Electronic Notes in Theoretical Computer Science 70 No. 2 (2002)
URL: `http://www.elsevier.nl/locate/entcs/volume70.html` 20 pages

Ambient Calculus and its Logic in the Calculus of Inductive Constructions

Ivan Scagnetto, Marino Miculan

Dipartimento di Matematica e Informatica, Università di Udine, Italy
scagnett@dimi.uniud.it,miculan@dimi.uniud.it

Abstract

The Ambient Calculus has been recently proposed as a model of mobility of agents in a dynamically changing hierarchy of domains. In this paper, we describe the implementation of the theory and metatheory of Ambient Calculus and its modal logic in the Calculus of Inductive Constructions. We take full advantage of Higher-Order Abstract Syntax, using the *Theory of Contexts* as a fundamental tool for developing formally the metatheory of the object system. Among others, we have successfully proved a set of *fresh renamings* properties, and formalized the connection between the Theory of Contexts and Gabbay-Pitts' "new" quantifier. As a feedback, we introduce a new definition of satisfaction for the Ambients logic and derive some of the properties originally assumed as axioms in the Theory of Contexts.

Introduction

A *Logical Framework* (LF) is a generic *logic specification language* where we can represent (encode) all the relevant notions of an object system: syntactic categories, terms, assertions, axiom and rule schemata, tactics, etc. Since the '80's, higher order intuitionistic type theories have been successfully experimented as Logical Frameworks. The basic idea is the "judgements-as-types, λ-terms-as-proofs" paradigm. A key technique commonly adopted for building adequate encodings is the *higher order abstract syntax* (HOAS), whereby binding operators are represented by constructors of higher order type [8,6,13]. Encoding the binders of the object language by means of the λ-abstraction operator of type-theoretic metalanguages frees the user from the burden of encoding the related machinery (i.e, α-conversion and capture avoiding substitution), since it is automatically provided by the LF itself.

However, it is well known that the HOAS-based encoding approach presents some drawbacks. For instance, object level variables cannot be encoded by means of an inductive type; otherwise, being equated to metalanguage variables, *exotic terms* arise [6, 13]. A second issue is related to the difficulty of

⋆ Work supported by italian MIUR COFIN 2001013518 *CoMeta* and UE WG 29001 *Types*.

reasoning by induction and using recursion over contexts, since they are rendered as functional terms. Finally, one looses the possibility of handling and proving properties over the mechanisms delegated to the metalanguage.

In order to overcome these drawbacks, in [11,10,22] a general methodology for reasoning on systems in HOAS is presented, based on an *axiomatic* syntactic standpoint. The gist is to extend the framework with a set of axioms, called the *Theory of Contexts*, capturing some basic and natural properties of *names* and *contexts*. According to our experience, these axioms allow for a smooth handling of schemata in HOAS. In fact, we feel that one of the main advantages of the axiomatic approach of the Theory of Contexts, is that it requires a very low mathematical and logical overhead.

Along this line of research, in this paper we present an encoding of the *Ambient Calculus* [2] and its modal logic as presented in [4] in the *Calculus of Inductive Constructions* (CIC), using the proof environment `Coq` [12]. We then use this implementation for formally derive several lemmata of [4], especially related to the notion of freshness of names and Gabbay-Pitts' "new" quantifier.

The present work is important both from the point of view of Logical Frameworks and of process algebras under many respects.

First, the encoding in a Logical Framework forces to spell out in full detail all aspects of an object system. This enlights problematic issues which are "swept under rug" on the paper, giving the possibility to identify and fix peculiar idiosyncrasies. This is particularly useful in the case of Ambient Calculus, which is a fairly new process algebra, still under development. In this setting, the LF perspective may suggest alternative (and cleaner) definitions of many fundamental concepts. For instance, the encoding of formulas sheds some light on the (inessential) difference between names and variables, and we introduce an alternative proof system in Natural Deduction style for satisfaction.

Moreover, many features of Ambients are still under discussion, and several variants (Safe, Boxed, . . .) are being proposed. The encoding of these different variants in a general metalanguage allows for a comparison of their features on a common ground. The work presented in this paper can be seen as the basis for further developments in several directions (e.g., new constructors, typing systems, different semantics, . . .).

On the LF side, we have the possibility to test the Theory of Contexts and its methodology over peculiar issues which have not been faced in previous case studies [11,14]; e.g., a modal logic, the presence of a sort of variables ranging over names, and the "new" quantifier "N" by Gabbay and Pitts [7]. Since both Nominal Logic and the Theory of Contexts aim to provide a framework for reasoning on nominal calculi, it is interesting to see how the Theory of Contexts allows for rendering the N quantifier and its properties.

From a more fundamental standpoint, an important result of this work regards the independence of the axioms of the Theory of Contexts. We have proved that some properties originally introduced as axioms in [11] can be actually derived from the others, using suitable inductions over sizes of terms.

$M ::=$	Capabilities	$P, Q, R ::=$	Processes
n	name	$(\nu n)P$	restriction
$in\ M$	can enter into M	$\mathbf{0}$	void
$out\ M$	can exit out of M	$P \vert Q$	composition
$open\ M$	can open M	$!P$	replication
ε	null	$M[P]$	ambient
$M.M'$	path	$M.P$	capability action
		$(n).P$	input action
		$\langle M \rangle$	output action

Fig. 1. Syntax of capabilities and processes.

Synopsis. A short remind of the object system is provided in Section 1. The HOAS-based encoding of Ambient Calculus and its logic is presented in Section 2. In Section 3 we present and discuss some developments of the metatheory of Ambient Calculus using the Theory of Contexts. Final conclusions, with related and future work, are in Section 4. Appendix A gives a brief account of the Calculus of Inductive Constructions and its implementation, `Coq`.

1 The Ambient Calculus and its Modal Logic

In this section we briefly recall the syntax and operational semantics of the Ambient Calculus and the syntax and satisfaction relation of the modal logic introduced in [4]. For notation and an informal description of the intended meaning of ambient processes and formulas, see [2, 4].

Processes, Structural Congruence, Reduction System

The basic syntactic categories of the Ambient Calculus are names, capabilities and processes (or agents), defined as in Figure 1. There are no binders among the capability constructors, while processes provide the usual binders, i.e., restriction and input action. Free names of capabilities and processes are defined as usual. Processes are identified up to α-conversion. We will denote by $P\{n \leftarrow M\}$, the capture avoiding substitution of the capability M for the free occurrences of n into P. Following the notation adopted in [4] we will denote by Λ the sort of names, and by ζ, Π the syntactic categories of capabilities and processes, respectively.

The operational semantics of processes is given in terms of a *reduction system*; processes are identified up to some minor (spatial) rearrangements specified by a structural congruence relation (Figure 2).

Formulas, Satisfaction

Logical formulas are defined by the grammar in Figure 3, where η can be a name n or a variable x. The sets of free names $fn(\mathcal{A})$ and free variables $fv(\mathcal{A})$

$$\frac{-}{P \equiv P}\ \text{(S Refl)} \qquad \frac{-}{(\nu n)(\nu m)P \equiv (\nu m)(\nu n)P}\ \text{(S Res Res)}$$

$$\frac{P \equiv Q}{Q \equiv P}\ \text{(S Symm)} \qquad \frac{-}{(\nu n)\mathbf{0} \equiv \mathbf{0}}\ \text{(S Res Zero)}$$

$$\frac{P \equiv Q, \quad Q \equiv R}{P \equiv R}\ \text{(S Trans)} \qquad \frac{n \notin fn(P)}{(\nu n)(P|Q) \equiv P|(\nu n)Q}\ \text{(S Res Par)}$$

$$\frac{P \equiv Q}{(\nu n)P \equiv (\nu n)Q}\ \text{(S Res)} \qquad \frac{n \neq m}{(\nu n)(m[P]) \equiv m[(\nu n)P]}\ \text{(S Res Amb)}$$

$$\frac{P \equiv Q}{P|R \equiv Q|R}\ \text{(S Par)} \qquad \frac{-}{P|\mathbf{0} \equiv P}\ \text{(S Par Zero)}$$

$$\frac{P \equiv Q}{!P \equiv !Q}\ \text{(S Repl)} \qquad \frac{-}{P|Q \equiv Q|P}\ \text{(S Par Comm)}$$

$$\frac{P \equiv Q}{n[P] \equiv n[Q]}\ \text{(S Amb)} \qquad \frac{-}{(P|Q)|R \equiv P|(Q|R)}\ \text{(S Par Assoc)}$$

$$\frac{P \equiv Q}{M.P \equiv M.Q}\ \text{(S Action)} \qquad \frac{-}{!\mathbf{0} \equiv \mathbf{0}}\ \text{(S Repl Zero)}$$

$$\frac{P \equiv Q}{(n).P \equiv (n).Q}\ \text{(S Input)} \qquad \frac{-}{!(P|Q) \equiv !P|!Q}\ \text{(S Repl Par)}$$

$$\frac{-}{\varepsilon.P \equiv P}\ \text{(S } \varepsilon\text{)} \qquad \frac{-}{!P \equiv P|!P}\ \text{(S Repl Copy)}$$

$$\frac{-}{(M.M').P \equiv M.M'.P}\ \text{(S .)} \qquad \frac{-}{!P \equiv !!P}\ \text{(S Repl Repl)}$$

$$\frac{-}{n[in\ m.P|Q]|m[R] \to m[n[P|Q]|R]}\ \text{(Red In)} \qquad \frac{P \to Q}{(\nu n)P \to (\nu n)Q}\ \text{(Red Res)}$$

$$\frac{-}{m[n[out\ m.P|Q]|R] \to n[P|Q]|m[R]}\ \text{(Red Out)} \qquad \frac{P \to Q}{P|R \to Q|R}\ \text{(Red Par)}$$

$$\frac{P' \equiv P, \quad P \to Q, \quad Q \equiv Q'}{P' \to Q'}\ \text{(Red} \equiv\text{)} \qquad \frac{P \to Q}{n[P] \to n[Q]}\ \text{(Red Amb)}$$

$$\frac{-}{(n).P|\langle M\rangle \to P\{n \leftarrow M\}}\ \text{(Red Comm)} \qquad \frac{-}{open\ n.P|n[Q] \to P|Q}\ \text{(Red Open)}$$

Fig. 2. Structural Congruence and Reduction System.

$\mathcal{A}, \mathcal{B}, \mathcal{C} ::= \mathbf{T}$	true	$\eta[\mathcal{A}]$	location	$\eta ® \mathcal{A}$	revelation
$\neg\mathcal{A}$	negation	$\mathcal{A}@\eta$	location adjunct	$\diamond\!\!\!\!\cdot\ \mathcal{A}$	somewhere modality
$\mathcal{A} \vee \mathcal{B}$	disjunction	$\mathcal{A} \oslash \eta$	revelation adjunct		
$\mathbf{0}$	inaction	$\mathcal{A} \triangleright \mathcal{B}$	composition adjunct	$\diamond\mathcal{A}$	sometime modality
$\mathcal{A}\|\mathcal{B}$	composition	$\forall x \mathcal{A}$	universal quantifier		

Fig. 3. Syntax of formulas.

of a formula $\mathcal{A}$ are defined as usual (Notice that there are no name binders). A formula $\mathcal{A}$ is closed if $fv(\mathcal{A}) = \emptyset$, while $\mathcal{A}\{\eta \leftarrow \mu\}$ denotes the substitution of a name or variable μ for another name or variable η (variables can range only over names). Formulas are identified up to α-conversion. We will denote by ϑ the sort of variables and by Φ the syntactic category of formulas.

$$
\begin{array}{rcl@{\qquad}rcl}
P \models \mathbf{T} & & & P \models \mathbf{0} & \iff & P \equiv \mathbf{0} \\
P \models \neg\mathcal{A} & \iff & \text{not } P \models \mathcal{A} & P \models \mathcal{A} \vee \mathcal{B} & \iff & P \models \mathcal{A} \text{ or } P \models \mathcal{B} \\
P \models \mathcal{A} \oslash n & \iff & (\nu n)P \models \mathcal{A} & P \models \mathcal{A}@n & \iff & n[P] \models \mathcal{A}
\end{array}
$$

$$
\begin{array}{rcl}
P \models \mathcal{A}|\mathcal{B} & \iff & \text{there exist } P', P'' \in \Pi \text{ s.t. } P \equiv P'|P'', P' \models \mathcal{A} \text{ and } P'' \models \mathcal{B} \\
P \models \mathcal{A} \rhd \mathcal{B} & \iff & \text{for all } P' \in \Pi, P' \models \mathcal{A} \text{ implies } P|P' \models \mathcal{B} \\
P \models n[\mathcal{A}] & \iff & \text{there exists } P' \in \Pi \text{ such that } P \equiv n[P'] \text{ and } P' \models \mathcal{A} \\
P \models n \circledR \mathcal{A} & \iff & \text{there exists } P' \in \Pi \text{ such that } P \equiv (\nu n)P' \text{ and } P' \models \mathcal{A} \\
P \models \Diamond\mathcal{A} & \iff & \text{there exists } P' \in \Pi \text{ such that } P \rightarrow^* P' \text{ and } P' \models \mathcal{A} \\
P \models \diamondsuit\mathcal{A} & \iff & \text{there exists } P' \in \Pi \text{ such that } P \downarrow^* P' \text{ and } P' \models \mathcal{A} \\
P \models \forall x\mathcal{A} & \iff & \text{for all } m \in \Lambda, P \models \mathcal{A}\{x \leftarrow m\}
\end{array}
$$

Fig. 4. Satisfaction.

The relation between processes and *closed* formulas is established by the *satisfaction relation*: $P \models \mathcal{A}$ means that the process P satisfies the closed formula $\mathcal{A}$ as specified by the rules in Figure 4. In particular, the satisfaction for the temporal modality is defined by means of the reduction relation ($\rightarrow^*$ is the reflexive and transitive closure of $\rightarrow$). As to the spatial modality instead, its satisfaction is given by means of the *nesting* relation $\downarrow$: we have $P \downarrow P'$ if and only if there exists a name n and a process P'' such that $P \equiv n[P']|P''$. The relation $\downarrow^*$ is then defined as the reflexive and transitive closure of $\downarrow$.

2 Encoding Ambients in `Coq`

In this section, we describe the HOAS-based encoding of the Ambient Calculus and its logic. For the full signature, see [21].

2.1 Names, Capabilities, Processes

```
Parameter name: Set.
Inductive cap: Set :=
    name2cap : name -> cap
  | in_cap   : cap -> cap
  | out_cap  : cap -> cap
  | open     : cap -> cap
  | eps      : cap
  | path    : cap->cap->cap.
```

Names of the Ambient Calculus will be represented by variables of `Coq` of type `name`; hence, the first constant in our signature Σ_A represents the sort of names. Capabilities are easily encoded by the plain (i.e., first order) inductive definition aside.

In the following, for $N \triangleq \{n_1, \ldots, n_k\} \subset \Lambda$ finite, we will denote by ζ_N the set $\{M \in \zeta \mid fn(M) \subseteq N\}$. Moreover, we will denote by Γ_N the `Coq` typing environment $\{\mathtt{n1} : \mathtt{name}, \ldots, \mathtt{nk} : \mathtt{name}\} \cup \{\mathtt{dij} : \tilde{}\,(\mathtt{ni} = \mathtt{nj}) \mid 1 \leq \mathtt{i} < \mathtt{j} \leq \mathtt{k}\}$.[1] Finally $\mathtt{cap}_N$ will represent the canonical forms `M` (i.e. $\beta\eta$-head normal forms) of type `cap` such that $\Gamma_N \vdash_{\Sigma_A} \mathtt{M} : \mathtt{cap}$. The following proposition stating the adequacy of the encoding is proved by usual inductions.

[1] This can be declared, e.g., by `Parameter n1,...,nk:name` followed by the statements `Axiom dij:~(ni=nj)` for $1 \leq i < j \leq k$.

Proposition 2.1 *For each $N \subset \Lambda$ finite, there is a compositional bijection ϵ^{ζ}_N (with inverse δ^{ζ}_N) between ζ_N and* `cap`$_N$*.*

The syntactic category of processes features two binders, namely, restriction and input action: in this case we take a higher-order encoding approach.

```
Inductive proc: Set :=
    nu     :(name -> proc) -> proc
  | nil     : proc
  | par     : proc -> proc -> proc
  | bang    : proc -> proc
  | ambient : cap -> proc -> proc
  | cap_act : cap -> proc -> proc
  | in_act :(name -> proc) -> proc
  | out_act : cap -> proc.
```

Since the arguments of `nu` and `in_act` are functions, α-conversion and capture avoiding substitution of names are automatically delegated to the metalanguage.

As in the case of the capabilities encoding, for $N \subset \Lambda$ finite, we will denote by Π_N the set $\{P \mid P \in \Pi, fn(P) \subseteq N\}$, and `proc`$_N$ will represent the canonical forms `P` of type `proc` such that $\Gamma_N \vdash_{\Sigma_A}$ `P : proc`. Then:

Proposition 2.2 *For each $N \subset \Lambda$ finite, there is a compositional bijection ϵ^{Π}_N (with inverse δ^{Π}_N) between Π_N and* `proc`$_N$*.*

Correctly, this correspondence covers also “non-well-formed”, meaningless terms, like *in* (*out* a).0, which still do belong to the syntax of the original calculus (Figure 1). These “wrong” terms can be ruled out by means of a type system, like e.g. *exchange types* [1]. Although we could address this problem in the present paper, we feel that the treatment of the theory and metatheory (e.g. subject reductions) of type systems for Ambients is interesting on its own and deserves an in-depth discussion, which we leave as future work.

Notice that, although capabilities can be exchanged in communications, we cannot take `in_act:(cap -> proc) -> proc` because this would give rise to exotic terms (see Appendix A). Thus we cannot delegate to the metalanguage the (capture-avoiding) substitution of capabilites for names, but only α-conversion and substitution of names for names. Substitution of capabilities for names in capabilities and processes must be encoded explicitly by means of two relations `subst_cap:cap->(name->cap)->cap` and `subst_proc:cap->(name->proc)->proc`. For instance, the intuitive meaning of `(subst_proc M P P’)` is that `P’` is the result of “filling the hole” in `P` with `M`. We show only two cases of the definition of `subst_proc`:

```
Inductive subst_proc [M:cap]: (name->proc) -> proc -> Prop :=
  subst_proc_nu  : (P:name->name->proc)(P’:name->proc)
             ((y:name)(subst_proc M [x:name](P x y) (P’ y))) ->
             (subst_proc M [x:name](nu (P x)) (nu P’))  ...
| subst_proc_par : (P,Q:name->proc)(P’,Q’:proc)
             (subst_proc M P P’) -> (subst_proc M Q Q’) ->
             (subst_proc M [x:name](par (P x) (Q x)) (par P’ Q’))
```

Proposition 2.3 (Adequacy of substitution) *Let $L \subset \Lambda$ finite, $M, N' \in$*

ζ_L, $N \in \zeta_{L \uplus \{n\}}$, $P' \in \Pi_L$, $P \in \Pi_{L \uplus \{n\}}$, then:

- $N\{n \leftarrow M\} = N'$ *iff* $\Gamma_L \vdash_{\Sigma_A} \_ : (\texttt{subst\_cap}\ \epsilon^{\zeta}_L(M)\ [\texttt{n} : \texttt{name}]\epsilon^{\zeta}_{L,n}(N)\ \epsilon^{\zeta}_L(N'))$,
- $P\{n \leftarrow M\} = P'$ *iff* $\Gamma_L \vdash_{\Sigma_A} \_ : (\texttt{subst\_proc}\ \epsilon^{\zeta}_L(M)\ [\texttt{n} : \texttt{name}]\epsilon^{\Pi}_{L,n}(P)\ \epsilon^{\Pi}_L(P'))$.

2.2 Formulas

The original syntax of formulas (Figure 3) introduces an additional sort of variables in order to have a universal quantifier and reasoning about open formulas. However, naïvely introducing an explicit type `var` for representing variables would yield several problems. First of all, one would need to duplicate the constructors $[\cdot]$, @, ® and ⃠ since one of their arguments may be either a name or a variable. Even worse, the universal quantifier would be represented by a constructor of type `(var->form)->form`; then we could not encode substitution of names for variables by means of the functional application of the metalanguage. Indeed, it would be impossible to apply a term of type `var->form` to a term of type `name`. Hence, we could delegate to the metalanguage only α-conversion of formulas.

The solution we adopt is to encode variables of the object language by means of `Coq` variables of type `name`, following a *full HOAS* paradigm like, e.g., for First Order Logic in LF [8]. Recall that the only λ-terms inhabiting `name` are metalanguage (`Coq`) variables. The difference between variables representing names and those representing object language variables is that no inequality assumptions are taken on the latter. Indeed, a variable is only a placeholder waiting to be replaced by a name, whence we cannot make a priori any assumptions on the nature of the name that will eventually replace it. On the other hand, metalanguage variables representing names come equipped with inequality judgments that allow to think of them as constants. In fact, a name is just a variable whose value is different from all others—what exactly this value *is*, it does not matter (see the notion of *equivariance* in [19]). Thus, the inductive type representing formulas is given as follows.

```
Inductive form: Set :=  T : form
  | neg        : form -> form
  | Or         : form -> form -> form
  | zero       : form
  | comp       : form -> form -> form
  | comp_adj   : form -> form -> form
  | loc        : name -> form -> form
  | loc_adj    : form -> name -> form
  | rev        : name -> form -> form
  | rev_adj    : form -> name -> form
  | sometime   : form -> form
  | somewhere  : form -> form
  | forall     : (name->form) -> form.
```

Given a finite set of variables $X \triangleq \{x_1, \ldots, x_n\} \subset \vartheta$, in the rest of this section we will denote by Γ_X the typing environment $\{\texttt{x1} : \texttt{name}, \ldots, \texttt{xn} : \texttt{name}\}$. Canonical forms `t` of type `form` such that $\Gamma_N, \Gamma_X \vdash_{\Sigma_A} \texttt{t} : \texttt{form}$ are denoted by $\texttt{form}_{N,X}$. Then:

Proposition 2.4 *For each $N \subset \Lambda$ finite, $X \subset \vartheta$, there is a compositional bijection $\epsilon^{\Phi}_{N,X}$ (with inverse $\delta^{\Phi}_{N,X}$) between $\Phi_{N,X}$ and* $\texttt{form}_{N,X}$.

2.3 Encoding freshness

Following the general pattern given in [10], we render the notion of freshness in our encoding (i.e., the fact that a given name does not occur free in a capability, process or formula) by means of three inductive predicates:

(i) `(notin_cap x M)` holds iff the name `x` does not occur in `M`;

(ii) `(notin_proc x P)` holds iff the name `x` does not occur in `P`;

(iii) `(notin_form x A)` holds iff the name `x` does not occur in `A`.

We report here only the (partial) definition of `notin_proc`:

```
Inductive notin_proc [m:name]: proc -> Prop :=
    notin_proc_nu   : (P:name->proc)
                      ((n:name)~m=n -> (notin_proc m (P n))) ->
                      (notin_proc m (nu P))
...
  | notin_proc_par  : (P,Q:proc)
                      (notin_proc m P) -> (notin_proc m Q) ->
                      (notin_proc m (par P Q))
...
```

2.4 Encoding of Structural Congruence and Reductions

The encoding of the structural congruence relation and of the reduction system is carried out in a straightforward way by two inductively defined predicates: `struct_eq` and `red`, respectively. Both definitions provide a distinct constructor for each rule defining the corresponding relation of the object language; hence we have 22 introduction rules for `struct_eq` and 8 introduction rules for `red`. We cannot report the whole definitions, but only two cases:

```
Inductive struct_eq: proc -> proc -> Prop :=  ...
| struct_res_par : (P:proc)(Q:name->proc)
            (struct_eq (nu [n:name](par P (Q n))) (par P (nu Q)))
...
```

Notice how the non-occurrence condition on n in (S Res Par) is automatically dealt with by the metalanguage.

```
Inductive red: proc -> proc -> Prop :=  ...
| red_comm : (P:name->proc)(M:cap)(P':proc)(subst_proc M P P') ->
             (red (par (in_act P) (out_act M)) P')
...
```

The adequacy of the encodings introduced in this section are stated by the following results, proved by standard structural inductions:

Proposition 2.5 (Adequacy of `struct_eq`) *Let $N \subset \Lambda$ finite, $P, Q \in \Pi_N$;*

(i) *(Soundness) if* $\Gamma_N \vdash_{\Sigma_A} \_ : (\texttt{struct\_eq P Q})$*, then* $\delta^{\Pi}_N(\texttt{P}) \equiv \delta^{\Pi}_N(\texttt{Q})$*;*

(ii) *(Completeness) if* $P \equiv Q$*, then* $\Gamma_N \vdash_{\Sigma_A} \_ : (\texttt{struct\_eq}\ \epsilon^{\Pi}_N(P)\ \epsilon^{\Pi}_N(Q))$*.*

Proposition 2.6 (Adequacy of red) *Let $N \subset \Lambda$ finite, $P, Q \in \Pi_N$:*

(i) *(Soundness) if* $\Gamma_N \vdash_{\Sigma_A} \_ : (\texttt{red P Q})$*, then* $\delta^{\Pi}_N(\texttt{P}) \to \delta^{\Pi}_N(\texttt{Q})$.

(ii) *(Completeness) if* $P \to Q$*, then* $\Gamma_N \vdash_{\Sigma_A} \_ : (\texttt{red}\ \epsilon^{\Pi}_N(P)\ \epsilon^{\Pi}_N(Q))$.

We skip the encoding of the nesting relation, which is a trivial `Coq` `Definition` using `struct_eq` (hence, the adequacy directly follows from Proposition 2.5).

2.5 Encoding of Satisfaction

In encoding the satisfaction relation, we cannot use directly an inductive definition since the introduction rules for $\neg$ and $\triangleright$ do not satisfy the positivity constraints imposed by the `Coq` type system on inductive constructors. Indeed, the encoding would look like the following:

```
Inductive sat: proc -> form -> Prop := ...
| sat_neg      : (P:proc)(A:form)~(sat P A) -> (sat P (neg A))
| sat_comp_adj : (P:proc)(A,B:form)
                 ((P':proc)(sat P' A) -> (sat (par P P') B))
                 -> (sat P (comp_adj A B)) ...
```

which is not admitted in CIC because the occurrences `(sat P A)` in `sat_neg` and `(sat P' A)` in `sat_comp_adj` are in negative position. In this section, we present and discuss three possibles ways for circumventing this obstacle.

Axiomatic solution Much in the spirit of the Edinburgh LF [8], one can drop inductive types and to fall back to the plain Calculus of Constructions. Thus, the `sat` judgment and its constructors are declared as axioms:

```
Parameter sat : proc -> form -> Prop.  ...
Axiom sat_neg: (P:proc)(A:form)~(sat P A) <-> (sat P (neg A)).
Axiom sat_comp_adj: (P:proc)(A,B:form)
                    ((P':proc)(sat P' A) -> (sat (par P P') B))
                    <-> (sat P (comp_adj A B)).  ...
```

The adequacy of this encoding is proved by usual inductions:

Proposition 2.7 (Adequacy of sat) *For* $N \subset \Lambda$ *finite,* $P \in \Pi_N$, $\mathcal{A} \in \Phi_{N,\emptyset}$,

(i) *(Soundness) if* $\Gamma_N \vdash_{\Sigma_A} \_ : (\texttt{sat P A})$*, then we have* $\delta^{\Pi}_N(\texttt{P}) \models \delta^{\Phi}_{N,\emptyset}(\texttt{A})$.

(ii) *(Completeness) if* $P \models A$*, then* $\Gamma_N \vdash_{\Sigma_A} \_ : (\texttt{sat}\ \epsilon^{\Pi}_N(P)\ \epsilon^{\Phi}_{N,\emptyset}(\mathcal{A}))$.

Notice that $\mathcal{A}$ ranges only over $\Phi_{N,\emptyset}$ (closed formulas), since the satisfaction relation is defined only on formulas with no free variables.

Mutual Inductive solution The axiomatic solution works fine as long as we do not need a structural induction principle on `sat`-derivations. Nevertheless, it is clear that an inductive definition of the satisfaction relation may be useful for future proof-theoretical investigations of the Modal Logic. Here we introduce a such alternative definition, which is suggested by the encoding

$$\frac{}{P \models_i \mathbf{T}}$$

$$\frac{P \equiv 0}{P \models_i \mathbf{0}} \qquad \frac{P \not\equiv 0}{P \not\models_i \mathbf{0}}$$

$$\frac{P \not\models_i \mathcal{A}}{P \models_i \neg\mathcal{A}} \qquad \frac{P \models_i \mathcal{A}}{P \not\models_i \neg\mathcal{A}}$$

$$\frac{P \models_i \mathcal{A}}{P \models_i \mathcal{A} \vee \mathcal{B}} \quad \frac{P \models_i \mathcal{B}}{P \models_i \mathcal{A} \vee \mathcal{B}} \qquad \frac{P \not\models_i \mathcal{A} \quad P \not\models_i \mathcal{B}}{P \not\models_i \mathcal{A} \vee \mathcal{B}}$$

$$\frac{n[P] \models_i \mathcal{A}}{P \models_i \mathcal{A}@n} \qquad \frac{n[P] \not\models_i \mathcal{A}}{P \not\models_i \mathcal{A}@n}$$

$$\frac{\text{for some } P' : P \equiv n[P'] \text{ and } P' \models_i \mathcal{A}}{P \models_i n[\mathcal{A}]} \qquad \frac{\text{for all } P',\ \text{if } P \equiv n[P'] \text{ then } P' \not\models_i \mathcal{A}}{P \not\models_i n[\mathcal{A}]}$$

$$\frac{\text{for some } P', P'' : P \equiv P'|P'', P' \models_i \mathcal{A}, P'' \models_i \mathcal{B}}{P \models_i \mathcal{A}|\mathcal{B}}$$

$$\frac{\text{for all } P', P'' :\ \text{if } P \equiv P'|P'' \text{ then } P' \not\models_i \mathcal{A} \text{ or } P'' \not\models_i \mathcal{B}}{P \not\models_i \mathcal{A}|\mathcal{B}}$$

$$\frac{\text{for all } m : P \models_i \mathcal{A}\{x \leftarrow m\}}{P \models_i \forall x.\mathcal{A}} \qquad \frac{\text{for some } m : P \not\models_i \mathcal{A}\{x \leftarrow m\}}{P \not\models_i \forall x.\mathcal{A}}$$

$$\frac{\text{for some } P' : P \rightarrow^* P' \text{ and } P' \models_i \mathcal{A}}{P \models_i \Diamond\mathcal{A}} \qquad \frac{\text{for all } P',\ \text{if } P \rightarrow^* P' \text{ then } P' \not\models_i \mathcal{A}}{P \not\models_i \Diamond\mathcal{A}}$$

$$\frac{\text{for all } P'.P' \not\models_i \mathcal{A} \text{ or } P|P' \models \mathcal{B}}{P \models_i \mathcal{A} \triangleright \mathcal{B}} \qquad \frac{\text{for some } P'.P \models_i \mathcal{A} \text{ and } \mathcal{P}|\mathcal{P}' \not\models \mathcal{B}}{P \not\models_i \mathcal{A} \triangleright \mathcal{B}}$$

$$\frac{\text{for some } P' : P \downarrow^* P' \text{ and } P' \models_i \mathcal{A}}{P \models_i \text{✧}\mathcal{A}} \qquad \frac{\text{for all } P',\ \text{if } P \downarrow^* P' \text{ then } P' \not\models_i \mathcal{A}}{P \not\models_i \text{✧}\mathcal{A}}$$

$$\frac{\text{for some } P' : P \equiv (\nu n)P' \text{ and } P' \models_i \mathcal{A}}{P \models_i n\circledR\mathcal{A}} \qquad \frac{\text{for all } P'.\ \text{if } P \equiv (\nu n)P' \text{ then } P' \not\models_i \mathcal{A}}{P \not\models_i n\circledR\mathcal{A}}$$

$$\frac{(\nu n)P \models_i \mathcal{A}}{P \models_i \mathcal{A} \oslash n} \qquad \frac{(\nu n)P \not\models_i \mathcal{A}}{P \not\models_i \mathcal{A} \oslash n}$$

Fig. 5. Mutual inductive satisfaction and unsatisfaction relations.

paradigm in inductive type theory. The idea is that occurrences of `sat` in negative position can be viewed as occurrences in positive position of a new judgment which represents "unsatisfaction". Thus, the *inductive* satisfaction system (Figure 5) derives two kinds of judgments: *satisfaction* ($P \models_i \mathcal{A}$) and *unsatisfaction* ($P \not\models_i \mathcal{A}$). Notice that $\not\models_i$ is a real judgment, not a metalogical abbreviation. Moreover, all occurrences of $\models_i$ and $\not\models_i$ are in positive position.

The following equivalence result is easily proved by mutual structural induction on formulas ($\Rightarrow$) and derivations ($\Leftarrow$):

Proposition 2.8 *For all P process, $\mathcal{A}$ closed formula: $P \models_i \mathcal{A}$ iff $P \models \mathcal{A}$, and $P \not\models_i \mathcal{A}$ iff not $P \models \mathcal{A}$.*

Functional solution Actually, all the properties proved in [4] do not rely on induction principles over derivations, thus for many applications the mutual inductive satisfaction system presented above may be an overkill. On the other hand, the axiomatic encoding is quite unsatisfactory from the point of view of the proof automatization. In fact, at a careful analysis the original system (Figure 4) can be seen as a deterministic, syntactic-driven unravelling of satisfaction judgments into "metalogical" statements. Following [15], this observation can be made precise in Coq by representing satisfaction by a function recursively defined on the syntax of formulas:

```
Fixpoint satF [P:proc;A:form]: Prop:=
<Prop>Cases A of    T => True
| (neg B) => (satF P B) -> False
| (Or A1 A2) => (satF P A1) \/ (satF P A2)
| (comp_adj A1 A2) => (P':proc)(satF P' A1)->(satF (par P P') A2)
| (forall B) => ((m:name)(satF P (B m)))     ...
end.
```

This approach is proved to be formally equivalent to the axiomatic one:

`Lemma SATF_SAT: (A:form)(P:proc)(satF P A) <-> (sat P A).`

As a corollary, the adequacy of `satF` follows from Proposition 2.7. Thus we can freely switch between the two encodings as needed during proofs developments. The functional presentation of the satisfaction relations allows for a certain degree of automatization. A goal `(satF P A)` can be automatically `Simpl`ified (i.e., using `Coq`'s tactic `Simpl`) to a proposition of atomic predicates, which can be dealt with simply using the semi-automatized tactics for predicate logic provided by the environment (`Auto`, `EAuto`, `Tauto`, `Prolog`, ...).

3 Formal Reasoning on Ambients in Coq

In this section, we describe the formal development of some properties proved in [4]. Since these properties are mostly about behaviour of names and contexts, we will extensively use the axioms of the Theory of Contexts [10].

3.1 The Theory of Contexts for Ambients

The first basic axiom we need is the decidability of the equality over names:

`Axiom dec_name: (x,y:name)x=y \/ ~x=y.`

This fact, tacitly assumed in many presentations of process algebras, is required in the development of the metatheory since many proofs proceed by discriminating on names (e.g. [16, Lemma 6] or [3, Lemma 4-6]). Another common assumption is that the set of names is infinite; our *unsaturation* axiom enforces this notion by stating that for any process and formula, we can always pick a new name not occurring in them:

```
Axiom unsat: (P:proc)(F:form)
             (Ex [n:name](notin_proc n P) /\ (notin_form n F)).
```

The remaining axiom schemata of the Theory of Contexts are β-expansion and extensionality. Here, we report the statements of two instantiations for the type `proc`:

```
Axiom proc_exp: (P:proc)(n:name)
          (Ex [P':name->proc](notin_proc n (nu P')) /\ P=(P' n)).
```

```
Axiom proc_ext: (P,Q:name->proc)(x:name)(notin_proc x (nu P)) ->
                (notin_proc x (nu Q)) -> (P x)=(Q x) -> P=Q.
```

We use β-expansion and extensionality also for the types `cap` and `form` and for contexts over them and over `proc`.

Other axioms of the Theory of Contexts Originally [11], the Theory of Contexts provided also the axioms of monotonicity and decidability of occur checking for processes. An important result we proved in this case study is that those properties are indeed derivable from the remaining axioms:

```
Lemma NOTIN_PROC_MONO: (P:name->proc)(x,y:name)
                 (notin_proc x (P y))->(notin_proc x (nu P)).
Lemma ISIN_PROC_MONO: (P:name->proc)(x,y:name)~x=y ->
                      (isin_proc x (P y)) -> (isin_proc x (nu P)).
Lemma NOTIN_PROC_DEC: (P:proc)(n:name)
                 (isin_proc n P) \/ (notin_proc n P).
```

The first two lemmata are proved by induction on the number of `proc`-constructors contained in `(P y)`. Notice that a naïve structural induction carried out using the built-in induction principle on processes provided by `Coq` is not sufficient. The reason is that the inductive hypothesis of structural induction applies only to proper subterms of the involved process, while substituting a name with another one (in the cases involving binders) breaks this relationship. On the other hand, the number of process-constructors is preserved by all renamings. `NOTIN_PROC_DEC` follows then by means of a structural induction on `P` using the monotonicity of `notin_proc` and its dual `isin_proc`[2] in the cases involving restriction and input action.

It is important to notice that these lemmata can be seen as *general results* about the Theory of Contexts. Indeed, since during the proof development we did not exploit any particular feature of the Ambient Calculus (a quick look at the `Coq` code will confirm this), these properties can be derived for the encoding of any language.

3.2 Fresh renaming properties

Some interesting results of [4] are the "fresh renaming" properties (i.e., the possibility of replacing a name with a fresh one in the arguments of a judgment) and the preservation of satisfaction under structural congruence:

Lemma 3.1 *Let $P \in \Pi$, $m, m' \in \Lambda$ s.t. $m' \notin fn(P)$, and $\mathcal{R} \in \{\equiv, \rightarrow, \downarrow\}$. For all P', if $P\mathcal{R}P'$ then $m' \notin fn(Q)$ and $P\{m \leftarrow m'\}\mathcal{R}Q\{m \leftarrow m'\}$. For all Q, if $P\{m \leftarrow m'\}\mathcal{R}Q$ then there is a P' such that $P\mathcal{R}P'$, $m' \notin fn(P')$ and $Q = P'\{m \leftarrow m'\}$.*

Lemma 3.2 *If $P \models \mathcal{A}$ and $P \equiv P'$, then $P' \models \mathcal{A}$.*

Lemma 3.3 *For all closed formulas $\mathcal{A}$, processes P, and names m, m', if $m' \notin fn(P) \cup fn(\mathcal{A})$ then $P \models \mathcal{A}$ iff $P\{m \leftarrow m'\} \models \mathcal{A}\{m \leftarrow m'\}$.*

[2] The intended meaning of `(isin_proc n P)` is, informally, "`n` occurs free in `P`".

All these properties have been formalized and fully verified in Coq (lemmata STRUCT_NOTIN and STRUCT_RW for $\equiv$, RED_NOTIN and RED_RW for $\rightarrow$, NEST_NOTIN and NEST_RW for $\downarrow$, SAT_UPTO_STRUCT_EQ and SAT_RW for $\models$). It is interesting to see how the previous lemmata about fresh renamings are rendered. For instance, the case of Lemma 3.1 for $\rightarrow$ splits in two parts:

```
Lemma RED_NOTIN: (A,B:proc)(red A B) ->
                 (x:name)(notin_proc x A)->(notin_proc x B).
Lemma RED_RW   : (P,Q:name->proc)
 (m:name)(notin_proc m (nu P)) -> (notin_proc m (nu Q)) ->
                                  (red (P m) (Q m)) ->
 (m':name)(notin_proc m' (nu P)) -> (notin_proc m' (nu Q)) ->
                                  (red (P m') (Q m')).
```

Representing P with `(P m)` and P' with `(Q m)` allows us to formalize both directions of Lemma 3.1 by means of RED_RW. Indeed, `m` and `m'` can be swapped without altering the meaning of the lemma, allowing to deduce `(red (P m) (Q m))` from `(red (P m') (Q m'))`, i.e., $P \rightarrow P'$ from $P\{m \leftarrow m'\} \rightarrow Q$ where $Q = P'\{m \leftarrow m'\}$. HOAS allows to express very naturally this possibility of *swapping* names preserving the validity of a property, i.e., the *equivariance property* introduced in [19] as the "fundamental assumption of Nominal Logic". The same considerations hold for the lemmata about $\equiv$, $\downarrow$ and $\models$.

3.3 Gabbay-Pitts' "new" quantifier

In [4] the Gabbay-Pitts fresh-name quantifier (И) is used in order to define formulas of the form $\text{И}x.\mathcal{A}$ meaning that "for fresh x, $\mathcal{A}$ holds". И is defined in [4] a syntactic shorthand for a more complex formula as follows:

$$\text{И}x.\mathcal{A} \triangleq \exists x.x\#(fnv(\mathcal{A}) \setminus \{x\}) \wedge x \circledR \mathbf{T} \wedge \mathcal{A}, \qquad (1)$$

where $x\#N$ is an abbreviation for the formula $\bigwedge_{y\in N}(x \neq y)$ meaning that x is a name fresh w.r.t. a given (finite) set of names N (recall that equality between names in ambient logic can be defined as $\eta = \mu \triangleq \eta[\mathbf{T}]@\mu$). Since a process P satisfies the formula $x\circledR\mathbf{T}$ iff x does not occur free into it, the first part of the definition simply states that x is a name fresh w.r.t. both the process P and the formula $\mathcal{A}$. However, as remarked in [4], since (1) contains in the right-hand side the set of free names and variables of $\mathcal{A}$, it is a meta-theoretical abbreviation rather than a definition *within* the logic. Hence, it must be always understood in its expanded form, where $x\#(fnv(\mathcal{A}) \setminus \{x\})$ is replaced by the corresponding conjunction of disequalities.

Having adopted a higher-order encoding approach, we cannot directly encode definition (1). The reason is that we cannot write a function fnv computing the set of free names of a formula, since recursive calls cannot cross abstractions over `name` (in the case regarding the `forall` constructor). In order to overcome this problem we add a new constructor to the type `form` corresponding to И:

```
... | new: (name -> form) -> form.
```

The problem of defining the behaviour of `new` is then shifted to the level of the satisfaction relation, where we add the following axiom and functional case:

```
Axiom sat_new: (P:proc)(A:name->form)
  (Ex [m:name](notin_proc m P) /\ (notin_form m (forall A)) /\
             (sat P (A m))) <-> (sat P (new A)).
Fixpoint satF [P:proc;A:form]: Prop:= <Prop>Cases A of  ...
| (new B) => (Ex [m:name](notin_proc m P) /\
                      (notin_form m (forall B)) /\ (satF P (B m)))
end.
```

It should be noted that all the previously proved properties still hold. Moreover, we can derive formally the following results (obviuosly `NEW_EXISTS` follows directly from `sat_new`):

```
Lemma EXISTS_FORALL: (P:proc)(A:name->form)
(Ex [m:name]((notin_proc m P)/\(notin_form m (forall A))/\(sat P (A m))))
<->((m:name)((notin_proc m P)/\(notin_form m (forall A)))->(sat P (A m))).
Lemma NEW_EXISTS: (A:name->form)(P:proc)(sat P (new A)) <->
(Ex [m:name]((notin_proc m P)/\(notin_form m (forall A))/\(sat P (A m)))).
Lemma NEW_FORALL: (A:name->form)(P:proc)(sat P (new A)) <->
((m:name)((notin_proc m P)/\(notin_form m (forall A)) -> (sat P (A m)))).
```

which essentially amount to the property establishing the double nature (existential-universal) of the И quantifier (Propositions 4-2 and 4-4 in [4]):

$$\begin{aligned} P \models \text{И}x.\mathcal{A} &\iff \exists m \in \Lambda . m \notin fn(P,\mathcal{A}) \text{ and } P \models \mathcal{A}\{x \leftarrow m\} \\ &\iff \forall x \in \Lambda . m \notin fn(P,\mathcal{A}) \text{ implies } P \models \mathcal{A}\{x \leftarrow m\} \end{aligned}$$

Also in this case our encoding based on HOAS and the Theory of Contexts turns out to be quite effective in deriving results about fresh renamings. This is confirmed also in the case of the logical properties of the И quantifier. Indeed, we proved very easily the following lemmata

```
Lemma NEW_NOT_F: (A:name->form)(P:proc)
(satF P (neg (new A))) <-> (satF P (new [x:name](neg (A x)))).
Lemma NEW_COMP_F: (A,B:name->form)(P:proc)
(satF P (new [x:name](comp (A x) (B x))))
                           <-> (satF P (comp (new A) (new B))).
```

which are the formal counterparts of the following properties:

$$P \models \neg\text{И}x.\mathcal{A} \text{ iff } P \models \text{И}x.\neg\mathcal{A} \qquad P \models \text{И}x.(\mathcal{A}|\mathcal{B}) \text{ iff } P \models (\text{И}x.\mathcal{A})|(\text{И}x.\mathcal{B})$$

These correspond to two corollaries of Proposition 4-3 in [4] expressed in terms of the satisfaction relation. In particular the second one is described in [4] as "of particular interest (and difficulty)"; however, with our encoding the formal proof is quite simple (a few lines of tactics).

The general "renaming property" pattern All the renaming lemmata illustrated in this subsection have very similar statements—the only differences are in the particular relation which we want to be preserved by renaming and eventually in the syntactic categories involved—and are formally proved in Coq by means of the same proof technique. Indeed, they are all instances of the following pattern:

$$\frac{\text{for some } x \notin \bigcup_{i=1}^{n} fn(C_i[\cdot]) : \mathcal{R}(C_1[x], \ldots, C_n[x])}{\text{for all } y \notin \bigcup_{i=1}^{n} fn(C_i[\cdot]) : \mathcal{R}(C_1[y], \ldots, C_n[y])} \tag{2}$$

where $\mathcal{R}$ is a given n-ary relation (e.g., structural congruence, capture-avoiding substitution, reduction relation etc.) and $C_1[\cdot] \ldots, C_n[\cdot]$ are variables ranging over contexts of given syntactic categories. Usually, this kind of properties are proved "with pencil and paper" by carrying out a structural induction either on the derivation of the premise $\mathcal{R}(C_1[x], \ldots, C_n[x])$ or on one of the arguments $C_i[x]$ $(1 \leq i \leq n)$ or a "measure" of an argument (e.g., the number of symbols it contains). However, Coq tactics (in particular those handling induction principles) deal not adequately with higher-order unification; hence, we are forced to prove a preliminary version of the renaming lemma where we introduce by hand the necessary unifications in order to recover sufficient information on the structure of the contexts $C_i[\cdot]$ from their instantiations $C_i[x]$. In other words, we "lift" structural information to the level of functional terms; in order to achieve this goal, the axioms of β-expansion and extensionality turn out to be indispensable (in fact, this is the original motivation of their introduction in [11]). This lifting follows a general pattern. First, we replace the original goal with the following:

$$\frac{\text{for some } x \notin \bigcup_{i=1}^{n} fn(C_i[\cdot]), T_1 = C_1[x], \ldots, T_n = C_n[x] : \mathcal{R}(T_1, \ldots, T_n)}{\text{for all } y \notin \bigcup_{i=1}^{n} fn(C_i[\cdot]) : \mathcal{R}(C_1[y], \ldots, C_n[y])} \tag{3}$$

where $T_1, \ldots, T_n$ are plain terms and $T_1 = C_1[x], \ldots, T_n = C_n[x]$ are the necessary unifications. Clearly we can infer (2) from (3) by taking $T_i = C_i[x]$.

During the proof, the inductive hypothesis gives us some structural information on $T_1, \ldots, T_n$. Then, using the β-expansion axiom, we can expand the latters into contexts applied to x yielding the equations $T_1 = T'_1[x], \ldots, T_n = T'_n[x]$ where $x \notin \bigcup_{i=1}^{n} fn(T'_i[\cdot])$. By transitivity we obtain the equations $C_i[x] = T'_i[x]$; thus, by extensionality, we get $C_i[\cdot] = T'_i[\cdot]$, i.e., the structural information we needed on the variable $C_i[\cdot]$. Such an information can then be transferred to the instantiations over y in the current goal in order to apply the suitable constructor of $\mathcal{R}$ and solve the subsequent subgoal by means of the inductive hypothesis.

This proof pattern should be automatizable by means of some *ad hoc* tactic.

3.4 The Hidden-Name Quantifier

For the sake of completeness, in this section we also give the encoding of the *hidden-name quantifier* ν. This allows one to define a formula $(\nu x)\mathcal{A}$, whose informal meaning is "for a name x (hidden in the underlying process), $\mathcal{A}$ holds". Thus, one can talk about restricted names in processes.

The formal definition in the modal logic is given by $(\nu x)\mathcal{A} \triangleq И x.x®\mathcal{A}$ [4]. Hence, the corresponding encoding is a straightforward Coq definition:

```
Definition nu_f: (name -> form) -> form :=
                 [A:name->form](new [x:name](rev x (A x))).
```

As the satisfaction relation is concerned, we have proved the formal equivalent of [4, Lemma 5-2]:

$$P \models (\nu x)\mathcal{A} \text{ iff } \exists m \in \Lambda . m \notin fn(P, \mathcal{A}) \wedge \exists P' \in \Pi . P \equiv (\nu m)P' \wedge P' \models \mathcal{A}\{x \leftarrow m\}$$

```
Lemma NU_F_SATF: (P:proc)(A:name->form)(satF P (nu_f A)) <->
(Ex [m:name]((notin_proc m P) /\ (notin_form m (forall A)) /\
(Ex [P':name->proc](struct_eq P (nu P')) /\ (satF (P' m) (A m))))).
```

The proof is trivial, since it relies on the definition of `nu_f` in terms of `new` and `rev`. On the other hand, Lemma 3.3 is needed in order to prove [4, Proposition 5-3]:

for all $n \in \Lambda$, $x \in \vartheta$, $P \in \Pi$ and closed $\mathcal{A} \in \Phi$,
$n \notin fn(P) \wedge P \models (\nu x)\mathcal{A}\{n \leftarrow x\} \Leftrightarrow \exists P' \in \Pi . P \equiv (\nu n)P' \wedge P' \models \mathcal{A}$

Observing that $n \notin fn(P)$ follows from the existence of a P' such that $P \equiv (\nu n)P'$, we can rearrange the previous statement as

for all $n \in \Lambda$, $x \in \vartheta$, $P \in \Pi$ and closed $\mathcal{A} \in \Phi$, if $n \notin fn(P)$ then
$P \models (\nu x)\mathcal{A}\{n \leftarrow x\} \Leftrightarrow \exists P' \in \Pi . P \equiv (\nu n)P' \wedge P' \models \mathcal{A}$.

Thus the formal equivalent, which we have formally proved, is the following:

```
Lemma NU_F_SATF_PROPER: (n:name)(P:proc)(A:name->form)
(notin_proc n P) -> (notin_form n (forall A)) -> (satF P (nu_f A))
<->(Ex [P':name->proc](struct_eq P (nu P'))/\(satF (P' n) (A n))).
```

4 Conclusions

In this paper we have presented a HOAS-based encoding of the Ambient Calculus and its Modal Logic [4] in the Calculus of Inductive Constructions. Many fundamental lemmata, mostly "fresh renamings" properties, have been formally proved in `Coq` using the Theory of Contexts [10].

In our opinion the present case study confirms a pleasant feature of HOAS-encodings, namely, the fact that object languages with binders can be encoded in a "purified" form, freed from inessential notions. As a consequence also the metatheoretic properties over names can be rendered in simpler, although equivalent, forms than the corresponding ones "on the paper".

This work has a bearing both on the object system and on the Logical Framework. A new system in natural deduction style for satisfaction has been introduced; its features (especially from a proof-theoretic point of view) deserve further investigations. On the other hand, we have described a general pattern for "fresh renaming" properties, which should be applicable to any nominal calculi. Also, some properties originally taken as axioms in the Theory of Contexts have been proved derivable from the others, thus reducing the minimum set of axioms one has to take.

Finally, the $\mathsf{И}$ quantifier has been faithfully rendered in `Coq`, and some of its properties formally proved, thus giving some insights about the connections between FM-based theories and the Theory of Contexts.

Related work. To our knowledge, this is the first formalization in a Logical Framework of the Ambient Calculus and of the satisfaction relation for the related modal logic. The closest process algebra which has been studied to some extent is the π-calculus; see [11,20,5,9] for some formalizations in Coq or Isabelle/HOL using HOAS, First Order Abstract Syntax or de Bruijn indexes. In particular, in [20] the monotonicity axiom is proved easily since the freshness predicate is not defined inductively, but as the negation of a free occurrence predicate (the analogous of our `isin_proc`). Instead, in Section 3 we derived monotonicity for `notin_proc` which is inductively defined. Moreover, using the proved properties of monotonicity of `isin_proc` and `notin_proc`, we have proved that occur checking is decidable (`NOTIN_PROC_DEC`). Therefore, the only classical feature we need for the Theory of Contexts is the decidability of the equality over names.

Future work. The present encoding can be seen as the basis for further developments. One interesting direction is the encoding of some typing systems among the many proposed in the literature (see, e.g., [1]), in order to get rid of "bad terms". Type systems for Ambients are a whole interesting topic on their own, raising deep questions such as subject reduction properties, *safeness* properties and so on. However, type systems like those presented in [1], are usually given in a sequent-style, while proof systems are best encoded in type-theory based metalanguages when they are in Natural Deduction style. Thus the first step would be the development of some equivalent formulation of existing type systems in Natural Deduction style, if possible.

A The Calculus of Inductive Constructions

The *Calculus of Inductive Constructions (CIC)* is an extension of the *Calculus of Constructions (CC)*, which can be defined as the PTS λC of Barendregt's λ-cube, with two sorts, *Prop* and *Set*. Under the *proposition-as-types, proofs-as-terms* paradigm, there is an isomorphism between propositions of intuitionistic higher-order logic and types of sort *Prop*. If A has type *Prop* then it represents a logical proposition; the fact that A is inhabited by a term M represents the fact that A holds. Each term M inhabiting A represents a *proof* of A. On the other hand, the sort *Set* is supposed to be the type of datatypes, such as naturals, lists, trees, booleans, etc. These types differ from those inhabiting *Prop* for their constructive contents.

Therefore, CC, as many similar Type Theories, can be fruitfully used as a general logic specification language, i.e. as a Logical Framework (LF) [8,17,18]. In an LF, following the "judgment-as-types" paradigm, 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.).

The Calculus of Inductive Constructions (implemented in the Coq system [12]) extends CC with some special constants which represent the definition, introduction and elimination of inductive types. For instance, the following definition of natural numbers (written in Gallina, Coq's specification language)

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

allows to define terms by "case analysis", like the following function:

```
Definition pred := [n:nat]Cases n of  O => O  |  (S u) => u  end.
```

where `[n:nat]` is Gallina notation for abstraction $\lambda n : nat$. Using these elimination schemata, Coq automatically states and proves the induction principle for each inductively defined type. For instance, the above definition yields the Peano induction principle "for free":

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

where `(n:nat)` is the notation for dependent product $\prod_{n:nat}$. This feature has been extensively used in the definition of logical connectives: we need only to specify the introduction rules, and we can prove the elimination rules from the elimination principle the system automatically provides us.

However, allowing for *any* inductive definition in CIC would yield non-normalizing terms, thus invalidating the standard proof of consistency of the system. Hence, inductive definitions are subject to the *positivity condition*, which (roughly) requires that the type we are defining does not occur in negative position in the type of any argument of any constructor. This condition ensures the soundness of the system, but it rules out also many sound inductive definitions. For instance, the following definition of λ-terms in *(full) higher-order abstract syntax*

```
Inductive L : Set := lam : (L->L) -> L | app : L -> L -> L.
```

is not well-formed, due to the negative occurrence of `L` in the type `L->L` of the argument of `lam`.

Another problem arising from the use of higher order abstract syntax together with inductive types is that of *exotic terms*. These are λ-terms which do not correspond to any object "on the paper", despite their types correspond to some syntactic category. Exotic terms are generated when a type has a higher-order constructor over an inductive type. A simple example is the following fragment of first-order logic:

```
Inductive i : Set := zero : i | one : i.
Inductive o : Set := ff : o | eq : i->i->o | forall : (i->o)->o.
Definition weird : o := (forall [x:i](Cases x of
                                        zero => ff
                                      | one  => (eq zero zero)
                                     end)).
```

The term `weird` does not correspond to any proposition of first order logic: there is no formula $\forall x \phi$ such that $\phi\{0/x\}$ and $\phi\{1/x\}$ are syntactically equal to "*ff*" and "$0 = 0$", respectively. Exotic terms are problematic in establishing the faithfulness of the formalization; usually, they have to be ruled out by means of auxiliary "validity" judgments [6, 20]. Another approach, which we have used in this paper, is to have the higher order constructors to range over types which are not inductive, so that there is no `Cases` to use as above.

A common implementation of CIC is `Coq`, an interactive proof assistant developed by the INRIA and other institutes. For a complete description, we refer to [12]. `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 judgments. 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.

References

[1] Cardelli, L., G. Ghelli and A. D. Gordon, *Types for the ambient calculus*, Information and Computation (2002), to appear. Special issue on TCS'2000.

[2] Cardelli, L. and A. D. Gordon, *Mobile ambients*, in: *Proc. FOSSACS '98*, Lecture Notes in Computer Science **1378** (1998), pp. 140–155.

[3] Cardelli, L. and A. D. Gordon, *Anytime, anywhere. Modal logics for mobile ambients*, in: *Proc. 27th ACM POPL*, 2000, pp. 365–377.

[4] Cardelli, L. and A. D. Gordon, *Logical properties of name restriction*, in: S. Abramsky, editor, *Proc. TLCA 2001*, LNCS **2044** (2001), p. 46.

[5] Despeyroux, J., *A higher-order specification of the pi-calculus*, in: *IFIP TCS*, 2000, pp. 425–439.

[6] Despeyroux, J., A. Felty and A. Hirschowitz, *Higher-order syntax in Coq*, in: *Proceedings of TLCA'95*, Lecture Notes in Computer Science **905** (1995).

[7] Gabbay, M. J. and A. M. Pitts, *A new approach to abstract syntax with variable binding*, Formal Aspects of Computing **?** (2001), pp. ?–?. To appear.

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

[9] Hirschkoff, D., *Bisimulation proofs for the π-calculus in the Calculus of Constructions*, in: *Proc. TPHOL'97*, LNCS **1275** (1997).

[10] Honsell, F., M. Miculan and I. Scagnetto, *An axiomatic approach to metareasoning on systems in higher-order abstract syntax*, in: *Proceedings of ICALP'01*, Lecture Notes in Computer Science **2076** (2001), pp. 963–978.

[11] Honsell, F., M. Miculan and I. Scagnetto, *π-calculus in (co)inductive type theory*, Theoretical computer science **239–285** (2001), pp. 239–285.

[12] INRIA, "The Coq Proof Assistant Reference Manual", `http://coq.inria.fr/`.

[13] Miculan, M., "Encoding logical theories of programs," Ph.D. thesis, Dipartimento di Informatica, Università di Pisa, Pisa, Italy (1997).

[14] Miculan, M., *Developing (meta)theory of lambda-calculus in the theory of contexts*, in: S. Ambler, R. Crole and A. Momigliano, editors, *Proc. MERLIN 2001*, Electronic Notes in Theoretical Computer Science **58.1** (2001), pp. 1–22.

[15] Miculan, M., *On the formalization of the modal μ-calculus in the Calculus of Inductive Constructions*, Information and Computation **164** (2001), pp. 199–231.

[16] Milner, R., J. Parrow and D. Walker, *A calculus of mobile processes*, Tech. Rep. ECS-LFCS-89-85, Dept. of Computer Science, University of Edinburgh (1989).

[17] Paulin-Mohring, C., *Inductive definitions in the system Coq; rules and properties*, in: M. Bezem and J. F. Groote, editors, *Proc. of Conference on Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science **664** (1993), pp. 328–345.

[18] Pfenning, F., *The practice of Logical Frameworks*, in: *Proc. CAAP'96*, Lecture Notes in Computer Science **1059** (1996), pp. 119–134.

[19] Pitts, A. M., *Nominal Logic: A first order theory of names and binding*, invited talk in: *Proc. TACS 2001*, Lecture Notes in Computer Science **2215** (2001), pp. 219–242.

[20] Röckl, C., D. Hirschkoff and S. Berghofer, *Higher-order abstract syntax with induction in Isabelle/HOL: Formalising the π-calculus and mechanizing the Theory of Contexts*, in: F. Honsell and M. Miculan, editors, *Proceedings of FoSSaCS 2001*, Lecture Notes in Computer Science **2030** (2001), pp. 359–373.

[21] Scagnetto, I., *Coq code for ambient and modal logic*, available at `http://www.dimi.uniud.it/~scagnett/Coq-Sources/ambient_modal_logic.v.gz`.

[22] Scagnetto, I., "Reasoning about Names in Higher-Order Abstract Syntax," Ph.D. thesis Cs 2002/4, Dipartimento di Matematica e Informatica, Università di Udine, Udine, Italy (2002).