* operator $x(y)$ and the *restriction* operator (νx) bind the occurrences of y in $x(y).P$ and $(\nu y)P$ respectively. Thus, for each process P we can define the sets of its *free names* $fn(P)$, *bound names* $bn(P)$ and *names* $n(P) \stackrel{\text{def}}{=} fn(P) \cup bn(P)$. Alpha equivalence of processes is defined as expected, and it is denoted by $\equiv_\alpha$. Let $X \subset \mathcal{N}$; $\mathcal{P}_X$ denotes the set $\{P \subset \mathcal{P} \mid fn(P) \subseteq X\}$. Traditionally, processes are *not* taken up to α-equivalence. Capture-avoiding substitution of a single name y in place of x in P is denoted by $P\{y/x\}$; we do not need simultaneous substitutions, because we shall not consider identifiers and definitional equations.

The above language is the language of π-calculus originally introduced in [25], apart from the *replication* operator "!" in place of identifiers and recursive rules, and the presence of the *mismatch* operator "$\neq$." We refer to [25] for an intuitive explanation of the meaning of the basic constructs. The process $!P$ behaves essentially as $P|P|P|\ldots$; it can be used to implement recursive processes. The process $[x \neq y]P$ behaves as P if x and y are different names, and as 0 otherwise. This operator is particularly useful in applications; in Section 3 we will see that its encoding raises some interesting issues.

There are four actions in π-calculus, defined by the syntax $\alpha ::= \tau \mid x(z) \mid \bar{x}y \mid \bar{x}(z)$. Their intuitive meaning is the following:

1. "silent" action: $P \xrightarrow{\tau} Q$ means that P can reduce itself to Q without interacting with other processes;

2. "free output:" $P \xrightarrow{\bar{x}y} Q$ means that P can reduce itself to Q emitting the name y on the channel x;

3. "input:" $P \xrightarrow{x(z)} Q$ means that P can receive from the channel x any name w and then evolve into $Q\{w/z\}$;

4. "bound output:" $P \xrightarrow{\overline{x}(z)} Q$ means that P can evolve into Q emitting on the channel x a name z, which is bound in P.

The channel x is called the *subject*, while z, y are the *objects*, or *parameters*. The functions $fn(\cdot)$ and $bn(\cdot)$ are extended to actions, as follows:

$$\begin{array}{ll} fn(x(z)) = fn(\bar{x}(z)) = \{x\} & fn(\bar{x}y) = \{x, y\} \\ fn(\tau) = bn(\tau) = bn(\bar{x}y) = \emptyset & bn(x(z)) = bn(\bar{x}(z)) = \{z\} \end{array}$$

As usual, $n(\alpha) \stackrel{\text{def}}{=} fn(\alpha) \cup bn(\alpha)$. The τ and free output actions are called *free*, the remaining ones are called *bound*.

The operational semantics of π-calculus can be given either as a labelled transition system (LTS), as in [25], or as a reduction system together with a *structural congruence* relation over processes, which identifies processes up to some "syntactic" detail (e.g., α-conversion, monoidal laws of the sum and parallel composition,...), as in [23]. In our encoding of the π-calculus in view of its use as a specification for an interactive proof development editor, we find it more convenient to give an LTS semantics. Congruence rules, in fact, are problematic from the point of view of top-down proof search—see Section 4.4 for more details.

There is a plethora of slightly different labelled transition systems for the late operational semantics of π-calculus, e.g. [25, 23, 33]. Here, we present the original one in [25]: the relation $\xrightarrow{\alpha}$ is the smallest relation over processes, satisfying the rules in Figure 1.

There are a number of different notions of observational equality that can be considered over processes. We shall focus only on *strong late bisimilarity*, which is the one most extensively discussed in the original paper on π-calculus [25]. It is defined as follows:

Definition 1 (Strong Late Bisimilarity) *A binary relation $\mathcal{S}$ on processes is a* strong late simulation *iff, for all P, Q processes, if $P\ \mathcal{S}\ Q$ then*

1. *if $P \xrightarrow{\alpha} P'$ and α is a free action, then for some Q', $Q \xrightarrow{\alpha} Q'$ and $P'\ \mathcal{S}\ Q'$;*

2. *if $P \xrightarrow{x(y)} P'$ and $y \notin n(P, Q)$, then for some Q', $Q \xrightarrow{x(y)} Q'$ and for all $w \in \mathcal{N}$: $P'\{w/y\}\ \mathcal{S}\ Q'\{w/y\}$;*

3. *if $P \xrightarrow{\overline{x}(y)} P'$ and $y \notin n(P, Q)$, then for some Q', $Q \xrightarrow{\overline{x}(y)} Q'$ and $P'\ \mathcal{S}\ Q'$.*

$\mathcal{S}$ is a strong late bisimulation *if both $\mathcal{S}$ and $\mathcal{S}^{-1}$ are strong late simulations.*

The strong late bisimilarity *is the binary relation $\dot{\sim}$ defined by*

$$P \dot{\sim} Q \iff \exists \mathcal{S}. \mathcal{S} \text{ strong late bisimulation and } (P\ \mathcal{S}\ Q)\,.$$

It is well-known that strong late bisimilarity can be defined as the greatest fixed point of a suitable monotonic operator over subsets of $\mathcal{P} \times \mathcal{P}$.

$$
\begin{array}{llll}
\text{IN} \ \dfrac{-}{x(z).P \xrightarrow{x(w)} P\{w/z\}} \ w \notin fn((\nu z)P) & & \text{OUT} \ \dfrac{-}{\overline{x}y.P \xrightarrow{\overline{x}y} P} \\
\text{PAR}_1 \ \dfrac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \ bn(\alpha) \cap fn(Q) = \emptyset & & \text{SUM}_1 \ \dfrac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'} \\
\text{PAR}_2 \ \dfrac{Q \xrightarrow{\alpha} Q'}{P|Q \xrightarrow{\alpha} P|Q'} \ bn(\alpha) \cap fn(P) = \emptyset & & \text{SUM}_2 \ \dfrac{Q \xrightarrow{\alpha} Q'}{P+Q \xrightarrow{\alpha} Q'} \\
\text{COM}_1 \ \dfrac{P \xrightarrow{x(z)} P' \quad Q \xrightarrow{\overline{x}y} Q'}{P|Q \xrightarrow{\tau} P'\{y/z\}|Q'} & & \text{COM}_2 \ \dfrac{P \xrightarrow{\overline{x}y} P' \quad Q \xrightarrow{x(z)} Q'}{P|Q \xrightarrow{\tau} P'|Q'\{y/z\}} \\
\text{RES} \ \dfrac{P \xrightarrow{\alpha} P'}{(\nu y)P \xrightarrow{\alpha} (\nu y)P'} \ y \notin n(\alpha) & & \text{REPL} \ \dfrac{P|!P \xrightarrow{\alpha} Q}{!P \xrightarrow{\alpha} Q} \\
\text{MATCH} \ \dfrac{P \xrightarrow{\alpha} P'}{[x=x]P \xrightarrow{\alpha} P'} & & \text{MISMATCH} \ \dfrac{P \xrightarrow{\alpha} P'}{[x \neq y]P \xrightarrow{\alpha} P'} \ x \neq y \\
\text{OPEN} \ \dfrac{P \xrightarrow{\overline{x}y} P'}{(\nu y)P \xrightarrow{\overline{x}(w)} P'\{w/y\}} \ \begin{array}{l} y \neq x \\ w \notin fn((\nu y)P') \end{array} & & \text{TAU} \ \dfrac{-}{\tau.P \xrightarrow{\tau} P} \\
\text{CLOSE}_1 \ \dfrac{P \xrightarrow{x(w)} P' \quad Q \xrightarrow{\overline{x}(w)} Q'}{P|Q \xrightarrow{\tau} (\nu w)(P'|Q')} & & \text{CLOSE}_2 \ \dfrac{P \xrightarrow{\overline{x}(w)} P' \quad Q \xrightarrow{x(w)} Q'}{P|Q \xrightarrow{\tau} (\nu w)(P'|Q')}
\end{array}
$$

Figure 1: Late operational semantics of π-calculus.

2 (Co)Inductive Type Theories as Logical Frameworks

In this section we present succinctly the logical and technological tools which will be used in the following sections. More specifically, we shall recall the main features of the proof assistant Coq, which is based on the type theory called *Calculus of (Co)Inductive Constructions* (briefly, $\mathrm{CC}^{(Co)Ind}$), introduced by Coquand and Huet and further extended by Paulin and Giménez. We refer the reader to [2] for further details.

In this paper we utilize the Calculus of (Co)Inductive Constructions as a *Logical Framework* [12], hence we shall end this section with a short discussion of the basic ideas of generic logical specification languages.

2.1 (Co)Inductive Type Theory

$\mathrm{CC}^{(Co)Ind}$ is an impredicative intuitionistic type theory, with dependent inductive and coinductive types. Formally, it is a system for deriving assertions of the shape $\Gamma \vdash_\Sigma M : T$, where Γ is a list of type assignments to variables, i.e. $x_1 : t_1, \ldots, x_n : t_n$; Σ is the *signature* (i.e, a list of typed constants); M is a λ-term and T is its type.

Using the *propositions-as-types, λ-terms-as-proofs* paradigm, $\mathrm{CC}^{(Co)Ind}$ can be viewed as a system for representing assertions of a higher-order intuitionistic logic and their proofs. This implies that valid assertions correspond to *inhabited* types (i.e., types for which there exists a *closed* term of that type), and moreover proof checking corresponds to type checking.

For lack of space, we shall not describe in detail the rich language of terms and types of $\mathrm{CC}^{(Co)Ind}$ or its properties. We shall only point out the important property of $\mathrm{CC}^{(Co)Ind}$, namely that checking whether a given term has a given type in $\mathrm{CC}^{(Co)Ind}$ is decidable. This is the crucial property which makes it possible to use $\mathrm{CC}^{(Co)Ind}$ as the core of a proof checker.

We shall now discuss briefly some features of $\mathrm{CC}^{(Co)Ind}$. *Simple inductive types* can be defined as follows:

`Inductive` ***ident*** `:` ***term*** `:=` ***ident1*** `:` ***term1*** `| ... |` ***identn*** `:` ***termn.***

The name *ident* is the name of the inductively defined object, and *term* is its type. The constructors of *ident* are *ident1*,...,*identn*, whose types are *term1*,...,*termn*, respectively. For instance, the set of natural numbers is defined as `Inductive nat :  Set := O : nat | S : nat -> nat.`

Types of constructors have to satisfy a *positivity condition*, which, roughly, requires that *ident* may occur only in strictly positive positions in the types of the arguments of *ident1*,...,*identn*. This condition ensures the soundness of the definition; for further details, see [2]. For instance, the following definition is not accepted:

```
Inductive D : Set := lam : (D -> nat) -> D.
```

Inductive definitions automatically provide *induction* and *recursion* principles over the defined type. These principles state that elements of the type are only those built by the given constructors. For instance, the automatically generated induction principle for `nat` is the well-known Peano principle:

```
nat_ind : (P:nat->Prop)(P O) ->
                       ((n:nat)(P n)->(P (S n))) -> (n:nat)(P n)
```

Objects of inductive types are well-founded, that is, they are always built by a finite unlimited number of constructors. *Coinductive* types arise by relaxing this condition: coinductive objects, in fact, can be *non-wellfounded*, in that they can have an infinite number of constructors in their structure. Hence, coinductive objects are specified by means of non-ending (but effective) processes of construction, expressed as "circular definitions." For example, the set of streams of natural numbers, is defined as

```
CoInductive  Stream : Set := seq : nat -> Stream -> Stream.
```

and the stream of all zeros is given by

```
CoFixpoint allzeros : Stream := (seq O allzeros).
```

Of course, since coinductive types are non-wellfounded, they do not have any induction principle. The only way for manipulating coinductive objects is by means of *case analysis* on the form of the outermost constructor. In order to ensure soundness of corecursive definitions, these have to satisfy a *guardedness condition* [2, 11, 10]. Roughly, the constant being defined may appear in the defining equation only within an argument of some of its constructors. "Short-circuit" definitions like `CoFixpoint X : Stream := X.` are not allowed.

An interesting possibility arises in $CC^{(Co)Ind}$, in connection with the propositions-as-types paradigm, due to the fact that proofs are first-class objects. Coinductive predicates can be rendered as coinductive types, and then these are propositions which have infinitely long (or circular) proofs. The guardedness condition on the well-formedness of infinite objects allows to make sense of such infinitely regressing proof arguments. One can consistently assume his thesis as a hypothesis provided its applications appear in the proof only when *guarded* by a constructor of the corresponding type. This is the propositional version of the *guarded induction principle* introduced by Coquand and Giménez [3, 11, 10] for reasoning on coinductive objects, and it is the constructive counterpart of coinductive proofs.

See Appendix A.9 for an extended example of how to use guarded induction in proof search.

2.2 The Coq proof assistant

Coq is an interactive proof assistant for the type theory $CC^{(Co)Ind}$, developed by the INRIA and other institutes. For a complete description, we refer to [2] and to the online documentation at `http://pauillac.inria.fr/coq/`. More specifically, Coq is an editor for interactively searching for an inhabitant of a type, in a top-down fashion by applying tactics step-by-step, backtracking if needed, and for verifying correctness of typing judgements.

Coq's specification language, `Gallina`, allows to express the type theory $CC^{(Co)Ind}$ in pure ASCII text, as follows:

$\lambda x : M.N$ is written `[x:M]N`	$\prod_{x:M} N$ is written `(x:M)N`
$(M\ N)$ is written `(M N)`	$M \to N$ is written `M -> N`

We will not give an independent syntax for $CC^{(Co)Ind}$, but we will use its `Gallina` formulation.

Given a signature written in `Gallina`, a proof search starts by entering

```
Lemma ident : goal.
```

where *goal* is the type representing the proposition to prove. At this point, Coq waits for commands from the user, in order to build the proof term which inhabits *goal* (i.e., the proof). To this end, Coq offers a rich set of *tactics*, e.g., introduction and application of assumptions, application of rules and previously proved lemmata, elimination of inductive objects, inversion of (co)inductive hypotheses and so on. These tactics allow the user to proceed in his proof search much like he would do informally. At every step, the type checking algorithm ensures the soundness of the proof. When the proof term is completed, it can be saved (by the command `Qed`) for future applications.

2.3 Logical Frameworks

Type Theories, such as the Edinburgh Logical Framework [12, 1] or the Calculus of (Co)Inductive Constructions [2] were especially designed, or can be fruitfully used, as a general logic specification language, i.e. as a Logical Framework (LF). In an LF, we can represent faithfully and uniformly all the relevant concepts of the inferential process in a logical system (syntactic categories, terms, variables, contexts, assertions, axiom schemata, rule schemata, instantiation, tactics, etc.) via the "judgements-as-types λ-terms-as-proofs" paradigm.

The key concept for representing assertions and rules is that of Martin-Löf's *hypothetico-general* judgement [18], which is rendered as a type of the dependent typed λ-calculus of the Logical Framework. The λ-calculus metalanguage of an LF supports *higher order abstract syntax* (HOAS) *à la* Church, i.e., syntax where language constructors may have higher order types. Using HOAS, *substitution*, α-conversion of bound variables and *instantiation of schemata* can be safely taken care of uniformly by the metalanguage [30].

Since LF's allow for higher order assertions (*judgements*) one can treat on a par axioms and rules, theorems and derived rules, and hence encode also generalized natural deduction systems in the sense of [35].

Encodings in LF's often provide the "normative" formalization of the system under consideration. The specification methodology of LF's, in fact, forces the user to make precise all tacit, or informal, conventions, which always accompany any presentation of a system.

Any interactive proof development environment for the type theoretic metalanguage of an LF (e.g. Coq [2], LEGO [31]), can be readily turned into one for a specific logic. We need only to fix a suitable environment (the *signature*), i.e. a declaration of typed constants corresponding to the syntactic categories, term constructors, judgements, and rule schemata. Such a generated editor allows the user to reason "under assumptions" and go about in developing a proof the way mathematicians normally reason: using hypotheses, formulating conjectures, storing and retrieving lemmata, often in top-down, goal-directed fashion.

3 Σ^π: A HOAS Formalization of π-calculus

In this section we describe Σ^π, a signature which encodes the theory of π-calculus. The formal proof development environment that it induces in Coq is adequate for reasoning *in* the π-calculus, i.e., for representing every processes and proving correctness of their transitions and equivalences. In Section 4.2 we will introduce a signature $\Sigma^{\pi+}$, extending Σ^π, appropriate also for meta-reasoning *on* the π-calculus.

We will present Σ^π in three stages: in Section 3.1 we describe the signature Σ_1^π, which encodes the syntax of processes; in Section 3.2 we extend Σ_1^π to Σ_2^π, which allows for the encoding of the labelled transition system. Finally, in Section 3.3 we further extend Σ_2^π to Σ^π so as to represent the strong late equivalence $\dot\sim$.

3.1 Encoding the syntax of the language

In order to take best advantage of the features of Coq, we use HOAS and inductive definitions as much as possible.

The first declaration in the signature Σ^π corresponds to $\mathcal{N}$, the set of names:

```
Parameter name : Set.
```

In view of the fact that binding operators are represented by higher-order terms, we cannot take `name` to be inductive. Otherwise, non-relevant, "exotic" parasite terms would arise [6, 22].

As far as the encoding of the theory of π-calculus is concerned, no other property about `name` is needed (Some properties will be eventually needed when dealing with the metatheory). The set `name` has no constructors; hence, names of π-calculus are represented by *variables* of $\mathrm{CC}^{(Co)Ind}$ of type `name`.

The next declaration in Σ^π is the inductive type representing the set of processes $\mathcal{P}$:

```
Inductive proc : Set :=
   nil      : proc
 | bang     : proc -> proc
 | tau_pref : proc -> proc
 | par      : proc -> proc -> proc
 | sum      : proc -> proc -> proc
 | nu       : (name -> proc) -> proc
 | match    : name -> name -> proc -> proc
 | mismatch : name -> name -> proc -> proc
 | in_pref  : name -> (name -> proc) -> proc
 | out_pref : name -> name -> proc -> proc.
```

Names of constructors of the above type recall the operators they encode; the suffix `_pref` in `tau_pref`, `in_pref` and `out_pref` distinguishes the latter from the analogous constructors for actions (see Section 3.2).

Following the principles of HOAS, we encode the binding operators ν and $x(\cdot)$ as functions over the higher-order type `name->proc`. This allows us to delegate directly to the metalanguage of $\mathrm{CC}^{(Co)Ind}$ α-conversion and capture-avoiding substitution. Hence, we do not need to implement explicitly an α-conversion mechanism, as instead is done in [13, 20] (see Section 5). Actually, the ν can be seen as "half λ": both ensure the *freshness* of bound names with respect to existing ones, but there is no notion of substitution on names bound by ν.

The encoding outlined above is not the only possible one—e.g., following a "pure LF" approach like in [14] (which inspired ours), we could encode the syntax of π-calculus without using any `Inductive` definitions, by taking each constructor as an `Axiom`atized constant. However, in order to deal with strong late equivalences between processes in a "pure LF" approach, we have then to axiomatize the *discrimination* and *injection* principles for process constructors,[2] as well as the induction principle over processes. On the other hand, such principles are immediately made available by the mechanism of inductive definitions in Coq.

Let Σ^π_1 be the signature consisting of the declarations given up to now in this section, and for $X = \{x_1, \ldots, x_n\} \subset \mathcal{N}$, let $\Gamma_X \stackrel{\mathrm{def}}{=} \mathtt{x_1} : \mathtt{name}, \ldots, \mathtt{x_n} : \mathtt{name}$, and

[2]For instance, the discrimination principle between "+" and "|" is $\forall P, Q \in \mathcal{P} : P + Q \neq P|Q$; the injection principle for "!" is $\forall P, Q \in \mathcal{P} : !P = !Q \Rightarrow P = Q$ (where $=$ is the syntactic equality, rendered in Coq by Leibniz equivalence). Of course, there is a discrimination principle for each pair of different constructors, and an injection principle for each constructor but 0.

$$
\begin{array}{rcl}
\varepsilon_X & : & \mathcal{P}_X \to \mathtt{proc}_X \\
\varepsilon_X(0) & = & \mathtt{nil} \\
\varepsilon_X(!P) & = & (\mathtt{bang}\ \varepsilon_X(P)) \\
\varepsilon_X(\tau.P) & = & (\mathtt{tau\_pref}\ \varepsilon_X(P)) \\
\varepsilon_X(P|Q) & = & (\mathtt{par}\ \varepsilon_X(P)\ \varepsilon_X(Q)) \\
\varepsilon_X(P+Q) & = & (\mathtt{sum}\ \varepsilon_X(P)\ \varepsilon_X(Q)) \\
\varepsilon_X((\nu x)P) & = & (\mathtt{nu}\ [\mathtt{x:name}]\varepsilon_{X,x}(P)) \\
\varepsilon_X([x=y]P) & = & (\mathtt{match\ x\ y}\ \varepsilon_X(P)) \\
\varepsilon_X([x\neq y]P) & = & (\mathtt{mismatch\ x\ y}\ \varepsilon_X(P)) \\
\varepsilon_X(x(y).P) & = & (\mathtt{in\_pref\ x}\ [\mathtt{y:name}]\varepsilon_{X,y}(P)) \\
\varepsilon_X(\bar{x}y.P) & = & (\mathtt{out\_pref\ x\ y}\ \varepsilon_X(P))
\end{array}
$$

$$
\begin{array}{rcll}
\delta_X & : & \mathtt{proc}_X \to \mathcal{P}_X & \\
\delta_X(\mathtt{nil}) & = & 0 & \\
\delta_X(\mathtt{bang\ t}) & = & !\delta_X(\mathtt{t}) & \\
\delta_X(\mathtt{tau\_pref\ t}) & = & \tau.\delta_X(\mathtt{t}) & \\
\delta_X(\mathtt{par}\ \mathtt{t}_1\ \mathtt{t}_2) & = & \delta_X(\mathtt{t}_1)|\delta_X(\mathtt{t}_2) & \\
\delta_X(\mathtt{sum\ t1\ t2}) & = & \delta_X(\mathtt{t}_1)+\delta_X(\mathtt{t}_2) & \\
\delta_X(\mathtt{nu\ t}) & = & (\nu z)\delta_{X,z}(\mathtt{t\ z}) & z = \mathit{fresh}(X) \\
\delta_X(\mathtt{match\ x\ y\ t}) & = & [x=y]\delta_X(\mathtt{t}) & \\
\delta_X(\mathtt{mismatch\ x\ y\ t}) & = & [x\neq y]\delta_X(\mathtt{t}) & \\
\delta_X(\mathtt{in\_pref\ x\ t}) & = & x(z).\delta_{X,z}(\mathtt{t\ z}) & z = \mathit{fresh}(X) \\
\delta_X(\mathtt{out\_pref\ x\ y\ t}) & = & \bar{x}y.\delta_X(t) &
\end{array}
$$

Figure 2: The encoding and decoding functions for the syntax of π-calculus.

let $\mathtt{proc}_X \stackrel{\text{def}}{=} \{\mathtt{t} \mid \Gamma_X \vdash \mathtt{t} : \mathtt{proc},\ \mathtt{t}\ \text{canonical}\}$ (where by "canonical term" we mean essentially long $\beta\eta$-head normal form, in the sense of [12]). The *encoding* and *decoding* functions $\varepsilon_X^{\Sigma_1^\pi}$, $\delta_X^{\Sigma_1^\pi}$ can be defined as in Figure 2 (for sake of simplicity, in the following we will drop the exponents). The map $\mathit{fresh} : \mathcal{P}_{<\omega}(\mathcal{N}) \to \mathcal{N}$ is a fixed "fresh name selection" function; the only condition *fresh* has to satisfy is that for any $X \subset \mathcal{N}$ finite, $\mathit{fresh}(X) \notin X$. A possible definition for *fresh* is the following: given an enumeration n_i of $\mathcal{N}$, let $\mathit{fresh}(X) = n_{\max\{i \mid n_i \in X\}+1}$. Our encoding of processes is faithful to the original system in the sense formalized by the following Theorem:

Theorem 1 (Adequacy of syntax) *For $X \subset \mathcal{N}$ finite:*

1. *ε_X is a compositional surjection from $\mathcal{P}_X$ to $\mathtt{proc}_X$;*
2. *δ_X is a compositional injection from $\mathtt{proc}_X$ to $\mathcal{P}_X$;*
3. *$\varepsilon_X \circ \delta_X = id_{\mathtt{proc}_X}$*
4. *$\forall P \in \mathcal{P}_X.(\delta_X \circ \varepsilon_X)(P) \equiv_\alpha P$*

Proof. Standard, using induction on the structure of processes and of normal forms of type $\mathtt{proc}$. □

As a consequence of the above Theorem, we have that α-equivalent processes are encoded by the same λ-term:

Corollary 1 *For all* $P_1, P_2 \in P_X$*:* $P_1 \equiv_\alpha P_2 \iff \varepsilon_X(P_1) = \varepsilon_X(P_2)$*.*

This can be considered to be a slightly non-standard feature of our presentation of the π-calculus. In fact, in the original system [25], α-equivalent processes are taken to be distinct. It is a matter of discussion whether this is essential.

Finally, we introduce in the signature two inductive predicates `isin` and `notin`, which reflect at the level of the language the metatheoretic properties of *occurrence* and *non occurrence* of a variable; their Coq code is reported in Appendix A.1. Roughly, `(isin x p)` holds iff the name `x` occurs free within the process `p`; dually for `notin`, which encodes the "freshness" of names in processes. It is interesting to point out that such predicates would not be needed if we were interested only in reasoning on the evaluation of processes without "mismatch" operators [14].

3.2 Encoding Late Operational Semantics

In this section we describe the encoding of the transition system in Figure 1. Differently from other approaches [13, 20], our encoding of the transition relation is higher-order and it follows the one in [14]. Here, moreover, we take advantage also of the inductive features of $\mathrm{CC}^{(Co)Ind}$.

The transition relation is rendered by two mutually defined inductive predicates `ftrans`, `btrans`, which take care of transitions involving free actions and bound actions, respectively. In the latter case, the result of the transition is not a process but a *process context*, i.e. a process with a hole, conveniently represented by a function `name->proc`. Their arity is therefore the following (the complete code appears in Section A.2):

```
Mutual Inductive ftrans : proc -> f_act -> proc -> Prop := ...
with   btrans : proc -> b_act -> (name -> proc) -> Prop := ...
```

where `f_act`, `b_act` are two inductive sets representing free actions and bound actions, respectively:

```
Inductive f_act:Set := tau : f_act | Out : name -> name -> f_act.
Inductive b_act:Set := In : name -> b_act | bOut : name -> b_act.
```

Notice that constructors of the sets defined above are all first-order, that is they do not bind any name in actions. In bound actions, only free names are mentioned, bound names (*i.e.*, the *objects* [25]) are represented indirectly as the "holes" of the result process of the bound transition.

Our choice of formalizing the transition relation by means of two predicates forces us to duplicate the rules (schemas) of Figure 1 which involve the schematic variable α. We have to formalize both a version for `ftrans` and one for `btrans`.

In our view, HOAS leads to a substantial clarification of the original syntax of π-calculus. Indeed, most of the side conditions in Figure 1 do not need to be explicitly encoded since they are automatically taken care of at the metalevel.

In our view, this use of HOAS lets us focus on the essence of π-calculus, and do away with tedious and unnatural bureaucratical details concerning names. This is the case, for instance, of the *scope extrusion* rule (CLOSE_2), which using first order syntax can be given in many different, albeit equivalent, forms, such as [25, rule (9)]. HOAS allows us to delegate to the metalanguage all freshness and non-occurrence issues, hence the formalizations of all these rules collapse into the same term of the metalanguage (*viz.*, `CLOSE2` of `ftrans`).

The way the metalanguage deals with fresh variables, however, needs some care. It is true that every time a new variable of type `name` is introduced in the proof environment, it is automatically chosen, by the metalanguage, to be different from any other pre-existent variable. However, this fact is not explicit ("known") at the object level. This information is not necessary in many uses of HOAS, *e.g.* in first order logic or in evaluating processes of π-calculus without mismatch. And we do not need any `isin`/`notin` predicates in this case. But in reasoning about many other aspects of π-calculus, such as the mismatch operator, strong late bisimilarity of processes, and especially for reasoning "on" the π-calculus itself, such explicit information is indispensable. In fact, many features and properties of π-calculus deal explicitly with freshness of names. Therefore, we need to reflect ("reify") this information at the object level. We achieve this by introducing freshness hypotheses (i.e., `notin` assumptions) on the locally quantified name. This amounts to strengthening the encoded versions of the rules to match the strength of the original ones. The most complex case is the encoding of the RES rule, which for `ftrans` is as follows:

```
fRES : (p1,p2:name -> proc)(a:f_act)(l:Nlist)
       ((y:name)(notin y (nu p1)) -> (notin y (nu p2)) ->
                (Nlist_notin y l) -> (f_act_notin y a) ->
                (ftrans (p1 y) a (p2 y)))
      -> (ftrans (nu p1) a (nu p2))
```

When we apply this rule, the locally bound variable `y` is automatically chosen different from all other variables. In order to reflect this fact at the object level, however, we need to introduce four extra assumptions. Those mentioning `notin` and `f_act_notin` state that `y` does not appear in `p1`, `p2` and `a`. The assumption `(Nlist_notin y l)` allows us to specify a finite set (actually, a list) of names the variable `y` has to differ from. This is sound because `y` is locally bound *after* `l`, and therefore `y` differs from all variables in `l`.

Of course, if we do not consider the mismatch operator or we are not interested in discussing in Coq strong late bisimilarity, the extra hypotheses can be safely dropped as in [14].

As was the case for the encoding of the syntax of the language in Section 3.1, also in the case of the operational semantics, the use of a Coq inductive definition is profitable. For instance, elimination and inversion tactics are immediately made available to us. Of course, at the price of some extra encoding or more elaborate proof search, the operational semantics could have been encoded also in a "pure LF" approach or using second order quantification.

Let Σ_2^π be the signature declared so far; it is adequate in the sense given by the following theorem:

Theorem 2 (Adequacy of `ftrans`, `btrans`)
Completeness: *Let $X \subset \mathcal{N}$ finite, $x, y \in \mathcal{N}$;*

1. *For all $P_1, P_2 \in \mathcal{P}_X$, if $P_1 \xrightarrow{\tau} P_2$ then there exists* `t` *canonical such that* $\Gamma_X \vdash_{\Sigma_2^\pi} \mathtt{t} : (\mathtt{ftrans}\ \varepsilon_X(P_1)\ \mathtt{tau}\ \varepsilon_X(P_2))$*;*

2. *For all $P_1, P_2 \in \mathcal{P}_X$, if $P_1 \xrightarrow{\bar{x}y} P_2$ then there exists* `t` *canonical such that* $\Gamma_X \vdash_{\Sigma_2^\pi} \mathtt{t} : (\mathtt{ftrans}\ \varepsilon_X(P_1)\ (\mathtt{Out\ x\ y})\ \varepsilon_X(P_2))$*;*

3. *For all $P_1 \in \mathcal{P}_X$ and $P_2 \in \mathcal{P}_{X,y}$, if $P_1 \xrightarrow{x(y)} P_2$ then there exists* `t` *canonical such that* $\Gamma_X \vdash_{\Sigma_2^\pi} \mathtt{t} : (\mathtt{btrans}\ \varepsilon_X(P_1)\ (\mathtt{In\ x})\ [\mathtt{y{:}name}]\varepsilon_{X,y}(P_2))$

4. *For all $P_1 \in \mathcal{P}_X$ and $P_2 \in \mathcal{P}_{X,y}$, if $P_1 \xrightarrow{\bar{x}(y)} P_2$ then there exists* `t` *canonical such that* $\Gamma_X \vdash_{\Sigma_2^\pi} \mathtt{t} : (\mathtt{btrans}\ \varepsilon_X(P_1)\ (\mathtt{bOut\ x})\ [\mathtt{y{:}name}]\varepsilon_{X,y}(P_2))$

Soundness: *Let $X \subset \mathcal{N}$ finite, and $x, y \in \mathcal{N}$;*

1. *For all $P_1, P_2 \in \mathcal{P}_X$, if there exists* `t` *canonical such that* $\Gamma_X \vdash_{\Sigma_2^\pi} \mathtt{t} : (\mathtt{ftrans}\ \varepsilon_X(P_1)\ \mathtt{tau}\ \varepsilon_X(P_2))$*, then there exists $P_2' \in \mathcal{P}_X$ such that $P_2' \equiv_\alpha P_2$ and $P_1 \xrightarrow{\tau} P_2'$;*

2. *For all $P_1, P_2 \in \mathcal{P}_X$, if there exists* `t` *canonical such that* $\Gamma_X \vdash_{\Sigma_2^\pi} \mathtt{t} : (\mathtt{ftrans}\ \varepsilon_X(P_1)\ (\mathtt{Out\ x\ y})\ \varepsilon_X(P_2))$*, then there exists $P_2' \in \mathcal{P}_X$ such that $P_2' \equiv_\alpha P_2$ and $P_1 \xrightarrow{\bar{x}y} P_2'$;*

3. *For all $P_1 \in \mathcal{P}_X$, $P_2 \in \mathcal{P}_{X,y}$, if there exists* `t` *canonical such that* $\Gamma_X \vdash_{\Sigma_2^\pi} \mathtt{t} : (\mathtt{btrans}\ \varepsilon_X(P_1)\ (\mathtt{In\ x})\ [\mathtt{y : name}]\varepsilon_{X,y}(P_2))$*, then there exist $z \in \mathcal{N}$ and $P_2' \in \mathcal{P}_{X,z}$ such that $(\nu z)P_2' \equiv_\alpha (\nu y)P_2$ and $P_1 \xrightarrow{x(z)} P_2'$;*

4. *For all $P_1 \in \mathcal{P}_X$, $P_2 \in \mathcal{P}_{X,y}$, if there exists* `t` *canonical such that* $\Gamma_X \vdash_{\Sigma_2^\pi} \mathtt{t} : (\mathtt{btrans}\ \varepsilon_X(P_1)\ (\mathtt{bOut\ x})\ [\mathtt{y : name}]\varepsilon_{X,y}(P_2))$*, then there exist $z \in \mathcal{N}$ and $P_2' \in \mathcal{P}_{X,z}$ such that $(\nu z)P_2' \equiv_\alpha (\nu y)P_2$ and $P_1 \xrightarrow{\bar{x}(z)} P_2'$;*

The proof of this result is by a long induction on the structure of derivations ($\Rightarrow$), and on the structure of normal forms ($\Leftarrow$).

The adequacy result is rather elaborate since the decoding of a successful transition is only "up-to α-equivalence" on the resulting process (see Corollary 1). The root of this awkwardness is in the fact that judgements in the original presentation of π-calculus in [25] are not closed under α conversion. For instance, no transition of the shape $(\nu x)P \xrightarrow{\alpha} (\nu w)P'$, with $x \neq w$ can be derived in that system. But even more subtle failures of closure under α-conversion can arise in connection with rule RES. Consider the formal lemma `weird` in Figure 3.a. A naïve interpretation of `weird` would be the derivation in Figure 3.b, but such a derivation is unsound because in the application of RES, w appears in $x(w)$. However a sound decoding of `weird` is the derivation in Figure 3.c.

a)

```
Variable x : name.
Lemma weird : (btrans (nu [w:name](in_pref x [z:name]nil))
                      (In x)
                      [w:name](nu [w:name]nil)).
Apply bRES with p2:=[u,v:name]nil l:=empty; Intros; Apply IN.
Qed.
```

b) $\boxed{\mathrm{RES}\dfrac{\mathrm{IN}\dfrac{-}{x(z)\xrightarrow{x(w)}0}}{(\nu w)x(z)\xrightarrow{x(w)}(\nu w)0}}$ c) $\boxed{\mathrm{RES}\dfrac{\mathrm{IN}\dfrac{-}{x(z)\xrightarrow{x(z)}0}}{(\nu w)x(z)\xrightarrow{x(z)}(\nu w)0}}$

Figure 3: Failure of closure under α-conversion.

3.3 Encoding of Strong Late Bisimilarity

In this section, we discuss the encoding of strong late bisimilarity of processes. In $\mathrm{CC}^{(Co)Ind}$ this can be done in various ways. We can either define strong late bisimilarity as a coinductive binary predicate over processes, or as the greatest fixed point, *à la* Tarski, of the appropriate operator on binary relations (over processes). Moreover, this latter approach can be carried out either using inductive definitions or using straight higher order logic. One can prove formally in Coq that these three approaches are equivalent as far as provability. However, they differ substantially from the point of view of practical proof search. Coq, in fact, provides different sets of built-in tactics for inductive and coinductive types, and for the case of the pure higher order definition, everything has to be derived from first principles. The three approaches therefore lead to different proof developments. Of course, the two versions of the greatest fixed point approach differ only in the amount of work that the user has to make. We present and discuss in detail the coinductive and the greatest fixed point approach (inductive version), which are the ones for which Coqprovides practical elimination and inversion tactics, and are therefore more appropriate from the point of view of proof development. We prove formally in $\mathrm{CC}^{(Co)Ind}$ their equivalence. In Section 4.3 we shall compare them also from a practical point of view.

3.3.1 The Guarded Approach

A key feature of $\mathrm{CC}^{(Co)Ind}$ (implemented by Coq V5.10 and later versions) with respect to its predecessors (LF [12], CC [4], CIC [29]) is the possibility of defining *coinductive types* [2, 11, 10]. We can take full advantage of this feature by defining directly the coinductive property of strong late bisimilarity. This can be achieved by means of just one `CoInductive` definition, which defines a class of coinductive sets parametrized on pairs of processes (P, Q): for all processes P and Q, it contains the set of all, either finitary or infinitary, proofs of equivalence of P and Q. The formal definition of the coinductive predicate `StBisim` representing strong late bisimilarity is given in Section A.3. Such a predicate has only one introduction rule, `sb`, which is the natural one derived from the definition of strong late bisimilarity (Definition 1): two processes `p,q` are bisimilar if, when one of them makes a transition, also the other does the same, and the reducts are still bisimilar.

Let Σ_3^π be the signature defined so far extended by the declaration given in Section A.3; the soundness of our encoding is given by the following theorem:

Theorem 3 (Soundness) *For all processes P, Q with $fn(P,Q) \subseteq \mathcal{X}$, if there exists a term* $\mathtt{t}$ *such that* $\Gamma_X \vdash_{\Sigma_3^\pi} \mathtt{t} : (\mathtt{StBisim}\ \varepsilon_X(P)\ \varepsilon_X(Q))$, *then P and Q are strongly bisimilar ($P \dot\sim Q$).*

Proof. Let $R \subseteq \mathcal{P} \times \mathcal{P}$ be defined as follows:

$$R \stackrel{\text{def}}{=} \{(P,Q) \mid \text{there exists } \mathtt{t} \text{ such that } \Gamma_X \vdash_{\Sigma_3^\pi} \mathtt{t} : (\mathtt{StBisim}\ \varepsilon_X(P)\ \varepsilon_X(Q))\}.$$

We prove that R is a bisimulation according to Definition 1. Let $(P,Q) \in R$ and $P \xrightarrow{\alpha} P'$ for some $P' \in \mathcal{P}$ and α free action, then by Theorems 1, 2 and by the definition of R, there exist $\mathtt{t}$ and $\mathtt{t}'$ such that

$$\Gamma_X \vdash_{\Sigma_3^\pi} \mathtt{t} : (\mathtt{ftrans}\ \varepsilon_X(P)\ \mathtt{a}\ \varepsilon_X(P')), \Gamma_X \vdash_{\Sigma_3^\pi} \mathtt{t}' : (\mathtt{StBisim}\ \varepsilon_X(P)\ \varepsilon_X(Q))$$

where `a` is `tau` if $\alpha = \tau$ or `(Out x y)` if $\alpha = \bar{x}y$. Hence, by the introduction rule of `StBisim`, there exists canonical terms $\mathtt{q}'$, $\mathtt{t}''$ and a term $\mathtt{t}'''$ such that:

$$\Gamma_X \vdash_{\Sigma_3^\pi} \mathtt{t}'' : (\mathtt{ftrans}\ \varepsilon_X(Q)\ \mathtt{a}\ \mathtt{q}'), \qquad \Gamma_X \vdash_{\Sigma_3^\pi} \mathtt{t}''' : (\mathtt{StBisim}\ \varepsilon_X(P')\ \mathtt{q}').$$

Hence, exploiting the definition of R and Theorems 1, 2, there exists $Q' \in \mathcal{P}$ such that $\varepsilon_X(Q') = \mathtt{q}'$, $Q \xrightarrow{\alpha} Q'$ and $(P', Q') \in R$. The case $Q \xrightarrow{\alpha} Q'$ is dealt with similarly. The case of bound actions follows closely that of free actions, using `btrans` in place of `ftrans`. So R is a bisimulation, whence for all $(P,Q) \in \mathcal{P} \times \mathcal{P}$, if $P\ R\ Q$ then $P \dot\sim Q$. □

It is worthwhile noticing that this soundness result does not depend on the particular formalization of the syntax and the operational semantics we have adopted, provided they are adequate. Indeed, we can change the encoding of the theory of π-calculus, e.g. by sticking to a pure LF approach (i.e., without using `Inductive` definitions), as in [14]. Nevertheless, as long as the adequacy results hold (Theorems 1, 2), also Theorem 3 holds.

The completeness of the encoding (i.e., the converse of Theorem 3) does not hold. Actually, $\dot\sim$ cannot be faithfully represented in *any* logical framework. Suppose that there is a complete encoding of $\dot\sim$; then, since type-checking is decidable, $\dot\sim$ would be semidecidable, which is absurd.

If we restrict ourselves to the set of processes without "!", namely the *finite agents*, then $\dot\sim$ is decidable, and a complete axiomatization is available [25, Section 3]. For such a fragment, our encoding of $\dot\sim$ should be complete.

3.3.2 The Greatest Fixed Point Approach

The approach described above fits neatly with the notion of strong late bisimilarity, but it can be carried out only in those intuitionistic logical frameworks featuring coinductive types, like $\mathrm{CC}^{(Co)Ind}$ or Alf. An alternative and more traditional approach is based on the fact that strong late bisimilarity can be encoded by formulating in the logical framework the definition of greatest fixed point in the style of Tarski. This approach has been adopted in many cases (see e.g. [32, 13, 9]), due to its generality. To carry out this alternative encoding, we need to use only the inductive fragment of $\mathrm{CC}^{(Co)Ind}$; therefore, this encoding can be readily translated in many other logical frameworks, *mutatis mutandis.*

Formally, we proceed as follows. First, we define the monotone operator $\mathcal{T}$ on relations between processes (see Section A.5). Notice that we could have done away with the `Inductive` definitions of `Op_StBisim` and used instead a simpler `Definition`, but the inductive version is more convenient with respect to the proof tactics of Coq.

Then, we define the ordering between relations `Inclus`:

```
Definition Inclus :=
  [R1,R2:proc->proc->Prop](P,Q:proc)(R1 P Q)->(R2 P Q).
```

Finally, we can characterize the strong late bisimilarity $\dot\sim$ as the greatest fixed point of `Op_StBisim`:

```
Inductive StBisim' [P,Q:proc] : Prop :=
       Co_Ind : (R:proc->proc->Prop)(Inclus R (Op_StBisim R)) ->
                (R P Q) -> (StBisim' P Q).
```

The above amounts to saying that in order to prove `StBisim' P Q`, we have to find a relation `R` which is included in `(Op_StBisim R)` and which holds on `P,Q`.

3.3.3 Internal (Cross) Adequacy

It is interesting to point out that, the two approaches outlined in the previous paragraphs can be formally proved equivalent in Coq itself:

```
Lemma Soundness  : (p1,p2:proc)(StBisim p1 p2)->(StBisim' p1 p2).
Lemma Completeness:(p1,p2:proc)(StBisim' p1 p2)->(StBisim p1 p2).
```

These results should be compared to the constructions carried out by Giménez in [11, Section 4.3].

From the practical point of view the above `Lemma`ta are very useful. They imply that in order to prove a given bisimilarity, it does not matter which encoding we use. We can choose either the greatest fixed point or the coinductive one, depending on which one we prefer or it is practically easier to use. Applying the proof terms of the cross adequacy `Lemma`ta we can convert strong late bisimulations of one kind into equivalent strong late bisimulations of the other kind.

Finally, using the `Completeness Lemma` and Theorem 3, we prove the soundness of the "greatest fixed point" encoding of strong late bisimilarity. Let Σ^π be the signature defined so far, extended by the declaration given in Section A.5 and those for the predicate `is_included` and `StBisim'`:

Corollary 2 (Soundness, 2) *For all processes P, Q with $fn(P,Q) \subseteq \mathcal{X}$, if there exists a term* `t` *such that* $\Gamma_X \vdash_{\Sigma^\pi} \mathtt{t} : (\mathtt{StBisim'}\ \varepsilon_X(P)\ \varepsilon_X(Q))$, *then P and Q are strongly bisimilar ($P \dot\sim Q$).*

4 A Formal Verification of [25, Section 2] in Coq

This is one of the most important sections of the present paper. In it we report on a very substantial case-study of formal verification development in Coq; namely, the theory of π-calculus developed in [25, Section 2]. This enterprise is significant from various viewpoints. To our knowledge, it is one of the largest case-studies, involving coinductive types. But furthermore it is, perhaps, the first serious attempt of using higher order syntax *à la* Church in metatheoretical studies outside type theory.

In order to achieve satisfactorily this latter end, we have to devise a theory of *contexts* for higher order syntax. The methodology that we follow in doing this is *axiomatic*. We prefer to introduce directly the necessary properties as axioms, rather than proving them on the basis of other, possibly inductive, principles that should themselves require some form of justification. However, at the end of Section 4.2 we shall briefly indicate possible ways for carrying out a formal justification of them. Our axioms make explicit some general properties of contexts and processes, which are normally taken for granted in informal reasoning: e.g. no process can mention all names, a name which does not occur is "generic",... We feel that this axiomatic approach is the appropriate one in formal verification, even if it leaves open the issue of justifying the axioms. But this, as is the case since the birth of Geometry in ancient Greece, is another story. A recommended methodological attitude is that of postulating only what is "strictly necessary", trying not to make unwanted ontological commitments. Moreover, our axiomatization has a general flavour and can be readily adapted to other metatheoretic treatments of systems in HOAS.

More specifically, in Section 4.1 we recall the theory of π-calculus appearing in [25, Section 2]. In Section 4.2, we extend the signature Σ^{π} to $\Sigma^{\pi+}$; this contains an axiomatization of the "theory of π-calculus contexts" to be used in order to establish formally the properties under consideration. In Section 4.3 we present the formal counterparts of [25, Section 2] that were actually formally verified in Coq. Finally in Section 4.4 we report on the proper verification activity, including statistical data, and we compare and contrast our formally verified theory development to the development "by hand" presented in [25].

4.1 The Theory of π-calculus developed in [25, Section 2]

Section 2 of [25] is the standard reference for the basic properties of π-calculus processes. It contains crucial Lemmata concerning the transition semantics of π-calculus, and the basic algebraic theory of strong late bisimilarity. In order to make it easier to follow the formal representation and verification of these results, to be carried out in the following subsections, we recall them here. Of course we have reformulated them, taking into account the fact that our syntax contains the mismatch operator, and features "!" in place of the original equational rewriting. We stick to the original numbering appearing in [25].

Theorem MPW 1 $\equiv_\alpha$ is a strong late bisimulation.

Theorem MPW 2' 1. $\dot\sim$ is an equivalence relation

2. If $P \dot\sim Q$ and α is a free action, then

$$\begin{array}{llll} \alpha.P \dot\sim \alpha.Q & P+R \dot\sim Q+R & P|R \dot\sim Q|R & !P \dot\sim !Q \\ [x=y]P \dot\sim [x=y]Q & [x\neq y]P \dot\sim [x\neq y]Q & (\nu w)P \dot\sim (\nu w)Q & \end{array}$$

3. If for all $v \in fn(P,Q,y)$, $P\{v/y\} \dot\sim Q\{v/y\}$, then $x(y).P \dot\sim x(y).Q$

Theorem MPW 3 $P+0 \dot\sim P \qquad P+P \dot\sim P$
$P+Q \dot\sim Q+P \qquad P+(Q+R) \dot\sim (P+Q)+R$

Theorem MPW 4' $!P \dot\sim P|!P$

Theorem MPW 5' If $x \neq y$, then $[x=y]P \dot\sim 0 \dot\sim [x\neq x]P$ and $[x=x]P \dot\sim P \dot\sim [x\neq y]P$.

Theorems MPW 6, 7

$$\begin{array}{ll} (\nu y)P \dot\sim P & \text{if } y \notin fn(P) \\ (\nu y)(\nu z)P \dot\sim (\nu z)(\nu y)P & \\ (\nu y)(P+Q) \dot\sim (\nu y)P + (\nu y)Q & \\ (\nu y)\alpha.P \dot\sim \alpha.(\nu y)P & \text{if } y \notin n(\alpha) \\ (\nu y)\alpha.P \dot\sim 0 & \text{if } y \text{ is the subject of } \alpha \end{array}$$

Theorems MPW 8, 9

$$\begin{array}{lll} P|0 \dot\sim P & P|Q \dot\sim Q|P & P|(Q|R) \dot\sim (P|Q)|R \\ (\nu y)(P|Q) \dot\sim (\nu y)P|Q & \text{if } y \notin fn(Q) & \\ (\nu y)(P|Q) \dot\sim (\nu y)P|(\nu y)Q & \text{if } y \notin fn(P)\cap fn(Q) & \end{array}$$

In [25] the following technical lemmata are used:

Lemma 1 If $P \xrightarrow{\alpha} P'$ then $fn(\alpha) \subseteq fn(P)$ and $fn(P') \subseteq fn(P) \cup bn(\alpha)$.

Lemma 2 If $P \xrightarrow{x(y)} Q$ (respectively, $P \xrightarrow{\bar{x}(y)} Q$), then for all $z \notin n(P)$ there exists Q' such that $Q' \equiv_\alpha Q\{z/y\}$ and $P \xrightarrow{x(z)} Q'$ (respectively, $P \xrightarrow{\bar{x}(z)} Q'$).

Lemma 3' If $P \xrightarrow{\alpha} P'$, $bn(\alpha) \cap fn(P'\{x/y\}) = \emptyset$, $x \notin fn(P)$ and $y \notin bn(\alpha)$, then there exists P'' such that $P'' \equiv_\alpha P'\{x/y\}$ and $P\{x/y\} \xrightarrow{\alpha\{x/y\}} P''$.

Lemma 4 If $P\{x/y\} \xrightarrow{\alpha} P'$, $x \notin fn(P)$, $bn(\alpha) \cap fn(P,x) = \emptyset$, then there exist Q, β such that $Q\{x/y\} \equiv_\alpha P'$, $\beta\{x/y\} = \alpha$ and $P \xrightarrow{\beta} Q$.

Lemma 5 Let $P \equiv_\alpha P'$; then, if α is a free action and $P \xrightarrow{\alpha} Q$, then there exists Q' such that $Q \equiv_\alpha Q'$ and $P' \xrightarrow{\alpha} Q'$; if $P \xrightarrow{x(y)} Q$ (respectively, $P \xrightarrow{\bar{x}(y)} Q$), then for all $z \notin n(P')$ there exists Q' such that $Q\{z/y\} \equiv_\alpha Q'$ and $P' \xrightarrow{x(z)} Q'$ (respectively, $P' \xrightarrow{\bar{x}(z)} Q'$).

Lemma 6 If $P \dot\sim Q$ and $w \notin fn(P,Q)$, then $P\{w/x\} \dot\sim Q\{w/x\}$.

Many of these lemmata are concerned with names and α-conversion. In particular, Lemmata 5 and 6 state that $\xrightarrow{\alpha}$ and $\dot{\sim}$ are preserved by α-conversion and by substitution of fresh names, respectively. The freshness condition on y in Lemma 3' differs from the original one in [25, Lemma 3], because of the presence of the mismatch operator in our syntax.

The basic technique used in [25] for proving these properties is that of exhibiting explicitly suitable strong late bisimulations. In some cases, this is achieved indirectly using appropriate alternative forms of bisimulations, such as bisimulations "up to restriction", which are subsequently proved to be included in $\dot{\sim}$. We shall not recall the many technical lemmata which appear in [25, Section 2] concerning these alternative forms of bisimulations. The formally verified proofs of the above results, which we will build using Coq will follow, in fact, a completely different pattern, based on the guarded induction principle.

4.2 $\Sigma^{\pi+}$: A signature for reasoning about the metatheory of π-calculus

Establishing metatheoretic properties using the higher order encoding Σ^{π} is straightforward only in some cases, *e.g.* symmetry, reflexivity of $\dot{\sim}$, and the monoidal laws of $+$ (see Section 4.3). In many other cases, the proofs of the metatheoretic properties are problematic. This is especially the case when the proofs "by hand" deal directly with those aspects, which HOAS encodings delegate to the metalanguage: namely, substitutions, freshness and α-conversion. In proving that $\dot{\sim}$ is transitive *i.e.* [25, Theorem 2(a)], for instance, when dealing with the restriction operator, one reasons directly on explicit substitutions of the bound name with a fresh one. This cannot be mimicked when using a HOAS encoding, since bound names are not even directly visible.

In order to handle adequately, these "name-related" metatheoretical results, we need to add to Σ^{π} some new axioms concerning names, processes and actions. Essentially, these reflect at the theory level the fact that our axiomatization of `isin` and `notin` captures correctly the informal notions of occurrence and non-occurrence (freshness), respectively. These axioms are all very natural and general, and the process of singling them out is a fruitful conceptual analysis of the informal use of terms and contexts.

As far as names are concerned, we postulate two axioms:

```
Axiom unsat : (p:proc)(Ex [x:name](notin x p)).
Axiom LEM_OC : (x:name)(p:proc)(isin x p) \/ (notin x p).
```

Axiom `unsat` states that no process can contain all the names; in other words, for each process we can always choose a name which does not occur in it. This is justified because in the π-calculus the set of names is assumed to be infinite. Axiom `LEM_OC` states that a name either occurs or does not occur in a given process. This adds a classical flavour to our encoding, allowing us to prove properties by case analysis on the comparison of names (Some important proofs proceed by case analysis on names—see, e.g., [25, Lemmata 3, 6]). This axiom is needed because name is not an inductive set, and there is no higher-order induction principle over proc. The simpler law of excluded middle over names, namely (x,y:name)(x=y)\/~(x=y), is derivable from LEM_OC, but not vice versa. Notice that if one tries to derive LEM_OC by induction on the syntax of processes, the case of the binding constructors fails.

As far as processes and actions are concerned, we have to deal with contexts. In particular, in our experience, the following general properties seem to be needed in proving the technical lemmata about $\dot{\sim}$ and $\longrightarrow$. By "phrase" we intend expressions of both process and action type:

β-expansion: given a phrase p and a name x, there is a context $q(\cdot)$ such that $q(x) = p$ and x does not occur in $q(\cdot)$.

Extensionality of contexts: two contexts are equal if they are equal on a fresh name; that is, if $p(x) = q(x)$ and $x \notin p(\cdot), q(\cdot)$, then $p = q$.

Monotonicity: if x does not appear in $p(y)$, then it does not appear in $p(\cdot)$.

Clearly, the above hold for contexts (of processes and actions), and indeed they can be informally proved by induction on the syntax. Nevertheless, most of their instances cannot be proved in $\mathrm{CC}^{(Co)Ind}$. In fact in a higher order encoding, contexts (of any syntactic sort) are represented as functions: *e.g.* a term of type name -> proc stands for a process with a hole, to be filled in by names. However there is no induction principle at the level of functional types in Type Theory and hence the above have to be taken as Axioms.

The full list of such axioms is given in Appendix A.6, along with some examples of their uses in proof developments.

In particular, as far as processes are regarded, the following have been assumed:

```
Axiom proc_ext : (p,q:name->proc)(x:name)
      (notin x (nu p)) -> (notin x (nu q)) -> (p x)=(q x) -> p=q.
Axiom proc_mono : (p:name->proc)(x,y:name)
      (notin x (p y))->(notin x (nu p)).
```

Axiom proc_ext could have been given equivalently over processes by formulating the conclusion as ...->(nu p)=(nu q), using equality over proc instead that over name->proc. The form we adopt highlights the equivalence of *contexts*, and not simply of formulæ, and it is uniform with the formulation of the higher-order instances of the same principle (see ho_proc_ext below). β-expansion can be proved from the corresponding β-expansion law for higher-order processes (processes with at most three holes):

```
Axiom ho_proc_exp: (p:name->proc)(x:name)
    (Ex [q:name->name->proc]
        (notin x (nu [y:name](nu (q y)))) /\ p=(q x)).
Axiom ho2_proc_exp : (p:name->name->proc)(x:name)
    (Ex [q:name->name->name->proc]
      (notin x (nu [y:name](nu [z:name](nu (q y z)))))/\p=(q x)).
```

```
Axiom ho_proc_ext: (p,q:name->name->proc)(x:name)
    (notin x (nu [y:name](nu (p y)))) ->
    (notin x (nu [y:name](nu (q y)))) -> (p x)=(q x) -> p=q.
```

For free and bound actions, only the extensionality and monotonicity properties have been postulated: in this case, the β-expansion law can be proved. These properties capture the idea that functions of type `name->proc` are actually "processes with a hole", which can be just filled in by any name without changing the structure of the process itself. This holds as long as `name` is *not* an inductive set; otherwise, it is easy to get an inconsistency by defining "exotic" functions by `Case` analysis. For instance, if we define `name` to be `nat`, we can take

$$\mathtt{x} \stackrel{\text{def}}{=} \mathtt{0} \qquad \mathtt{y} \stackrel{\text{def}}{=} (\mathtt{S\ 0}) \qquad \mathtt{p} \stackrel{\text{def}}{=} [\mathtt{z}:\mathtt{name}]\mathtt{nil}$$

$$\mathtt{q} \stackrel{\text{def}}{=} \mathtt{[z{:}name]{<}proc{>}Case\ z\ of\ nil\ [y{:}name](par\ nil\ nil)\ end}$$

By these definitions, `(p x)=(p y)=(q x)=nil`, while `(q y)=(par nil nil)`. Then, by extensionality one can easily prove `nil=(par nil nil)`, which is inconsistent since `proc` is inductive.

It is worthwhile noticing that these axioms are not peculiar to the π-calculus. On the contrary, they are proper to the HOAS approach itself, when one tries to capture a general theory of contexts. *Mutatis mutandis*, this set of axioms should be readily applicable to other formalizations based on HOAS, whenever in need of reasoning on those details HOAS takes care of at the metalevel.

Of course, one can ask for a formal justification of the above axioms. First of all we point out that, a model, based on categories of presheaves, which validates these axioms has been just recently developed by Martin Hofmann [16]. Another line of justification is the following. Consider the term model of the signature Σ^π, i.e., our basic signature without the axioms in question. In such model, each type `name->...->name->proc` is interpreted by a set of canonical λ-terms, without the `Case` constructor. It can be proved by induction on the syntax of these terms, that our axioms hold in this model, i.e. the term model of Σ^π is a model for the axioms. Of course, in this model the types corresponding to the axioms are not inhabited. But now, we can apply a strong form of "reflection", and internalize these properties by introducing the remaining axioms of $\Sigma^{\pi+}$. This is the kind of reflection principle that one would invoke, say, in claiming the consistency of the existence of an inaccessible cardinal from the bare consistency of the axioms of Set Theory.

Notice that, as pointed out by Hofmann, axioms `proc_ext`, `unsat` and `LEM_OC` would be inconsistent if we assumed the *Axiom of Unique Choice*:

```
Axiom UC : (A,B:Set)(R:A->B->Prop)
           ((a:A)(EX b:B | (R a b)/\((b':B)(R a b')->b=b')))
           ->(EX f:A->B | (a:A)(R a (f a))).
```

In Coq, axiom `UC` is not derivable. It could be derived only if the two kinds `Prop` and `Set` were identified. But then, our metalogic would use quite a weird notion of existential, far removed from standard intuition. No wonder our axioms, dictated by ordinary reasoning, would then be inconsistent.

4.3 Verifying [25, Section 2] using $\Sigma^{\pi+}$

In this section we comment on the experiments on the formal theory of π-calculus, which we have carried out in Coq using the encoding $\Sigma^{\pi+}$, and in particular on the computer assisted verification of all the formal counterpart of [25, Section 2]. All the properties formally proved are listed in Appendix A.8[3].

The proofs of all these properties but for TRANS, the congruence of $\dot\sim$ with respect to the !-constructor and some laws about ν, are quite simple. First of all we do not need to deal with low level details about names, and moreover, by exploiting fully the coinductive features offered by Coq in combination with top-down refinement, we can proceed directly without the need to introduce auxiliary notions of bisimulations. This is the case, for instance, of the associativity of $|$. In [25], this is proved by introducing a new kind of bisimulation, called *strong late bisimulation up to restriction and* $\dot\sim$, which has to be proved to be a strong late bisimulation as well. In the encoding presented in this paper, instead, we do not need to introduce any further notion of bisimilarity. A suitable bisimulation between $(P|Q)|R$ and $P|(Q|R)$ can be built in Coq directly and interactively, by means of guarded applications of the coinductive hypothesis, i.e., by a "coinductive" ("circular") argument (see Appendix A.9).

Both in the proof of transitivity of $\dot\sim$ and of the commutativity of "$|$" we use a coinductive argument which ultimately relies on [25, Lemma 6]. This is rendered in Coq as follows:

```
Lemma Lemma6: (p,q:name->proc)(z:name)
       (notin z (nu p)) -> (notin z (nu q)) ->
       (StBisim (p z) (q z)) ->
 (w:name)~(w=z)->(notin w (nu p)) -> (notin w (nu q)) ->
       (StBisim (p w) (q w)).
```

Lemma 6 reflects a reasoning style which is used frequently when dealing with "schematic derivations": one freely replaces every occurrence of a given variable by a fresh one. These are precisely the kind of properties which require the extra axioms introduced in Section 4.2 in order to be proved. The (quite long) Coq proof of `Lemma6` has another peculiarity, in that it exploits the possibility of switching between the greatest fixed point and coinductive encodings of $\dot\sim$ by means of the "cross adequacy" result of Section 3.3.3. Indeed, the proof of `(StBisim (p w) (q w))` is reduced to the existence of a bisimulation between `(p w)` and `(q w)`, which is built inductively following the sketch in [25]. One may wonder whether `Lemma6` could be proved directly by a coinductive proof. Although the existence of a (coinductive) proof term for `Lemma6` is eventually proved, it is not easy to derive it directly by means of the `Cofix` tactic. The main problem is that one would like to apply the coinductive hypothesis twice, in a nested manner; but, this violates the restrictive guardedness conditions enforced by Coq. Therefore, a direct, coinductive proof would not follow the natural pattern one adopts informally, on the paper.

[3]The Coq code is available at `http://www.dimi.uniud.it/~scagnett/picalculus/`

It is interesting to point out that, because of our higher order encoding, we may have to express and prove properties which have no natural correspondence in the ordinary π-calculus. One of these is the fact that η-equivalence is a strong late bisimulation:

```
Lemma eta_slb: (p:name->proc)(StBisim (nu p) (nu [x:name](p x))).
```

Although the decoding of `(nu [x:name](p x))` is the same as that of `(nu p)`, this property is important in the formal reasoning because it allows to normalize process encodings to their canonical form. This has turned out to be essential in the proof of some laws about ν (e.g., Theorems 8, 9).

The use of HOAS clashes especially in connection with the fact that Coq tactics do not deal adequately with higher order unification. For example, in the proofs of Lemmata 3 and 4, one cannot simply rely on the mutual induction principle generated by the system. These are too weak, and cannot be applied to goals which involve context variables. In order to overcome this drawback, we introduce by hand the appropriate unifications.

The proof of the congruence of $\dot{\sim}$ with respect to the !-constructor needs particular care because of the difficulty, which we pointed out earlier, related to the use of top-down proof search in connection with rules which do not satisfy a "sub-formula" property. In this case, the problematic rule is REPL$\frac{P|!P \xrightarrow{\alpha} Q}{!P \xrightarrow{\alpha} Q}$. Any inversion tactic introduces as subgoals new instances of the conclusion, producing an infinite regress. We overcame this difficulty by proving in Coq the (formal equivalent of the) following normal form theorem:[4]

Theorem 4 *For $P, Q \in \mathcal{P}$ and α action such that $!P \xrightarrow{\alpha} Q$, then*

1. *if $\alpha \neq \tau$ then there exist Q' such that $P \xrightarrow{\alpha} Q'$ and $Q = P^n|Q'|!P$ for some n;*
2. *if $\alpha = \tau$, then one of the following holds:*
   - *a) $\exists Q'$ such that $P \xrightarrow{\tau} Q'$ and $Q = P^n|Q'|!P$, for some n;*
   - *b) $\exists Q', Q''$ such that $P \xrightarrow{\bar{x}y} Q'$, $P \xrightarrow{x(z)} Q''$ and, for some n, m, $Q = P^n|(Q'|(P^m|(Q''\{y/z\}|!P)))$ or $Q = P^n|(Q''\{y/z\}|(P^m|(Q'|!P)))$;*
   - *c) $\exists Q', Q''$ such that $P \xrightarrow{\bar{x}(y)} Q'$, $P \xrightarrow{x(z)} Q''$ and, for some n, m, $Q = P^n|(\nu y(Q'|(P^m|(Q''\{y/z\}|!P))))$ or $Q = P^n|(\nu y(Q''\{y/z\}|(P^m|(Q'|!P))))$.*

This theorem, which is proved by induction on the structure of Q, allows us to invert effectively predicates of the shape $!P \xrightarrow{\alpha} P'$, deducing the structure of P'. This is crucial in the proof of the congruence of $\dot{\sim}$ with respect to !.

Notice that the use of lists of variables in the encoding of the restriction rule is necessary. Consider for example the proof of Lemma 3'. The argument goes, as usual, by induction on the depth of derivations of $\dot{\longrightarrow}$; the crucial step is in the case of RES rule. The hypotheses are $P \equiv (\nu z)P_1$, $P' \equiv (\nu z)P_1'$, $P \xrightarrow{\alpha} P'$. Then, by inverting the latter, we have $z \notin fn(\alpha)$

[4] For the sake of simplicity, in this proposition we denote $P|\ldots|P$ by P^n.

and $P_1 \xrightarrow{\alpha} P_1'$. Now, let z' be a fresh name, i.e., $z' \notin fn((\nu z)P_1, P_1\{x/y\}, x)$. Then, we have $P\{x/y\} \equiv (\nu z')P_1\{z'/z\}\{x/y\}$ and, by induction hypothesis: $P_1\{z'/z\}\{x/y\} \xrightarrow{\alpha\{x/y\}} P_1'' \equiv_\alpha P_1'\{z'/z\}\{x/y\}$. Hence, knowing that $z' \notin n(\alpha\{x/y\})$, we conclude $P\{x/y\} \xrightarrow{\alpha\{x/y\}} (\nu z')P_1'' \equiv_\alpha P'\{x/y\}$.

In formalizing the previous argument, we need to enrich the local environment of the (encoding of the) RES rule with a generic list of names, in order to choose z' such that $z' \notin fn((\nu z)P_1, P_1\{x/y\}, x)$. This list will contain all the names which do not appear in the processes involved in the transition, but are in the proof environment and may come into play later; hence, z' must be different from these names as well. In the case of the proof of Lemma 3', this list contains only the name x, which is introduced in the processes by the substitution.

We conclude this section by giving some statistics on our formal development in `Coq`. All data refer to the following environment: Sun Enterprise Server 450 with two UltraSPARC processors at 300MHz, 256MB RAM, 513MB swap space, with almost no other process running; `Coq` V6.2, in native mode.

Number of proofs:	90
Size of source code:	~350 kB
Length of proofs:	maximum: ~ 57kB (Lemma 3) average : ~ 3.9kB minimum : 178byte (`Soundness`)
Broadest proof tree:	42 main subgoals (`ASS_PAR`)

Times of compilation	
Theory:	42.3 sec
Cross adequacy:	39 sec
Theory of contexts:	38 sec
Lemmata 1–6:	1h 2m 31sec
Metatheory:	1h 1m 19sec
Congruence of $\dot\sim$ w.r.t. !:	11m 26sec
Maximum memory consumption:	187MB

4.4 Comparing [25, Sec. 2] and their formal counterparts

In this section we briefly compare our formal development of the basic theory of $\dot\sim$ to the "handmade" version appearing in [25, Section 2], emphasizing what we have gained and learnt from this experience.

In the first place our formal development has forced us to spell out the proofs in all details. In effect, we have found two imperfections in the proofs given in [25]. First, in the proof of Lemma 3, in the RES case, the fact that $z' \notin v(\alpha\sigma)$ does not follow from the hypotheses $z' \notin \mathrm{FV}((\nu z)P_1, P_1\sigma)$ and $z'\sigma = z'$. Instead, this z' has to be chosen fresh also with respect to the names in $\alpha\sigma$. The proof can be easily fixed by requiring this extra condition explicitly (without any loss of generality). In our formalization, this extra condition has been reflected in the (encoding of the) rule RES, by introducing a generic list of names and assuming that the bound variable does not appear in it. Secondly, a minor omission appears in the proof of transitivity, which does not mention the use of

Lemma 6. In our formalization this Lemma turns out to be indispensable in the case of bound output transitions. Remarkably for hand made proofs, these are the only imperfections that we found in the otherwise extraordinarily detailed and exceptionally accurate and exhaustive proofs given in [25].

As we pointed out in the previous subsection, many of the formal proofs adopt a completely different approach to that of [25]. This is the case of those properties which are proved in [25] using bisimulations "up-to-something", such as commutativity of $|$, of ν, *etc.*. Using the `Cofix` tactic, one does not need to introduce explicitly these auxiliary notions of bisimulations and to show that they are included in $\dot{\sim}$. One can directly prove the properties by means of natural coinductive arguments, using the thesis as hypothesis, accordingly to the Guarded Induction principle [3, 11, 10]. As remarked in [3], in a sense such proofs are easier than the corresponding ones in [25, Section 2], because they are directly guided by the definition of bisimulation. They all follow a common pattern, given by the introduction rule of strong late bisimilarity (`sb`). One does not need to produce in advance a bisimulation containing the two processes. Such a bisimulation is gradually built interactively during the proof in a way completely transparent for the user.

The use of HOAS has a tradeoff.

On one hand we have a simplification of many arguments. The most obvious case is that of Theorem 1 of [25], which asserts that α-equivalence is a bisimulation. This theorem simply disappears. The same happens for Lemma 2 and Lemma 5. Another example is given by Lemma 3 which is stated in two different ways in [25, 24]. These two versions differ only on a hypothesis about bound names. Using HOAS these hypotheses do not need to be formulated explicitly since they are enforced by the metalanguage itself. In our formal translation we do not need to worry about which formulation to choose, since both versions are formalized by the same type. For the same reason, Lemma 3 and Lemma 4 collapse into the same formalization.

On the other hand the use of HOAS is problematic when establishing metatheoretic properties involving exactly those notions which are delegated to the metalanguage, e.g. substitution of names, freshness and α-conversion. To overcome these difficulties, we had to introduce in Section 4.2 a partial axiomatization of the theory of contexts (the full list appears in Section A.8). In informal reasoning, these properties are usually taken for granted, but in a formal approach they have to be postulated.

Finally, we point out, once more, one of the main difference between the formal top-down development and the version "by hand", *i.e.* the impracticality of a system based on structural congruences. In effect, in the literature, the operational semantics of π-calculus is given with the structural congruence relation, but this is not suitable for semiautomatic proof search, because it can lead to non-well founded proof trees. Rules like $\mathrm{EQ}\frac{P\equiv P' \quad P' \xrightarrow{\alpha} Q' \quad Q'\equiv Q}{P \xrightarrow{\alpha} Q}$ can always be applied without reducing the complexity of the goal. Suppose we have to prove a given goal by inverting the hypothesis $H : (P \xrightarrow{\alpha} Q)$, i.e., we want to reason by case analysis on the way this hypothesis can be inferred. Since the EQ rule is always applicable, if we invert H we find ourselves back with a proof-search problem similar to the one we started with. Namely, we have to prove the goal in a context containing the premises of EQ, *i.e.* $H_1 : P \equiv P'$, $H_2 : P' \xrightarrow{\alpha} Q'$, $H_3 : Q' \equiv Q$. The inversion of H_2 arises the same problem, while the inversion of H_1 leads us to apply *ad infinitum* the symmetry rule, switching between $P \equiv P'$ and $P' \equiv P$ back and forth. Luckily there exist also purely transitional presentations of π-calculus, which are the ones we have used.

5 Conclusions

In this paper we have presented a faithful HOAS encoding of the π-calculus, inspired by [14], suitable for use in logical frameworks based on intuitionistic type-theory. In our view, such a representation has several advantages when compared to other more traditional approaches used in formal developments of π-calculus, namely [20] and [13]. First of all, the extensive and careful use of the higher order syntax frees us from the tedious encoding of the ordinary mechanisms involved in the handling of bound names, because these are automatically inherited from the metalevel. This solution offers a smooth and simple treatment both of the syntax of processes and of transitions and produces what we think is a clean and intuitive representation of the system. But moreover, when we are indeed forced to mention explicitly, through the `isin`, `notin` predicates, side conditions on freshness, we are sure that we are facing an essential peculiarity of the π-calculus in the handling of bound variables. This is the case, for instance, of the restriction operator.

The main drawback of HOAS is the difficulty of dealing with metatheoretic issues concerning names in process contexts, *i.e.* terms of type `name->proc`. As a consequence, some natural metatheoretic properties involving substitution and freshness of names inside proofs and processes, cannot be proved inside the framework and instead have to be postulated. Soundness and completeness of our axiomatization of this elementary theory of contexts deserves further investigation. In any case, Martin Hofmann has recently announced that our axioms are validated in suitable pre-sheaves models [16].

We have investigated in detail two ways of encoding strong late bisimulations: one by means of an inductive encoding of Tarski's definition of greatest fixed point and one by means of a `CoInductive` predicate. While the former approach is more widely applicable, the latter takes full advantage of the pragmatic features offered by Coq's coinductive tactics, based on the Guarded Induction Principle of Coquand and Giménez [3, 11, 10]. As originally anticipated by Coquand, this tactic allows for more straightforward development of equivalence proofs, which does not force the user to produce in advance a bisimulation. Moreover, we can also do without having to introduce auxiliary generalized forms of strong late bisimulation, like the "up-to restriction and strong late bisimilarity" bisimulation introduced in [25].

Finally, we have presented a formal development of a nontrivial fragment of the theory of π-calculus in Coq, using our encoding, which essentially amounts to [25, Section 2].

In this paper we have dealt only with strong late bisimilarity. However, the ideas and techniques that we have used are quite general and they can be readily applied also to the case e.g. of strong early semantics (see Appendix A.4) or weak semantics (see e.g. [17]).

Summing up, we can claim that we have not only a faithful proof editor for π-calculus, but also a practical "workbench" for reasoning on the calculus itself.

Related work The solutions we have adopted are quite different from other formal developments of π-calculus in proof development environments.

In [20], the π-calculus is encoded in Isabelle/HOL by means of a plain first-order, inductive approach. Binding syntactic constructor (such as the input prefix) are represented by first-order constructors; therefore, all syntactic operations (such as substitution) cannot be delegated to the metalevel, but have to be defined "by hand". Bisimulations are defined by directly translating the coinductive definitions in a inductive setting, as it is done in Section 3.3.2.

In [13], a fragment of the polyadic π-calculus is encoded in Coq using a radically different approach. Names are represented by means of de Bruijn indexes, and therefore there is no need to introduce a specific set of names or higher-order syntactic constructors. This encoding is not close to the intuitive syntax of processes and a non trivial additional technical machinery is needed in order to manipulate indexes during communications. The author proves formally, in a purely inductive setting, a very interesting set of properties about the "!" operator and the encoding of λ-calculus into the π-calculus, using Sangiorgi's theory of progressions and up-to-context techniques, in an inductive setting.

Formalized metatheoretic reasoning on systems in HOAS has been explored by other authors, *e.g.* [3, 4, 14, 16, 28]. But apart from [16], none of their approaches, in their current state of development, appears to be general enough to sustain the metatheoric investigations of the π-calculus presented in this paper.

Current and future work The encoding in $\mathrm{CC}^{(Co)Ind}$ of the π-calculus presented in this paper is only one step of a wider research programme on building computerized tools for reasoning about processes algebras.

There are many open problems concerning the present system. We just recall a few: a complete formal development in HOAS of a theory of contexts, and an analysis of induction principles for higher order types.

There are also some pragmatic constraints on the current implementation of Coq, which partially affect the friendliness of our workbench for π-calculus. For instance we mention the fact that nested applications of the coinductive hypotheses are not allowed, and that the available instances of higher order unification are not sufficiently general. These limitations should be hopefully removed in future releases of Coq.

The current multitude of variants of the π-calculus, brings about the serious problem of how to develop a general proof editor for mobile processes. One could think of considering encodings not only of weaker notions of bisimulation, but also polyadic versions, and asynchronous versions of π-calculus, etc.. The issue of building proofs systems for general action calculi should be addressed.

For the time being we have considered only the polyadic π-calculus, as presented in [23]. However, the higher order presentation of this system seems to bring forward a host of radically new delicate issues. The main issue we have to face is the adequacy problem arising from the encoding of the five syntactic categories introduced by Milner in the original system (namely normal processes, processes, abstractions, concretions and agents). These categories are defined in a dependent mutual manner; so a normal process can also be considered as a process or as an abstraction (whose arity is zero) and so on. The straightforward way of encoding such dependencies in a dependent types theory, like $\mathrm{CC}^{(Co)Ind}$, is by means of coercion operators. However this yields different canonical terms (even belonging to the same type) representing the same object. Obviously a "standard" form of adequacy cannot be achieved in this case (there cannot exist a compositional bijection between the objects of the polyadic π-calculus and the $\mathrm{CC}^{(Co)Ind}$ terms representing them). Despite this problem, we can recover a form of adequacy introducing the concept of *encoding relations* which are simply the formalization of the *one-to-many* correspondence between the syntactic objects of the polyadic π-calculus and the $\mathrm{CC}^{(Co)Ind}$ terms encoding them.

Another difficult problem arises in connection with the representation of Milner's *pseudo-application* which embodies the complexity of communicating several names at the same time along a channel. In the original syntax this relies on a multiple substitution of (data) names into (bound) names, but in $\mathrm{CC}^{(Co)Ind}$ abstractions and applications are monadic. Exploiting higher order syntax, however, it is possible to decompose the polyadic communication into several monadic ones, without the appearance of an unwelcomed composed substitution.

Acknowledgements We are grateful to Joëlle Despeyroux, Mauro Felchero, Eduardo Giménez, Martin Hofmann, Marina Lenisa, Christine Paulin-Mohring, Randy Pollack, Davide Sangiorgi and the anonymous referees for their helpful hints and discussions. A preliminary version of this paper was presented at the IC-EATCS School *Models and Paradigms for Concurrency*, held in Udine (Italy), 15-19 September 1997.

References

[1] A. Avron, F. Honsell, I. A. Mason, and R. Pollack. Using Typed Lambda Calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9:309–354, 1992.

[2] *The Coq Proof Assistant Reference Manual - Version 6.2*. INRIA, May 1998. Available at `ftp://ftp.inria.fr/INRIA/coq/V6.2/doc`.

[3] T. Coquand. Infinite objects in type theory. In [36], pages 62–78.

[4] T. Coquand and G. Huet. The calculus of constructions. *Information and Control*, 76:95–120, 1988.

[5] C. Cornes and D. Terrasse. Automating Inversion and Inductive Predicates in Coq. In [37], pages 85–104.

[6] J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-order syntax in Coq. In *Proc. of TLCA'95*. LNCS 905, Springer-Verlag, 1995.

[7] J. Despeyroux, F. Pfenning, and C. Schürmann. Primitive recursion for higher order abstract syntax. CMU-CS-96-172, School of Computer Science, Carnegie Mellon University, Pittsburgh. Aug. 1996.

[8] M. Felchero. Sistemi di transizione in teoria dei tipi coinduttivi. Laurea's thesis, Università di Udine, Italy, July 1996. In Italian.

[9] H. Geuvers. Inductive and coinductive types with iteration and recursion. Available at `http://www.dcs.ed.ac.uk/lfcsinfo/research/types_bra/proc/proc92.dvi.gz`, June 1992.

[10] E. Giménez. An application of co-inductive types in Coq: Verification of the Alternating Bit Protocol. In [37], pages 134–152.

[11] E. Giménez. Codifying guarded recursion definitions with recursive schemes. In *Proc. of TYPES'94*. LNCS 996, Springer-Verlag, 1995.

[12] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, Jan. 1993.

[13] D. Hirschkoff. Bisimulation proofs for the π-calculus in the Calculus of Constructions. In *Proc. TPHOL'97*. LNCS 1275, Springer-Verlag, 1997.

[14] F. Honsell, M. Lenisa, U. Montanari, and M. Pistore. Final semantics for the π-calculus. In *Proc. PROCOMET'98*. IFIP and Chapman & Hall, 1998.

[15] F. Honsell and M. Miculan. A natural deduction approach to dynamic logics. In [37], pages 165–182.

[16] M. Hofmann. Semantical analysis of higher-order abstract syntax. In *Proc. 14^{th} LICS*. IEEE, 1999.

[17] M. Lenisa. *Themes in Final Semantics*. PhD thesis, Dipartimento di Informatica, Università di Pisa, Italy, Mar. 1998.

[18] P. Martin-Löf. On the meaning of the logical constants and the justifications of the logic laws. TR 2, Dip. di Matematica, Università di Siena, 1985.

[19] R. McDowell and D. Miller. A logic for reasoning with higher-order abstract syntax. In *Proc. 12^{th} LICS*. IEEE, 1997.

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

[21] M. Miculan. The expressive power of structural operational semantics with explicit assumptions. In [36], pages 292–320.

[22] M. Miculan. *Encoding Logical Theories of Programs*. PhD thesis, Dipartimento di Informatica, Università di Pisa, Italy, Mar. 1997.

[23] R. Milner. The polyadic π-calculus: a tutorial. In *Logic and Algebra of Specification*, volume 94 of *NATO ASI Series F*. Springer-Verlag, 1993.

[24] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. Tech. Rep. ECS-LFCS-89-85, Dept. of Computer Science, Univ. of Edinburgh, June 1989.

[25] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Inform. and Comput.*, 100(1):1–77, 1992.

[26] R. Milner, J. Parrow, and D. Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 114(1):149–171, 1993.

[27] T. Nipkow, and L. C. Paulson. *Isabelle-91 – System abstract*. In *Proceedings of CADE 11*, pages 673–676. LNCS 607, Springer-Verlag, 1992.

[28] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. OUP, 1990.

[29] C. Paulin-Mohring. Inductive definitions in the system Coq; rules and properties. In *Proc. of TLCA'94*, pages 328–345. LNCS 664, Springer-Verlag, 1993.

[30] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proc. of ACM SIGPLAN '88*, pages 199–208, June 1988. ACM.

[31] R. Pollack. *The Theory of LEGO*. PhD thesis, Univ. of Edinburgh, 1994.

[32] A. Rossi. Tipi di dati coinduttivi: i reali. Laurea's thesis, Università di Udine, Italy, 1994.

[33] D. Sangiorgi. A theory of bisimulation for the π-calculus. *Acta Informatica*, 33:69–97, 1996.

[34] I. Scagnetto. Rappresentazione di algebre di processi in Logical Frameworks. Laurea's thesis, Università di Udine, Mar. 1997. In italian.

[35] P. Schroeder-Heister. A natural extension of natural deduction. *J. Symbolic Logic*, 49(4):1284–1300, Dec. 1984.

[36] *Proceedings of TYPES'93*. LNCS 806, Springer-Verlag, 1994.

[37] *Proceedings of TYPES'95*. LNCS 1158, Springer-Verlag, 1996.

A Coq code

A.1 The "occur check" predicates

```
Inductive isin [x:name] : proc -> Prop :=
    isin_bang   : (p:proc)(isin x p) -> (isin x (bang p))
  | isin_tau    : (p:proc)(isin x p) -> (isin x (tau_pref p))
  | isin_par1   : (p,q:proc)(isin x p) -> (isin x (par p q))
  | isin_par2   : (p,q:proc)(isin x q) -> (isin x (par p q))
  | isin_sum1   : (p,q:proc)(isin x p) -> (isin x (sum p q))
  | isin_sum2   : (p,q:proc)(isin x q) -> (isin x (sum p q))
  | isin_nu     : (p:name->proc)
                  ((z:name)(isin x (p z))) -> (isin x (nu p))
  | isin_match1 : (p:proc)(y,z:name)
                  (isin x p) -> (isin x (match y z p))
  | isin_match2 : (p:proc)(y:name)(isin x (match x y p))
  | isin_match3 : (p:proc)(y:name)(isin x (match y x p))
  | isin_mismatch1 : (p:proc)(y,z:name)
                  (isin x p) -> (isin x (mismatch y z p))
  | isin_mismatch2 : (p:proc)(y:name)(isin x (mismatch x y p))
  | isin_mismatch3 : (p:proc)(y:name)(isin x (mismatch y x p))
  | isin_in1    : (p:name->proc)(y:name)
                  ((z:name)(isin x (p z))) -> (isin x (in_pref y p))
  | isin_in2    : (p:name->proc)(isin x (in_pref x p))
  | isin_out1   : (p:proc)(y,z:name)
                  (isin x p) -> (isin x (out_pref y z p))
  | isin_out2   : (p:proc)(y:name) (isin x (out_pref x y p))
  | isin_out3   : (p:proc)(y:name) (isin x (out_pref y x p)).

Inductive notin [x:name] : proc -> Prop :=
    notin_nil   : (notin x nil)
  | notin_bang  : (p:proc)(notin x p) -> (notin x (bang p))
  | notin_tau   : (p:proc)(notin x p) -> (notin x (tau_pref p))
  | notin_par   : (p,q:proc)(notin x p)->(notin x q)->(notin x (par p q))
  | notin_sum   : (p,q:proc)(notin x p)->(notin x q)->(notin x (sum p q))
  | notin_nu    : (p:name->proc)
                  ((z:name)~(x=z) -> (notin x (p z)))
                  -> (notin x (nu p))
  | notin_match : (p:proc)(y,z:name) ~(x=y) -> ~(x=z) ->
                  (notin x p)->(notin x (match y z p))
  | notin_mismatch: (p:proc)(y,z:name) ~(x=y) -> ~(x=z) ->
                  (notin x p) -> (notin x (mismatch y z p))
  | notin_in    : (p:name->proc)(y:name) ~(x=y) ->
                  ((z:name)~(x=z) -> (notin x (p z)))
                  -> (notin x (in_pref y p))
  | notin_out   : (p:proc)(y,z:name) ~(x=y) -> ~(x=z) ->
                  (notin x p) -> (notin x (out_pref y z p)).

Inductive f_act_notin [x:name] : f_act -> Prop :=
    f_act_notin_tau : (f_act_notin x tau)
  | f_act_notin_Out : (y,z:name)~x=y->~x=z->(f_act_notin x (Out y z)).
```

```
Definition f_act_notin_ho :=
       [x:name][a:name->f_act]((y:name)~x=y->(f_act_notin x (a y))).

Inductive b_act_notin [x:name] : b_act -> Prop :=
    b_act_notin_In   : (y:name)~(x=y)->(b_act_notin x (In y))
  | b_act_notin_bOut : (y:name)~(x=y)->(b_act_notin x (bOut y)).


Definition b_act_notin_ho :=
       [x:name][a:name->b_act]((y:name)~x=y->(b_act_notin x (a y))).
```

A.2 The Transition System of π-calculus

```
Mutual Inductive ftrans : proc -> f_act -> proc -> Prop :=
  TAU       : (p:proc)(ftrans (tau_pref p) tau p)
| OUT       : (p:proc)(x,y:name)
                (ftrans (out_pref x y p) (Out x y) p)
| fSUM1     : (p1,p2,p:proc)(a:f_act)(ftrans p1 a p)
                -> (ftrans (sum p1 p2) a p)
| fSUM2     : (p1,p2,p:proc)(a:f_act)(ftrans p2 a p)
                -> (ftrans (sum p1 p2) a p)
| fPAR1     : (p1,p2,p:proc)(a:f_act)(ftrans p1 a p)
                -> (ftrans (par p1 p2) a (par p p2))
| fPAR2     : (p1,p2,p:proc)(a:f_act)(ftrans p2 a p)
                -> (ftrans (par p1 p2) a (par p1 p))
| fMATCH    : (x:name)(p,q:proc)(a:f_act)
                (ftrans p a q) -> (ftrans (match x x p) a q)
| fMISMATCH : (x,y:name)(p,q:proc)(a:f_act)~(x=y)
                -> (ftrans p a q) -> (ftrans (mismatch x y p) a q)
| fBANG     : (p,q:proc)(a:f_act)(ftrans (par p (bang p)) a q)
                -> (ftrans (bang p) a q)
| COM1      : (p1,p2,q2:proc)(q1:name -> proc)(x,y:name)
                (btrans p1 (In x) q1)
                  -> (ftrans p2 (Out x y) q2)
                    -> (ftrans (par p1 p2) tau (par (q1 y) q2))
| COM2      : (p1,p2,q1:proc)(q2:name -> proc)(x,y:name)
                (ftrans p1 (Out x y) q1)
                  ->(btrans p2 (In x) q2)
                    -> (ftrans (par p1 p2) tau (par q1 (q2 y)))
| fRES      : (p1,p2:name -> proc)(a:f_act)(l:Nlist)
                ((y:name)(notin y (nu p1)) -> (notin y (nu p2)) ->
                    (Nlist_notin y l) -> (f_act_notin y a) ->
                    (ftrans (p1 y) a (p2 y)))
                -> (ftrans (nu p1) a (nu p2))
| CLOSE1    : (p1,p2:proc)(q1,q2:name -> proc)(x:name)
                (btrans p1 (In x) q1) ->
                  (btrans p2 (bOut x) q2) ->
                    (ftrans (par p1 p2) tau (nu [z:name](par (q1 z)
                                                             (q2 z))))
| CLOSE2    : (p1,p2:proc)(q1,q2:name -> proc)(x:name)
```

```
               (btrans p1 (bOut x) q1) ->
                 (btrans p2 (In x) q2) ->
                   (ftrans (par p1 p2) tau (nu [z:name](par (q1 z)
                                                            (q2 z))))
with btrans : proc -> b_act -> (name -> proc) -> Prop :=
  IN        : (p:name -> proc)(x:name)(btrans (in_pref x p) (In x) p)
| bSUM1     : (p1,p2:proc)(a:b_act)(p:name -> proc)
              (btrans p1 a p) -> (btrans (sum p1 p2) a p)
| bSUM2     : (p1,p2:proc)(a:b_act)(p:name -> proc)
              (btrans p2 a p) -> (btrans (sum p1 p2) a p)
| bPAR1     : (p1,p2:proc)(a:b_act)(p:name -> proc)(btrans p1 a p)
                  -> (btrans (par p1 p2) a [x: name](par (p x) p2))
| bPAR2     : (p1,p2:proc)(a:b_act)(p:name -> proc)(btrans p2 a p)
                  -> (btrans (par p1 p2) a [x: name](par p1 (p x)))
| bMATCH    : (x:name)(p:proc)(a:b_act)(q:name -> proc)
                (btrans p a q) -> (btrans (match x x p) a q)
| bMISMATCH : (x,y:name)(p:proc)(a:b_act)(q:name -> proc)
                ~(x=y) -> (btrans p a q) -> (btrans (mismatch x y p) a q)
| bBANG     : (p:proc)(a:b_act)(q:name -> proc)
                (btrans (par p (bang p)) a q) -> (btrans (bang p) a q)
| bRES      : (p1:name -> proc)(a:b_act)
                (p2:name -> name -> proc)(l:Nlist)
                  ((y:name)(notin y (nu p1)) -> (Nlist_notin y l) ->
                    (notin y (nu [z:name](nu (p2 z)))) ->
                    (btrans (p1 y) a (p2 y)))
                      -> (btrans (nu p1) a [z:name](nu (p2 z)))
| OPEN      : (p1,p2:name -> proc)(x:name)
                ((y:name)(notin y (nu p1)) -> (notin y (nu p2)) ->
                  ~x=y -> (ftrans (p1 y) (Out x y) (p2 y)))
                    -> (btrans (nu p1) (bOut x) p2).
```

A.3 Coinductive encoding of Strong Late Bisimilarity

```
CoInductive StBisim : proc -> proc -> Prop :=
sb : (p,q:proc)
     (((a:f_act)
        (((p1:proc)(ftrans p a p1)->
          (Ex [q1:proc]((ftrans q a q1) /\ (StBisim p1 q1))))
        /\ ((q1:proc)(ftrans q a q1)->
           (Ex [p1:proc]((ftrans p a p1) /\ (StBisim p1 q1))))))
      /\ ((x:name)
          (((p1:name->proc)(btrans p (In x) p1)->
            (Ex [q1:name->proc]((btrans q (In x) q1)
                             /\((y:name)(StBisim (p1 y) (q1 y))))))
         /\((q1:name->proc)(btrans q (In x) q1)->
             (Ex [p1:name->proc]((btrans p (In x) p1)
                             /\((y:name)(StBisim (p1 y) (q1 y))))))))
      /\((x:name)
         (((p1:name->proc)(btrans p (bOut x) p1)->
           (Ex [q1:name->proc]((btrans q (bOut x) q1)
                    /\((y:name)(notin y (nu p1)) -> (notin y (nu q1))
```

```
                                        -> (StBisim (p1 y) (q1 y))))))
        /\((q1:name->proc)(btrans q (bOut x) q1)->
             (Ex [p1:name->proc]((btrans p (bOut x) p1)
                    /\((y:name)(notin y (nu p1)) -> (notin y (nu q1))
                                        -> (StBisim (p1 y) (q1 y))))))))
    )->(StBisim p q).
```

A.4 Coinductive Encoding of Strong Early Bisimilarity

Early semantics $P \xrightarrow{\alpha}_e Q$ is obtained by replacing COM_1, COM_2 by the following:

$$\text{E-IN}\ \frac{-}{x(z).P \xrightarrow{xy}_e P\{y/z\}}$$

$$\text{E-COM}_1\ \frac{P \xrightarrow{xz}_e P' \quad Q \xrightarrow{\bar{x}z}_e Q'}{P|Q \xrightarrow{\tau}_e P'|Q'} \qquad \text{E-COM}_2\ \frac{P \xrightarrow{\bar{x}z}_e P' \quad Q \xrightarrow{xz}_e Q'}{P|Q \xrightarrow{\tau}_e P'|Q'}$$

These rules are encoded in the natural way; in the definition of `ftrans`, just replace rules `COM1`, `COM2` by the following:

```
  EIN   : (p:name->proc)(x,y:name)(ftrans (in_pref x p) (fIn x y) (p y))
| ECOM1 : (p1,p2,q1,q2:proc)(x,y:name)
             (ftrans p1 (fIn x y) q1) -> (ftrans p2 (Out x y) q2)
                 -> (ftrans (par p1 p2) tau (par q1 q2))
| ECOM2 : (p1,p2,q1,q2:proc)(x,z:name)
             (ftrans p1 (Out x z) q1) ->(ftrans p2 (fIn x z) q2)
                 -> (ftrans (par p1 p2) tau (par q1 q2))
```

where `fIn : name -> name -> f_act` is a new constructor of type `f_act`.

Definition 2 *A binary relation $\mathcal{S}$ on processes is a* strong early simulation *if $P\mathcal{S}Q$ implies that for all P', α such that $P \xrightarrow{\alpha}_e P'$ and $bn(\alpha) \cap fn(P,Q) = \emptyset$, there exists Q' such that $Q \xrightarrow{\alpha}_e Q'$ and $P'\mathcal{S}Q'$.*

The relation $\mathcal{S}$ is a strong early bisimulation *if both $\mathcal{S}$ and $\mathcal{S}^{-1}$ are strong early simulations. P, Q are* strong early bisimilar *(written $P \dot{\approx} Q$) if $P\mathcal{S}Q$ for some strong early bisimilarity $\mathcal{S}$.*

Its encoding is the following:

```
CoInductive StEBisim : proc -> proc -> Prop :=
seb : (p,q:proc)
   ( ((a:f_act)
        (((p1:proc)(ftrans p a p1)->
           (Ex [q1:proc]((ftrans q a q1) /\ (StEBisim p1 q1))))
        /\ ((q1:proc)(ftrans q a q1)->
            (Ex [p1:proc]((ftrans p a p1) /\ (StEBisim p1 q1))))))
      /\((x:name)
           ((p1:name->proc)(btrans p (In x) p1)->
             (Ex [q1:name->proc](btrans q (In x) q1)
                    /\((y:name)(notin y (nu p1)) -> (notin y (nu q1))
                                        -> (StEBisim (p1 y) (q1 y)))))
         /\((q1:name->proc)(btrans q (In x) q1)->
```

```
            (Ex [p1:name->proc](btrans p (In x) p1)
                     /\((y:name)(notin y (nu p1)) -> (notin y (nu q1))
                                     -> (StEBisim (p1 y) (q1 y))))))
      /\((x:name)
          ((p1:name->proc)(btrans p (bOut x) p1)->
            (Ex [q1:name->proc](btrans q (bOut x) q1)
                     /\((y:name)(notin y (nu p1)) -> (notin y (nu q1))
                                     -> (StEBisim (p1 y) (q1 y)))))
         /\((q1:name->proc)(btrans q (bOut x) q1)->
            (Ex [p1:name->proc](btrans p (bOut x) p1)
                     /\((y:name)(notin y (nu p1)) -> (notin y (nu q1))
                                     -> (StEBisim (p1 y) (q1 y))))))
   ) -> (StEBisim p q).
```

A.5 The operator $\mathcal{T}$ of Strong Late Bisimilarity

```
Inductive Op_StBisim [R:proc -> proc -> Prop] : proc -> proc -> Prop :=
op_sb : (p,q:proc)
        (((a:f_act)
           (((p1:proc)(ftrans p a p1)->
             (Ex [q1:proc]((ftrans q a q1) /\ (R p1 q1))))
            /\
            ((q1:proc)(ftrans q a q1)->
             (Ex [p1:proc]((ftrans p a p1) /\ (R p1 q1))))))
         /\((x:name)
           (((p1:name->proc)(btrans p (In x) p1)->
             (Ex [q1:name->proc]((btrans q (In x) q1)
                                 /\((y:name)(R (p1 y) (q1 y))))))
            /\
            ((q1:name->proc)(btrans q (In x) q1)->
             (Ex [p1:name->proc]((btrans p (In x) p1)
                                 /\((y:name)(R (p1 y) (q1 y))))))))
         /\((x:name)
           (((p1:name->proc)(btrans p (bOut x) p1)->
             (Ex [q1:name->proc]((btrans q (bOut x) q1)
                    /\((y:name)(notin y (nu p1)) -> (notin y (nu q1))
                                              -> (R (p1 y) (q1 y))))))
            /\
            ((q1:name->proc)(btrans q (bOut x) q1)->
             (Ex [p1:name->proc]((btrans p (bOut x) p1)
                  /\((y:name)(notin y (nu p1)) -> (notin y (nu q1))
                                            -> (R (p1 y) (q1 y))))))))
        )->((Op_StBisim R) p q).

Definition Inclus := [R1,R2:proc->proc->Prop]
                     (p1,p2:proc)(R1 p1 p2)->(R2 p1 p2).

Inductive StBisim' [p1,p2:proc] : Prop :=
       Co_Ind : (R:proc->proc->Prop)
                (Inclus R (Op_StBisim R)) ->
                (R p1 p2) -> (StBisim' p1 p2).
```

A.6 Theory of contexts

```
(* a process cannot contain all names *)
Axiom unsat : (p:proc)(Ex [x:name](notin x p)).

(* Law of excluded middle (decidability) of occur check predicates *)
Axiom LEM_OC : (x:name)(p:proc)(isin x p) \/ (notin x p).

(* Extensionality of contexts *)
Axiom proc_ext : (p,q:name->proc)(x:name)
       (notin x (nu p)) -> (notin x (nu q)) -> (p x)=(q x) -> p=q.

(* If a name does not occur in an applied context,
   then it cannot occur in the context itself *)
Axiom proc_mono :
        (p:name->proc)(x,y:name)(notin x (p y))->(notin x (nu p)).

(* Same as above, for free actions *)
Axiom f_act_ext : (a,b:name->f_act)(x:name)
                (f_act_notin_ho x a) -> (f_act_notin_ho x b)->
                (a x)=(b x) -> a=b.
Axiom f_act_mono : (a:name->f_act)(x,y:name)
      (f_act_notin x (a y)) -> (f_act_notin_ho x a).


(* Same as above, for bound actions *)
Axiom b_act_ext : (a,b:name->b_act)(x:name)
                (b_act_notin_ho x a) -> (b_act_notin_ho x b)->
                (a x)=(b x) -> a=b.
Axiom b_act_mono : (a:name->b_act)(x,y:name)
      (b_act_notin x (a y)) -> (b_act_notin_ho x a).


(* Extensionality of processes contexts *)
Axiom ho_proc_ext: (p,q:name->name->proc)(x:name)
      (notin x (nu [y:name](nu (p y)))) ->
      (notin x (nu [y:name](nu (q y)))) -> (p x)=(q x) -> p=q.

(* Beta expansions *)
Axiom ho_proc_exp : (p:name->proc)(x:name)
        (Ex [q:name->name->proc]
            (notin x (nu [y:name](nu (q y))))/\p=(q x)).
Axiom ho2_proc_exp : (p:name->name->proc)(x:name)
        (Ex [q:name->name->name->proc]
            (notin x (nu [y:name](nu [z:name](nu (q y z)))))/\p=(q x)).
```

In the following we present some uses of these axioms.

β-expansion. Let us consider the proof of the transitivity of $\dot{\sim}$ (case of *bound output* actions): we must prove $\forall\ x.\ x \notin P(\cdot), R(\cdot).P(x)\ \dot{\sim}\dot{\sim}\ R(x)$ knowing that $\forall\ x.\ x \notin P(\cdot), Q(\cdot).\ P(x) \dot{\sim} Q(x)$ and $\forall\ x.\ x \notin Q(\cdot), R(\cdot).\ Q(x) \dot{\sim} R(x)$ hold.

Obviously, in order to prove this goal, it would be natural to show that $P(x) \dot{\sim} Q(x)$ and $Q(x) \dot{\sim} R(x)$ hold (for $x \notin P(\cdot), R(\cdot)$). Since $x \notin P(\cdot), R(\cdot)$ does not necessarily imply $x \notin Q(\cdot)$, we need the expansion axiom $(\forall\, Q(\cdot).\, \forall\, x.\, \exists Q'(\cdot)(\cdot).\, x \notin Q'(\cdot)(\cdot) \wedge Q \equiv Q'(x))$ and Lemma 6.

So let $Q'(\cdot)(\cdot)$ be a context such that $x \notin Q'(\cdot)(\cdot) \wedge Q \equiv Q'(x)$. By definition $P(x) \dot{\sim}\dot{\sim} R(x)$ iff there is some process P' such that $P(x) \dot{\sim} P'$ and $P' \dot{\sim} R(x)$. We will prove that such P' is indeed the process $Q'(w)(x)$ where $w \neq x$, $w \notin Q'(\cdot)(\cdot), P(\cdot), R(\cdot)$. For the sake of simplicity we will prove only $P(x) \dot{\sim} Q'(w)(x)$ (the other subgoal being analogous). This is accomplished applying Lemma 6 with a fresh name z (where $z \notin P(\cdot), Q'(\cdot)(\cdot)$, $z \neq x, w$), which yields the subgoal $P(z) \dot{\sim} Q'(w)(z)$. Now we can conclude applying again Lemma 6 to obtain $P(z) \dot{\sim} Q'(x)(z)$, i.e., our hypothesis[5].

Extensionality. Let us consider Lemma 3': as usual for properties concerning the LTS, we proceed by induction on the derivation depth of the judgement $P \xrightarrow{\alpha} Q$. On paper the proof of the case of a τ-transition is as follows: $P \equiv \tau.P_0$, so $P\{x/y\} \equiv \tau.P_0\{x/y\}$, hence we can infer $P\{x/y\} \xrightarrow{\tau} P_0\{x/y\}$.

In Coq the above argument is rendered as follows: the term `p=(p' y)` coincides with `(tau_pref p0)`, `a=(a' y)` coincides with `tau` and `q=(q' y)` coincides with `p0`. Hence `(p' y)=(tau_pref (q' y))`. To conclude we need extensionality (`Axiom proc_ext`) since we must prove the equivalent of $P\{x/y\} = \tau.P_0\{x/y\}$, i.e., `(p' x)=(tau_pref (q' x))` (an analogous argument applies to the transition label).

Monotonicity. Let us consider the following lemma: if $P \xrightarrow{\alpha} P'$ and $x \notin fn(P)$, then $x \notin fn(\alpha) \cup fn(P')$. Again the proof is by induction on the derivation depth of the transition judgement; when we deal with the case of the RES rule, we have that $(z)P_1 \xrightarrow{\alpha} (z)P_1'$ ($P \equiv (z)P_1$, $P' \equiv (z)P_1'$), $x \notin fn(P)$. By inverting the hypothesis, we deduce $P_1 \xrightarrow{\alpha} P_1'$ and $z \notin n(\alpha)$. By induction hypothesis we know that $\forall\, y \notin fn(P_1)$ we have that $y \notin fn(\alpha) \cup fn(P_1')$. Using these two facts, we have then $x \notin fn(\alpha) \cup fn((z)P_1')$.

The above argument is natural "on paper", but in Coq P, P' correspond, respectively, to `(nu p1)`, `(nu p1')`. So P_1 and P_1' are represented by `(p1 x1)`, `(p1' x1)` where `x1` satisfies the following properties of freshness: `(notin x1 (nu p1))`, `(notin x1 (nu p1'))`, `~x1=x`.

The part of the thesis regarding the transition label is easy (as on paper), but in order to prove `(notin x (nu p1'))` we need monotonicity (`Axiom proc_mono`) since we only know `(notin x (nu p1))` (by hypothesis) and `(notin x (p1' x1))` (by induction hypothesis).

A.7 Technical lemmata for the π-calculus

```
(* A form of Lemma 1 for free and bound transitions *)
Lemma FTR_L1: (p,q:proc)(a:f_act)(x:name)(notin x p)->
                (ftrans p a q)->((f_act_notin x a)/\(notin x q)).

Lemma BTR_L1: (p:proc)(q:name->proc)(a:b_act)(x:name)(notin x p)->
                (btrans p a q)->((b_act_notin x a)/\(notin x (nu q))).

(* Lemmata 3 and 4 for free and bound transitions*)
```

[5] Recall that $P(z) =_\beta ((\lambda u : name.\ P(z))\ w)$ and $Q'(w)(z) =_\beta ((\lambda u : name.\ Q'(u)(z))\ w)$.

```
Lemma FTR_L3: (p,q:name->proc)(a:name->f_act)(x:name)
           (notin x (nu p))->(notin x (nu q))->(f_act_notin_ho x a)
           ->(ftrans (p x) (a x) (q x))
           ->(y:name)
               (notin y (nu p))->(notin y (nu q))->
               (f_act_notin_ho y a)->(ftrans (p y) (a y) (q y)).

Lemma BTR_L3: (p:name->proc)(q:name->name->proc)(a:name->b_act)(x:name)
           (notin x (nu p))->(notin x (nu [z:name](nu (q z))))->
             (b_act_notin_ho x a)
           ->(btrans (p x) (a x) (q x))
           ->(y:name)
               (notin y (nu p))->(notin y (nu [z:name](nu (q z))))->
               (b_act_notin_ho y a)->(btrans (p y) (a y) (q y)).

(* Lemma 6 *)
Lemma L6: (p,q:name->proc)(z:name)
         (notin z (nu p)) -> (notin z (nu q)) ->
         (StBisim (p z) (q z)) ->
 (w:name)~(w=z)->(notin w (nu p)) -> (notin w (nu q)) ->
         (StBisim (p w) (q w)).
```

A.8 Toolkit for the π-calculus

```
Section equivalence. (* Theorem 2'.1 *)
Variables p,q,r : proc.
Lemma REF : (StBisim p p).
Lemma SYM : (StBisim p q) -> (StBisim q p).
Lemma TRANS : (StBisim p q) -> (StBisim q r) -> (StBisim p r).
End equivalence.

Section structural_congruence.  (* Theorem 2'.2, 2'.3 *)
Variables p,q,r : proc.
Hypothesis  H : (StBisim p q).

Lemma TAU_S : (StBisim (tau_pref p) (tau_pref q)).
Lemma SUM_S : (StBisim (sum p r) (sum q r)).
Lemma PAR_S : (StBisim (par p r) (par q r)).
Lemma BANG_S : (StBisim (bang p) (bang q)).

Variables x,y : name.
Lemma MATCH_S : (StBisim (match x y p) (match x y q)).
Lemma MISMATCH_S : (StBisim (mismatch x y p) (mismatch x y q)).
Lemma OUT_S : (StBisim (out_pref x y p) (out_pref x y q)).

Variable p',q' : name->proc.
Lemma NU_S : ((z:name)(notin z (nu p')) -> (notin z (nu q'))
                      -> (StBisim (p' z) (q' z)))
             -> (StBisim (nu p') (nu q')).
Lemma IN_S: (notin y (nu p')) -> (notin y (nu q')) ->
           ((z:name)
```

```
             ((isin z (nu p'))\/(isin z (nu q'))\/z=y) ->
             (StBisim (p' z) (q' z))) ->
            (StBisim (in_pref x p') (in_pref x q')).
End structural_congruence.

Section monoidal_sum. (* Theorem 3 *)
Variables p,q,r : proc.
Lemma ID_SUM : (StBisim (sum p nil) p).
Lemma IDEM_SUM : (StBisim (sum p p) p).
Lemma COMM_SUM :  (StBisim (sum p q) (sum q p)).
Lemma ASS_SUM : (StBisim (sum p (sum q r)) (sum (sum p q) r)).
End monoidal_sum.


Section bang_unfolding. (* Theorem 4' *)
Lemma BANG_UNF : (p:proc)(StBisim (bang p) (par p (bang p))).
End bang_unfolding.

Section matching_laws. (* Theorem 5' *)
Variables p : proc.
Variables x,y : name.
Lemma MATCH1 : (x=y)->(StBisim (match x y p) p).
Lemma MATCH2 : ~(x=y)->(StBisim (match x y p) nil).
Lemma MISMATCH1 : ~(x=y)->(StBisim (mismatch x y p) p).
Lemma MISMATCH2 : (x=y)->(StBisim (mismatch x y p) nil).
End matching_laws.

Section restriction_laws. (* Theorems 6, 7 *)
Variable p : proc.
Variable p',q: name->proc.
Variable p'': name->name->proc.
Lemma NU_P: (StBisim (nu [x:name]p) p).
Lemma NU_COMM:(StBisim (nu [y:name](nu [z:name](p'' y z)))
                       (nu [z:name](nu [y:name](p'' y z)))).
Lemma NU_SUM :(StBisim (nu [y:name](sum (p' y) (q y)))
                       (sum (nu p') (nu q))).
Lemma NU_TAU_PREF: (StBisim (nu [x:name](tau_pref (p' x)))
                            (tau_pref (nu p'))).
Lemma NU_OUT_PREF: (x,y:name)
        (StBisim (nu [z:name](out_pref x y (p' z)))
                 (out_pref x y (nu p'))).
Lemma NU_IN_PREF: (x:name)
        (StBisim (nu [y:name](in_pref x (p'' y)))
                 (in_pref x [z:name](nu [y:name](p'' y z)))).
Lemma NU_NIL1: (x:name)(StBisim (nu [y:name](out_pref y x p)) nil).
Lemma NU_NIL2: (StBisim (nu [y:name](in_pref y p')) nil).
End restriction_laws.

Section monoidal_par. (* Theorem 8 *)
Variables p,q,r : proc.
```

```
Lemma ID_PAR  : (StBisim (par p nil) p).
Lemma COMM_PAR: (StBisim (par p q) (par q p)).
Lemma ASS_PAR : (StBisim (par (par p q) r) (par p (par q r))).
Variables p'  : name->proc.
Lemma NU_EXTR : (StBisim (nu [y:name](par (p' y) q)) (par (nu p') q)).
Lemma NU_PAR1 : (StBisim (nu [x:name](par (p' x) q))
                         (par (nu p') (nu [x:name]q))).
Lemma NU_PAR2 : (StBisim (nu [x:name](par p (p' x)))
                         (par (nu [x:name]p) (nu p'))).
End monoidal_par.
```

A.9 An example proof in $\Sigma^{\pi+}$

In this section, we develop formally one of the cases in the proof of the associativity of $|$. In [25, Theorem 8(d)], this property is proved using a bisimulation up-to $\dot{\sim}$ and restriction. In our approach, instead, we take advantage of the `Cofix` tactic.

In order to make the proof fragment more readable and to allow a comparison with the corresponding proof on paper, we insert comments to explain the effect of the tactics applied, and we list the proof environments which are incrementally generated. It is worthwhile noticing that the main difference between the formal development of the proof in Coq and its "informal" counterpart is the incremental building of the necessary bisimulation with no need of any explicit reference to bisimulations "up-to".

```
Lemma ASS_PAR:(p,q,r:proc)(StBisim (par (par p q) r) (par p (par q r))).
Proof.
1 subgoal
  ============================
   (p,q,r:proc)(StBisim (par (par p q) r) (par p (par q r)))

(* First of all we apply the Cofix tactic which introduces
 * the thesis among the hypotheses; then we apply the introduction
 * rule sb in order to ensure that all the subsequent applications
 * of the thesis will be guarded, hence legal *)

ASS_PAR < Cofix; Intros; Apply sb; Do 3 Try (Split; Intros).
6 subgoals

  ASS_PAR : (p,q,r:proc)(StBisim (par (par p q) r) (par p (par q r)))
  p : proc
  q : proc
  r : proc
  a : f_act
  p1 : proc
  H : (ftrans (par (par p q) r) a p1)
  ============================
   (EX q1:proc | (ftrans (par p (par q r)) a q1)/\(StBisim p1 q1))

subgoal 2 is:
 (EX p1:proc | (ftrans (par (par p q) r) a p1)/\(StBisim p1 q1))
subgoal 3 is:
 (EX q1:name->proc |
```

```
    (btrans (par p (par q r)) (In x) q1)
    /\((y:name)(StBisim (p1 y) (q1 y))))
subgoal 4 is:
 (EX p1:name->proc |
    (btrans (par (par p q) r) (In x) p1)
    /\((y:name)(StBisim (p1 y) (q1 y))))
subgoal 5 is:
 (EX q1:name->proc |
    (btrans (par p (par q r)) (bOut x) q1)
    /\((y:name)
        (notin y (nu p1))->(notin y (nu q1))->(StBisim (p1 y) (q1 y))))
subgoal 6 is:
 (EX p1:name->proc |
    (btrans (par (par p q) r) (bOut x) p1)
    /\((y:name)
        (notin y (nu p1))->(notin y (nu q1))->(StBisim (p1 y) (q1 y))))

(* We now have 3 + 3 subgoals to consider (in fact each case comes
 * with its symmetric) corresponding to the clauses in the
 * definition of strong late bisimilarity *)

(* ...some cases omitted... *)

(* case 6 in Milner, Parrow, Walker, Appendix 4.2, Theorem 8(d) *)

subgoal 1 is:

  ASS_PAR : (p,q,r:proc)(StBisim (par (par p q) r) (par p (par q r)))
  p : proc
  q : proc
  r : proc
  a : f_act
  p1 : proc
  p3 : proc
  q1 : name->proc
  q2 : name->proc
  x : name
  H : (btrans p (bOut x) q1)
  H1 : (btrans q (In x) q2)
  ============================
   (EX q0:proc |
      (ftrans (par p (par q r)) tau q0)
      /\(StBisim (par (nu [z:name](par (q1 z) (q2 z))) r) q0))

(* Let (P|Q)|R --tau--> P'; we supply explicitly
 * the process Q' corresponding to P' *)
ASS_PAR < Exists (nu [_:name](par (q1 _) (par (q2 _) r))); Split.

subgoal 1 is:
```

```
  ASS_PAR : (p,q,r:proc)(StBisim (par (par p q) r) (par p (par q r)))
  p : proc
  q : proc
  r : proc
  a : f_act
  p1 : proc
  p3 : proc
  q1 : name->proc
  q2 : name->proc
  x : name
  H : (btrans p (bOut x) q1)
  H1 : (btrans q (In x) q2)
  ============================
   (ftrans (par p (par q r)) tau
     (nu [_:name](par (q1 _) (par (q2 _) r))))

subgoal 2 is:
   (StBisim (par (nu [z:name](par (q1 z) (q2 z))) r)
     (nu [_:name](par (q1 _) (par (q2 _) r))))

(* we prove the first subgoal, which claims that P|(Q|R) --tau--> Q' *)
ASS_PAR < Change (ftrans (par p (par q r)) tau
ASS_PAR <          (nu [_:name](par (q1 _) ([u:name](par (q2 u) r) _))));
ASS_PAR < Apply CLOSE2 with x; [Assumption | Apply bPAR1; Assumption].

(* the remaining goal claims  P' ~ Q': *)
subgoal 1 is:
  ASS_PAR : (p,q,r:proc)(StBisim (par (par p q) r) (par p (par q r)))
  p : proc
  q : proc
  r : proc
  a : f_act
  p1 : proc
  p3 : proc
  q1 : name->proc
  q2 : name->proc
  x : name
  H : (btrans p (bOut x) q1)
  H1 : (btrans q (In x) q2)
  ============================
   (StBisim (par (nu [z:name](par (q1 z) (q2 z))) r)
     (nu [_:name](par (q1 _) (par (q2 _) r))))

(* Now we directly prove that P' ~ Q'.  Notice that we do not introduce
 * any auxiliary notion of bisimulation "up-to": the necessary
 * bisimulation is built implicitly by applying the coinductive
 * hypothesis ASS_PAR *)

ASS_PAR < Apply SYM;
ASS_PAR < Apply TRANS with (nu [z:name](par (par (q1 z) (q2 z)) r)).
```

```
subgoal 1 is:
  ASS_PAR : (p,q,r:proc)(StBisim (par (par p q) r) (par p (par q r)))
  p : proc
  q : proc
  r : proc
  a : f_act
  p1 : proc
  p3 : proc
  q1 : name->proc
  q2 : name->proc
  x : name
  H : (btrans p (bOut x) q1)
  H1 : (btrans q (In x) q2)
  ============================
   (StBisim (nu [_:name](par (q1 _) (par (q2 _) r)))
     (nu [z:name](par (par (q1 z) (q2 z)) r)))

ASS_PAR < Apply NU_S; Intros; Apply SYM; Apply ASS_PAR.
(* where Lemma NU_S states that StBisim is a congruence with
 * respect to nu, see Appendix A.8 *)
subgoal 1 is:

  ASS_PAR : (p,q,r:proc)(StBisim (par (par p q) r) (par p (par q r)))
  p : proc
  q : proc
  r : proc
  a : f_act
  p1 : proc
  p3 : proc
  q1 : name->proc
  q2 : name->proc
  x : name
  H : (btrans p (bOut x) q1)
  H1 : (btrans q (In x) q2)
  ============================
   (StBisim (nu [z:name](par (par (q1 z) (q2 z)) r))
     (par (nu [z:name](par (q1 z) (q2 z))) r))

ASS_PAR < Change (StBisim (nu [z:name](par ([_:name](par (q1 _)
ASS_PAR <                                                   (q2 _) z) r))
ASS_PAR <                  (par (nu [z:name](par (q1 z) (q2 z))) r));
ASS_PAR < Apply NU_EXTR.
(* where NU_EXTR is the scope extrusion law, see Appendix A.8 *)

(* end of the case - other cases omitted *)
Qed.
```