Università degli Studi di Udine
Dipartimento di Matematica e Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis

# Reasoning about names in Higher-Order Abstract Syntax

Candidate:
Ivan Scagnetto

Supervisor:
Furio Honsell

October 31, 2001

# Abstract

The continuously growing application of computer science to a wide range of human activities yields several issues. Besides sociological, philosophical or other more or less arbitrary speculations, there is the pragmatic need of controlling the activity and predicting the behaviour of complex systems devoted to critical tasks or, more simply, to the accomplishment of online financial transactions. Most of the time, programmers convince themselves that their code meets the original specifications by informal arguments or by testing the software over some sample of input data. However, even at the dawn of the computer era, this approach has been considered unsatisfactory since it cannot ensure the absence of errors under all circumstances.

On the other hand the formal methods of mathematics provide a rigorous way for reasoning and understanding the behaviour of programs, languages, and complex systems. Many logics and calculi have arisen in order to deal with a plethora of problems and properties. However, their application to real cases is often cumbersome and error prone due to the overwhelming complexity and the subtleties involved.

The field of Computer Aided Formal Reasoning (CAFR) is a research branch whose aim is to study and implement tools allowing one to develop formal proofs in a computer-assisted way. So doing, nothing can be "swept under the rug" (as it often happens with proofs carried out with "pencil and paper") and, once the proof is finished, its correctness is ensured, i.e., certified by the system.

Recently Logical Frameworks (LFs) based on constructive type theory have emerged as general metalanguages for encoding and formally reasoning about formal systems by means of the *propositions-as-types*, *proofs-as-terms* paradigm. The latter allows one to reduce the problem of *proof checking* to that of *type checking* opening the way to a mechanized implementation of LFs.

The contribution of this thesis is the proposal of an axiomatic theory which, in conjunction with the Higher-Order Abstract Syntax (HOAS) encoding approach, allows one to adequately encode and reason over a large class of formal systems. We hope that this work will help in the understanding and application of LFs.

**Keywords:** Formal Methods, Computer Aided Formal Reasoning; Logical Frameworks, Type Theory, Coq, Higher-Order Abstract Syntax; Functor Categories.

# Contents

# 1
# Introduction

The increasing complexity of modern software and hardware systems generates the non trivial problem of the certification of their behaviour; indeed formally ensuring that a given system is doing exactly what it is designed for imposes an overwhelming burden of details making it likely that errors will arise. A large number of states, for example, often implies that checking even a simple transition may be not only error prone, but far beyond the human skills.

So there is the need for automated tools helping people to carry out the difficult task of formally reasoning about computerized systems. The aim of Computer Aided Formal Reasoning (CAFR) research is exactly the study and implementation of such tools. In computer science there are many logics which are useful for reasoning about software and hardware systems (e.g. propositional, first-order, higher-order, modal, temporal and linear logics, set-theory, etc.). In general there are two main possible approaches when facing the problem of implementing an automated tool for one of those formal systems. The first consists of building it from scratch, but this immediately appears to be a daunting task since there are many mechanisms to be implemented and, when a different formal system is considered, all the work must be repeated. Just to give an idea of the difficulties imposed by this strategy, it is sufficient to consider the syntax level, where support must be provided for representing terms, formulæ, derivation rules, binding operators, and substitution. Moreover, at the level of formal proofs there are more complicated techniques that must be implemented such as proof construction and proof checking tools, instantiation of rule schemes and all the machinery checking the applicability of rules provided with context-sensitive side conditions. Obviously the result will be perfectly tailored for the special case (more efficient, more reminiscent of the original syntax of the implemented system etc.), but even a minor change or extension of the original system can lead to the problem of redoing the whole effort from start, i.e., the approach lacks flexibility. An alternative way is the development of a general framework capturing the main common features of a large class of logics in such a way that a great amount of work is done once and for all.

In particular a class of *Logical Frameworks* based on the notion known as *Curry-Howard isomorphism*[1] [dB70, How80] has revealed itself to be a suitable basis for the synthesis of interactive environments where logic-independent reasoning is possible: without entering into the details, these frameworks are built upon the idea that types can be interpreted not only

[1] Also known as *propositions-as-types* principle.

as a specification of partial correctness properties about a program, but also as *propositions*. Following this intuition then it seems natural to interpret terms as *proofs* of the proposition associated with their type. Proceeding further, we have that a proposition is true if the corresponding type is inhabited[2]; thus *proof checking* reduces to *type checking*[3] and if the latter is decidable (this is true in all the Logical Frameworks based on Type Theory) the whole process can be mechanized and implemented on a machine. Moreover, since the rules of the type systems of the underlying $\lambda$-calculi are usually given in Natural Deduction style, it follows that the implemented system gives rise to a Natural Deduction Proof System helping the user in the task of finding the proof term by means of a top-down process[4]. The latter begins with the main goal (i.e. the proposition to be proved) and proceeds by transforming it into (simpler) subgoals through the use of *tactics*, i.e., functional programs and ending when all the current subgoals are instances of axioms. Often tactics are completely automated and require no interaction with the user, in other cases the latter needs to give some "hints" to the machine in order to solve the goal. However, in practice this is better than carrying out the proof by hand on the paper; the user has to pay attention only when encoding a formal system into the framework[5]; all the subsequent work is then guaranteed to be error free.

It follows from this rough introduction that Type Theory based Logical Frameworks can be thought of as general purpose programming languages allowing the synthesis of *proof assistants* from a *signature* (provided by the user) containing the encoding of a formal system. Many useful mechanisms are automatically made available by the underlying metalanguage[6] of the Logical Framework: unification, pattern matching, recursive functions definition, natural deduction style reasoning etc. The philosophy which inspired the Edinburgh Logical Framework [HHP93] goes a little further; in fact the encoding methodology suggested to the user allows one to delegate to the metalanguage also the common notions of $\alpha$-conversion and capture-avoiding substitution. To illustrate this point, let us consider the case of encoding the syntax of untyped $\lambda$-calculus:

$$M ::= x \mid M_1 M_2 \mid \lambda x.M,$$

where $x \in \mathcal{V}$ ($\mathcal{V}$ is an infinite set of variables). In general, to encode a logical system in a Type Theory based logical framework, the user must assign types to a set of constants representing the syntax constructors and the judgments with their derivation rules. In the abovementioned case (untyped $\lambda$-calculus) a naïve encoding would take an inductive set (like $\mathbb{N}$) as the set representing variables and map the binding operator $\lambda$ to a term constructor $lam$ of type $var \to tm \to tm$ (if $tm$ is the type chosen to represent $\lambda$-calculus terms and $var$ the type representing variables). This is intuitive since the term $lam$-constructor takes as arguments one variable ($x$) and one term ($m$) in which the variable $x$ will be bound exactly as the original $\lambda$-operator does. However, there are several drawbacks; indeed so doing one must then provide an additional encoding of the notions of free and bound variable and of the mechanisms of $\alpha$-conversion and capture-avoiding substitution.

[2] The absurdity then corresponds to the empty type, i.e., the type with no inhabitants.

[3] Formally, a logical framework is a system allowing the derivation of judgments of the form $\Gamma \vdash_\Sigma t : T$, where $\Gamma \equiv x_1 : T_1, \ldots, x_n : T_n$ is an assignment of types to free variables and $\Sigma$, the signature, is a set of typed constants.

[4] The Natural Deduction style was originally conceived by Gentzen [Gen69] and, according to its inventor, it "reflects as accurately as possible the actual logical reasoning involved in mathematical proofs".

[5] This "attention" amounts to the proof of an adequacy theorem stating that the encoding function establishes a bijective correspondence between the objects of the formal system and the so called "canonical" terms of the framework.

[6] Usually a typed $\lambda$-calculus with dependent types, i.e., types depending on other types or terms (e.g. the type of vectors of length $n$).

The described encoding approach, known as *first-order*, clearly complicates the task of representing a formal system in Type Theory based Logical Frameworks. In fact with a more complicated system, like a process algebra, carrying out formal proofs with a first-order encoding requires an overwhelming preliminary effort to prove often trivial (but very long) lemmata about basic syntactical properties (see e.g. [Hir97] where 600 out of 800 proved lemmata involve the handling of de Bruijn indexes).

In contrast the Higher Order Abstract Syntax (HOAS) [Chu40] encoding approach allows one to completely delegate all the syntactical details to the metalanguage making the encoding elegant and clean. Indeed the main idea is to encode binding operators (like $\lambda$) with constants whose domain is of functional type; so doing the variables of the object language are identified with the metavariables of the logical framework. For example, in the case of the untyped $\lambda$-calculus the signature encoding its syntax would be the following:

$$\begin{array}{rcl} tm & ::= & app : tm \to tm \to tm \\ & \mid & lam : (tm \to tm) \to tm \end{array}$$

Notice the type of the higher-order $lam$ constructor taking a meta-level function as argument. So the term $lam([x : tm]x)$[7] encodes the $\lambda$-term $\lambda x.x$ and the notion of $\beta$-reduction can be expressed in a very natural and elegant way saying that $lam(f, t)$ reduces to $f(t)$ without the need to specify what are free and bound variables and capture-avoiding substitution.

To summarize, we can say that, in order to fruitfully represent the language of formal systems in a form closely related to its intended semantics, it is useful to go beyond the limits imposed by language descriptions in BNF-style. Indeed the latter approach is too much "parsing oriented": there is a great amount of information which is essentially useless for language processing. First-order abstract syntax is one step towards a representation of formal systems languages where the structure of a phrase reflects its semantic commitments. However, this approach is not completely satisfactory since it cannot automatically account for variable binding related mechanisms (e.g. notions of free and bound variables, $\alpha$-conversion, capture-avoiding substitution, schemes and their instantiation). Higher-order abstract syntax encodings instead, allowing one to represent binding operators by means of constructors of higher-order type, conveniently delegate such machinery to the underlying metalanguage. Hence the object language can be encoded in an elegant way, being freed from the many inessential side conditions necessary to avoid name clashes and other issues involving binding structures.

However, it is well known that the advantages of the HOAS-encoding approach have a price to pay for. The first drawback is that, being equated to metalanguage variables, object level variables cannot be defined inductively without introducing exotic terms [DFH95, Mic97].

A similar difficulty arises with contexts, which are rendered as functional terms. Reasoning by induction and definition by recursion on object level terms is therefore problematic. Ironically, the last drawback is that one looses the possibility of reasoning about the properties which are delegated to the metalanguage, e.g., substitution and $\alpha$-equivalence.

In the literature there are several approaches aiming at reconciling HOAS with these issues; they are based on different techniques such as modal types, functor categories, permutation models of ZF, etc. [DPS96, FPT99, Hof99, GP99, Gab00, Pit01a, MM01]. The purpose of this thesis is to investigate another approach in this direction, namely, the so-called Theory of Contexts originally conceived in [HMS01b] for metareasoning about a HOAS-encoding of

[7] Square brackets are used to indicate $\lambda$-abstraction in the logical framework in order to distinguish it from $\lambda$-abstraction in the untyped $\lambda$-calculus.

the $\pi$-calculus [MPW92]. Following [HMS01a], we will present the theory in broad generality as a suitable framework for representing and reasoning about *nominal calculi.* The latter are a class of formal systems based upon the central notion of name/variable and featuring binding mechanisms on the latter, in order to simulate the creation and handling of protected (private) resources or simply to represent placeholders for something that will eventually be instantiated at a later time.

Our approach is axiomatic: we add on top of a pre-existing dependent typed $\lambda$-calculus a set of natural properties allowing for a smooth treatment of syntactic contexts, giving access to some of the mechanisms about the handling of names delegated to the metalevel.

Obviously, as for any axiomatic theory, one upmost concern is related to its consistency. As pointed out by Hofmann [Hof99], in order to prove it we have to resort to a quite complicated construction related to the categorical notion of tripos [HJP80, Pit81]. However, in this thesis we prefer to carry out the construction at an elementary level, in order to give the possibility of understanding it even to readers without a deep knowledge of category theory and in particular to the users of logical frameworks.

As to the completeness of the Theory of Contexts, it is an open problem. So far, we do not know exactly what is its expressive power. In order to grasp some hints, we have developed several complex case studies which have revealed it to be very fruitful. Indeed, the analysis of the proof techniques developed during the abovementioned experiences yielded a first result towards a better comprehension of the expressiveness of the Theory of Contexts (Chapter 4 § 4.5.2). More precisely, we derived higher-order induction principles from first-order ones and the axioms of the Theory of Contexts.

Since the results presented in this thesis have been developed in collaboration with other researchers, we will briefly recall our main original contributions:

1. the results about the independence and the expressiveness of the axioms of the Theory of Contexts (see Section 4.5 and Appendix A);

2. all the results appearing in Chapter 5 whose proofs appear in Appendix C (except Theorem 5.9); the proofs of the Theorems 5.2, 5.6, 5.7, 5.8 (i.e. the consistency of the categorical model validating the Theory of Contexts) and of the results in Section 5.7.1 (validity of first-order induction);

3. the case studies appearing in Chapter 6.

**Structure of the thesis.** This thesis consists of seven chapters (including this introduction) and three appendices. We start by reviewing some basic notions about type theory based logical frameworks in Chapter 2. In particular we focus on $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$(an extension of CC [CH88, Hue92, Hue94, Tay88] with primitive support for (co)inductive types [CP90, PM93, Gim94, Wer94]) and its implementation `Coq` [TCDT01]. Subsequently, in Chapter 3, we make a brief survey of the main encoding methodologies in general and of HOAS-based approaches in particular. Moreover, we will highlight the fundamental issues arising from the attempt of using HOAS-encodings in inductive settings.

In Chapter 4 we introduce a generalized form of the Theory of Contexts in $\Upsilon$, a simple type theory *à la Church* [Chu40], featuring (higher-order) induction/recursion principles. The incompatibility of the Theory of Contexts with the *Axiom of Unique Choice* (AC!) is discussed in in Section 4.4. In Section 4.5 we give the results obtained from our case studies on the independence (Section 4.5.1) and expressiveness (Section 4.5.2) of the axioms of the Theory of Contexts. Comparisons with the related work on the synthesis of frameworks and tools for metareasoning about calculi with binders are carried out in Section 4.6.

Chapter 5 is devoted to the construction of the functorial model $\mathcal{U}$ for validating the properties of the Theory of Contexts. In Section 5.9 we compare the idea of Hofmann [Hof99] (which is the basis of our work for the construction of the model $\mathcal{U}$) with [FPT99] and [GP99] under a categorical viewpoint, trying to highlight the formal links between these papers. All the material contained in this chapter is taken from [BHH$^+$01].

Chapter 6 describes two complex case studies on the applicability of the Theory of Contexts, carried out in `Coq`. The first is about the development of the metatheory of $\alpha$-equivalence for the untyped $\lambda$-calculus. In particular we formally prove that three alternative formulations of the notion of $\alpha$-equivalence are indeed equivalent. The second case study is a large work on the HOAS-encoding of the Ambient Calculus and of the satisfaction relation of the related Modal Logic introduced in [CG01]. Once again, the Theory of Contexts plays a fundamental rôle in deriving a set of fresh renaming properties lying at the heart of the metatheory of the Ambient Calculus.

Final remarks and future work appear in Chapter 7.

Appendix A contains the full `Coq` code about the derivability of a higher-order induction principle from usual first-order induction principles and the axioms of the Theory of Contexts in the specific case of a HOAS-encoding of untyped $\lambda$-calculus.

Some basic definitions of category theory appear in Appendix B, while in Appendix C we gathered longer (and tedious) proofs of some results used in the construction of the functorial model in Chapter 5.

# 2

# Type Theory-based Logical Frameworks

In [Chu33] Church proposed a general theory of functions and logic as a foundation for mathematics. Even if the whole system was proved to be inconsistent in [KR35], the functional part, universally known under the name of $\lambda$-calculus, became *the* model of functional computation [Chu41]. In the $\lambda$-calculus there are no types, i.e., every expression can be applied to every argument (even to itself); whence, it is sometimes called *untyped* $\lambda$-calculus. Indeed, in [Cur34] and [Chu40] two typed versions of the $\lambda$-calculus are introduced. In the former document terms are essentially those of untyped $\lambda$-calculus; then to every term it is possible to associate a set of possible types (including the empty set), following predefined rules of type assignment. Hence, systems following such a paradigm are also called *systems of type assignment* or typed $\lambda$-calculi *à la Curry*. On the other hand, following the paradigm proposed in [Chu40], we obtain the systems known as typed $\lambda$-calculi *à la Church* where terms are annotated with their corresponding type, i.e., they carry type information with them; moreover, every term has usually a unique type associated with it (while in systems of type assignment a term determines a set of possible types).

Type theory-based logical frameworks arise from the *Curry-Howard isomorphism*[1] (independently introduced in [dB70] and [How80]) which allows one to think of types not only as partial correctness specifications of programs (terms), but also as *propositions*. Whence, a given term can be interpreted as a proof of the proposition associated with its type; it follows that a proposition is true (i.e, there is a proof of it) if the corresponding type is inhabited. Moreover, since logical systems can be viewed as calculi for building proofs of a given set of basic judgments, for those type theories featuring dependent types, it is possible to use the *judgments-as-types* principle [ML85, HHP93], which can be regarded as the metatheoretic analogue of the Curry-Howard isomorphism, in order to fruitfully use type theories as *general logic programming languages*, i.e., a logical framework (LF). This consists of representing basic judgments with suitable types of the LF; whence proofs are represented by terms whose type represents in turn the judgment that they prove. Moreover, dependent types allow one to uniformly extend basic judgment forms to two higher-order forms introduced by Martin-Löf, namely, the *hypothetical* (representing consequence) and the *schematic* (representing

[1] Also known as *propositions-as-types* principle.

generality). Hence, all the relevant parts of an inference system can be faithfully represented in a logical framework: syntactic categories, terms, judgments, axiom and rule schemata etc.

In this chapter we will introduce the basic notions underlying type theory-based logical frameworks in order to make easier the understanding of the material contained in the rest of this document.

## 2.1 Pure Type Systems

Typed $\lambda$-calculi *à la Church* can be generally described as *Pure Type Systems* (PTSs). Such a formalism emerged from the independent work of Berardi and Terlouw [2]. The basic language of a PTS is that of *pseudo-terms*:

**Definition 2.1 (Pseudo-terms)** *Let $V$ be an infinite set of variable symbols, ranged over by x,y,z, and $C$ an infinite set of constant symbols, ranged over by c. The set of pseudo-terms $\mathcal{T}$, ranged over by $M$, $N$, $A$, $B$, $C$, is specified by the following grammar:*

$$\mathcal{T} \qquad M ::= x \mid c \mid MN \mid \lambda x{:}A.M \mid \Pi x{:}A.B,$$

*where the variable $x$ is bound in $\lambda x{:}A.M$ and $\Pi x{:}A.B$.*

Given any pseudo-term $M$, its set of free variables (denoted by $FV(M)$) is defined as usual, keeping in mind that the only binders are the abstraction operator ($\lambda$) and the dependent type constructor ($\Pi$).

**Definition 2.2 (Pseudo-environments)** *Let $\mathcal{T}$ be a set of pseudo-terms,*

- *a* statement *is of the form $M : A$ (where $M$, $A \in \mathcal{T}$); $M$ is called the* subject *and $A$ the* predicate *of $M : A$;*
- *a* declaration *is of the form $x{:}A$ (where $x \in V$ and $A \in \mathcal{T}$);*
- *a* pseudo-environment[3] *$\Gamma$ is a finite list of declarations where all the subjects are distinct; pseudo-environments and their domains (sets of subjects occurring in $\Gamma$) are inductively defined by the following rules:*
  - *the empty environment $\langle\rangle$ is a pseudo-environment ($dom(\langle\rangle) = \emptyset$);*
  - *if $\Gamma$ is a pseudo-environment, $x$ is a variable such that $x \notin dom(\Gamma)$ and $A$ is a pseudo-term, then $\langle\Gamma, x{:}A\rangle$ is a pseudo-environment ($dom(\langle\Gamma, x{:}A\rangle) = dom(\Gamma) \cup \{x\}$).*

*In the following we will write "$x_1{:}A_1, \ldots, x_n{:}A_n$" instead of "$\langle \ldots \langle\langle\rangle, x_1{:}A_1\rangle, \ldots, x_n{:}A_n\rangle$". Moreover, given two pseudo-environments $\Gamma$ and $\Delta$, we will denote by $\Gamma \subseteq \Delta$ the fact that each statement $x{:}A$ of $\Gamma$ is also a statement of $\Delta$. Finally, $\Gamma, \Delta$ stands for the pseudo-environment obtained appending the list of statements of $\Delta$ to that of $\Gamma$.*

**Definition 2.3** *Let $\mathcal{T}$ be a set of pseudo-terms, the* specification *of a PTS is a triple $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}\rangle$, where:*

[2] Both were approaching the problem of finding a method to generate all the systems of the $\lambda$-cube [Bar92].

[3] In this document we prefer to use the expression *environment* instead of the more traditional *context* in order to avoid confusion, because the latter will be used in subsequent chapters to denote *syntactic contexts*, i.e., terms with holes.

1. General axioms and rules:

$$\begin{array}{ll}
\text{(Axiom)} & \langle\rangle \vdash c{:}s \quad (c : s) \in \mathcal{A} \\
\text{(Start rule)} & \dfrac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x{:}A} x \notin \Gamma \\
\text{(Weakening rule)} & \dfrac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma, x : B \vdash M : A} x \notin \Gamma \\
\text{(Application rule)} & \dfrac{\Gamma \vdash M : (\Pi x : A.B) \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B[N/x]} \\
\text{(Abstraction rule)} & \dfrac{\Gamma, x : A \vdash M : B \quad \Gamma \vdash (\Pi x : A.B) : s}{\Gamma \vdash (\lambda x : A.M) : (\Pi x : A.B)} \\
\text{(Conversion rule)} & \dfrac{\Gamma \vdash M : A \quad \Gamma \vdash B' : s \quad B =_\beta B'}{\Gamma \vdash M : B'}
\end{array}$$

2. Specific rules:

$$(s_1, s_2, s_3) \text{ rule} \quad \frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi x : A.B) : s_3} \quad \text{where } (s_1, s_2, s_3) \in \mathcal{R}$$

Figure 2.1: Typing axioms and rules.

1. *$\mathcal{S}$ is a subset of $C$ (elements of $S$ are called* sorts*);*
2. *$\mathcal{A}$ is a set of* axioms *of the form $c : s$ (where $c \in C$ and $s \in \mathcal{S}$);*
3. *$\mathcal{R}$ is a set of* rules *of the form $(s_1, s_2, s_3)$ (where $s_1$, $s_2$, $s_3 \in \mathcal{S}$); in the case that $s_2 = s_3$ we simply write $(s_1, s_2)$ instead of $(s_1, s_2, s_3)$.*

As usual, in the following, we will denote by $\rightarrow_\beta$ the relation of (one-step) $\beta$-reduction and by $=_\beta$ the corresponding equivalence, also known as $\beta$-conversion.

The next step is to define the PTS (with $\beta$-conversion) determined by a specification $\langle \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$:

**Definition 2.4** *A specification $\langle \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ determines a PTS (with $\beta$-conversion), denoted by $\lambda S = \lambda(S, A, R)$. The latter is a typed $\lambda$-calculus* à la Church *whose typing judgment is a triple $\langle \Gamma, M, A \rangle$, written $\Gamma \vdash M : A$, where $\Gamma$ is a pseudo-environment and $M$, $A$ are pseudo-terms. If $\Gamma \vdash M : A$ is derivable by means of the rules in Figure 2.1, $\Gamma$ is said to be a (legal) environment and $M$, $A$ are (legal) terms.*

*A PTS is called* singly sorted *if:*

1. *$(c{:}s_1)$, $(c{:}s_2) \in \mathcal{A}$ implies $s_1 = s_2$;*
2. *$(s_1, s_2, s_3)$, $(s_1, s_2, s_3') \in \mathcal{R}$, implies $s_3 = s_3'$.*

In the literature there are also Pure Type Systems with $\beta\eta$-conversion, where some care has to be taken in defining the rules of the equality judgment, otherwise one looses the Church-Rosser property (see [HHP93] for example):

**Theorem 2.1 (Church-Rosser property)** *If $M \to_\beta^* M'$[4] and $M \to_\beta^* M''$, then there exists $N$ such that $M' \to_\beta^* N$ and $M'' \to_\beta^* N$.*

Since their introduction, PTSs have been deeply investigated; for the sake of completeness we recall here some of their main properties (the interested reader is referred to [Bar92, Geu93]):

**Free variable lemma.** Let $\Gamma \vdash M{:}A$, then $FV(M) \cup FV(A) \subseteq dom(\Gamma)$.

**Transitivity lemma.** If $\Gamma$ is a legal context such that $\Gamma \vdash x_i{:}A_i$ $(1 \leq i \leq n)$, $\Delta \triangleq x_1{:}A_1, \dots x_n{:}A_n$ and $\Delta \vdash A{:}B$, then $\Gamma \vdash A{:}B$.

**Substitution lemma.** If $\Gamma, x{:}A, \Delta \vdash B{:}C$ and $\Gamma \vdash D{:}A$, then $\Gamma, \Delta[D/x] \vdash B[C/x]{:}C[D/x]$.

**Thinning lemma.** Let $\Gamma$ and $\Delta$ be two legal environments such that $\Gamma \subseteq \Delta$ and $\Gamma \vdash A{:}B$, then $\Delta \vdash A{:}B$.

**Generation lemma.**
1. $\Gamma \vdash c{:}C$ implies that there exists $s \in \mathcal{S}$ such that $C =_\beta s$ and $(c{:}s) \in \mathcal{A}$;
2. $\Gamma \vdash x{:}C$ implies that there exist $s \in \mathcal{S}$ and $B$ such that $B =_\beta C$, $\Gamma \vdash B{:}s$ and $(x{:}B) \in \Gamma$;
3. $\Gamma \vdash (\Pi x{:}A.B){:}C$ implies that there exists $(s_1, s_2, s_3) \in \mathcal{R}$ such that $\Gamma \vdash A{:}s_1$, $\Gamma, x{:}A \vdash B{:}s_2$ and $C =_\beta s_3$;
4. $\Gamma \vdash (\lambda x{:}A.M){:}C$ implies that there exist $s \in \mathcal{S}$ and $B$ such that $\Gamma \vdash (\Pi x{:}A.B){:}s$, $\Gamma, x{:}A \vdash M{:}B$ and $C =_\beta (\Pi x{:}A.B)$;
5. $\Gamma \vdash (MN){:}C$ implies that there exist $A$ and $B$ such that $\Gamma \vdash M{:}(\Pi x{:}A.B)$, $\Gamma \vdash N{:}A$ and $C =_\beta B[N/x]$.

**Subject Reduction theorem.** If $\Gamma \vdash M{:}A$ and $M \to_\beta^* N$, then $\Gamma \vdash N{:}A$.

**Condensing lemma.** If $\Gamma, x{:}A, \Delta \vdash M{:}B$ and $x \notin FV(\Delta) \cup FV(M) \cup FV(B)$, then $\Gamma, \Delta \vdash M{:}B$.

**Uniqueness of type for singly sorted PTSs.** If a PTS is singly sorted, $\Gamma \vdash M{:}A$ and $\Gamma \vdash M{:}B$, then $A =_\beta B$.

We have seen that rules in a PTS specification have the form $(s_1, s_2, s_3)$; in the particular case when $\mathcal{S} = \{*, \Box\}$ (called respectively the sort of *terms* and the sort of *types*) and all the rules in $\mathcal{R}$ have the form $(s_1, s_2, s_2)$ (whence, they can be simply written as $(s_1, s_2)$), we obtain all the systems of the so-called $\lambda$-*cube* (see Figure 2.2) by choosing which among all the possible rules are allowed for a particular system. Such combinations are listed in Table 2.1.

As we can see all the systems have the basic rule $(*, *)$ which allows one to abstract terms over terms. More sophisticated levels of abstraction can be reached by adding some of the other rules, e.g., all the systems on the upper face of the $\lambda$-cube feature the rule $(\Box, *)$ allowing one to derive impredicative types, like $\Pi A{:}*.(A \to A)$[5]:

$$\dfrac{\langle\rangle \vdash *{:}\Box \qquad \dfrac{\dfrac{\langle\rangle \vdash *{:}\Box}{A{:}* \vdash A{:}*} \qquad \dfrac{\dfrac{\langle\rangle \vdash *{:}\Box}{A{:}* \vdash A{:}*} \quad \dfrac{\langle\rangle \vdash *{:}\Box}{A{:}* \vdash A{:}*}}{A{:}*, x{:}A \vdash A{:}*}}{A{:}* \vdash (A \to A){:}*}}{\langle\rangle \vdash (\Pi A{:}*.(A \to A)){:}*}$$

[4] By $\to_\beta^*$ we denote the reflexive and transitive closure of $\to_\beta$.

[5] In general, $A \to B$ is an abbreviation for $\Pi x : A.B$ when $x$ does not occur in $B$.

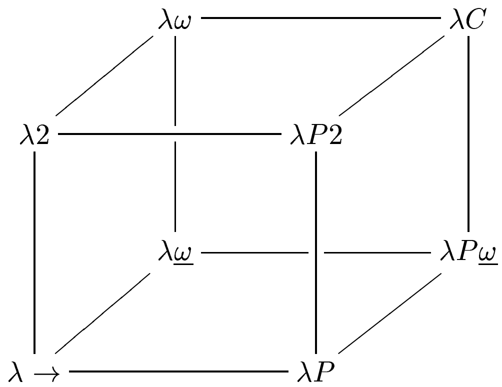


Figure 2.2: The $\lambda$-cube.

| **System** | **Set of rules** | | | |
|---|---|---|---|---|
| $\lambda \rightarrow$ | $(*,*)$ | | | |
| $\lambda 2$ | $(*,*)$ | $(\Box,*)$ | | |
| $\lambda P$ | $(*,*)$ | | $(*,\Box)$ | |
| $\lambda P2$ | $(*,*)$ | $(\Box,*)$ | $(*,\Box)$ | |
| $\lambda \underline{\omega}$ | $(*,*)$ | | | $(\Box,\Box)$ |
| $\lambda \omega$ | $(*,*)$ | $(\Box,*)$ | | $(\Box,\Box)$ |
| $\lambda P\underline{\omega}$ | $(*,*)$ | | $(*,\Box)$ | $(\Box,\Box)$ |
| $\lambda C$ | $(*,*)$ | $(\Box,*)$ | $(*,\Box)$ | $(\Box,\Box)$ |

Table 2.1: Rules for the systems of the $\lambda$-cube.

The expression *impredicative type* is justified in this case for the following reasons:

1. $\Pi A{:}*.(A \rightarrow A)$, being a cartesian product of types, is indeed a type $((\Pi A{:}*.(A \rightarrow A)) : *)$;

2. however, since the product is over all the possible types, it includes $\Pi A{:}*.(A \rightarrow A)$ which is in the process of being defined, whence the impredicativity of the definition.

The structure of the $\lambda$-cube is very useful since it allows one to understand at a glance the expressive power of each PTS labeling one of its vertices. For instance, the PTS $\lambda C$ at the top right vertex is the only one featuring all the possible rules; hence it represents the most general type theory. In the literature $\lambda C$ is also known as the *Calculus of Constructions (CC)* (introduced by Coquand and Huet in [CH88]); other well-known systems related to those of the $\lambda$-cube are the simply typed $\lambda$-calculus [Chu40] corresponding to $\lambda \rightarrow$, the *System F* introduced in [Gir72] and corresponding to $\lambda 2$, and the Edinburgh Logical Framework corresponding to $\lambda P$ [HHP93].

The most interesting property enjoyed by the systems of the $\lambda$-cube is the *strong normalization.*

**Definition 2.5** *Let $\lambda S$ be a PTS, then $\lambda S$ is strongly normalizing ($\lambda S \models SN$) if $\Gamma \vdash M{:}A$ implies that both $M$ and $A$ are strongly normalizing ($SN(M)$ and $SN(A)$), i.e., there are no infinite $\beta$-reductions starting from $M$, $A$.*

As anticipated, we have the following theorem:

**Theorem 2.2 (Strong normalization for the $\lambda$-cube)** *For every system in the $\lambda$-cube, the following hold:*

1. *if $\Gamma \vdash M{:}A$, then $SN(M)$ and $SN(A)$;*
2. *$\Gamma \vdash M{:}A$ and $\Gamma \triangleq x_1{:}A_1, \ldots, x_n{:}A_n$ imply $SN(A_i)$ for $1 \leq i \leq n$.*

Strong normalization and the Church-Rosser property entail the decidability of type checking: to test if $U =_\beta V$, reduce both to their unique normal forms and check if they are identical up to some renamings of the bound names (i.e., up to $\alpha$-equivalence). This is the most desirable property of a type theory-based logical framework since it allows one to implement an automatized proof-checker. Indeed, by the Curry-Howard isomorphism, proof-checking (that is, verifying that a given proof is indeed a proof of a given proposition) reduces to type checking (that is, verifying that the term corresponding to the given proof inhabits the type corresponding to the given proposition).

## 2.2 The Calculus of (Co)Inductive Constructions ($\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$)

In this section we will take a closer look at a particular logical framework, namely, the *Calculus of (Co)Inductive Constructions (*$\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$*)*, an impredicative intuitionistic type theory which extends the original CC [CH88, Hue92, Hue94, Tay88] with primitive support for (co)inductive types (see [CP90, PM93, Gim94, Wer94]).

### 2.2.1 $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$ as a PTS

The specification of $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$ as a PTS is actually more complicated w.r.t. the one of CC. Indeed, $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$ features two basic sorts, namely, `Prop` and `Set`; the former is intended to be the type of logical propositions, predicates or judgments, while the latter is supposed to be the type of datatypes (e.g., natural numbers, lists, trees etc.). Since also `Prop` and `Set` can be manipulated as ordinary terms, they must have a type. However, in order to avoid the well known Girard's paradox, it is not possible to assume that these sorts are inhabited by themselves; hence $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$ provides an infinite hierarchy of universes denoted by $\mathtt{Type}(i)$ for any natural $i$. The hierarchy is such that $\mathtt{Prop}{:}\mathtt{Type}(0)$, $\mathtt{Set}{:}\mathtt{Type}(0)$ and $\mathtt{Type}(i)\ {:}\mathtt{Type}(i+1)$. Summarizing, we have the following PTS-specification for $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$:

- $\mathcal{S} \triangleq \{\mathtt{Prop}, \mathtt{Set}\} \cup \{\mathtt{Type}(i) \mid i \in \mathbb{N}\}$
- $\mathcal{A} \triangleq \{\mathtt{Prop} : \mathtt{Type}(0), \mathtt{Set} : \mathtt{Type}(0)\} \cup \{\mathtt{Type}(i) : \mathtt{Type}(i+1) \mid i \in \mathbb{N}\}$
- $\mathcal{R} \triangleq \{\mathtt{Prop}, \mathtt{Set}\}^2 \cup \{(\mathtt{Type}(i), \mathtt{Type}(j), \mathtt{Type}(k)) \mid i, j, k \in \mathbb{N}, i, j \leq k\}$

### 2.2.2 (Co)Inductive definitions

The most appealing feature of $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$ is the possibility of defining types inductively; for instance one can introduce a new inhabitant of `Set` by declaring that it is the least set closed under the application of a given family of constructors. Since inductive definitions are possible in any extension of the (second order) PTS $\lambda 2$ by means of higher-order impredicative quantifications, one could conjecture that the abovementioned feature of $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$ is indeed superfluous. However, as remarked in [CP90], impredicative inductive types have two major drawbacks:

1. the impossibility of proving the induction principle;

2. when recursive definitions are called for, the conversion rule for natural numbers yielded by an impredicative definition is $rec(x, f)(S(n)) = f(\phi(n), rec(x, f)(n))$ instead of the expected $rec(x, f)(S(n)) = f(n, rec(x, f)(n))$ (where $\phi$ is a term such that $\phi(0) = 0$ and $\phi(S(n)) = S(\phi(n))$). It follows that, since $\phi(n)$ and $n$ are intensionally equal, but not convertible, programs (terms) containing expressions like $\phi^k(n)$ are inefficient since the reduction of $\phi^k(S(n))$ to $S(\phi^k(n))$ requires $k$ steps.

Coquand and Paulin-Mohring introduced in [CP90, PM93] a solution of this problem by extending the language of pseudo-terms of CC with special constants representing the definition, introduction and elimination of inductive types:

$$\mathcal{T} \qquad M ::= \ldots \mid \mathtt{Ind}(x{:}A)\{A_1|\ldots|A_n\} \mid \mathtt{Constr}(i, A) \mid \mathtt{Elim}(M, A)\{M_1|\ldots|M_n\}$$

where $i$ ranges over natural numbers. The informal meaning of the new pseudo-term constructors is the following:

- $\mathtt{Ind}(x{:}A)\{A_1|\ldots|A_n\}$ represents an inductive type (denoted by $x$) of sort $A$ whose constructors are given by $A_1, \ldots, A_n$;

- if $I \triangleq \mathtt{Ind}(x{:}A)\{A_1|\ldots|A_n\}$, then $\mathtt{Constr}(i, I)$ has type $A_i$ and represents the $i$-th constructor of the latter;

- $\mathtt{Elim}(M, A)\{M_1|\ldots|M_n\}$ is a term defined by induction/recursion over a term $M$ belonging to an inductively defined type, where $M_1, \ldots, M_n$ are the $n$ branches corresponding to the constructors of the latter.

Extending the language with new constructors requires to extend the typing system and the rules stating the equivalence of terms as well. Moreover, a new term reduction rule, called *$\iota$-reduction*, is needed in order to represent the computational content of inductive types (i.e., a way for performing computations with recursively defined functions is needed).

The same approach has been adopted by Giménez in [Gim94] in order to provide a better support for *coinductive types*[6] w.r.t. the classical approach of representing them by means of higher-order impredicative quantifications. Hence, the language of pseudo-terms has been further extended with new constructors[7]:

- $\mathtt{CoInd}(x{:}A)\{A_1|\ldots|A_n\}$ represents a coinductive type (denoted by $x$) of sort $A$ whose constructors are given by $A_1, \ldots, A_n$;

- if $I \triangleq \mathtt{CoInd}(x{:}A)\{A_1|\ldots|A_n\}$, then $\mathtt{Constr}(i, I)$ has type $A_i$ and represents the $i$-th constructor of the latter (this is not a new constructor, but simply an extension of `Constr` to coinductive types);

- `CoFix` $f{:}A := M$ where $f$ may occur in $M$ (under certain conditions briefly discussed below) represents a way of defining a new coinductive object denoted by $f$ through a "circular" equation (in [Gim94] it is illustrated how to encode such equations by means of more "conventional" *corecursion rules*). It is important to notice that the aim of the constructor `CoFix` is to generate elements of a coinductive type (instead of eliminating them).

[6]Coinductive types can be thought of as sets whose elements may be potentially infinite (circular) like, e.g., streams (i.e., infinite sequences) of natural numbers.

[7]The notation used in [Gim94] is slightly different from that in [PM93]; here we prefer to unify them in order to highlight the duality between inductive and coinductive types.

Like in the case of inductive types, the new constructors require extensions to the typing system and to the equivalence rules. Moreover, $\iota$-reductions must be carefully extended to take into account also a new form of redex involving coinductive objects defined by means of the `CoFix` constructor. The logic underlying the systems of the $\lambda$-cube is constructive, while infinite objects are non constructive, i.e., they cannot be represented effectively. However, it is sometimes possible to represent the process generating infinite objects if such a process is effective. This is the reason that allows us to say that coinductive types allow one to represent potentially infinite objects: defining equations built with the $CoFix$ constructor can thus be viewed as *generation rules* of infinite terms. It follows that, in order to *compute* with coinductive types, we must allow one to *unfold*[8] such equations in order to yield some information that can be subsequently used to, e.g., build a term. However, allowing the unfolding of coinductive definitions without restrictions means loosing strong normalization and, as a consequence, loosing the decidability of type checking. In order to avoid this, Giménez in [Gim94] imposes that unfolding of coinductive definitions can happen only in a *lazy* way, i.e., when they are the argument of a *case analysis* constructor[9].

Since the `CoFix` constructor does not impose any constraint on $M$ in `CoFix` $f{:}A := M$, it would be possible to introduce definitions like `CoFix` $f{:}A := f$ which would yield non-normalizing terms even if reductions of coinductive definitions are allowed only in case analysis statements. In order to avoid this problem `CoFix`-definitions must satisfy a *guarded-by-constructors* condition[10] which is a *syntactic criterion* derived from the *Guarded Induction Principle* (introduced in [Coq93]). Without entering into the details, the intuition behind such a criterion is that a legal defining equation of a coinductive object must yield at every reduction step some computational content. For instance, in the case of a `CoFix` equation defining a stream of naturals, at every reduction step a new piece of the stream must be generated, i.e., it must be possible to obtain an arbitrarily long (although finite) prefix of the stream with a finite number of reductions. Thus definitions like `CoFix` $f{:}A := f$ are rejected by the system. This leads to the previously mentioned idea of representing the processes generating infinite objects, provided they are effective.

The extended grammar defining the language of pseudo-terms allows for arbitrary declarations of (co)inductive types. Some of them can easily lead to inconsistencies (i.e., to inhabit the void type[11] corresponding to the absurdity). To ensure the soundness of recursive type declarations (both inductive and coinductive ones) the constructors must satisfy some conditions, i.e., they must be *forms of constructor* w.r.t. the new defined type. Keeping in mind that in $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$ $MN$ is written $(M\ N)$ and $\Pi x{:}M.N$ is written $(x{:}M)N$, these conditions are formally expressed in the following way (where $\vec{x}{:}\vec{M}$ is an abbreviated notation for $x_1{:}M_1,\ldots,x_n{:}M_n$ and $|\vec{M}|$ stands for the length of $\vec{M}$):

**Definition 2.6** *The variable $X$ occurs strictly positively in the term $P$ if $P \equiv (\vec{x}{:}\vec{M})(X\vec{N})$ and $X$ does not occur free in $M_i$ $\forall i \in |\vec{M}|$, nor in $N_j$ $\forall j \in |\vec{N}|$.*

[8] By unfolding an equation of the form `CoFix` $f{:}A := M$, we mean the operation of substituting it for the free occurrences of $f$ in the body $M$ ($M[(\texttt{CoFix}\ f{:}A := M)/f]$).

[9] Since this thesis does not aim to study coinductive types, we do not insist further on this topic and we refer the interested reader to [Gim94, Gim96]

[10] Roughly, the *guarded-by-constructors* condition amounts to ensure that the occurrences of $f$ in $M$ must be guarded (i.e., they must be preceded) by a constructor of the corresponding coinductive type.

[11] The void type is defined as $\bot \triangleq \texttt{Ind}(x{:}\texttt{Prop})\{\}$; the corresponding non-dependent elimination principle is $(P{:}\texttt{Prop}\bot \rightarrow P)$ saying that from $\bot$ we can derive any proposition.

**Definition 2.7** *Let $X$ be a variable. The terms which are a form of constructor w.r.t. $X$ are generated by the syntax:*

$$Co ::= (X\ \vec{N}) \mid P \rightarrow Co \mid (x{:}M)Co$$

*with the following restrictions for $X$: it does not occur free in $N_i$ $\forall i \in |\vec{N}|$, it is strictly positive in $P$, and it does not occur free in $M$.*

## 2.3 The proof assistant Coq

The `Coq` system [TCDT01] is a proof assistant whose underlying metalanguage is $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$. It is the result of over ten years of research at INRIA[12].

In 1984, Coquand and Huet wrote the first implementation of the Calculus of Constructions in CAML (a functional language belonging to the ML family and developed at INRIA). The core of the system was a proof-checker, known as *Constructive Engine*, which allowed the declaration of axioms and parameters, the definition of mathematical types and objects and the explicit construction of proof-objects represented as $\lambda$-terms. Then, a section mechanism, called the *Mathematical Vernacular*, allowing one to develop mathematical theories in a hierarchical way was added. At the same time an interactive *theorem prover* executing tactics written in CAML was implemented; by means of this tool proofs could be built progressively in a *top-down* style, generating subgoals and backtracking when needed. Moreover, the basic set of tactics could be extended by the user.

With the introduction of inductive types by Paulin-Mohring, a new set of tactics allowing one to carry out proofs by induction was added. The implementation of a module for compiling programs extracted from proofs in CAML is due to Werner. Starting from version V5.10 the `Coq` system supports coinductive types as well, implementing a powerful tactic `Cofix` which allows the user to interactively build proof involving coinductive predicates in a natural way, without having to exhibit a priori a bisimulation like it is usually done with "pencil and paper".

The essential features of the system are:

1. a logic metalanguage allowing one to represent formal systems;

2. a powerful proof engine assisting the user in the task of formally reasoning about the encoded systems;

3. a *program extractor* yielding functional programs from constructive proofs.

Since program extraction is not a topic of this thesis, we will only provide a brief introduction to the first two features in order to make more readable the material contained in Chapter 6, where we will illustrate two case studies developed in `Coq`(for a full and detailed introduction to the system the reader is referred to [TCDT01]).

The proof engine allows one to interactively develop distinct partial proofs in parallel; indeed, the structure of proof-trees is a combined representation of $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$ trees and of abstract syntax trees of *tactic scripts*, allowing one to explore proofs at various levels of detail. Other interesting features are:

- the possibility of programming new tactics,

[12]Institut National de Recherche en Informatique et Automatique

- a file system service allowing the separate compilation of modules yielding *image files* that can be loaded quickly without being processed again by the system,
- *pretty printing* services,
- recursive and mutually recursive definitions,
- completely automatized inversion tactics for (co)inductive types,
- powerful tactics, e.g., `Omega` allows one to automatically solve formulæ of Presburger arithmetic,
- native support for coinductive types and coinductive proofs.

### 2.3.1 Terms, types and sorts

In the previous sections we have seen that a fundamental features of type theories is that they allow one to manipulate only two categories of objects: terms and types. The latter specify the classes the former can belong to; every object must belong to a type. In $CC^{(Co)Ind}$ (the underlying metalanguage of `Coq`) there are no syntactic distinctions between terms and types since they can be defined in a mutual way and similar constructions can be applied to both categories. Hence, also the types of `Coq` must have a type: this is the reason for the introduction of sorts which are constants of the system. In Section 2.2.1 we introduced the sorts of $CC^{(Co)Ind}$ which are infinitely many in order to avoid Girard's paradox. However, in `Coq` the user does not have to worry about the indexes $i$ of `Type`($i$), since the latter are automatically handled by the system. Hence, from the user's viewpoint we have `Prop:Type`, `Set:Type` and `Type:Type`.

### 2.3.2 The `Gallina` specification language

In this section we will briefly recall the specification language of `Coq`, called `Gallina`, which allows one to develop mathematical theories handling axioms, hypotheses, parameters, constants definitions, functions, predicates and so on. The usual operations of typed $\lambda$-calculi are rendered in `Gallina` as follows:

- $\lambda x : M.N$ is written `[x:M]N`,
- $(MN)$ is written `(M N)` (the application is left associative),
- $M \to N$ is written `M -> N`,
- $\Pi_{x:M}N$ is written `(x:M)N`.

#### Declarations

The system waits for user's commands by means of a prompt (`Coq <`); it is possible to enrich the current environment through the commands `Variable`, `Hypothesis`, `Axiom` and `Parameter`[13] (the use of `Axiom` and `Hypothesis` is recommended for logical entities, while in the remaining cases `Variable` and `Parameter` should be used). For instance, declaring a variable ranging over natural numbers (a predefined type in `Coq`) can be done with the following command:

[13]When a section is closed, the objects declared by means of the first two commands are automatically discharged.

```
Coq < Variable n:nat.
```

The system communicates that the operation has been carried out successfully by emitting the following message:

```
n is assumed
```

The `Check` command allows one to control the type of a previously declared variable:

```
Check n.
```

and in the case of the previous declaration we obtain:

```
n:nat
```

### Definitions

Definitions differ from declarations in the sense that the former associate a name to a term correctly typable in the current environment, while the latter associate a type to a name. It follows that the name of a defined object can be replaced at any time by the body of the corresponding definition (this is commonly known as $\delta$-reduction). The general form of a definition is the following:

$$\texttt{Definition}\ ident := term.$$

The command `Print` applied to the name of a defined object yields the body of the definition and its type.

Inductive types are introduced as follows:

$$\texttt{Inductive}\ ident : term := ident_1 : term_1 \mid \ldots \mid ident_n : term_n.$$

*ident* is the name of the new object, *term* is its type and the identifiers $ident_1, \ldots, ident_n$ (whose types are, respectively, $term_1, \ldots, term_n$) represent its constructors. As we noticed in Section 2.2.2, inductive definition must satisfy some syntactic conditions in order to avoid inconsistencies (see Definition 2.7).

For instance the set of natural numbers can be introduced as follows:

```
Coq < Inductive nat : Set := O : nat | S : nat -> nat.
nat_ind is defined
nat_rec is defined
nat_rect is defined
nat is defined
```

The `Coq` system automatically generates eliminations principles for the new inductive type: `nat_ind` is associated to the sort `Prop`, `nat_rec` to the sort `Set` and `nat_rect` to the sort `Type`. The first elimination scheme represents the well known *Peano's induction principle*:

```
Coq < Check nat_ind.
nat_ind :
   (P:nat->Prop)(P O)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)
```

On the other hand, `nat_rec` encodes primitive recursion on natural numbers.

In an analogous way it is possible to define new coinductive types:

$$\texttt{CoInductive}\ ident : term := ident_1 : term_1 \mid \ldots \mid ident_n : term_n.$$

As for inductive definitions the constructors types must satisfy the constraints of Definition 2.7. Obviously, in this case no elimination schemes are yielded by the systems.

In both cases (inductive and coinductive) it is possible to introduce parametric and mutual definitions. For further details the reader can refer to [TCDT01].

### Case expressions

A *case expression* is a destructive operation whose underlying idea is taking a term $m$ belonging to a recursive type $I$ (inductive or coinductive) and building a term of type $(P\ m)$ (depending on $m$) by cases. The syntax is the following:

$$< term > \texttt{Case}\ term\ \texttt{of}\ \underbrace{term_1 \cdots term_n}_{\text{n expressions}} \texttt{end}$$

The preceding term, in the case that $m = (c_i\ t_1 \cdots t_p)$, reduces to $(term_i\ t_1 \cdots t_p)$.

### `Fixpoint` and `CoFixpoint`

Fixed point definitions on inductive objects can be introduced as follows:

$$\texttt{Fixpoint}\ ident[ident_1 : term_1] : term_2 := term_3.$$

The intuitive semantics of the previous definitions is that *ident* represents a recursive function with argument $ident_1$ of type $term_1$ (must be inductive) such that $(ident\ ident_1)$ is equivalent to $term_3$ of type $term_2$. Hence, the type of *ident* is $(ident_1 : term_1)term_2$. Obviously, fixed point definitions must satisfy some syntactic constraints in order to ensure that the defined function always terminates. More precisely, the constraints amount to the *structurally smaller calls* principle, i.e, the recursive calls of the function must apply proper subterms of the recursive argument. Moreover, in order to preserve strong normalization, fixed point definitions can reduce only when the recursive argument is a term beginning with a constructor. Fixed point definitions can also be parametric, mutually defined and defined by *pattern matching* (for more details, the reader can refer to [TCDT01]).

The `Gallina` specification language also allows one to build infinite objects by means of the `CoFixpoint` command, implementing the `CoFix` constructor of $CC^{(Co)Ind}$ introduced in Section 2.2.2. The syntax is analogous to that of the `Fixpoint` command, while the associated reduction rule is lazily defined in order to preserve the strong normalization property of the system as we briefly recalled in Section 2.2.2.

### The proof editing mode

When `Coq` enters the so-called *proof editing mode*, the system prompt changes from `Coq <` to `ident <`, where `ident` is the name of the theorem one wants to prove. Besides the commands briefly recalled so far, other specific commands are available in proof editing mode. At any instant the proof context is represented by a set of subgoals to prove (initially this set contains only the theorem) and by a set of hypothesis (initially the list is empty). Both sets are manipulated and modified through tactics; when the proof development is completed (i.e., when the set of subgoals is empty), the system notifies the user with the message `Subtree proved!` and the command `Qed` (or `Save`) is made available in order to store the proof in the current environment for later use in subsequent theorems.

The commands starting the proof editing mode are the following:

- `Goal` *term*. In this case *term* and the associated name is `Unnamed_thm`.

- `Theorem` *ident* : *term*. This command has the same effect of `Goal`, with the exception of naming the current goal *ident* (other variants are `Lemma`, `Remark` and `Fact`).

### Predefined logic objects

As outlined before, terms having as sort `Prop` represent propositions, while predicates are rendered by means of dependent types and, when applied to the right arguments, yield propositions. In `Coq`, like in traditional logics, both propositions and predicates can be combined together in order build other propositions and predicates by means of *logic connectives*. The most commonly used are rendered as follows in `Coq`:

- The true constant is denoted by `True` and it is inductively defined by the non-dependent type which is always inhabited:

  ```
  Coq < Inductive True : Prop := I : True.
  ```

- The false constant is denoted by `False` and is defined by the empty inductive type (i.e., the type with no inhabitants):

  ```
  Coq < Inductive False : Prop := .
  ```

- The negation of a proposition in intuitionistic logic is equivalent to say that we can derive the absurdity from it:

  ```
  Coq < Definition not := [A:Prop] A -> False.
  ```

- The logic conjunction is denoted by `A /\ B` and is defined by:

  ```
  Coq < Inductive and [A,B:Prop] : Prop
  Coq <     := conj : A -> B -> A /\ B.
  ```

- The logic disjunction is denoted by `A\/B` and is defined by:

  ```
  Coq < Inductive or [A,B:Prop] : Prop
  Coq <     := or_introl : A -> A \/ B
  Coq <      | or_intror : B -> A \/ B.
  ```

- The logic implication between proposition is rendered through the non dependent type constructor `->`.

- Universal quantification on a predicate `P` is rendered by means of dependent types. For instance the formula $\forall x \in A.P(x)$ is encoded by `(x:A)P`.

- Existential quantification is inductively defined as follows:

  ```
  Coq < Inductive ex [A:Set;P:A->Prop] : Prop
  Coq <     := ex_intro : (x:A)(P x) -> (ex A P).
  ```

It is worth noticing that the definitions of implication and universal quantification are a direct consequence of the Curry-Howard isomorphism.

Equality is also inductively defined:

```
Coq < Inductive eq [A:Set;x:A] : A->Prop
Coq <     := refl_equal : (eq A x x).
```

Usually `(eq ? x y)` is written `x=y` in `Coq`. The abovementioned definition is due to Christine Paulin-Mohring and amounts to the least reflexive relation[14].

[14] It is formally provable that the given definition of equality coincides with Leibniz definition of equality, stating that two object are equal if and only if they satisfy the same properties. Hence, the equality of `Coq` is often referred to as Leibniz equivalence.

### Proof tactics

Generally, an inference rule represents a link between a conclusion and one or more premises: this characterization allows one to read it in two different ways. The former amounts to the classical *forward reasoning* and consists of assuming the premises in order to derive the conclusion. The second interpretation is less immediate, but more useful in the field of computer assisted proof development; indeed, the so-called *backward reasoning* proceeds from the conclusion to the premises, starting from the idea that in order to prove the conclusion, it is first necessary to derive the premises. `Coq` tactics work in this way, replacing the current *goal* (the conclusion) with one or more *subgoals* (the premises). So doing, the proof tree is progressively built starting from the root and the subgoals obtained by the application of a given tactic represent the roots of the related subtrees. The proof of a subgoal happens by means of axioms, hypothesis of the current proof environment or previously proved results.

The `Coq` system provides a great number of tactics which can be grouped as follows:

1. Exact tactics:

   - `Exact` *term*: this tactic can be applied to any goal; if `T` is the current goal and `p` is a term of type `U`, then `Exact p` succeeds if and only if `T` and `U` are convertible types.
   - `Assumption`: another tactic applicable to any goal: it automatically searches in the current proof environment if there is a hypothesis whose type coincides with the current goal.

2. Basic tactics:

   - `Intro`: it can be applied to any goal having the form of a product; in the case that the latter is a dependent type `(x:T)U`, the effect of the tactic is to add to the current proof environment the hypothesis `x:T` or `xn:T`, with `xn` fresh in the case `x` is already present. On the other hand if the goal is a non-dependent product `T->U`, the tactic will introduce in the current proof environment a hypothesis of type `T` (if a hypothesis name is not supplied by the user, it will be automatically chosen by `Coq`). The variant `Intros` repeats the tactics `Intro` as much as possible.
   - `Apply` *term*: when applied to any goal, this tactic tries to unify the current goal with the conclusion of the type of *term*, where the latter is a term well formed in the current environment. If the unification succeeds, then the tactic yields as many subgoals as the instantiations of the premises of the type of *term*. An extensively used variant is `Apply` term `with` $term_1 \cdots term_n$, which allows one to instantiate all the premises of the type of *term* which are not deducible from the unification.
   - `Cut` *term*: this tactic is very useful in the proof development; it can be applied to any goal and allows one to prove the current goal `T` as a consequence of `U`; indeed `Cut U` replaces `T` with two subgoals, namely, `U->T` and `U`.

3. Introduction tactics:

   - `Constructor i`: if the head of the conclusion of the current goal is an inductive constant `I`, it is possible to apply the tactic `Constructor i` (where `i` is a number less or equal to the number of constructors of `I`) whose effect is the same of the sequence `Intros; Apply ci` (where `ci` is the *i*-th constructor of `I`).

4. Elimination tactics: these tactics are specialized for proofs by induction and case analysis.

   - `Elim` *term*: this tactic can be applied to any goal under the condition that the type of *term* is an inductive constant; then, depending on the type of the goal, it applies the appropriate destructor. Like in the case of the `Apply` tactic, there is a variant allowing the user to specify the values for the dependent premises of the elimination scheme which cannot be deduced by the matching operation. (`Elim` *term* `with` $term_1 \cdots term_n$).
   - `Case` *term*: the tactic can be applied in order to carry out a proof by case analysis; hence, the type of *term* must be inductive or coinductive.

5. Inversion tactics: when the current proof involves (co)inductive predicates, it is very common to fall into one of the following cases:

   - in the proof environment there is an inconsistent instance of a (co)inductive predicate; hence, the current goal can be proved by absurdity;
   - in the proof environment there is an instance $(I\ \vec{t})$ of a (co)inductive predicate $I$ and we want to derive, for each constructor $c_i$ of $I$, all the necessary conditions that should hold for $(I\ \vec{t})$ to be proved by $c_i$.

   Generally, given an instance $(I\ t_1 \cdots t_n)$ of a (co)inductive predicate, the derivation of the necessary conditions such that $(I\ t_1 \cdots t_n)$ holds is called *inversion*. Inversion tactics can be classified in three categories:

   (a) tactics inverting an instance of a (co)inductive predicate without storing in the current environment the inversion lemma: `Inversion`, `Simple Inversion` and `Inversion_clear`;

   (b) tactics generating and storing in the current environment the inversion lemma used to invert an instance of a (co)inductive predicate: `Derive Inversion` and `Derive Inversion_clear`;

   (c) tactics inverting an instance of a (co)inductive predicate using an already defined inversion lemma: `Use Inversion`.

6. Conversion tactics:

   - `Change` *term*: it can be applied to any goal; `Change U` replaces the current goal `T` with `U` if and only if `U` is legal and `T` and `U` are convertible.
   - `Simpl`: if `T` is the current goal, this tactic first applies the $\beta\iota$-reductions, then unfolds transparent constants[15] and finally applies again $\beta\iota$-reductions until possible.
   - `Unfold` *ident*: this tactic replaces all the occurrences of the transparent constant *ident* with the body of the corresponding definition in the current goal (this process is also called $\delta$-reduction).

7. Automatic tactics:

[15] Constants become non-transparent by applying the `Opaque` command.

- `Auto`: this tactic uses a *Prolog-like resolution* in the sense that it first tries the tactic `Assumption`, then if the latter fails it applies the tactic `Intro` until the current goal is an atomic one (adding the generated hypotheses as hints); then it tries each of the tactics associated to the head symbol of the goal starting from those with lower costs. The whole process is recursively applied to the generated subgoals.
- `Trivial`: this is a variant of `Auto` which is not recursive and tries only hints whose cost is zero (see below the `Hint` command).

`Auto` and `Trivial` use *hints* organized in several databases in order to automatically solve the current goal. Each database maps *head symbols* o *hints list* (consisting in a collection of tactics); each hint has a cost[16] and a pattern and `Auto` use it if the conclusion of the current goal match its pattern. In the case that several distinct hints are applicable at the same time, the ones with lower costs are tried first. Such a list can be extended by the user using the command `Hint name: database:=hint_definition`, where `hint_definition` can be any of the following:

- `Resolve` *term*: depending on whether the type of *term* is a product or not, the tactic `Apply` term or `Exact` term is added to the hint list associated with the head symbol of the type of *term* in the specified database.
- `Immediate` *term*: the tactic `Apply` *term*; `Trivial` is added to the hint list associated with the head symbol of the type of *term* in the specified database.
- `Constructors` *ident*: if *ident* is an inductive type, then all its constructors are added as hints of type `Resolve`.
- `Unfold` *ident*: the tactic `Unfold` *ident* is added to the hints list that will be used only in the case that the head constant of the current goal is *ident* itself.
- `Extern` *num pattern tactic*: this command allows one to add an arbitrary tactic to a hint list by specifying the cost (*num*), the pattern and the tactic to apply.

Obviously, we have not covered the whole set of tactics provided by `Coq` (e.g. we have not mentioned coinductive tactics, since the case studies presented in Chapter 6 do not use them), but we hope to have given an idea of how the proof development with this system can be carried out. We conclude this section by recalling that tactics can be combined by means of some operators:

- `Do` *num tac* repeats *num* times the tactic *tac*;
- $tac_1; tac_2$ applies the tactic $tac_2$ to each of the subgoals generated by $tac_1$ (we already encountered this operator in the previous list);
- $tac$; $[tac_1| \cdots |tac_n]$ applies the tactic $tac_i$ to the $i$-th subgoal generated by $tac$;
- `Try` *tac* allows one to apply the tactic *tac* without considering the eventual failure of the latter.

[16] Given by the number of subgoals generated by the corresponding tactic.

# 3

# Encoding methodology

Using a type theory based logical framework as a specification language for representing formal systems requires, as a first step, to choose an encoding methodology. Indeed, object languages typically consist of two components:

- the syntax, i.e., the syntactic categories like, e.g., names/variables, terms etc.
- the semantics which is formulated by a series of judgments over syntactic entities, e.g, labeled transition systems for process algebras, the $\beta$-reduction of $\lambda$-calculus, satisfaction and validity relations for the first-order logic etc.

In order to faithfully represent an object language, both components must be encoded. In this chapter we will illustrate the possible approaches, focusing in particular on the *Higher-Order Abstract Syntax* (HOAS) approach and the *judgments-as-types principle*. The notions we are going to recall are applicable to any logical framework having at least the expressive power of the Edinburgh LF [HHP93] (except of course when we speak of inductive definitions: in this case the reference framework we consider is $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$).

## 3.1 Representing expressions

The general approach in representing the syntax of an object language is to introduce an LF type for each syntactic category and to declare a constant for each expression-forming construct of the object language. The collection of types and constants introduced in this way is called the *signature* of the object language. This approach allows one to establish a bijective correspondence between the syntactic entities of the object language and the so called *canonical forms* of the corresponding type in the LF. Since the systems of the $\lambda$-cube are strongly normalizing (see the previous chapter), they provide a notion of *normal form* (i.e., a term that cannot be further reduced). Canonical forms, w.r.t. a typing environment and a signature, are a stronger notion corresponding to long $\beta\eta$-normal forms, i.e., to normal forms where each constant and variable occurrence is *fully applied*[1] w.r.t. the given typing

[1]Without entering into the details of the formal definition [HHP93], an occurrence of a constant or variable $\xi$ is fully applied in a legal term $M$ w.r.t. a typing environment $\Gamma$ and a signature $\Sigma$ if and only if it is of the form $\xi M_1 \cdots M_n$, where $n$ is the *arity* of $\xi$. The arity of a kind or type is the number of $\Pi$ in the prefix of its normal form; the arity of a constant occurrence in a signature is the arity of its type in that

$$\begin{array}{rcl} \epsilon^{\Lambda}_X & : & \Lambda_X \longrightarrow tm_X \\ \epsilon^{\Lambda}_X(x) & \triangleq & x \\ \epsilon^{\Lambda}_X(MN) & \triangleq & (app\ \epsilon^{\Lambda}_X(M)\ \epsilon^{\Lambda}_X(N)) \\ \epsilon^{\Lambda}_X(\boldsymbol{\lambda} x.M) & \triangleq & (lam\ \lambda x{:}tm.\epsilon^{\Lambda}_{X,x}(M)) \end{array} \qquad \begin{array}{rcl} \delta^{\Lambda}_X & : & tm_X \longrightarrow \Lambda_X \\ \delta^{\Lambda}_X(x) & \triangleq & x \\ \delta^{\Lambda}_X((app\ M\ N)) & \triangleq & \delta^{\Lambda}_X(M)\delta^{\Lambda}_X(N) \\ \delta^{\Lambda}_X((lam\ M)) & \triangleq & \boldsymbol{\lambda} x.\delta^{\Lambda}_{X,x}(Mx) \end{array}$$

Figure 3.1: Encoding and decodings maps for untyped $\lambda$-calculus.

environment and signature. The abovementioned bijective correspondence is given in terms on an encoding map $\epsilon$ and a decoding map $\delta$ and is usually required to be compositional (i.e., it must commute with substitution).

### 3.1.1 Higher-Order Abstract Syntax (HOAS)

If the object language features binding operators, it is possible to represent them by means of constants whose type is functional. Thus, binders are rendered by the binder of the underlying metalanguage of the LF (i.e., the $\lambda$-abstraction operator). This approach allows one to represent variables of the object language by metavariables of the LF of a suitable type. Hence, the related machinery of $\alpha$-conversion and capture-avoiding substitution are shifted to the metalevel, freeing the user from an explicit implementation.

For instance, in the case of untyped $\lambda$-calculus, the syntax is defined by the following grammar (there are two syntactic categories: variables $\mathcal{V}$, ranged over by $x, y, z, \ldots$, and terms $\Lambda$):

$$\Lambda \qquad M, N ::= x \mid MN \mid \boldsymbol{\lambda} x.M$$

Since there is a binder in the object language (denoted by a bold $\lambda$ to distinguish it from the $\lambda$-operator of the logical framework), the corresponding HOAS-signature is the following:

$$\Sigma_\Lambda \triangleq \{tm{:}*,\ app{:}tm \to tm \to tm,\ lam{:}(tm \to tm) \to tm\}$$

Given $X \triangleq \{x_1, \ldots, x_n\} \subset \mathcal{V}$ finite, we denote by $\Lambda_X$ the set $\{M \in \Lambda \mid fv(M) \subseteq X\}$ (where $fv(M)$ denotes free variables of $M$, as usual) and by $tm_X$ the canonical forms of type $tm$ such that $\Gamma_X \vdash_{\Sigma_\Lambda} M{:}tm$ holds[2] (where $\Gamma_X = x_1{:}tm, \ldots, x_n{:}tm$). As we can see from the encoding and decoding maps in Figure 3.1 (it is assumed that in the clauses involving $\boldsymbol{\lambda}$ and $lam$ $x$ is chosen to be fresh, i.e., $x \notin X$; moreover, $X, x$ is a compact notation for $X \cup \{x\}$), the abstraction constructor of the object language is represented by the constant $lam$ taking functions from $tm$ to $tm$ as arguments. This allows one to represent variables of untyped $\lambda$-calculus with metavariables of the LF of type $tm$. As anticipated, the HOAS-based approach allows one to automatically delegate to the metalanguage the mechanisms of $\alpha$-conversion and capture-avoiding substitution. For instance, in the case of $\lambda$-calculus we have:

**$\alpha$-conversion** : the terms $(lam\ \lambda x.M)$ and $(lam\ \lambda y.M\{y/x\})$ (where $y$ is a fresh variable) are identified by the LF. Thus, in case of a name clash, like in $(app\ (lam\ \lambda x.M)\ x)$, the metalanguage automatically renames the bound occurrences of $x$ with a fresh variable $y$, yielding the term $(app\ (lam\ \lambda y.M\{y/x\})\ x)$. In the case of a "traditional" encoding

signature. Similarly, the arity of a variable occurrence in a typing environment is the arity of its type in that environment, while the arity of a bound variable occurrence is the arity of the type label attached to its binding occurrence.

[2]In the following $\Gamma \vdash_{\Sigma_\Lambda} M{:}tm$ stands for $\Gamma, \Sigma_\Lambda \vdash M{:}tm$. It is usually preferred to keep the typing environment distinct from the signature, since the latter contains constants, while the former contains variables.

instead, keeping bound variables distinct from free ones is left to the user with all the related mistakes commonly made with "pencil and paper".

**capture-avoiding substitution** : this operation is also delegated to the metalanguage; indeed, we can render the $\beta$-reduction of $(\boldsymbol{\lambda}x.M)N$ to $M\{N/x\}$ by saying that $(app\ (lam\ \overline{M})\ \overline{N})$ reduces to $\overline{MN}$, where $X \triangleq fv(M) \cup fv(N)$, $(lam\ \overline{M}) \triangleq \epsilon^{\Lambda}_{X}(\boldsymbol{\lambda}x.M)$ and $\overline{N} \triangleq \epsilon^{\Lambda}_{X}(N)$ (since $\overline{M}$ has type $tm \to tm$). Hence, capture-avoiding substitution is modeled by functional application in the logical framework. Again, the user is freed from an explicit encoding of this notion.

The adequacy of the encoding of the syntax of untyped $\lambda$-calculus is given by the following proposition:

**Proposition 3.1 (Adequacy of syntax)** *The encoding $\epsilon^{\Lambda}_{X}$ is a bijection between $\Lambda_X$ and $tm_X$. Moreover, the encoding is compositional in the sense that for $M$ a $\lambda$-term with free variables in $X \triangleq \{x_1, \dots, x_n\}$ and $N_1, \dots, N_n$ $\lambda$-terms with free variables in $Y$ the following holds:*

$$\epsilon^{\Lambda}_{Y}(M\{N_1/x_1, \dots, N_n/x_n\}) = \epsilon^{\Lambda}_{X}(M)\left\{\epsilon^{\Lambda}_{Y}(N_1)/x_1, \dots, \epsilon^{\Lambda}_{Y}(N_n)/x_n\right\}$$

*Proof.* Clearly, $\epsilon^{\Lambda}_{X}$ is injective by definition. Surjectivity follows form the definition of $\delta^{\Lambda}_{X}$ and an inspection of canonical forms $\xi M_1 \cdots M_n$ of type $tm$. Indeed, the only choice for $\xi$ is given by $app$ and $lam$; hence, the types of these constants allow one to conclude that $\delta^{\Lambda}_{X}$ is total and well-defined. An easy induction on the structure of $M$ allows to prove that $\delta^{\Lambda}_{X}(\epsilon^{\Lambda}_{X}(M)) = M$. Compositionality is proved by means of another straightforward induction on terms of the object language. $\square$

### 3.1.2 HOAS and Inductive Definitions

So far, there is no assumption about the type theory used to encode object languages (this is reflected by the fact that we illustrated the example about the untyped $\lambda$-calculus with the generic notation used in the previous chapter to describe the language of terms of PTSs). However, even if one could rely on a logical framework as simple as the Edinburgh LF, it is often useful for "real" and complex case studies to choose a LF with the capability of automatically providing induction principles on sets and propositions. This feature is particularly relevant in view of a formal development of the metatheory of the encoded language, since many proofs with "pencil and paper" are carried out by means of structural inductions on the syntax.

At a first sight, it seems that using a HOAS-based encoding approach in a LF featuring inductive types is the best solution for carrying out formal developments with the minimum effort, delegating as much as possible to the metalevel ($\alpha$-conversion, capture-avoiding substitution and the generation of suitable induction principles). Unfortunately, a "close encounter" between HOAS and inductive definitions can yield several problems. If, for instance, we choose the `Coq` system [TCDT01] as our logical framework, we cannot encode the untyped $\lambda$-calculus as we did in the previous section taking advantage of inductive definitions. Indeed, the encoding would be the following:

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

However, as we recalled in the previous chapter (Section 2.2.2), the underlying metalanguage of `Coq` ($CC^{(Co)Ind}$) rejects the previous definition since there is a negative occurrence of `tm` in the type of `lam`; thus the latter is not a *form of constructor* (see Definition 2.7). The reason for prohibiting this kind of constructors is to avoid non-normalizing terms like the following:

```
Definition F: tm -> tm :=
    [x:tm]Case x of [t,t':tm](app t t') [f:tm->tm](f (lam f)) end
```

$$(\mathtt{F}\ (\mathtt{lam}\ \mathtt{F})) : \mathtt{tm} \rightarrow_{\beta\delta\iota} (\mathtt{F}\ (\mathtt{lam}\ \mathtt{F})) : \mathtt{tm} \rightarrow_{\beta\delta\iota} \ldots$$

In this case, if one does not want to drop inductive definitions and resort to an axiomatic encoding, the only feasible solution in `Coq` is to introduce a specific type `var` for the identifiers (variables). So doing, it is possible to introduce the inductive type:

```
Inductive tm: Set:=
    is_var : var -> tm
  | app    : tm -> tm -> tm
  | lam    : (var -> tm) -> tm
```

It is worth noticing that variables of the object language are now represented by `Coq` metavariables of type `var` (instead of type `tm`). A first drawback of this solution is that only $\alpha$-equivalence can be delegated to the metalevel. Indeed, the `lam` constructor accepts as arguments abstractions of type `var -> tm` (instead of `tm -> tm`). It follows that functional application can model only substitution of variables for variables (instead of substitution of compound terms for variables which needs to be implemented by the user). This encoding approach is sometimes called *half-HOAS*, while the former (with $lam : (tm \rightarrow tm) \rightarrow tm$) is called *full-HOAS*.

It is important to notice that if `var` is an inductive type, a serious problem arises. For instance, if we take the built-in `Coq` type encoding natural numbers for representing variables (hence, `is_var` has type `nat->tm`), we can define the following terms:

```
Definition weird1: tm :=(lam [x:nat]Case x of
                                    (* 0 *)      (is_var O)
                                    (* (S n) *) [n:var](is_var (S n))
                                    end)

Definition weird2: tm :=(lam [x:nat]Case x of
                                    (* 0 *)      (is_var O)
                                    (* (S n) *) [n:var](is_var n)
                                    end)
```

Both `weird1` and `weird2` are in head normal form and are not the encoding of any term of the untyped $\lambda$-calculus. However, the first is extensionally equivalent[3] to the $\lambda$-term `(lam`

[3] Following [DFH95], extensional equivalence is defined as the following `Coq` predicate:

```
Inductive eq_tm   :  tm -> tm -> Prop :=
  eq_tm_var       :  (x : var)(eq_tm (is_var x) (is_var x))
| eq_tm_app       :  (m1, m2, n1, n2 : tm)(eq_tm m1 n1) -> (eq_tm m2 n2) ->
                     (eq_tm (app m1 m2) (app n1 n2))
| eq_tm_lam       :  (M, N : var-> tm)((x : var)(eq_tm (M x) (N x))) ->
                     (eq_tm (lam M) (lam N)).
```

`[x:nat](is_var x))` (i.e., the encoding of $\boldsymbol{\lambda} x.x$) and could be acceptable. Instead, `weird2` clearly does not correspond (even up to extensional equality) to any untyped $\lambda$-term[4] Hence, we have defined an *exotic term*, i.e., a canonical term not corresponding to any entity of the object language; in other words we have lost the adequacy of the encoding.

There are two possible solutions to this problem; the former is to take `var` as an open set:

```
Parameter var: Set.
```

In this way the case constructor cannot be applied to a variable of type `var`, automatically ruling out exotic terms[5]. On the other hand, if we need to define `var` inductively, it is necessary to rule out exotic terms "manually" by means of a "validity" judgment holding only for `Case`-free terms [DFH95]. Then all the theorems developed in the framework must be decorated with these validity judgments.

### 3.1.3 Alternatives to HOAS

In this section we will briefly illustrate some alternatives to the HOAS encoding approach. Indeed, HOAS is well suited for representing formal systems with binders under the implicit condition that binding operators of the object language behave similarly to the binder of the metalanguage. If this is not the case one can try to enforce eventual context-dependent conditions not expressible by means of the type system with some auxiliary judgments (this notion is explained in Section 3.2). Another solution is to drop the HOAS approach in favor of other encoding techniques.

In the latter case, a first possibility is to adopt a First-Order Abstract Syntax (FOAS) solution, where the type of every constructor is "flat", i.e., no functions can be taken as arguments, only plain terms. In the case that the object language has no binders, this is a satisfactory approach; for instance, the languages of natural numbers and (finite) lists of natural numbers are given by the following grammars:

$$\begin{array}{llcl} \mathit{nat} & n & ::= & 0 \mid Sn \\ \mathit{list} & l & ::= & \langle\rangle \mid n.l \end{array}$$

Their encodings are usually given as follows:

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

Inductive nat_list : Set :=
   empty : nat_list
 | cons  : nat -> nat_list -> nat_list.
```

However, in presence of binders a FOAS approach is very unsatisfactory; for example a first-order encoding of the untyped $\lambda$-calculus is the following:

[4]If there would be a term $M$ corresponding to `weird`, then we would have $(M\ i_0) \to_\beta i_0$, while $(M\ i_{n+1}) \to_\beta i_n$ (where $(i_n)_n$ is an enumeration of variables such that the $n$-th identifier $i_n$ is represented by the $n$-th natural number), but this is absurd in the theory of the $\lambda$-calculus.

[5]Exotic terms arise in presence of a higher-order constructor (like `lam`) abstracting over an inductive type. In this situation it is possible to apply the `Case` operator of the metalanguage over an abstract variable, yielding a head normal form which does not correspond to the encoding of any entity of the object language.

```
Inductive tm: Set :=
    is_var : var -> tm
  | app    : tm  -> tm -> tm
  | lam    : var -> tm -> tm.
```

As we anticipated above, types are "flattened" (i.e., no higher-order types are considered) with the consequence that the resulting encoding is too fine-grained. Indeed, even if $\boldsymbol{\lambda} x.x$ is $\alpha$-equivalent to $\boldsymbol{\lambda} y.y$, i.e, the terms can be identified, the respective encodings (`(lam x (is_var x))` and `(lam y (is_var y))`) are in general completely different. Thus, the burden of encoding $\alpha$-equivalence of terms and the relative metatheory is left to the user. The first case study in Chapter 6 is a clear witness of the complexity of such a formal development. Moreover, it is easy to imagine that in the case of a more complex language (e.g., the Ambient Calculus) the formal encoding and development of the metatheory of $\alpha$-equivalence becomes a daunting task, even for the most patient being living on Earth. Obviously, also capture-avoiding substitution must be implemented from scratch.

An alternative approach allowing to solve the abovementioned problem of $\alpha$-conversion is to adopt the so-called *de Bruijn notation*. Using de Bruijn indexes, variables simply disappear and consequently there is no more $\alpha$-conversion, since $\lambda$-terms become diadic trees of natural numbers. The corresponding encoding in `Coq` is the following [Hue92, DH94]:

```
Inductive tm : Set :=
    ref : nat -> tm
  | app : tm  -> tm -> tm
  | lam : tm  -> tm.
```

Terms of type `tm` are trees whose leaves are labeled by naturals and internal nodes are labeled either by `app` or by `lam`. Hence, the behaviour of the binder of the object language is rendered by manipulating natural numbers (indexes); hence, $\alpha$-conversion is automatically enforced by de Bruijn notation. However, substitution must be manually implemented and also the handling of indexes is left to the user. To give an idea of the work that must be accomplished in order to get things working, it is sufficient to cite [Hir97], where a formalization of a fragment of the polyadic $\pi$-calculus is carried out by means of de Bruijn indexes and 600 of 800 proved lemmata concern technical manipulations of such indexes. However, the worst consequence of using de Bruijn notation is that the encoding and the subsequent formal development of the metatheory are very hard to read and to translate back to the original object language.

## 3.2 Representing proofs

So far, we discussed the representation of syntactic categories, but the final goal of encoding an object language in a type theory based LF is to obtain a proof assistant helping the user in formally proving properties about the encoded system. Hence, the treatment of rules and proofs lies at the heart of a logical framework and, following [HHP93] the key notion is that of *judgment* (or assertion) stressed by Martin-Löf [ML85]. Indeed, logical systems can be viewed as calculi for building proofs of a collection of basic (or atomic) judgments[6]. Following this idea, the principle known as *judgments-as-types* (which can be viewed as the

[6] For instance, in the untyped $\lambda$-calculus there is the judgment of $\beta$-reduction relating a term containing some redexes with the corresponding reducts. Another example of an atomic judgment is the assertion that a formula is true in first-order logic.

metathoretic analogue of the propositions-as-types principle) represents basic judgments of the object language with suitable types and their proofs as $\lambda$-terms whose type corresponds to the judgment they prove.

The basic set of judgments can be extended to two higher-order forms introduced by Martin-Löf: the former is the *hypothetical* judgment representing a form of consequence (if $J_1$ and $J_2$ are judgments, then $J_1 \vdash J_2$ means that $J_2$ is provable under the assumption of $J_1$), while the second is the *schematic* judgment representing generality (if $C$ is a category of the language and $J(x)$ is a judgment involving a variable $x$ ranging over elements of $C$, then $\bigwedge_{x\in C} J(x)$ means that $J(x)$ is provable uniformly in $x$). A LF providing dependent types can easily render such higher-order forms as follows:

$$\overline{J_1 \vdash J_2} = \overline{J_1} \to \overline{J_2}$$

Thus the hypothetical judgment is rendered by means of the functional space constructor ($\to$); generality, on the other hand, is rendered by means of the dependent types constructor ($\Pi$):

$$\overline{\bigwedge_{x\in C} J(x)} = \Pi x{:}\overline{C}.\overline{J(x)}$$

where $\overline{C}$ is the type representing the syntactic category $C$.

We said above that the hypothetical judgment expresses a form of consequence. In traditional logic, usually there is only one form of basic judgment and the notion of consequence is intended between sets or multisets of basic judgments instances. Hypothetical judgments instead are more general since they allow one to consider notions of consequence involving different basic judgments. The classical example, taken from [HHP93], is to consider the basic judgments of S4, namely, $\phi$ *true* and $\phi$ *valid*; then it is possible to consider the "hybrid" consequence notion $\phi$ *valid* $\vdash \phi$ *true*. Indeed, the latter is neither an instance of the truth consequence relation (expressing consequence in each single world) nor of the validity consequence relation (expressing consequence in all worlds).

Since derivations of basic judgments in the object language are carried out by means of rule systems, the latter must also be represented in order to "mimick" proofs with "pencil and paper". This task is carried out by associating to each rule of the object language a constant of higher type taking as arguments the values of the parameters and the proofs of the premises. For instance $\beta$-reduction in one step ($\to_\beta$) is defined by means of the following rules:

$$\begin{array}{ll}
(\boldsymbol{\lambda} x.M)N \to_\beta M[N/x] & (\beta - \text{REDEX}) \\
\dfrac{M \to_\beta N}{ZM \to_\beta ZN} & (\beta - \text{APP1}) \\
\dfrac{M \to_\beta N}{MZ \to_\beta NZ} & (\beta - \text{APP2}) \\
\dfrac{M \to_\beta N}{\boldsymbol{\lambda} x.M \to_\beta \boldsymbol{\lambda} x.N} & (\beta - \text{LAM})
\end{array}$$

Assuming that the syntax has been encoded using a (full) HOAS-based approach (see Section 3.1.1), we add to the signature the constant $beta : tm \to tm \to *$ and the following ones, representing the previously introduced $\to_\beta$-rules:

$$beta\_REDEX \quad : \quad \Pi m{:}tm \to tm.\Pi n{:}tm.(beta\ (app\ (lam\ m)\ n)\ (m\ n))$$

$$\begin{array}{lcl} beta\_APP1 & : & \Pi m{:}tm.\Pi n{:}tm.\Pi z{:}tm.(beta\ m\ n) \to (beta\ (app\ z\ m)\ (app\ z\ n)) \\ beta\_APP2 & : & \Pi m{:}tm.\Pi n{:}tm.\Pi z{:}tm.(beta\ m\ n) \to (beta\ (app\ m\ z)\ (app\ n\ z)) \\ beta\_LAM & : & \Pi m{:}tm \to tm.\Pi n{:}tm \to tm. \\ & & (\Pi z{:}tm.(beta\ (m\ z)\ (n\ z))) \to (beta\ (lam\ m)\ (lam\ n)) \end{array}$$

It is worth noticing the rôle played by the schematic judgment $\Pi z{:}tm.(beta\ (m\ z)\ (n\ z))$ in $beta\_LAM$: since the type of $m$ is functional ($tm \to tm$) it is required that $m$ applied to a generic term $z$ of type $tm$ $\beta$-reduces to $n$ applied to the same $z$ in order to conclude that $(lam\ m)$ $\beta$-reduces to $(lam\ n)$ (in other words we need to "fill the hole" of $m$ and $n$ before applying $beta$ to them).

Obviously, in order to have a faithful encoding of untyped $\lambda$-calculus and one step $\beta$-reduction, it is necessary to ensure a correspondence between proofs "on paper" and formal proofs in the LF. There are two alternative possibilities, differing in the "strength" of the abovementioned correspondence:

1. we may require that proofs of basic judgments of the object language are strictly related to proofs in the LF, i.e, by canonical terms whose types correspond to those of the encoded basic judgments. We will illustrate this notion of correspondence by means of our running example of untyped $\lambda$-calculus and one step $\beta$-reduction. First of all, we introduce a language of *proof expressions* as follows:

   $$\Pi ::= \beta - \mathrm{REDEX}_{x,M,N} \mid \beta - \mathrm{APP1}_{M,N,Z}(\Pi) \mid \beta - \mathrm{APP2}_{M,N,Z}(\Pi) \mid \beta - \mathrm{LAM}_{x,M,N}(\Pi)$$

   In the preceding grammar $x$ is a binding occurrence: in $\beta - \mathrm{REDEX}_{x,M,N}$, occurrences of $x$ in $M$ are bound, while in $\beta - \mathrm{LAM}_{x,M,N}(\Pi)$ occurrences of $x$ are bound in $M$, $N$ and $\Pi$ (proof expressions differing only in their bound variables are identified). Since not all proof expressions are valid, we introduce the rules depicted in Figure 3.2 in order to infer judgments of the form $X \vdash \Pi{:}M \to_\beta N$ which is to be read as "$\Pi$ is a valid proof of $M \to_\beta N$ w.r.t. the proof environment $X$" (where $X$ is a finite set of variables such that $fv(M) \cup fv(N) \subseteq X$)[7]. Finally, we define the encoding $\epsilon^\Pi_X$ map in Figure 3.3 (where $(beta\ M\ N)_X$ denotes the set of canonical terms $t$ such that $\Gamma_X \vdash_\Sigma t{:}(beta\ M\ N)$) and we prove the following proposition:

   **Proposition 3.2 (Adequacy for proofs, I)** *For every $X \subset \mathcal{V}$, and $M, N \in \Lambda_X$, the encoding $\epsilon^\Pi_{X,M,N}$ is a bijection between valid proofs of $M \to_\beta N$ and canonical terms $t$ of type $beta(\epsilon^\Lambda_X(M), \epsilon^\Lambda_X(N))$ such that $\Gamma_X \vdash_\Sigma t : beta(\epsilon^\Lambda_X(M), \epsilon^\Lambda_X(N))$. Moreover, $\epsilon^\Pi_{X,M,N}$ is compositional in the sense that for $\Pi$ a proof of $M \to_\beta N$ with $m, N \in \Lambda_X \triangleq \{x_1, \ldots, x_n\}$ and $M_1, \ldots, M_n \in \Lambda_Y$ the following holds:*

   $$\begin{array}{ll} & \epsilon^\Pi_{Y,M\{M_1/x_1,\ldots,M_n/x_n\},N\{M_1/x_1,\ldots,M_n/x_n\}}(\Pi\{M_1/x_1,\ldots,M_n/x_n\}) \\ = & \epsilon^\Pi_{X,M,N}(\Pi)\left\{\epsilon^\Lambda_Y(M_1)/x_1,\ldots,\epsilon^\Lambda_Y(M_n)/x_n\right\} \end{array}$$

   *Proof.* A tedious, but straightforward induction on the structure of the valid proof expression $\Pi{:}M \to_\beta N$ allows to establish both that $\epsilon^\Pi_{X,M,N}(\Pi)$ is a canonical form of type $beta(\epsilon^\Lambda_X(M), \epsilon^\Pi_X(N))$ and that $\epsilon^\Pi_{X,M,N}$ is injective. Surjectivity is proved by

[7] In the following we will denote by $\Pi_{X,M,N}$ the set of proof expressions such that $X \vdash \Pi{:}M \to_\beta N$.

defining a decoding map $\delta^{\Pi}$ that is left inverse to $\epsilon^{\Pi}$:

$$\begin{aligned}
&\delta^{\Pi}_{X,M,N} : (beta\ M\ N)_X \longrightarrow \Pi_{X,\delta^{\Lambda}_X(M),\delta^{\Lambda}_X(N)}\\
&\delta^{\Pi}_{X,M,N}((beta_REDEX\ M\ N)) \triangleq \beta-\mathrm{REDEX}_{x,\delta^{\Lambda}_{X,x}(Mx),\delta^{\Lambda}_X(N)}\\
&\delta^{\Pi}_{X,(app\ Z\ M),(app\ Z\ N)}((beta_APP1\ M\ N\ Z\ t)) \triangleq\\
&\qquad \beta-\mathrm{APP1}_{\delta^{\Lambda}_X(M),\delta^{\Lambda}_X(N),\delta^{\Lambda}_X(Z)}(\delta^{\Pi}_{X,\delta^{\Lambda}_X(M),\delta^{\Lambda}_X(N)}(t))\\
&\delta^{\Pi}_{X,(app\ M\ Z),(app\ N\ Z)}((beta_APP2\ M\ N\ Z\ t)) \triangleq\\
&\qquad \beta-\mathrm{APP2}_{\delta^{\Lambda}_X(M),\delta^{\Lambda}_X(N),\delta^{\Lambda}_X(Z)}(\delta^{\Pi}_{X,\delta^{\Lambda}_X(M),\delta^{\Lambda}_X(N)}(t))\\
&\delta^{\Pi}_{X,(lam\ M),(lam\ N)}((beta_LAM\ M\ N\ t)) \triangleq\\
&\qquad \beta-\mathrm{LAM}_{x,\delta^{\Lambda}_{X,x}(Mx),\delta^{\Lambda}_{X,x}(Nx)}(\delta^{\Pi}_{\langle X,x\rangle,\delta^{\Lambda}_{X,x}(Mx),\delta^{\Lambda}_{X,x}(Nx)}(tx))
\end{aligned}$$

By inspection on the possible canonical forms and the signature so far introduced and by the definition of valid proof expressions, it follows that $\delta^{\Pi}$ is defined and total. A structural induction on $\Pi$ allows to prove both $\delta^{\Pi}_{X,\epsilon^{\Lambda}_X(M),\epsilon^{\Lambda}_X(N)}(\epsilon^{\Pi}_{X,M,N}(\Pi)) = \Pi$ and the compositionality property. □

2. More often, the strict correspondence previously exemplified is not required. Indeed, it may suffice to prove that for each proof of the object language there is a canonical proof-term of the corresponding type and vice versa. This amounts to the approach of *proof irrelevance*, i.e, we do not require to reflect faithfully each derivation of the object language in the LF, but only the existence of a *witness* for each derivation. For instance, in the case of untyped $\lambda$-calculus and one step $\beta$-reduction the adequacy is stated by the following proposition:

   **Proposition 3.3 (Adequacy for proofs, II)** *Let $X \subset \mathcal{V}$ finite, $M, N \in \Lambda_X$,*

   *(a) (Soundness) if there exists $t$ canonical such that $\Gamma_X \vdash_\Sigma t{:}(beta\ M\ N)$, then we have $\delta^{\Lambda}_X(M) \to_\beta \delta^{\Lambda}_X(N)$.*

   *(b) (Completeness) if $M \to_\beta N$, there is a canonical form $t$ such that $\Gamma_X \vdash_\Sigma t : (beta\ \epsilon^{\Lambda}_X(M)\ \epsilon^{\Lambda}_X(N))$.*

   *Proof.* The argument is an induction on the structure of the canonical form $t$ (Soundness), and a structural induction on the derivation of $M \to_\beta N$ (Completeness). □

   This approach is adopted in the case studies of Chapter 6 and in [HMS01b].

It is worth noticing that, even if the example proposed (encoding of one step $\beta$-reduction) has been carried out in a minimal setting (without exploiting inductive definitions) following the approach introduced in [HHP93], the same techniques can be adopted, e.g., for proving the adequacy of a `Coq` encoding taking advantage of the (co)inductive features provided by the system. For a more thorough discussion on the topics so far recalled in this chapter we refer the reader to [AHMP92, HHP93, Mic97].

## 3.3 Pragmatic remarks

It should be clear from the arguments discussed in previous sections that representing a formal system in a type theory based logical framework is not a trivial task and many issues

$$\frac{fv(M) \setminus \{x\}, \cup fv(N) \subseteq X \qquad x \notin X}{X \vdash \beta - \mathrm{REDEX}_{x,M,N}:(\boldsymbol{\lambda} x.M) \to M[N/x]} \quad (\text{V-}\beta\text{-REDEX})$$

$$\frac{X \vdash \Pi{:}M \to N \qquad fv(Z) \subseteq X}{X \vdash \beta - \mathrm{APP1}_{M,N,Z}(\Pi){:}ZM \to ZN} \quad (\text{V-}\beta\text{-APP1})$$

$$\frac{X \vdash \Pi{:}M \to N \qquad fv(Z) \subseteq X}{X \vdash \beta - \mathrm{APP2}_{M,N,Z}(\Pi){:}MZ \to NZ} \quad (\text{V-}\beta\text{-APP2})$$

$$\frac{X, x \vdash \Pi{:}M \to N \qquad x \notin X}{X \vdash \beta - \mathrm{LAM}_{x,M,N}(\Pi){:}\boldsymbol{\lambda} x.M \to \boldsymbol{\lambda} x.N} \quad (\text{V-}\beta\text{-LAM})$$

Figure 3.2: Valid proof expressions for one step $\beta$-reduction.

$$\begin{array}{rcl}
\epsilon^{\Pi}_{X,M,N} & : & \Pi_{X,M,N} \longrightarrow (beta\ \epsilon^{\Lambda}_{X}(M)\ \epsilon^{\Lambda}_{X}(N))_X \\
\epsilon^{\Pi}_{X,M,N}(\beta - \mathrm{REDEX}_{x,M,N}) & \triangleq & (beta_REDEX\ \lambda x{:}tm.\epsilon^{\Lambda}_{X,x}(Mx)\ \epsilon^{\Lambda}_{X}(N)) \\
\epsilon^{\Pi}_{X,ZM,ZN}(\beta - \mathrm{APP1}_{M,N,Z}(\Pi)) & \triangleq & (beta_APP1\ \epsilon^{\Lambda}_{X}(M)\ \epsilon^{\Lambda}_{X}(N)\ \epsilon^{\Lambda}_{X}(Z)\ \epsilon^{\Pi}_{X,M,N}(\Pi)) \\
\epsilon^{\Pi}_{X,MZ,NZ}(\beta - \mathrm{APP2}_{M,N,Z}(\Pi)) & \triangleq & (beta_APP2\ \epsilon^{\Lambda}_{X}(M)\ \epsilon^{\Lambda}_{X}(N)\ \epsilon^{\Lambda}_{X}(Z)\ \epsilon^{\Pi}_{X,M,N}(\Pi)) \\
\epsilon^{\Pi}_{X,\lambda x.M,\lambda x.N}(\beta - \mathrm{LAM}_{x,M,N}(\Pi)) & \triangleq & (beta_LAM\quad \lambda x{:}tm.\epsilon^{\Lambda}_{X,x}(M) \\
 & & \qquad\qquad \lambda x{:}tm.\epsilon^{\Lambda}_{X,x}(N)\epsilon^{\Pi}_{X,M,N}(\Pi))
\end{array}$$

Figure 3.3: Encoding map for proofs of one step $\beta$-reduction.

can arise depending both on the object language and on the chosen encoding approach. We have seen how HOAS allows to delegate the treatment of binders to the metalanguage of the chosen framework, freeing the user from an explicit encoding of the relative machinery.

In general, the problem of exploiting as much as possible of the framework's metalevel, in order to avoid reinventing the wheel whenever a new object language has to be encoded, is covered by several works in the literature. For instance, in [BGG+92] a new terminology is introduced in order to classify encoding approaches. More precisely, the expression *shallow embedding* is used to denote the use of the syntactic infrastructure of the metalanguage in order to represent the semantics of the object language[8], without explicitly encoding the syntax (in other words, the object language is simply "translated" into appropriate denotations of the logical framework). On the other hand, representing syntactic categories of the object language as a defined type is called a *deep embedding* approach. The dichotomy *deep vs. shallow* is recalled in [Mel95], where it is stated that shallow embeddings are "particularly well suited to reasoning about applications", but cannot be fruitfully used to carry out a formal development of metatheoretic properties of the object language. The reason is that a direct translation of a formal system in terms of constructs of the metalanguage does not allow one to reason or even make reference to the objects or properties which have been delegated.

While a HOAS-based encoding approach is to be considered a deep embedding (in the sense of [BGG+92]), there are some drawbacks that may induce one to classify it as a shallow embedding [RHB01]. Indeed, representing binders by means of functional constants and

[8] In the specific case of [BGG+92] the problem was to formalize hardware description languages in HOL (Higher-Order Logic).

identifying object level identifiers (variables or names) with metavariables of the metalanguage, have the unpleasant effect of making the delegated machinery involving the treatment of bound names unavailable during the proof development activity. Moreover, the lack of adequate induction/recursion principle on higher-order types in most logical frameworks makes practically impossible to formally derive all those properties of the object language which are usually carried out by means of structural induction over contexts (represented by functional terms in the logical framework). These are the main reasons inducing many users of type theory-based logical frameworks to prefer first-order encodings or de Bruijn indexes for metareasoning about encoded systems.

An attempt to overcome the abovementioned issues is proposed in [GM96], where two *layers* are introduced. In the first layer a first-order implementation of the $\lambda$-calculus is carried out together with a formalization of $\alpha$-conversion and substitution. Hence, on top of this layer, it is possible to use a HOAS-based encoding approach to represent any formal system without worrying about exotic terms and negative occurrences. Moreover, using the manual implementation of the $\lambda$-calculus as a metalanguage all the machinery delegated by a HOAS encoding is still available to the user during the proof development activity. However, it is clear that the implementation of the first layer is to be carried out by means of a first-order embedding (or, even worse, by means of de Bruijn indexes) with all the related difficulties and technicalities.

In [DPS96] an extension of the "traditional" metalanguage of type theory based logical frameworks is proposed in order to allow primitive recursive functional types. More precisely, since primitive recursion cannot be allowed on any term whose type is higher-order (otherwise paradoxes can arise), the authors impose some constraints on functional types on which primitive recursion can be safely defined. This is carried out by decomposing the primitive recursive function space, denoted by $A \Rightarrow B$, into a modal type operator and a parametric function space: $A \Rightarrow B = (\Box A) \rightarrow B$. This approach is inspired by linear logic where a similar decomposition of the intuitionistic implication $A \supset B$ is accomplished in terms of a modal operator and a linear function space $(!A) \multimap B$.

Another proposal for allowing the definition of recursive functions in presence of higher-order encodings is introduced in [Sch01], where a new type system for the Edinburgh LF (called $\mathcal{T}_\omega^+$) is presented. The main idea is to "weaken" the *closed world assumption*[9] which is implied by the positivity condition builtin in all logical frameworks based on inductive definitions. Indeed, the closed world assumption does not allow one to "traverse" $\lambda$-abstractions in recursive calls, while this is needed in presence of HOAS-encodings. Therefore, a new property, called the *regular world assumption*, is introduced, allowing to recursively define functions with open arguments depending on parameters. Essentially, the latter must conform to an a priori specified regular grammar, which rules out "dangerous" definitions and makes the whole construction work. The novelty of the approach, w.r.t. [DPS96], is that it supports dependent types and does not require to add new modalities to the metalanguage.

The goal of this thesis is to propose a "cheaper" (although effective) way to efficiently metareason about formal systems using a HOAS approach. In particular, we will focus on the formalization of *nominal calculi*, a class of object languages whose central notion is that of name and name binding. In the next chapter we will introduce the so-called *Theory of Contexts*, i.e., a set of axioms stating some fundamental properties of contexts over names which can be easily embedded in any logical framework supporting the introduction of new axioms. Using a half-HOAS encoding approach, those axioms allow one to handle contexts

[9] This property amounts to require that arguments to functions must be closed, i.e., functions cannot be defined on free parameters.

like it is usually done in proofs with "pencil and paper". For instance, we will see that they allow one to derive a higher-order induction principle on functional terms over names starting from the usual induction principle over plain terms automatically provided by `Coq`. The efficacy of the Theory of Contexts has already been verified in [HMS01b], where the metatheory of the $\pi$-calculus strong late bisimilarity has been formally proved in `Coq`. The same framework has been used in [Mic01a] in order to formally develop the metatheory of capture-avoiding substitution for a HOAS-based encoding of untyped $\lambda$-calculus. In Chapter 6 we will provide two more complex case studies in order to illustrate its applicability and flexibility.

# 4

# A Theory of Contexts

In this chapter, following [HMS01a], we introduce an *axiomatic* Theory of Contexts with the aim of providing the user with the capability of meta-reasoning over HOAS-based encodings of nominal languages. The main advantage of an axiomatic approach is the possibility to implement it in any type theory based logical framework supporting the introduction of new axioms (e.g. the system Coq [TCDT01]), without having to modify the framework itself.

Obviously, an utmost concern about the Theory of Contexts (like for any axiomatic theory) is its consistency. This problem will be solved in Chapter 5 with the construction of a functorial model, following the lines of the seminal work by Hofmann [Hof99].

The chapter is structured as follows: we start with an informal introduction to the properties of the Theory of Contexts (Section 4.1). Then, in Section 4.2 we introduce the class of languages we focus on, namely, the so-called *nominal calculi*. In order to give a sufficient degree of generality to our methodology of representing and reasoning about nominal calculi, we do not stick to a particular logical framework like, e.g., CIC or HOL, but we introduce the generic system $\Upsilon$ in Section 4.3. In the latter, we formalize the Theory of Contexts and other useful recursion/induction principles. Some discussion on the *caveats* needed in order to ensure the consistency of the Theory of Contexts is carried out in Section 4.4. Remarks on the design choices for $\Upsilon$, independence and expressiveness of the properties and principles introduced are gathered in Section 4.5. Finally, some pointers to the related work appear in Section 4.6.

## 4.1 The axioms

The Theory of Contexts we introduce focuses on structural properties of syntactic contexts over a set of variables and/or names. Hence, informally speaking, the objects of our study will be terms with "holes" which can be filled in by variables/names to become plain terms. In order to obtain a non-trivial theory, we require that the set of variables is such that for any term there always exists a fresh variable with respect to it[1] (i.e. a variable not occurring in the given term):

**unsaturation:** $\forall M.\exists x.x \notin fv(M)$.

[1] In the following we will denote by $\notin$ the non occurrence predicate; more precisely $x \notin M$ means that the name/variable $x$ does not occur free in the term $M$.

That is, we require that there is always a fresh name in any given syntactic context. The intuition behind this axiom is that terms are finite objects; hence, a single term cannot contain all the possible variables. Obviously, since we are interested in meta-reasoning over nominal languages, we want the equality over names to be decidable. This property can be stated formally as follows:

**LEM$_{\mathbf{var}}$:** $\forall x, y. x = y \vee x \neq y$,

In a classical context, LEM$_{\text{var}}$ is obviously an instance of the law of excluded middle; hence, it does not need to be explicitly stated. On the other hand, in an intuitionistic setting it represents the minimum classical flavour needed in order to allow a meta-theoretic reasoning about encodings of nominal languages.

Usually, the implementations of the most widely used Logical Frameworks do not provide any support for metareasoning over contexts, that is there are neither induction nor recursion principles over higher-order terms. Hence, it is practically impossible to carry out formal proofs by reasoning over the structure of contexts. This is precisely the problem referred by the axioms of $\beta$-expansion and extensionality of Leibniz's equality:

**$\beta$-expansion:** $\forall M, x. \exists N[\cdot]. x \notin fv(N[\cdot]) \wedge M = N[x]$,

**extensionality:** $\forall M[\cdot], N[\cdot], x. x \notin fv(M[\cdot]) \cup fv(N[\cdot]) \Rightarrow M[x] = N[x] \Rightarrow M[\cdot] = N[\cdot]$.

Intuitively, the axiom of $\beta$-expansion provides the capability of freely generating a new context $N[\cdot]$ starting from a plain term $M$ and a name $x$, whereas the axiom of extensionality allows to state the equality of two contexts when their applications to a fresh name are equal. Thus, we have a rather simple machinery allowing us to introduce and to reason about syntactical properties of contexts.

In order to give an intuitive idea of the expressive power of the Theory of Contexts, we can consider the case of a typical property of nominal calculi, namely, the possibility of freely replacing a name (variable) with a fresh one in a proof, still preserving the validity of the latter. Hence, if we consider, for example, the case of $\pi$-calculus, for a given predicate $R$ over processes we may want to prove $R(P[y])$ knowing that $R(P[x])$ holds (where neither $x$ nor $y$ occur free in the process contexts $P[\cdot]$ and $Q[\cdot]$). Obviously, a proof of this kind heavily relies upon syntactical arguments and is usually carried out by means of an induction on the depth of inference of the judgment $R(P[x])$. Since the definitions of predicates on processes are usually driven by their syntactic structure, this implies in turn that in a generic induction step we will know something about the structure of $P[x]$ and we must use this information in order to derive something about the structure of $P[y]$ and prove the current subgoal. Thus, let us suppose we know that $P[x] \triangleq \overline{x}u.Q|x(v).R$ holds[2] (where $x \neq u$ and $x \neq v$); then we can use the axiom of *$\beta$-expansion* in order to infer that there exist two contexts $Q'[\cdot]$ and $R'[\cdot]$ such that $x$ does not occur free in them and, moreover, $Q = Q'[x]$ and $R = R'[x]$. It follows, by means of extensionality, that $P[\cdot] = \overline{[\cdot]}u.Q'[\cdot]|[\cdot](v).R'[\cdot]$. Whence, our initial subgoal $R(P[y])$ can be rewritten as $R(\overline{y}u.Q'[y]|y(v).R'[y])$. At this point, it will be clear which rule of $R$ is applicable in order to reduce the subgoal to a structurally smaller one which can be solved by means of the inductive hypothesis.

[2]For the reader not familiar with the $\pi$-calculus, we recall that the expression $\overline{x}u.Q|x(v).R$ represents two processes, namely $\overline{x}u.Q$ and $x(v).R$, running in parallel. More precisely, the first one is trying to send a name $u$ on channel $x$, while the second is waiting (on the same channel) for a name which will replace $v$ in $R$.

It is worth noticing that the axioms of $\beta$-expansion and extensionality can be generalized to yield schemata of axioms holding for contexts of an arbitrary arity (i.e. with more than one "hole"[3]). This fact turns out to be rather useful in applications, as we will see in Chapter 6.

In the next sections we will give a rigorous treatment of the arguments informally introduced so far together with the guidelines to implement a HOAS-based encoding of a nominal calculus within the framework of the Theory of Contexts.

## 4.2 Nominal calculi

In the following we call *nominal calculi* those object logics which seize upon the notion of name[4]. Typical examples are processes algebras (e.g. the $\pi$-calculus [MPW92], the Ambient-calculus [CG98], the Spi-Calculus [AG99] etc.) because, as it is stated in [Mil93], the act of naming implicitly implies a notion of concurrency i.e. the coexistence of the *namer* and the *named*.

In this section we give a formal definition of the notion of *nominal calculus*, providing the terminology and the tools needed in order to accommodate a general treatment of a rather broad class of object languages.

The first kind of basic entities of a generic nominal calculus is represented by a set of names. They are the most primitive objects since they have no structure; we assume that they are infinitely many in order to be able to pick a new name whenever this is needed. Moreover, we require that it is always possible to decide whether two names are equal or not. Obviously, there can be many separate sets of names; more formally, we have the following definition:

**Definition 4.1** *A* names set $\upsilon$ *is an infinite enumerable set of objects with a decidable equality.*

We will denote names sets with $\upsilon$, possibly with indexes like in $\upsilon_i$ or primes like in $\upsilon'$. Elements of names sets are ranged over by $n, m, \ldots$ or $x, y, z, \ldots$.

**Definition 4.2** *A* names base *is a finite set* $V = \{\upsilon_1, \ldots, \upsilon_k\}$ *of names sets. A* stage *over* $V = \{\upsilon_1, \ldots, \upsilon_k\}$ *is a collection* $X = \{X_1, \ldots, X_k\}$ *such that* $X_i \subset \upsilon_i$ *finite for* $i = 1, \ldots, k$.

Before introducing basic types, we need to define some constraints on the kind of constructors allowed for them: this is carried out by means of a notion of *nominal arity.*

**Definition 4.3** *Given a names base* $V \triangleq \{\upsilon_1, \ldots, \upsilon_k\}$ *and a collection of type symbols* $T \triangleq \{\iota_1, \ldots, \iota_l\}$*, each nominal arity* $\alpha$ *over* $V$ *and* $T$ *has the form* $\tau_1 \times \ldots \times \tau_n \to \iota$*, where* $\iota \in T$, $n \geq 0$ *(if* $n = 0$*, then* $\tau_1 \times \ldots \times \tau_n \to \iota = \iota$*) and, for* $1 \leq j \leq n$*, either* $\tau_j \in V$ *or* $\tau_j = \upsilon_{j_1} \times \ldots \times \upsilon_{j_{n_j}} \to \sigma_j$ *where* $\upsilon_{j_h} \in V$ $(1 \leq h \leq n_j)$*,* $n_j \geq 0$ *(if* $n_j = 0$*, then* $\upsilon_{j_1} \times \ldots \times \upsilon_{j_{n_j}} \to \sigma_j = \sigma_j$*) and* $\sigma_j \in T$*. We call the symbol* $\iota$ *in* $\alpha = \tau_1 \times \ldots \times \tau_n \to \iota$ *the* ending type *of the nominal arity* $\alpha$*.*

[3]The notion of arity of a context will be made clear shortly. So far, it is important to keep in mind that $\overline{[\cdot]}u.Q'[\cdot]||[\cdot](v).R'[\cdot]$ is a context with four occurrences of *one* hole. When we speak of a context with "more than one hole" we mean something like $\overline{[\cdot]}u.Q'[\cdot]||[\bullet](v).R'[\cdot]$, where the occurrences denoted by $[\cdot]$ are different from the one denoted by $[\bullet]$; hence, we have a context with two holes. Indeed, in order to instantiate such a context it is necessary to supply two arguments: the former will fill the occurrences marked by $[\cdot]$, while the latter will fill the one marked by $[\bullet]$.

[4]For the sake of simplicity, we will always refer to the notion of "name", even if some object languages, which can be classified as nominal calculi, employ the term "variable". Indeed, it is clear that the different terminology denotes the same notion.

$$\frac{x_1 \in \upsilon_1, \ldots, x_n \in \upsilon_n}{\langle x_1, \ldots, x_n\rangle \in \upsilon_1 \times \cdots \times \upsilon_n} \quad \text{(n-TUPLE)}$$

$$\frac{\langle x_1, \ldots, x_n\rangle \in \upsilon_1 \times \cdots \times \upsilon_n, M \in \iota}{(x_1, \ldots, x_n).M \in \upsilon_1 \times \cdots \times \upsilon_n \to \iota} \quad \text{(n-CTXT)}$$

$$\frac{M_1 \in \tau_1, \ldots, M_n \in \tau_n}{\langle M_1, \ldots, M_n\rangle \in \tau_1 \times \cdots \times \tau_n} \quad \text{(TUPLE)}$$

$$\frac{-}{c^{\iota} \in \iota} \quad \text{(CONST)}$$

$$\frac{\langle M_1, \ldots, M_n\rangle \in \tau_1 \times \cdots \times \tau_n}{c^{\tau_1 \times \cdots \times \tau_n \to \iota}(M_1, \ldots, M_n) \in \iota} \quad \text{(APP)}$$

Figure 4.1: Term forming rules

**Definition 4.4** *If a nominal arity $\alpha \triangleq \tau_1 \times \ldots \times \tau_n \to \iota$ is such that, for some $j$, $\tau_j = \upsilon_{j_1} \times \ldots \times \upsilon_{j_{n_j}} \to \sigma_j$ and $n_j > 0$, then $\alpha$ is said to be a* binding arity.

Basic types (e.g. terms, processes etc.) are denoted by $\iota$, possibly with indexes like in $\iota_i$ or primes like in $\iota'$. Every basic type $\iota$ has some constructors associated with it specifying how to build legal terms of type $\iota$. These may take as arguments names, terms (belonging to $\iota$ or to another basic type) and contexts (i.e. abstractions over names) as specified by the following definition:

**Definition 4.5** *Given a names base $V \triangleq \{\upsilon_1, \ldots, \upsilon_k\}$, let $I$ be the set of* basic types *$\{\iota_1\langle c^{\alpha_{\iota_1,1}}_{\iota_1,1}, \ldots, c^{\alpha_{\iota_1,m_1}}_{\iota_1,m_1}\rangle, \ldots, \iota_l\langle c^{\alpha_{\iota_l,1}}_{\iota_l,1}, \ldots, c^{\alpha_{\iota_l,m_l}}_{\iota_l,m_l}\rangle\}$ where each element*[5] *consists of a type name $\iota_i$ and a list of constructors $\langle c^{\alpha_{\iota_i,1}}_{\iota_i,1}, \ldots, c^{\alpha_{\iota_i,m_i}}_{\iota_i,m_i}\rangle$ such that for every $c^{\alpha_{\iota_i,j}}_{\iota_i,j}$ the corresponding nominal arity $\alpha_{\iota_i,j}$ (over $V$ and $\{\iota_1, \ldots, \iota_l\}$) has ending type $\iota_i$.*

*We denote the set of constructors of a basic type $\iota \in I$ by $Constr(\iota) = \{c^{\alpha_1}_1, \ldots, c^{\alpha_m}_m\}$; moreover, if $\alpha$ is a binding arity, then $c$ is said to be a binding constructor or simply a binder.*

Intuitively, constructors allow one to build terms of a basic type from other terms as specified by the corresponding nominal arity. More formally, given a names base $V$ and a set of basic types $I$, we have the term forming rules in Figure 4.1 (where $\upsilon_1, \ldots \upsilon_n \in V$, $\iota \in I$, $\tau$ and $\tau'$ are generic argument types as in the previous definitions and $n \geq 1$).

Contexts with $n$ holes ($n$-ary contexts) are denoted by $(x_1, \ldots, x_n).M$ and are derived by means of Rule (n-CTXT) starting from plain terms and lists of names to abstract on. Hence, if we consider the $n$-ary context $(x_1, \ldots, x_n).M$, the occurrences of names $x_1, \ldots x_n$ in $M$ are bound. Moreover, if $M$ is a $n$-ary context of the form $(y_1, \ldots, y_n).N$, $M[x_1, \ldots, x_n]$ denotes the substituted term $N[x_1/y_1, \ldots, x_n/y_n]$.

In nominal calculi names under the scope of a binder play a special rôle; indeed, they can be considered as placeholders or markers. It follows that the actual name used for the argument of a binder is not important as long it is different from all other names occurring in the syntactical context. Hence, terms of basic types can be taken up to $\alpha$-conversion, i.e., they can be considered syntactically equal if they differ only for bound names.

[5] We will often denote an element of $I$ by its name, omitting the list of constructors.

**Definition 4.6** *A nominal calculus $N$ is a triple $\langle V, I, L\rangle$ where $V$ is a names base, $I$ is a set of basic types and $L$ is the set of terms whose type belongs to $I$ derivable by the rules in Figure 4.1 up to $\alpha$-equivalence. Given any $\iota \in I$, we denote by $L^\iota$ the subset of $L$ of terms of type $\iota$. Moreover, if $X$ is a stage over $V$, we denote by $L_X$ (respectively, $L^\iota_X$) the subset of $L$ (respectively, $L^\iota_X$) of terms with free names in $X$.*

**Examples.** In order to make things clear, we give some examples of common object languages which can be seen as nominal calculi.

**Untyped $\lambda$-calculus.** $N_\lambda \triangleq \langle\{\upsilon\}, \{\Lambda\langle var^{\upsilon\to\Lambda}, app^{\Lambda\times\Lambda\to\Lambda}, lam^{(\upsilon\to\Lambda)\to\Lambda}\rangle\}\rangle$. The *lam* constructor is a binder.

**$\pi$-calculus.**

$$\begin{array}{rcll} N_\pi & \triangleq & \langle\{\eta\}, & \{P\langle 0^P, out^{\eta\times\eta\times P\to P}, in^{\eta\times(\eta\to P)\times P\to P}, \tau^{P\to P}, \\ & & & \quad \nu^{(\eta\to P)\to P}, !^{P\to P}, |^{P\times P\to P}, +^{P\times P\to P}, =^{\eta\times\eta\times P\to P} \\ & & & \rangle\} \\ & & \rangle & \end{array}$$

In this case we have two binders, namely, the input prefix ($in$) and the restriction operator ($\nu$).

**Ambient Calculus with Ambient Logic.**

$$\begin{array}{rcll} N_{Amb} & \triangleq & \langle\{\eta, \upsilon\}, & \{C\langle name^{\eta\to C}, in^{C\to C}, out^{C\to C}, \\ & & & \quad open^{C\to C}, \epsilon^C, path^{C\times C\to C} \\ & & & \rangle, \\ & & & P\langle \nu^{(\eta\to P)\to P}, 0^P, |^{P\times P\to P}, !^{P\to P}, amb^{C\times P\to P}, \\ & & & \quad cap^{C\times P\to P}, in_a^{(\eta\to P)\to P}, out_a^{C\to P} \\ & & & \rangle, \\ & & & F\langle \mathbf{T}^F, \neg^{F\to F}, \vee^{F\times F\to F}, \mathbf{0}^F, |^{F\times F\to F}, \\ & & & \quad \rhd^{F\times F\to F}, [\cdot]_\eta^{\eta\times F\to F}, @_\eta^{F\times\eta\to F}, \circledR_\eta^{\eta\times F\to F}, \\ & & & \quad \oslash_\eta^{F\times\eta\to F}, [\cdot]_\upsilon^{\upsilon\times F\to F}, @_\upsilon^{F\times\upsilon\to F}, \circledR_\upsilon^{\upsilon\times F\to F}, \\ & & & \quad \oslash_\upsilon^{F\times\upsilon\to F}, \Diamond^{F\to F}, \diamondsuit^{F\to F}, \forall^{(\upsilon\to F)\to F} \\ & & & \rangle\} \\ & & \rangle & \end{array}$$

This is an example of a nominal calculus where the names base consists of two distinct names sets: names ($\eta$) and variables ($\upsilon$). Then we have three basic types: capabilities ($C$), processes ($P$) and formulæ ($F$). While there are no binders for the capabilities type, processes have two binding constructors, i.e., input action ($in_a$) and restriction ($\nu$). Formulæ have only one binder, namely, the universal quantification ($\forall$) whose arity is $(\upsilon \to F) \to F$. Notice that, since in the Ambient Logic some constructors (namely, $[\cdot]$, $@$, $\circledR$ and $\oslash$) can take as arguments either a name or a variable, we are forced to specify two distinct versions (denoted by the $\eta$, $\upsilon$ subscripts).

## 4.3 The system $\Upsilon$

In order to give a general methodology for developing HOAS-based encodings of nominal calculi, we do not stick on a particular logical framework like, e.g., CIC or HOL, but we introduce the system $\Upsilon$. The latter represents the minimum type theory-based logical framework needed to encode and metareason about nominal calculi. Indeed, the underlying type theory is not strictly intrinsic, whence the machinery we are going to describe can be easily adapted to any sufficiently expressive logical framework. The advantage of keeping things simple is that we will not be overwhelmed by features inessential to our purposes, while all the focus will be on the basic mechanisms.

The system $\Upsilon$ is a theory of Simple Types/Higher Order Logic *à la* Church on a given signature $\Sigma$.

### 4.3.1 Syntax

In this section we give the definition of the basic syntactical notions of $\Upsilon$, i.e., signatures, simple types, terms and (typing) environments.

**Definition 4.7** *A* type signature $\Sigma_t$ *is a finite list of atomic type symbols* $\sigma_1, \ldots, \sigma_n$.

**Definition 4.8** Simple types *over a type signature* $\Sigma_t$ *are ranged over by* $\sigma$, $\tau$, *possibly with indexes or primes, and are defined by the following abstract syntax (where* $\sigma \in \Sigma_t$ *and* $o$ *is a distinct atomic symbol for the type of propositions):*

$$\tau ::= o \mid \sigma \mid \tau \to \tau$$

*We assume that there is a distinct countably infinite set of variables for each simple type.*

**Definition 4.9** *A* constant signature $\Sigma_c$ *is a finite list of constant symbols with simple types* $c{:}\tau_1, \ldots, c_m{:}\tau_m$

**Definition 4.10** *A* signature *consists of a pair* $\Sigma = \langle \Sigma_t, \Sigma_c \rangle$, *where* $\Sigma_t$ *is a type signature and* $\Sigma_c$ *is a constant signature.*

**Definition 4.11** *Terms over a signature* $\Sigma = \langle \Sigma_t, \Sigma_c \rangle$ *are ranged over by* $M$, $N$, $P$, $Q$, $R$ *and* $p$, $q$, $r$ *in the case of propositions (possibly with indexes or primes); they are defined by the following abstract syntax (where* $c{:}\sigma \in \Sigma_c$*):*

$$M ::= x \mid MN \mid \lambda x{:}\tau.M \mid c \mid M \Rightarrow N \mid \forall_\tau.M.$$

*We assume that there is an infinite set of variables ranged over by* $x, y, z, \ldots$. *Terms are taken up to* $\alpha$*-equivalence, while capture-avoiding substitution of* $N$ *for* $x$ *in* $M$ *is denoted by* $M[N/x]$.

Notice that, for what concerns the logical connectives, we keep as primitive only implication and universal quantification; as usual in higher-order logic, the remaining ones can be defined as abbreviations as depicted in Figure 4.2 (obviously, they make sense because we are in a classical framework rather than in an intuitionistic one).

**Definition 4.12** *A* (typing) environment $\Gamma$ *is a finite set of typing assertions over distinct variables denoted by* $\{x_1{:}\sigma_1, x_2{:}\sigma_2, \ldots, x_n{:}\sigma_n\}$, *possibly without curly brackets. The* domain *of an environment* $x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n$ *is the set of variables* $\{x_1, \ldots, x_n\}$.

$$
\begin{array}{rcl}
\forall x{:}\sigma.p & \triangleq & \forall_\sigma(\lambda x{:}\sigma.p) \\
p \wedge q & \triangleq & \neg(p \Rightarrow \neg q) \\
\bot & \triangleq & \forall x{:}o.x \\
p \vee q & \triangleq & \neg p \Rightarrow q \\
\neg p & \triangleq & p \Rightarrow \bot \\
p \Leftrightarrow q & \triangleq & (p \Rightarrow q) \wedge (q \Rightarrow p) \\
\exists x{:}\sigma.p & \triangleq & \neg\forall x{:}\sigma.\neg p \\
M =^\sigma N & \triangleq & \forall x{:}\sigma \to o.\ xM \Rightarrow xN
\end{array}
$$

Figure 4.2: Syntactic abbreviations.

$$
\frac{-}{\Gamma, x:\sigma \vdash_\Sigma x:\sigma}\ \text{(VAR)} \qquad \frac{\Gamma, x:\sigma' \vdash_\Sigma M:\sigma}{\Gamma \vdash_\Sigma \lambda x{:}\sigma'.M : \sigma' \to \sigma}\ \text{(ABS)}
$$

$$
\frac{-}{\Gamma \vdash_\Sigma c:\sigma}(c:\sigma) \in \Sigma\ \text{(CONST)} \qquad \frac{\Gamma \vdash_\Sigma M:\sigma' \to \sigma \quad \Gamma \vdash_\Sigma N:\sigma'}{\Gamma \vdash_\Sigma MN:\sigma}\ \text{(APP)}
$$

$$
\frac{\Gamma \vdash_\Sigma M:\sigma \to o}{\Gamma \vdash_\Sigma \forall_\sigma.M:o}\ (\forall) \qquad \frac{\Gamma \vdash_\Sigma M:o \quad \Gamma \vdash_\Sigma N:o}{\Gamma \vdash_\Sigma M \Rightarrow N:o}\ (\Rightarrow)
$$

Figure 4.3: Typing rules.

### 4.3.2 Judgments

In order to define the set of *legal* terms of $\Upsilon$, we need a typing judgment of the form $\Gamma \vdash_\Sigma M{:}\tau$ stating that we can derive the term $M$ of type $\tau$ starting from the environment $\Gamma$ and the signature $\Sigma$ using the rules in Figure 4.3.

Since $\Upsilon$ is a full blown higher-order logic, we need a specific *truth judgment* $\Gamma \vdash_\Sigma p$ in order to express the fact that a proposition $p$ holds in the environment $\Gamma$ and signature $\Sigma$. The rules for deriving truth judgments are given in Hilbert-style and are depicted in Figure 4.4.

### 4.3.3 HOAS-Encodings and Adequacy

In this section we will show how HOAS-based encodings of nominal calculi are rendered in the system $\Upsilon$. First of all we recall again the general guidelines of the higher-order abstract syntax paradigm (see [HHP93]):

- a basic $\Upsilon$-type is associated to each syntactic category;
- object level names are represented by metalanguages variables;
- a constant is declared for each expression-forming construct;
- contexts are represented by functions;
- name-binding operators are represented by constants of functional type (i.e. binders take contexts as arguments);
- contexts instantiation and capture-avoiding substitution are rendered by meta-level application of the underlying typed $\lambda$-calculus of the framework;
- as a consequence $\alpha$-conversion is automatically granted by the metalanguage.

$$\frac{\Gamma \vdash_\Sigma p : o \quad \Gamma \vdash_\Sigma q : o \quad \Gamma \vdash_\Sigma r : o}{\Gamma \vdash_\Sigma (p \Rightarrow q \Rightarrow r) \Rightarrow (p \Rightarrow q) \Rightarrow p \Rightarrow r} \text{(S)} \qquad \frac{\Gamma, x : \sigma \vdash_\Sigma M : \sigma' \quad \Gamma \vdash_\Sigma N : \sigma}{\Gamma \vdash_\Sigma (\lambda x{:}\sigma.M)N =^{\sigma'} M[N/x]} (\beta)$$

$$\frac{\Gamma \vdash_\Sigma p : o \quad \Gamma \vdash_\Sigma q : o}{\Gamma \vdash_\Sigma p \Rightarrow q \Rightarrow p} \text{(K)} \qquad \frac{\Gamma \vdash_\Sigma M : \sigma \to \sigma'}{\Gamma \vdash_\Sigma \lambda x{:}\sigma.Mx =^{\sigma \to \sigma'} M} x \notin FV(M) \quad (\eta)$$

$$\frac{\Gamma \vdash_\Sigma P : \sigma \to o \quad \Gamma \vdash_\Sigma M : \sigma}{\Gamma \vdash_\Sigma \forall_\sigma(P) \Rightarrow PM} (\forall\text{-E}) \qquad \frac{\Gamma, x : \sigma \vdash_\Sigma M : \sigma' \quad \Gamma, x : \sigma \vdash_\Sigma N : \sigma'}{\Gamma \vdash_\Sigma (\forall x{:}\sigma.M =^{\sigma'} N) \Rightarrow \lambda x{:}\sigma.M =^{\sigma \to \sigma'} \lambda x{:}\sigma.N} (\xi)$$

$$\frac{\Gamma \vdash_\Sigma p : o}{\Gamma \vdash_\Sigma \neg\neg p \Rightarrow p} \text{(DN)} \qquad \frac{\Gamma \vdash_\Sigma p : o \quad \Gamma, x : \sigma \vdash_\Sigma p \Rightarrow q}{\Gamma \vdash_\Sigma p \Rightarrow \forall x{:}\sigma.q} \text{(Gen)}$$

$$\frac{\Gamma \vdash_\Sigma p \Rightarrow q \quad \Gamma \vdash_\Sigma p}{\Gamma \vdash_\Sigma q} \text{(MP)}$$

Figure 4.4: Logical axioms and rules.

As we noticed in the previous chapter, it is clear that the most pleasant feature of HOAS-based encodings is that the user is freed from the daunting task of implementing by hand the mechanisms of $\alpha$-conversion and capture-avoiding substitution for each object logic.

Let $N \triangleq \langle V, I, L\rangle$ be a nominal calculus, then, to encode it in $\Upsilon$, we must provide the following items:

- one basic $\Upsilon$-type for each element of the names base $V \triangleq \{\upsilon_1, \ldots, \upsilon_k\}$; for the sake of simplicity we will denote by $\upsilon_i$ both the names set and the type representing it;
- one basic $\Upsilon$-type for each element of $I \triangleq \{\iota_1\langle c_{\iota_1,1}^{\alpha_{\iota_1,1}}, \ldots, c_{\iota_1,m_1}^{\alpha_{\iota_1,m_1}}\rangle, \ldots, \iota_l\langle c_{\iota_l,1}^{\alpha_{\iota_l,1}}, \ldots, c_{\iota_l,m_l}^{\alpha_{\iota_l,m_l}}\rangle\}$; for the sake of simplicity we will denote by $\iota_i$ both the basic type name of the object language and the basic $\Upsilon$-type representing it;
- one typed constant $c{:}Curry(\alpha)$ for each constructor $c^\alpha \in Constr(\iota)$ (where $\iota \in I$ and $Curry(\alpha)$ is the simple type of $\Upsilon$ obtained by currifying[6] the arity $\alpha$).

More precisely, we have the following definition:

**Definition 4.13** *Let $N \triangleq \langle V, I, L\rangle$ be a nominal calculus, we define the type signature $\Sigma_t(N)$ as the set of atomic type symbols $\{\upsilon \mid \upsilon \in V\} \cup \{\iota \mid \iota \in I\}$. Then the constant signature $\Sigma_c(N)$ over $N$ is defined as the set $\{c{:}Curry(\alpha) \mid c^\alpha \in Constr(\iota), \iota \in I\}$.*

Now, we are ready to introduce the encoding map $\epsilon$ which allows us to establish a compositional bijective correspondence between syntactical entities of a nominal calculus $N$ and $\beta\eta$-normal forms of basic $\Upsilon$-types in $\Sigma_t(N)$. This is a fundamental point that every encoding map must ensure in order to faithfully represent the chosen object language.

**Definition 4.14** *Given a nominal calculus $N \triangleq \langle V, I, L\rangle$ and a stage $X \triangleq \{X_1, \ldots, X_k\}$ over $V \triangleq \{\upsilon_1, \ldots, \upsilon_k\}$, for every $\upsilon_i \in V$, let $(n_j)_j$ be an enumeration of $\upsilon_i$ and $(x_j)_j$ be an enumeration of variables of type $\upsilon_i$ in $\Upsilon$. Then, the function $\epsilon_X^{\upsilon_i}$, mapping names in $X_i$ to variables of type $\upsilon_i$ in $\Upsilon$, is defined by $\epsilon_X^{\upsilon_i}(n_j) \triangleq x_j$ for $n_j \in X_i$*[7].

[6] The function $Curry$ is needed because the only constructor of the simple types of $\Upsilon$ is the arrow. Hence, we must encode type pairs by currifying arities containing them. Indeed, since a nominal arity $\alpha$ has the shape $\tau_1 \times \cdots \times \tau_n \to \iota$, we have $Curry(\alpha) = Curry(\tau_1) \to \cdots \to Curry(\tau_n) \to \iota$, where each $\tau_i$ being equal to $\upsilon_{i1} \times \cdots \times \upsilon_{im_i} \to \sigma_i$ may need to be currified as well.

[7] For the sake of simplicity we will write $\epsilon_X^\upsilon(x) \triangleq x$, i.e., we will denote by the same symbol both the object level name and the $\Upsilon$-variable of type $\upsilon$ encoding it.

*For every $\iota \in I$, the definition of the encoding map $\epsilon^\iota_X$ (mapping elements of $L^\iota_X$ to terms in $\Upsilon$ of type $\iota$) depends on the constructors of $\iota$. In particular, for each $c^\alpha \in Constr(\iota)$, we can distinguish the following cases according to the shape of $\alpha$:*

$\alpha = \iota$**:** $\epsilon^\iota_X(c^\iota) \triangleq c$,

$\alpha = \tau_1 \times \cdots \times \tau_n \to \iota$ $(n \geq 1)$**:** $\epsilon^\iota_X(c^{\tau_1\times\cdots\times\tau_n\to\iota}(t_1,\ldots,t_n)) \triangleq (c\ \epsilon^{\tau_1}_X(t_1)\ \cdots\ \epsilon^{\tau_n}_X(t_n))$,

*where $\epsilon^{\tau_i}_X$ is defined as follows, according to the shape of $\tau_i$:*

$\tau_i = \sigma_i$ $(\sigma_i \in V \cup I)$**:** $\epsilon^{\tau_i}_X(t_i) \triangleq \epsilon^{\sigma_i}_X(t_i)$,

$\tau_i = \upsilon_{i1} \times \cdots \times \upsilon_{im_i} \to \sigma_i$ $(m_i \geq 1)$**:**

$$\epsilon^{\upsilon_{i1}\times\cdots\times\upsilon_{im_i}\to\sigma_i}_X((x_{i1},\ldots,x_{im_i}).t_i) \triangleq \lambda x_{i1}:\upsilon_{i1}.\ldots\lambda x_{im_i}:\upsilon_{im_i}.\epsilon^{\sigma_i}_{Y^i}(t_i)$$

*where $Y^i_l \triangleq X_l \cup \{x_{ij} \mid \upsilon_{ij} = \upsilon_l, x_{ij} \in \upsilon_l \setminus X_l, j = 1,\ldots,m_i\}$ and $\sigma_i \in I$.*

As anticipated, we have the following result:

**Theorem 4.1** *Let $N \triangleq \langle V, I, L\rangle$ be a nominal calculus, let $X$ be a stage over $V$, let $\Sigma(N) \triangleq \langle \Sigma_t, \Sigma_c\rangle$ be the signature encoding $N$ and let $\Gamma(X) \triangleq \{x : \upsilon_i \mid x \in X_i, i = 1\ldots n\}$. For each type $\iota \in I$, the map $\epsilon^\iota_X$ is a compositional*[8] *bijection between $L^\iota_X$ and the set of terms $\{M \mid M$ is in $\beta\eta$-normal form and $\Gamma(X) \vdash_{\Sigma(N)} M : \iota\}$.*

*Proof.* (Sketch) By definition, $\epsilon^\iota_X$ is an injective function mapping every element of $L^\iota_X$ to a $\beta\eta$-normal form of type $\iota$. In order to prove that $\epsilon^\iota_X$ is bijective, it remains to show its surjectivity. This can be accomplished by defining an inverse map $\delta^\iota_X$ by recursion on the structure of $\beta\eta$-normal forms of type $\iota$:

$$\begin{array}{lcl}
\delta^\iota_X(c) & \triangleq & c^\iota \quad \text{if } \Gamma(X) \vdash_{\Sigma(N)} c : \iota \\
\delta^\iota_X((c\ M_1\ \cdots\ M_n)) & \triangleq & c^{\tau_1\times\cdots\times\tau_n\to\iota}(\delta^{Curry(\tau_1)}_X(M_1),\ldots,\delta^{Curry(\tau_n)}_X(M_n)) \\
& & \text{if } n \geq 1,\ \Gamma(X) \vdash_{\Sigma(N)} c : Curry(\tau_1\times\cdots\times\tau_n\to\iota) \text{ and} \\
& & \Gamma(X) \vdash_{\Sigma(N)} M_1 : Curry(\tau_1),\ldots,\Gamma(X) \vdash_{\Sigma(N)} M_n : Curry(\tau_n)
\end{array}$$

where $\delta^{Curry(\tau_i)}_X$ is defined as follows, according to the shape of $\tau_i$:

$\tau_i = \sigma_i$ $(\sigma_i \in V \cup I)$**:** $\delta^{\tau_i}_X(M_i) \triangleq \delta^{\sigma_i}_X(M_i)$ (in particular if $\sigma_i = \upsilon_i \in V$, then $M_i$ is a variable $x$ and $\delta^{\upsilon_i}_X(x) \triangleq x$),

$\tau_i = \upsilon_{i1} \times \cdots \times \upsilon_{im_i} \to \sigma_i$ $(m_i \geq 1)$**:**

$$\delta^{\upsilon_{i1}\to\cdots\to\upsilon_{im_i}\to\sigma_i}_X(\lambda x_{i1}:\upsilon_{i1}.\ldots\lambda x_{im_i}:\upsilon_{im_i}.M) \triangleq (x_{i1},\ldots,x_{im_i}).\delta^{\sigma_i}_{Y^i}(M)$$

where $Y^i_l \triangleq X_l \cup \{x_{ij} \mid \upsilon_{ij} = \upsilon_l, x_{ij} \in \upsilon_l \setminus X_l, j = 1,\ldots,m_i\}$ and $\sigma_i \in I$.

Clearly, a $\beta\eta$-normal form of type $\iota$ derivable from the context $\Gamma(X)$ and the signature $\Sigma(N)$ must have the form $(\xi\ M_1 \cdots M_k)$, where $\xi$ is a constant and $k$ is its arity. Hence, it follows that $\delta^\iota_X$ is total and well defined.

The fact that $\delta^\iota_X(\epsilon^\iota_X(t))$ holds can be proved by structural induction on $t$. The same technique can also be used to show that $\epsilon^\iota_X$ is compositional. □

[8]We recall that compositionality means that if $t$ is a term with free names in $X = \{x_1,\ldots,x_n\}$, then for any other stage $Y = \{y_1,\ldots,y_n\}$ we have that $\epsilon^\iota_Y(t[y_1/x_1,\ldots,y_n/x_n]) = \epsilon^\iota_X(t)[y_1/x_1,\ldots,y_n/x_n]$.

**Examples.** Some HOAS-based encodings in $\Upsilon$ of the nominal calculi used in the examples of Section 4.2 are given here in order to make things clear:

**Untyped $\lambda$-calculus.** $\Sigma_t(N_\lambda) \triangleq \{var, term\}$, $\Sigma_c(N_\lambda) \triangleq \{is\_var : var \to term,\ app : term \to term \to term,\ lam : (var \to term) \to term\}$.

$$\begin{array}{lcl}
\epsilon^v_X(x) & \triangleq & x \\
\epsilon^\Lambda_X(var(x)) & \triangleq & (is_var\ \epsilon^v_X(x)) \\
\epsilon^\Lambda_X(app(M,N)) & \triangleq & (app\ \epsilon^\Lambda_X(M)\ \epsilon^\Lambda_X(N)) \\
\epsilon^\Lambda_X(lam((x).M)) & \triangleq & (lam\ \lambda x{:}var.\epsilon^\Lambda_{X\uplus\{x\}}(M))
\end{array}$$

**$\pi$-calculus.** $\Sigma_t(N_\pi) \triangleq \{name, proc\}$,

$$\begin{array}{lcllcl}
\Sigma_c(N_\pi) & \triangleq & \{ & nil & : & proc, \\
 & & & out & : & name \to name \to proc \to proc, \\
 & & & in & : & name \to (name \to proc) \to proc, \\
 & & & tau & : & proc \to proc, \\
 & & & nu & : & (name \to proc) \to proc, \\
 & & & bang & : & proc \to proc, \\
 & & & par & : & proc \to proc \to proc, \\
 & & & sum & : & proc \to proc \to proc, \\
 & & & match & : & name \to name \to proc \to proc \\
 & & \} & & &
\end{array}$$

$$\begin{array}{lcl}
\epsilon^\eta_X(n) & \triangleq & n \\
\epsilon^P_X(0) & \triangleq & nil \\
\epsilon^P_X(out(m,n,P)) & \triangleq & (out\ \epsilon^\eta_X(m)\ \epsilon^\eta_X(n)\ \epsilon^P_X(P)) \\
\epsilon^P_X(in(m,(n).P)) & \triangleq & (in\ \epsilon^\eta_X(m)\ \lambda n{:}\eta.\epsilon^P_{X\uplus\{n\}}(P)) \\
\epsilon^P_X(\tau(P)) & \triangleq & (tau\ \epsilon^P_X(P)) \\
\epsilon^P_X(\nu((n).P)) & \triangleq & (nu\ \lambda n{:}\eta.\epsilon^P_{X\uplus\{n\}}(P)) \\
\epsilon^P_X(!(P)) & \triangleq & (bang\ \epsilon^P_X(P)) \\
\epsilon^P_X(P|Q) & \triangleq & (par\ \epsilon^P_X(P)\ \epsilon^P_X(Q)) \\
\epsilon^P_X(P+Q) & \triangleq & (sum\ \epsilon^P_X(P)\ \epsilon^P_X(Q)) \\
\epsilon^P_X([m=n]P) & \triangleq & (match\ \epsilon^\eta_X(m)\ \epsilon^\eta_X(n)\ \epsilon^P_X(P))
\end{array}$$

**Ambient calculus with ambient logic.** $\Sigma_t(N_{Amb}) \triangleq \{name, var, cap, proc, form\}$,

$$\begin{array}{lcllcl}
\Sigma_c(N_{Amb}) & \triangleq & \{ & is_name & : & name \to cap, \\
 & & & in & : & cap \to cap, \\
 & & & out & : & cap \to cap, \\
 & & & open & : & cap \to cap, \\
 & & & epsilon & : & cap, \\
 & & & path & : & cap \to cap \to cap, \\
 & & & nu & : & (name \to proc) \to proc, \\
 & & & nil & : & proc,
\end{array}$$

$$
\begin{array}{lcl}
\quad par & : & proc \to proc \to proc, \\
\quad bang & : & proc \to proc, \\
\quad amb & : & cap \to proc \to proc, \\
\quad cap\_act & : & cap \to proc \to proc, \\
\quad in\_act & : & (name \to proc) \to proc, \\
\quad out\_act & : & cap \to proc, \\
\quad true & : & form, \\
\quad not & : & form \to form, \\
\quad or & : & form \to form \to form, \\
\quad zero & : & form, \\
\quad comp & : & form \to form \to form, \\
\quad comp\_adj & : & form \to form \to form, \\
\quad loc_n & : & name \to form \to form, \\
\quad loc\_adj_n & : & form \to name \to form, \\
\quad rev_n & : & name \to form \to form, \\
\quad rev\_adj_n & : & form \to name \to form, \\
\quad loc_v & : & var \to form \to form, \\
\quad loc\_adj_v & : & form \to var \to form, \\
\quad rev_v & : & var \to form \to form, \\
\quad rev\_adj_v & : & form \to var \to form, \\
\quad sometime & : & form \to form, \\
\quad somewhere & : & form \to form, \\
\quad forall & : & (var \to form) \to form \\
\}
\end{array}
$$

$$
\begin{array}{lcl}
\epsilon^{\eta}_{X}(n) & \triangleq & n \\
\epsilon^{v}_{X}(x) & \triangleq & x \\
\\
\epsilon^{C}_{X}(name(n)) & \triangleq & (is\_name\ \epsilon^{\eta}_{X}(n)) \\
\epsilon^{C}_{X}(in(M)) & \triangleq & (in\ \epsilon^{C}_{X}(M)) \\
\epsilon^{C}_{X}(out(M)) & \triangleq & (out\ \epsilon^{C}_{X}(M)) \\
\epsilon^{C}_{X}(open(M)) & \triangleq & (open\ \epsilon^{C}_{X}(M)) \\
\epsilon^{C}_{X}(\epsilon) & \triangleq & epsilon \\
\epsilon^{C}_{X}(path(M,N)) & \triangleq & (path\ \epsilon^{C}_{X}(M)\ \epsilon^{C}_{X}(N)) \\
\\
\epsilon^{P}_{X}(\nu((n).P)) & \triangleq & (nu\ \lambda n{:}\eta.\epsilon^{P}_{X\uplus\{n\}}(P)) \\
\epsilon^{P}_{X}(0) & \triangleq & nil \\
\epsilon^{P}_{X}(P|Q) & \triangleq & (par\ \epsilon^{P}_{X}(P)\ \epsilon^{P}_{X}(Q)) \\
\epsilon^{P}_{X}(!(P)) & \triangleq & (bang\ \epsilon^{P}_{X}(P)) \\
\epsilon^{P}_{X}(amb(M,P)) & \triangleq & (amb\ \epsilon^{C}_{X}(M)\ \epsilon^{P}_{X}(P)) \\
\epsilon^{P}_{X}(cap(M,P)) & \triangleq & (cap\_act\ \epsilon^{C}_{X}(M)\ \epsilon^{P}_{X}(P)) \\
\epsilon^{P}_{X}(in_a((n).P)) & \triangleq & (in\_act\ \lambda n{:}\eta.\epsilon^{P}_{X\uplus\{n\}}(P)) \\
\epsilon^{P}_{X}(out_a(M,P)) & \triangleq & (out\_act\ \epsilon^{C}_{X}(M)\ \epsilon^{P}_{X}(P)) \\
\\
\epsilon^{F}_{X}(\mathbf{T}) & \triangleq & true \\
\epsilon^{F}_{X}(\neg(A)) & \triangleq & (not\ \epsilon^{F}_{X}(A)) \\
\epsilon^{F}_{X}(A \vee B) & \triangleq & (or\ \epsilon^{F}_{X}(A)\ \epsilon^{F}_{X}(B)) \\
\epsilon^{F}_{X}(\mathbf{0}) & \triangleq & zero
\end{array}
$$

$$
\begin{array}{lcl}
\epsilon^F_X(A|B) & \triangleq & (comp\ \epsilon^F_X(A)\ \epsilon^F_X(B)) \\
\epsilon^F_X(A \rhd B) & \triangleq & (comp\_adj\ \epsilon^F_X(A)\ \epsilon^F_X(B)) \\
\epsilon^F_X([n]_\eta A) & \triangleq & (loc_n\ \epsilon^\eta_X(n)\ \epsilon^F_X(A)) \\
\epsilon^F_X(A@_\eta n) & \triangleq & (loc\_adj_n\ \epsilon^F_X(A)\ \epsilon^\eta_X(n)) \\
\epsilon^F_X(n®_\eta A) & \triangleq & (rev_n\ \epsilon^\eta_X(n)\ \epsilon^F_X(A)) \\
\epsilon^F_X(A \oslash_\eta n) & \triangleq & (rev\_adj_n\ \epsilon^F_X(A)\ \epsilon^\eta_X(n)) \\
\epsilon^F_X([n]_\upsilon A) & \triangleq & (loc_\upsilon\ \epsilon^\upsilon_X(n)\ \epsilon^F_X(A)) \\
\epsilon^F_X(A@_\upsilon n) & \triangleq & (loc\_adj_\upsilon\ \epsilon^F_X(A)\ \epsilon^\upsilon_X(n)) \\
\epsilon^F_X(n®_\upsilon A) & \triangleq & (rev_\upsilon\ \epsilon^\upsilon_X(n)\ \epsilon^F_X(A)) \\
\epsilon^F_X(A \oslash_\upsilon n) & \triangleq & (rev\_adj_\upsilon\ \epsilon^F_X(A)\ \epsilon^\upsilon_X(n)) \\
\epsilon^F_X(\Diamond(A)) & \triangleq & (sometime\ \epsilon^F_X(A)) \\
\epsilon^F_X(\diamondsuit(A)) & \triangleq & (somewhere\ \epsilon^F_X(A)) \\
\epsilon^F_X(\forall((x).A)) & \triangleq & (forall\ \lambda x{:}\eta.\epsilon^F_{X \uplus \{x\}}(A))
\end{array}
$$

### 4.3.4 Logic

As we said in Section 4.3.2 the judgment $\Gamma \vdash_\Sigma p$ express the fact that proposition $p$ holds in the environment $\Gamma$ and signature $\Sigma$. Besides the logical axioms and rules depicted in Figure 4.4, there are also the axioms of the Theory of Contexts which grant to the system $\Upsilon$ a remarkable expressive power for meta-reasoning about properties of HOAS-based encodings. However, before introducing those axioms, we need to define a non-occurrence predicate $\notin^\iota_\upsilon$. Intuitively, the latter allows to express the fact that a given name/variable does not occur free in a given term; more precisely, $x \notin^\iota_\upsilon M$ holds if and only if the name/variable of type $\upsilon$ does not occur free in the term $M$ of type $\iota$. It is important to notice that we do not attempt to give a logical characterisation of the notion of non-occurrence that is independent of object-level syntax. Instead it should be clear that the formal definition of $\notin^\iota_\upsilon$ depends both on $\iota$ and $\upsilon$, i.e., on a nominal calculus $N$ which must be known before defining the non-occurrence predicate. More precisely, the definition is syntax-driven in the sense that it depends on the constructors of type $\iota$. Indeed, it is a customary approach in higher-order logic to define predicates by means of higher-order quantifications and monotone operators.

**Definition 4.15** *$x \notin^\iota_\upsilon M \triangleq \forall R{:}\upsilon \to \iota \to o.(\forall y{:}\upsilon.\forall N{:}\iota.(T_{\notin^\iota_\upsilon}\ R\ y\ N) \Rightarrow (R\ y\ N)) \Rightarrow (R\ x\ M)$, where $T_{\notin^\iota_\upsilon}$ is an operator defined as follows:*

$$T_{\notin^\iota_\upsilon} : (\upsilon \to \iota \to o) \to (\upsilon \to \iota \to o) \triangleq \lambda R{:}\upsilon \to \iota \to o.\lambda x{:}\upsilon.\lambda M{:}\iota.\bigvee_{i=1}^{|Constr(\iota)|} C_i,$$

*where each $C_i$ (for $i = 1, \ldots, |Constr(\iota)|$) is a clause corresponding to a constructor $c_i$ : $Curry(\alpha_i)$ belonging to the type $\iota$ as follows:*

$\alpha = \iota$**:** *$C_i \triangleq M =^\iota c$;*

$\alpha = \tau_1 \times \cdots \times \tau_k \to \iota$**:** *$C_i \triangleq (\exists N_1{:}\tau_1.\ldots.\exists N_k{:}\tau_k.M =^\iota (c\ N_1 \cdots N_k) \wedge \bigwedge_{j=1}^k H_j)$, where each $H_j$ depends on the shape of $\tau_j$:*

> $\tau_j = \upsilon$**:** *$H_j \triangleq \neg(x =^\upsilon N_j)$;*
>
> $\tau_j = \upsilon_{j1} \times \cdots \times \upsilon_{jm_j} \to \sigma_j$**:** *$H_j \triangleq (\forall y_1{:}\upsilon_{j1}.\ldots.\forall y_{m_j}{:}\upsilon_{jm_j}.\neg(x =^\upsilon y_{k_1}) \Rightarrow \cdots \neg(x =^\upsilon y_{k_j}) \Rightarrow (R\ x\ (N_j\ y_1 \cdots y_{m_j})))$, where the indexes $k_1, \ldots, k_j$ are precisely those belonging to the set $\{1, \ldots, m_j\}$ such that $\upsilon_{jk_l} = \upsilon$ for each $l = 1, \ldots, j$.*

$$\frac{\Gamma \vdash_{\Sigma(N)} M : \iota}{\Gamma \vdash_{\Sigma(N)} \exists x{:}\upsilon . x \not\in^{\iota}_{\upsilon} M} \quad (Unsat^{\upsilon}_{\iota})$$

$$\frac{\Gamma \vdash_{\Sigma(N)} M : \upsilon \to \tau \quad \Gamma \vdash_{\Sigma(N)} N : \upsilon \to \tau \quad \Gamma \vdash_{\Sigma(N)} x : \upsilon}{\Gamma \vdash_{\Sigma(N)} x \not\in^{\upsilon\to\tau}_{\upsilon} M \Rightarrow x \not\in^{\upsilon\to\tau}_{\upsilon} N \Rightarrow (M\ x) =^{\tau} (N\ x) \Rightarrow M =^{\upsilon\to\tau} N} \quad (Ext^{\upsilon}_{\tau})$$

$$\frac{\Gamma \vdash_{\Sigma(N)} M : \tau \quad \Gamma \vdash_{\Sigma(N)} x : \upsilon}{\Gamma \vdash_{\Sigma(N)} \exists N{:}\upsilon \to \tau . x \not\in^{\upsilon\to\tau}_{\upsilon} N \wedge M =^{\tau} (N\ x)} \quad (\beta\_exp^{\upsilon}_{\tau})$$

where $\tau = \upsilon_{i_1} \to \cdots \to \upsilon_{i_k} \to \iota$ and $\Sigma(N)$ means that the axioms are parameterized by the particular nominal calculus $N$ being considered.

Figure 4.5: Axiom schemata for the Theory of Contexts.

*Whence, we have the following definition:*

$$\not\in^{\iota}_{\upsilon} \triangleq \lambda x{:}\upsilon.\lambda M{:}\iota.\forall R{:}\upsilon \to \iota \to o.(\forall y{:}\upsilon.\forall N{:}\iota.(T_{\not\in^{\iota}_{\upsilon}}\ R\ y\ N) \Rightarrow (R\ y\ N)) \Rightarrow (R\ x\ M)$$

*It is trivial to verify that $T_{\not\in^{\iota}_{\upsilon}}$ is monotone (i.e. for all relations $R : \upsilon \to \iota \to o$, we have $T_{\not\in^{\iota}_{\upsilon}}(R) \subseteq R$, where $\subseteq$ can be defined as $\lambda R : \upsilon \to \iota \to o.\lambda S : \upsilon \to \iota \to o.\forall x : \upsilon.\forall M : \iota.(R\ x\ M) \Rightarrow (S\ x\ M)$), whence $\not\in^{\iota}_{\upsilon}$ is the least relation $R$ such that $T_{\not\in^{\iota}_{\upsilon}}(R) \subseteq R$ holds.*

**Example.** In the case of the encoding of untyped $\lambda$-calculus proposed in Section 4.3.3, the operator $T_{\not\in^{term}_{var}}$ is defined as follows:

$$\begin{aligned} T_{\not\in^{term}_{var}} &: (var \to term \to o) \to (var \to term \to o) \\ &\triangleq \lambda R{:}var \to term \to o.\lambda x{:}var.\lambda t{:}term. \\ &\quad (\exists y{:}var.t =^{term} (is\_var\ y) \wedge \neg(x =^{var} y)) \vee \\ &\quad (\exists t_1{:}term.\exists t_2{:}term.t = (app\ t_1\ t_2) \wedge (R\ x\ t_1) \wedge (R\ x\ t_2)) \vee \\ &\quad (\exists t'{:}var \to term.t = (lam\ t') \wedge (\forall y{:}var.\neg(x =^{\upsilon} y) \Rightarrow (R\ x\ (t'\ y)))) \end{aligned}$$

Given Definition 4.15, it is possible to define the binary predicate $\in^{\iota}_{\upsilon}$[9] either as the negation of $\not\in^{\iota}_{\upsilon}$ or by means of another syntax-driven monotone operator. For the sake of simplicity, in the following we stick to the first choice (however, the two approaches are provably equivalent in $\Upsilon$, see Section 4.5 for the details):

$$\in^{\iota}_{\upsilon} \triangleq \lambda x{:}\upsilon.\lambda M{:}\iota.\neg(x \not\in^{\iota}_{\upsilon} M)$$

It is possible to "lift" non-occurrence predicates to higher-order terms as follows:

$$\begin{aligned} \not\in^{\upsilon_1 \to \cdots \upsilon_n \to \iota}_{\upsilon} \quad \triangleq \quad & \lambda x{:}\upsilon.\lambda M{:}\upsilon_1 \to \cdots \upsilon_n \to \iota.(\forall y_1{:}\upsilon_1.\ldots\forall y_n{:}\upsilon_n.\neg(x =^{\upsilon} y_{k_1}) \Rightarrow \\ & \cdots \Rightarrow \neg(x =^{\upsilon} y_{k_m}) \Rightarrow x \not\in^{\iota}_{\upsilon} (M\ y_1 \cdots y_n)) \quad (n \geq 1), \end{aligned}$$

where $\{k_1, \ldots, k_m\} \subseteq \{1, \ldots, n\}$ and $\upsilon_{k_l} = \upsilon$ for $1 \leq l \leq m$.

So far, we have defined all the machinery needed to formally introduce in the system $\Upsilon$ the axioms of the Theory of Contexts: they are depicted in Figure 4.5 (in the following we will sometimes use the word *extensionality* to denote $Ext^{\upsilon}_{\tau}$ and *$\beta$-expansion* to denote $\beta\_exp^{\upsilon}_{\tau}$). Obviously, they are schemata of axioms, being parameterized by the particular nominal calculus being considered. This is due to the fact that the non-occurrence predicates used to define them depend on object-level syntax as we remarked at the beginning of this section. It is worth noticing that we do not need to explicitly assume the axioms $\mathrm{LEM}_{\mathrm{var}}$ and $\mathrm{LEM}_{\in}$ presented in Section 4.1, since in $\Upsilon$ we have full classical logic.

[9] Intuitively, $x \in^{\iota}_{\upsilon} M$ means that the name/variable $x : \upsilon$ occurs free in the term $M : \iota$.

$$\frac{\iota \in I}{\Gamma \vdash_{\Sigma(N)} \forall P{:}\iota \to o.C_1 \Rightarrow \ldots \Rightarrow C_m \Rightarrow \forall x{:}\iota.(P\ x)} \quad (Ind^{\iota})$$

where $m = |Constr(\iota)|$, and for each $1 \le i \le m$:

1. $c_i^{\tau_{i,1}\times\ldots\times\tau_{i,n_i}\to\iota} \in Constr(\iota)$
2. $\{j_1,\ldots,j_{k_i}\} \triangleq \{j \mid 1 \le j \le n_i \text{ and } \tau_{i,j} = \upsilon_{i,j,1}\times\cdots\times\upsilon_{i,j,m_{i,j}} \to \iota\}$
3. $C_i \triangleq (\forall x_1{:}\tau'_{i,1}.\cdots\forall x_{n_i}{:}\tau'_{i,n_i}.Q_{i,j_1} \Rightarrow \cdots \Rightarrow Q_{i,j_{k_i}} \Rightarrow (P\ (c_i\ x_1 \cdots x_{n_i})))$, where
   (a) $\tau'_{i,j} \triangleq \begin{cases} \upsilon_{i,j,1} \to \cdots \to \upsilon_{i,j,m_{i,j}} \to \iota_i & \text{if } \tau_{i,j} = \upsilon_{i,j,1}\times\cdots\times\upsilon_{i,j,m_{i_j}} \to \iota_i \\ \tau_{i,j} & \text{otherwise} \end{cases}$
   (b) $Q_{i,j_l} \triangleq (\forall y_1{:}\upsilon_{i,j_l,1}.\cdots\forall y_{m_{i,j_l}}{:}\upsilon_{i,j_l,m_{i,j_l}}.(P\ (x_{i,j_l}\ y_1\ \cdots\ y_{m_{i,j_l}})))$ (for $1 \le l \le k_i$)

Figure 4.6: First-order induction principle.

### 4.3.5 Induction in $\Upsilon$

Since in nominal calculi many proofs are carried out by means of structural induction over terms, $\Upsilon$ also provides such appropriate induction principles. In Figure 4.6 we introduce the definition of an induction scheme for any nominal calculus[10]. Moreover, the latter can be naturally extended to higher-order terms, i.e., contexts. For instance in Figure 4.7 we give the induction scheme for simple contexts, i.e., terms of type $\upsilon \to \iota$. In Chapter 5 we will introduce (and justify) induction principles over contexts of type $\upsilon^n \to \iota$ for any $n$ in the case where the type $\iota$ represents processes of a $\pi$-calculus fragment. Since the terms $t_j$ ($1 \le j \le n_i$) are non deterministically defined, when $\tau_{i_j}$ coincides with $\upsilon$ both the "$x_j$" and "$x$" cases apply. Hence, if $c_i^{\tau_{i,1}\times\cdots\times\tau_{i,n_i}\to\iota}$ is a constructor such that $\tau_{i,j}$ coincides $k$ times with $\upsilon$, then the induction principle of $\iota$ has $2^k$ premises related to that constructor.

**Examples.** In order to make clear how induction principles can be generated starting from the abstract definitions in Figure 4.6 and Figure 4.7, we give here two examples illustrating the first-order ($Ind^{term}$) and higher-order ($Ind^{var\to term}$) induction principles for the encoding of untyped $\lambda$-calculus presented in Section 4.3.3:

$$\frac{term \in I}{\begin{array}{l}\Gamma \vdash_{\Sigma(N_\lambda)} \forall P{:}term \to o. \\ \quad (\forall v{:}var.(P\ (is\_var\ v))) \Rightarrow \\ \quad (\forall t{:}term.\forall t'{:}term.(P\ t) \Rightarrow (P\ t') \Rightarrow (P\ (app\ t\ t'))) \Rightarrow \\ \quad (\forall t{:}var \to term.(\forall v{:}var.(P\ (t\ v))) \Rightarrow (P\ (lam\ t))) \Rightarrow \\ \quad \forall t{:}term.(P\ t)\end{array}} \quad (Ind^{term})$$

As we can see, there is one premise for each constructor belonging to type $term$ in $Ind^{term}$. For what concerns the higher-order induction principle $Ind^{var\to term}$ instead, notice that the

[10] In the case that some syntactic categories are mutually defined (e.g., trees and forests), we may want to reason about several inductive properties mutually defined, one for each syntactic category. A generalization of the induction scheme in Figure 4.6 to a mutual one is given in [HMS01a].

$$\frac{\iota \in I}{\Gamma \vdash_{\Sigma(N)} \forall P{:}(\upsilon \to \iota) \to o.C_1 \Rightarrow \ldots \Rightarrow C_m \Rightarrow \forall x{:}\upsilon \to \iota.(P\ x)} \quad (Ind^{\upsilon\to\iota})$$

where $m = \sum_{i=1}^{|Constr(\iota)|} 2^{cv(c_i)}$, $cv(c_i)$ is the number of times that $\tau_{i,j}$ coincides with $\upsilon$ in $\tau_{i,1}, \ldots, \tau_{i,n_i}$ if $\tau_{i,1} \times \cdots \times \tau_{i,n_i} \to \iota$ is the arity of $c_i$ and for each $1 \le i \le m$, $C_i$ is a distinct premise obtained from a constructor $c^{\tau_1 \times \cdots \times \tau_n \to \iota}$ as follows:

1. $\{j_1, \ldots, j_k\} \triangleq \{j \mid 1 \le j \le n \text{ and } \tau_j = \upsilon_{j,1} \times \cdots \times \upsilon_{j,m_j} \to \iota\}$

2. $C_i \triangleq (\forall x_1{:}\tau'_1. \cdots \forall x_n{:}\tau'_n.Q_{j_1} \Rightarrow \cdots \Rightarrow Q_{j_k} \Rightarrow (P\ \lambda x{:}\upsilon.(c_i\ t_1 \cdots t_n)))$, where

   (a) $\tau'_j \triangleq \begin{cases} \upsilon \to \upsilon_{j,1} \to \cdots \to \upsilon_{j,m_j} \to \iota & \text{if } \tau_j = \upsilon_{j,1} \times \cdots \times \upsilon_{j,m_j} \to \iota \\ \upsilon_{j,1} \to \cdots \to \upsilon_{j,m_j} \to \iota' & \text{if } \tau_j = \upsilon_{j,1} \times \cdots \times \upsilon_{j,m_j} \to \iota' \text{ and } \iota' \neq \iota \\ \tau_j & \text{otherwise} \end{cases}$

   (b) $Q_{j_l} \triangleq (\forall y_1{:}\upsilon_{j_l,1}. \cdots \forall y_{m_{j_l}}{:}\upsilon_{j_l,m_{j_l}}.(P\ \lambda x{:}\upsilon.(x_{j_l}\ x\ y_1\ \cdots\ y_{m_{j_l}})))$ $(1 \le l \le k)$

   (c) $t_j = \begin{cases} (x_j\ x) & \text{if } \tau_j = \upsilon_{j,1} \times \cdots \times \upsilon_{j,m_j} \to \iota \\ x_j & \text{if } \tau_j = \upsilon_{j,1} \times \cdots \times \upsilon_{j,m_j} \to \iota' \text{ where } \iota' \neq \iota \text{ or } \tau_j \in V \setminus \{\upsilon\} \\ x_j \text{ or } x & \text{if } \tau_j = \upsilon \end{cases}$

Figure 4.7: Higher-order induction principle for terms of type $\upsilon \to \iota$.

*is_var* constructor yields two premises in $Ind^{var\to term}$, since its type is $var \to term$:

$$\frac{term \in I}{\begin{array}{l} \Gamma \vdash_{\Sigma(N_\lambda)} \forall P{:}(var \to term) \to o. \\ \quad (\forall x{:}var.(P\ \lambda y{:}var.(is\_var\ x))) \Rightarrow \\ \quad (P\ is\_var) \Rightarrow \\ \quad (\forall t{:}var \to term.\forall t'{:}var \to term.(P\ t) \Rightarrow (P\ t') \Rightarrow \\ \quad\quad (P\lambda x{:}var.(app\ (t\ x)\ (t'\ x)))) \Rightarrow \\ \quad (\forall t{:}var \to var \to term.(\forall y{:}var.(P\ \lambda x{:}var.(t\ x\ y))) \Rightarrow \\ \quad\quad (P\ \lambda x{:}var.(lam\ (t\ x)))) \Rightarrow \\ \quad \forall t{:}var \to term.(P\ t) \end{array}} \quad (Ind^{var\to term})$$

### 4.3.6 Functions in Υ

Despite the fact that the motivations which led to the formulation of the Theory of Contexts did not take into account the problem of programming with datatypes including binding structures, Υ also accommodates useful principles of (higher-order) recursion.

Indeed, given any basic type $\iota$ we can consistently extend the framework Υ with a set of typing and equivalence rules defined according to the type of the constructors of $\iota$.

**Definition 4.16** *Let $N \triangleq \langle V, I, L\rangle$ be a nominal calculus, $J \subseteq I$ a subset of basic types, $\tau$ a simple type over $\Sigma_t(N)$ and $\Gamma$ a typing environment, then a $J$-*elimination scheme *over $\tau$ (in $\Gamma$) is a collection of terms $F^\tau_J \triangleq \{f_c \mid c^\alpha \in Constr(\iota),\ \iota \in J\}$ such that the following holds:*

$$\Gamma \vdash f_c : \tau'_1 \to \cdots \to \tau'_n \to \tau \text{ for each } \iota \in J,\ c^{\tau_1 \times \cdots \times \tau_n \to \iota} \in Constr(\iota)$$

$$\frac{F^\tau_J \ J\text{-elimination scheme over } \tau \text{ in } \Gamma,\ \iota \in J}{\Gamma \vdash_{\Sigma(N)} \hat{F}^\tau_\iota : \iota \to \tau} \quad (\hat{F}^\tau_\iota)$$

$$\frac{\Gamma \vdash_{\Sigma(N)} \hat{F}^\tau_\iota : \iota \to \tau}{\Gamma \vdash_{\Sigma(N)} \forall t_1{:}Curry(\tau_1).\dots\forall t_n{:}Curry(\tau_n).(\hat{F}^\tau_\iota\ (c\ t_1 \dots t_n)) =^\tau (f_c\ M_1 \dots M_n)} \quad (\hat{F}^\tau_\iota\text{-}eq_c)$$

where $c^{\tau_1 \times \dots \times \tau_n \to \iota} \in Constr(\iota)$, $f_c \in F^\tau_J$ and for $i = 1, \dots, n$:

$$M_i \triangleq \begin{cases} \lambda x_{i_1}{:}\upsilon_{i_1} \dots \lambda x_{i_{m_i}}{:}\upsilon_{i_{m_i}}.(\hat{F}^\tau_{\iota_i}\ (t_i\ x_{i_1} \dots x_{i_{m_i}})) & \text{if } \tau_i = \upsilon_{i_1} \times \dots \times \upsilon_{i_{m_i}} \to \iota_i \\ & \text{and } \iota_i \in J \\ t_i & \text{otherwise} \end{cases}$$

Figure 4.8: First-order recursion typing and equivalence rules.

*where* $\tau'_i \triangleq \begin{cases} \upsilon_{i_1} \to \dots \to \upsilon_{i_{m_i}} \to \tau & \textit{if } \tau_i = \upsilon_{i_1} \times \dots \times \upsilon_{i_{m_i}} \to \iota_i \textit{ and } \iota_i \in J \\ \upsilon_{i_1} \to \dots \to \upsilon_{i_{m_i}} \to \iota_i & \textit{if } \tau_i = \upsilon_{i_1} \times \dots \times \upsilon_{i_{m_i}} \to \iota_i \textit{ and } \iota_i \notin J \\ \tau_i & \textit{otherwise} \end{cases}$

*Then, given a $J$-elimination scheme $F^\tau_J$ over $\tau$ (in $\Gamma$), for each $\iota \in J$, we introduce a new symbol $\hat{F}^\tau_\iota$ denoting the* $F^\tau_J$-defined recursive map over *$\iota$ whose typing and equivalence rules appear in Figure 4.8.*

The previous definition generalizes the usual notion of recursion to the case of mutually defined recursive maps and to terms possibly containing contexts, i.e., functional terms over names.

In $\Upsilon$, differently from most logical frameworks, it is also possible to accommodate recursion principles for (higher-order) contexts. The following definitions allow one to define recursive functions over $n$-ary contexts of type $\upsilon^n \to \iota$ where $\upsilon \in V$, $\iota \in I$ and $\upsilon^n$ is a shorthand for $\underbrace{\upsilon \to \dots \to \upsilon}_{n \text{ times}}$.

**Definition 4.17** *Let $N \triangleq \langle V, I, L\rangle$ be a nominal calculus, $\alpha = \tau_1 \times \dots \times \tau_n \to \iota$ a nominal arity of a constructor belonging to a basic type in $I$, $\upsilon \in V$ and $n \geq 1$. Then we have the following conventions:*

1. *we denote by $k$ the number of indexes $1 \leq i_1 \leq \dots \leq i_k \leq n$ ($k \geq 0$) such that $\tau_{i_j} = \upsilon$;*

2. *let $L(\alpha) \triangleq \{0,1\}^k$ be the set of binary strings of length $k$, which we call the* labels for *$\alpha$ (thus, $|L(\alpha)| = 2^k$); for $j = 1 \dots k$, the $j$-th component of a label $l$ is denoted by $l_{i_j}$, that is it has the same index of the occurrence of $\upsilon$ in $\tau_1 \times \dots \times \tau_n$ it refers to;*

3. *we denote by $l \bullet (\tau'_1 \to \dots \to \tau'_n \to \tau)$ the type obtained from $\tau'_1 \to \dots \to \tau'_n \to \tau$ by eliminating $\tau'_{i_j}$ if $l_{i_j} = 0$.*

**Definition 4.18** *Let $N \triangleq \langle V, I, L\rangle$ be a nominal calculus, $\upsilon \in V$, $n \geq 1$, $J \subseteq I$ a subset of basic types, $\tau$ a simple type over $\Sigma_t(N)$, and let $\Gamma$ be a typing environment. Then, a* $\upsilon^n J$-elimination scheme over $\tau$ (in $\Gamma$) *is a family of terms $F^\tau_{\upsilon^n,J} = \{f^l_c \mid c^\alpha \in Constr(\iota), \iota \in$*

$$\frac{F^{\tau}_{\upsilon^n,J}\ \upsilon^n J\text{-elimination scheme over } \tau \text{ in } \Gamma,\ \iota \in J}{\Gamma \vdash_{\Sigma(N)} \hat{F}^{\tau}_{\upsilon^n,\iota} : (\upsilon^n \to \iota) \to \tau} \quad (\hat{F}^{\tau}_{\upsilon^n,\iota})$$

$$\frac{\Gamma \vdash_{\Sigma(N)} \hat{F}^{\tau}_{\upsilon^n,\iota} : (\upsilon^n \to \iota) \to \tau}{\Gamma \vdash_{\Sigma(N)} Q_1.Q_2.(\hat{F}^{\tau}_{\upsilon^n,\iota}\ \lambda\vec{x}{:}\upsilon.(c\ N_1 \dots N_n)) =^{\tau} (f^l_c\ M_1 \dots M_n)} \quad (\hat{F}^{\tau}_{\upsilon^n,\iota}\text{-}eq^l_c)$$

where $c^{\tau_1\times\cdots\times\tau_n\to\iota} \in Constr(\iota)$, $l \in L(\tau_1 \times \cdots \times \tau_n \to \iota)$, $f^l_c \in F^{\tau}_{\upsilon^n,J}$, $Q_1$ is a sequence of universal quantifications of the form $\forall t_i{:}\upsilon^n \to \tau'_i$ for each $i \in \{i \mid 1 \leq i \leq n, \tau_i \notin V\}$ and $Q_2$ is a sequence of universal quantifications of the form $\forall y_i : \tau_i$ for each $i \in \{i \mid 1 \leq i \leq n, \tau_i \in V\}$, $\lambda\vec{x}{:}\upsilon$ is a shorthand for $\lambda x_1{:}\upsilon \dots \lambda x_n{:}\upsilon$ and for $i = 1, \dots, n$:

$$\tau'_i \triangleq \begin{cases} \upsilon_{i_1} \to \cdots \to \upsilon_{i_{m_i}} \to \iota_i & \text{if } \tau_i = \upsilon_{i_1} \times \cdots \times \upsilon_{i_{m_i}} \to \iota_i \\ \tau_i & \text{otherwise} \end{cases}$$

$$N_i = \begin{cases} (t_i\ \vec{x}) & \text{if } \tau_i \notin V \\ y_i & \text{if } \tau_i \in V \text{ and } (\tau_i \neq \upsilon \text{ or } l_i = 1) \\ x & \text{if } \tau_i = \upsilon \text{ and } l_i = 0 \end{cases}$$

$$M_i = \begin{cases} \lambda x_{i_1}{:}\upsilon_{i_1} \dots \lambda x_{i_{m_i}}{:}\upsilon_{i_{m_i}}.(\hat{F}^{\tau}_{\upsilon^n,\iota_i}\ \lambda\vec{x}{:}\upsilon(t_i\ \vec{x}\ x_{i_1} \dots x_{i_{m_i}})) & \\ \qquad\qquad & \text{if } \tau_i = \upsilon_{i1} \times \cdots \times \upsilon_{im_i} \to \iota_i \text{ and } \iota_i \in J \\ y_i & \text{if } \tau_i \in V \text{ and } (\tau_i \neq \upsilon \text{ or } l_i = 1) \\ (nothing) & \text{if } \tau_i = \upsilon \text{ and } l_i = 0 \\ (t_i\ \vec{x}) & \text{otherwise} \end{cases}$$

where $\vec{x}$ (in applications) stands for $x_1 \dots x_n$.

Figure 4.9: Higher-order recursion typing and equivalence rules.

*$J, l \in L(\alpha)\}$ such that, for each $\iota \in J$, $c^{\tau_1\times\cdots\times\tau_n\to\iota} \in Constr(\iota)$ and $l \in L(\tau_1 \times \cdots \times \tau_n \to \iota)$, the following holds:*

$$\Gamma \vdash f^l_c : l \bullet (\tau'_1 \to \cdots \to \tau'_n \to \tau)$$

$$\textit{where } \tau'_i \triangleq \begin{cases} \upsilon_{i_1} \to \cdots \to \upsilon_{i_{m_i}} \to \tau & \textit{if } \tau_i = \upsilon_{i_1} \times \cdots \times \upsilon_{i_{m_i}} \to \iota_i \textit{ and } \iota_i \in J \\ \upsilon_{i_1} \to \cdots \to \upsilon_{i_{m_i}} \to \iota_i & \textit{if } \tau_i = \upsilon_{i_1} \times \cdots \times \upsilon_{i_{m_i}} \to \iota_i \textit{ and } \iota_i \notin J \\ \tau_i & \textit{otherwise} \end{cases}$$

Hence, for each constructor $c^{\alpha}$, there are $|L(\alpha)|$ terms in $F^{\tau}_{\upsilon^n,J}$.

**Definition 4.19** *Let $N \triangleq \langle V, I, L\rangle$ be a nominal calculus, $\upsilon \in V$, $n \geq 1$, $J \subseteq I$ a subset of basic types, $\tau$ a simple type over $\Sigma_t(N)$, $\Gamma$ a typing environment and $F^{\tau}_{\upsilon^n,J}$ a $\upsilon^n J$-elimination scheme over $\tau$ (in $\Gamma$), then, for each $\iota \in J$, we introduce a new symbol $\hat{F}^{\tau}_{\upsilon^n,\iota}$ denoting the* $F^{\tau}_{\upsilon^n,J}$-defined recursive map over *$\iota$ whose typing and equivalence rules appear in Figure 4.9.*

We end this section by giving an example of a function recursively definable in $\Upsilon$.

**Example.** Let us consider again the encoding of untyped $\lambda$-calculus of Section 4.3.3 and let us choose a typing environment $\Gamma$ and a term $N$ such that $\Gamma \vdash_{\Sigma(N_\lambda)} N : term$ holds. In this example we want to define in the signature $\Sigma(N_\lambda)$, by higher-order recursion, the substitution $\cdot[N/\cdot]$. The latter will be modeled by a function taking as argument a term of type $var \to term$, i.e., a term with a designated hole and yielding a term where $N$ has been substituted for the hole (see [Hof99]). Since we have three constructors belonging to the type $term$, the corresponding three sets of labels are $L(var \to term) = \{0, 1\}$, $L(term \to term \to term) = L((var \to term) \to term) = \{\langle\rangle\}$. Thus, let $F^{term}_{var,\{term\}} \triangleq \{f^0_{is\_var}, f^1_{is\_var}, f_{app}, f_{lam}\}$ where $f^0_{is\_var} \triangleq N$, $f^1_{is\_var} \triangleq is\_var$, $f_{app} \triangleq app$, $f_{lam} \triangleq lam$. Then, $F^{term}_{var,\{term\}}$ is a $var\{term\}$-elimination scheme over $term$ in $\Gamma$, such that $\Gamma \vdash_{\Sigma(N_\lambda)} \hat{F}^{term}_{var,term} : (var \to term) \to term$ and the following are derivable:

$$\frac{\Gamma \vdash_{\Sigma(N_\lambda)} \hat{F}^{term}_{var,term} : (var \to term) \to term}{\Gamma \vdash_{\Sigma(N_\lambda)} \hat{F}^{term}_{var,term}(is\_var) =^{term} N}$$

$$\frac{\Gamma \vdash_{\Sigma(N_\lambda)} \hat{F}^{term}_{var,term} : (var \to term) \to term}{\Gamma \vdash_{\Sigma(N_\lambda)} \forall y{:}var.\hat{F}^{term}_{var,term}(\lambda x{:}var.(is\_var\ y)) =^{term} (is\_var\ y)}$$

$$\frac{\Gamma \vdash_{\Sigma(N_\lambda)} \hat{F}^{term}_{var,term} : (var \to term) \to term}{\begin{array}{l}\Gamma \vdash_{\Sigma(N_\lambda)} \forall M_1{:}var \to term.\forall M_2{:}var \to term.\\ \qquad \hat{F}^{term}_{var,term}(\lambda x{:}var.(app\ (M_1\ x)\ (M_2\ x)) =^{term} (app\ \hat{F}^{term}_{var,term}(M_1)\ \hat{F}^{term}_{var,term}(M_2))\end{array}}$$

$$\frac{\Gamma \vdash_{\Sigma(N_\lambda)} \hat{F}^{term}_{var,term} : (var \to term) \to term}{\begin{array}{l}\Gamma \vdash_{\Sigma(N_\lambda)} \forall M{:}var \to var \to term.\\ \qquad \hat{F}^{term}_{var,term}(\lambda x{:}var.(lam\ (M\ x))) =^{term} (lam\ \lambda x{:}var.\hat{F}^{term}_{var,term}(M\ x)\end{array}}$$

Hence, for any $M$ term and $x$ variable, $\hat{F}^{term}_{var,term}(\lambda x{:}var.M)$ is equal to the term obtained from $M$ by replacing every free occurrence of $x$ by $N$.

## 4.4 The Axiom of Unique Choice

As originally pointed out in [Hof99] for the case of $\lambda$-calculus, a rather surprising tradeoff of our natural framework for treating contexts is the following:

**Proposition 4.1** *The Axiom of Unique Choice*

$$\frac{\Gamma \vdash_\Sigma P : \tau_1 \to \tau_2 \to o}{\Gamma \vdash_\Sigma (\forall x{:}\tau_1.\exists y{:}\tau_2.(P\ x\ y) \wedge \forall z{:}\tau_2.(P\ x\ z) \Rightarrow y =^{\tau_2} z) \Rightarrow \exists f{:}\tau_1 \to \tau_2.\forall x{:}\tau_1.(P\ x\ (f\ x))} \quad \text{(AC!)}$$

*is inconsistent with the Theory of Contexts.*

*Proof.* Let us consider the case of the $\pi$-calculus encoding (see the examples in Section 4.3.3), then, by $Unsat^{name}_{proc}$, we can infer the existence of two fresh names $u'$, $v'$; hence, we can define the term $R \triangleq \lambda u : name.\lambda q : proc.\lambda x : name.\lambda p : proc.(x =^{name} u \wedge p =^{proc} 0) \vee (\neg x =^{name} u \wedge p =^{proc} q)$. It is easy to show that, for all $p' : proc$, $(R\ u'\ p') : name \to proc \to o$ is a functional binary relation. At this point we can prove, by means of $Ext^{name}_{proc}$ and AC!, that the proposition $\forall p : proc.p =^{proc} 0$ holds; indeed, from AC! we can deduce the existence of a function $f : name \to proc$ such that, for all $x : name$, $((R\ u'\ p)\ x\ (f\ x))$ holds. Hence, by $Ext^{name}_{proc}$, we can prove that $f =^{name \to proc} \lambda x : name.p$ because for any fresh name $w$ we have that $(f\ w) =^{proc} p$. Then we have that, for all names $y$, $(f\ y) =^{proc} ((\lambda x : name.p)\ y) =^{proc} p$ holds, whence we may conclude, since $(f\ u') =^{proc} 0$.

At this point the contradiction follows because, as a special case, we have that $0|0 =^{proc} 0$ while *proc* is an inductive type (the constructors are disjoint[11]). □

Proposition 4.1 highlights a weak point of $\Upsilon$ w.r.t. the framework introduced by Gabbay and Pitts [GP99, GP01, Gab00], that is, there are (recursive) functions which cannot be defined as such in our system. The reason is that we model name-abstractions by means of total functions of names (instead of partial functions only defined for sufficiently fresh names) and this fact cannot coexists with the axioms of the Theory of Contexts without reducing the set of definable functions. More precisely $\Upsilon$ does not allow one to define functions whose definitions need freshly generated names, since there are no means for generating a "fresh name" at the term level, while we can use $Unsat^{v}_{\iota}$ for generating fresh names at the logical level. Nevertheless, $n$-ary functions of this kind can be represented in $\Upsilon$ as $(n+1)$-ary relations, as in the next Example.

**Example.** Let us consider the encoding of untyped $\lambda$-calculus introduced in Section 4.3.3) and the function $count : term \rightarrow nat$ which takes as argument a term $M$ of type $term$ and returns the number[12] of occurrences of free variables occurring in $M$. The corresponding elimination scheme over $nat$ should be $f_{is\_var} \triangleq \lambda x : var.1$, $f_{app} \triangleq \lambda n : nat.\lambda n' : nat.n + n'$, $f_{lam} \triangleq \lambda g : var \rightarrow nat.(g\ z) \dot{-} 1$, where $\dot{-}1$ denotes the predecessor function over natural numbers. However, the above definition cannot be expressed in $\Upsilon$ since the fresh variable $z$, needed in the definition of $f_{lam}$, is not definable. We do not have a mechanism working at the level of datatypes for generating fresh names on the spot, like Gabbay and Pitts' *fresh* operator [GP99]. It is straightforward that, in the presence of such a fresh operator, $f_{lam}$ can be defined as $\lambda g : var \rightarrow nat.fresh\ z.(g\ z) \dot{-} 1$. However, we can represent $f_{lam}$ as a binary relation $R_{lam} : (var \rightarrow nat) \rightarrow nat \rightarrow o$ defined as

$$R_{lam}(g, n) \triangleq \exists z : nat.z \notin^{var \rightarrow nat} g \wedge (g\ z) \dot{-} 1 =^{nat} n$$

the existence of the fresh variable $z$ being granted by $Unsat^{var}_{nat}$. Hence, the *fresh* operator can be mimicked at the logical level by our $Unsat^{v}_{\iota}$ axiom scheme.

Luckily, $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$ and its implementation `Coq` do not validate AC!. Hence, the Theory of Contexts can be consistently axiomatized into them. Interestingly, in the case that `Prop=Set`, AC! is derivable in `Coq` as it is showed in the following short proof script:

```
Inductive myEx [A:Set; P:A->Set] : Set :=
   myEx_intro : (x:A)(P x)->(myEx A P).

Inductive myAnd [A:Set; B:Set] : Set :=  myAnd_conj : A->B->(myAnd A B).

Inductive myEq [A:Set; x:A] : A->Set :=  myEq_refl_equal : (myEq A x x).

Definition witness := [A,B:Set][P:B->Set][p:(myEx B P)]
```

[11] The disjointness of constructors of an inductive type is derivable in $\Upsilon$. Indeed, in the case at hand, for example, in order to prove that $|$ and $0$ are disjoint, it suffices to define a function $discr : proc \rightarrow o$ such that $discr(P) = \top$ (we have not defined the *true* connective, but this can be easily done in the usual way) if $P$ is of the form $Q|R$ and $discr(P) = \bot$ otherwise. Then, from $0|0 =^{proc} 0$, the definition of Leibniz equality (see Figure 4.2) and the definition of *discr*, we can deduce that $\top \Rightarrow \bot$ holds. At this point $\bot$ follows by modus ponens (MP).

[12] Here, we assume the availability of the type of natural numbers. Although we have not defined them in $\Upsilon$, they can be easily added to the metalanguage.

```
  Cases p of (myEx_intro x q) => x end.

Lemma AC_U: (A,B:Set)(R':A->B->Set)
            ((a:A)(myEx B [b:B](myAnd (R' a b)
                              ((b':B)(R' a b')-> (myEq B b b')))))->
            (myEx A->B [f:A->B](a:A)(R' a (f a))).
Proof.
Intros; Split with
[a:A](witness A B [x:B](myAnd (R' a x) (b':B)(R' a b')->(myEq B x b'))
             (H a)); Intro; Unfold witness; (Elim (H a); Intros);
(Elim p; Intros); Assumption.
Qed.
```

It is important to stress that names/variables cannot be represented by inductive types (this is the reason for the distinction we made in Section 4.2 between names bases and inductive types). Otherwise, it is easy to get an inconsistency by defining "exotic" functions by case analysis. Indeed, the argument used in the proof of Proposition 4.1 ultimately relies on the existence of a function able to distinguish between two names. Such a function can be recovered as follows in, e.g., Coq (we are using the same signature of the $\pi$-calculus of Section 4.3.3):

```
Definition name:=nat.
Definition x:= O.
Definition y:=(S O).
Definition p:=[z:name]nil.
Definition q:=[z:name]Case z of nil [y:name](par nil nil) end.
```

By the preceding definitions we have `(p x)=(p y)=(q x)=nil`, but `(q y)=(par nil nil)`. Then, applying $Ext_\iota^\upsilon$ we can easily prove `nil=(par nil nil)`, whence the absurdity.

The abovementioned subtleties clearly represent the need for a model proving the consistency of the Theory of Contexts. This will be the topic of the subsequent chapter, where functor categories will be used to build such a model.

## 4.5 Investigating the Theory of Contexts

In this section we will illustrate the results so far obtained in the study of the expressiveness and independence of the properties of the Theory of Contexts. Indeed, since its birth, there have been some refinements; hence, we think that it will be useful to summarize the current results, which are rather interesting.

### 4.5.1 Independence

In the original presentation of the Theory of Contexts [HMS01b], there was another rather useful axiom stating the monotonicity[13] of $\not\in_\upsilon^\iota$. Moreover, the decidability of checking the

[13] The name of this property may suggest that there is an order being preserved. Indeed, $(M\ y)$ is a plain term of type $\iota$ while $M$ is a unary context of type $\upsilon \to \iota$. Hence, if we decide that terms of type $\iota$ are "smaller" than terms of type $\upsilon \to \iota$ (which, in turn, are smaller than terms of type $\upsilon \to \upsilon \to \iota$ and so on), we have that the axiom of monotonicity preserves such an order since the truth value of $x \not\in_\upsilon^\iota (M\ y)$ is smaller or equal to that of $x \not\in_\upsilon^{\upsilon\to\iota} M$. In fact, if $x \not\in_\upsilon^\iota (M\ y)$ is false, then $x \not\in_\upsilon^{\upsilon\to\iota} M$ may be true or false, while if $x \not\in_\upsilon^\iota (M\ y)$ is true, then $x \not\in_\upsilon^{\upsilon\to\iota} M$ must also be true.

occurrence of a name in a term was assumed. These axioms can be rendered in $\Upsilon$ as follows:

$$\frac{-}{\Gamma \vdash_\Sigma \forall M{:}\upsilon \to \iota.\forall x{:}\upsilon.\forall y{:}\upsilon.x \not\in^\iota_\upsilon (M\ y) \Rightarrow x \not\in^{\upsilon\to\iota}_\upsilon M} \quad (\mathrm{MON}_{\not\in^\iota_\upsilon})$$

$$\frac{-}{\Gamma \vdash_\Sigma \forall x{:}\upsilon.\forall M{:}\iota.x \in^\iota_\upsilon M \vee x \not\in^\iota_\upsilon M} \quad (\mathrm{LEM}_{\not\in^\iota_\upsilon})$$

Indeed, the decidability of equality over names is derivable from $\mathrm{LEM}_{\not\in^\iota_\upsilon}$ (recall that the underlying logic of `Coq` is intuitionistic; hence, classical axioms must be explicitly assumed).

As it is noticed in [HMS01a], we discovered that $\mathrm{MON}_{\not\in^\iota_\upsilon}$ is indeed derivable by means of a structural induction over contexts (i.e., using $Ind^{\upsilon\to\iota}$). Another possibility is to proceed as follows: by a structural induction over terms (i.e., using $Ind^\iota$) it is possible to infer the following result:

$$\vdash_\Sigma SEP{:}\forall M{:}\iota.\forall x{:}\upsilon.\forall y{:}\upsilon.(x \in^\iota_\upsilon M) \Rightarrow (y \not\in^\iota_\upsilon M) \Rightarrow \neg x =^\upsilon y.$$

Then using $\mathrm{LEM}_{\not\in^\iota_\upsilon}$ and $SEP$, it is possible to deduce the following auxiliary lemmata:

$$\begin{array}{lcl} \vdash_\Sigma A1 & : & \forall M{:}\upsilon \to \iota.\forall x{:}\upsilon.(x \not\in^\iota_\upsilon (M\ x)) \Rightarrow (x \not\in^{\upsilon\to\iota}_\upsilon M) \\ \vdash_\Sigma A2 & : & \forall M{:}\upsilon \to \iota.\forall x{:}\upsilon.\forall y{:}\upsilon.\neg x =^\upsilon y \Rightarrow (x \not\in^\iota_\upsilon x(M\ y)) \Rightarrow \\ & & \forall z{:}\upsilon.\neg x =^\upsilon z \Rightarrow \neg y =^\upsilon z \Rightarrow (x \not\in^\iota_\upsilon (M\ z)) \end{array}$$

At this point $\mathrm{MON}_{\not\in^\iota_\upsilon}$ easily follows from the decidability of equality over names and the previous auxiliary lemmata.

We remark here that monotonicity of $\in^\iota_\upsilon$ (defined as the negation of $\not\in^\iota_\upsilon$), i.e.

$$\forall M{:}\upsilon \to \iota.\forall x{:}\upsilon.\forall y{:}\upsilon.\neg x =^\upsilon y \Rightarrow x \in^\iota_\upsilon (M\ y) \Rightarrow x \in^{\upsilon\to\iota}_\upsilon M$$

is trivially derivable by exploiting the constructive definition of $\not\in^\iota_\upsilon$. The choice of defining $\in^\iota_\upsilon$ in terms of $\not\in^\iota_\upsilon$ rather than giving an independent constructive definition is motivated by the fact that in nominal calculi a crucial rôle is played by freshness (i.e. non-occurrence) of names within terms. However, it would be clearly possible to give a constructive definition of such a predicate as follows[14] (in the following for the sake of readability, we drop the indexes $\iota$ and $\upsilon$ in $\in^\iota_\upsilon$):

$$\in^c \triangleq \lambda x{:}\upsilon.\lambda M{:}\iota.\forall R{:}\upsilon \to \iota \to o.(\forall y{:}\upsilon.\forall N{:}\iota.(T_{\in^c}(R)\ y\ N) \Rightarrow (R\ y\ N)) \Rightarrow (p\ x\ M)$$

where

$$T_{\in^c} : (\upsilon \to \iota \to o) \to (\upsilon \to \iota \to o) \triangleq \lambda R{:}\upsilon \to \iota \to o.\lambda x{:}\upsilon.\lambda M{:}\iota.\bigvee_{i=1}^{|Constr(\iota)|} C_i,$$

where each $C_i$ (for $i = 1, \ldots, |Constr(\iota)|$) is a clause corresponding to a constructor $c_i$ : $Curry(\alpha_i)$ belonging to the type $\iota$ as follows:

$\alpha = \iota$: $C_i$ is empty (or alternatively $\top$);

$\alpha = \tau_1 \times \cdots \times \tau_k \to \iota$: $C_i \triangleq \exists N_1{:}\tau_1.\ldots.\exists N_k{:}\tau_k.M =^\iota (c\ N_1 \cdots N_k) \wedge \bigvee_{j=1}^k H_j$, where each $H_j$ depends on the shape of $\tau_j$:

$\tau_j = \upsilon$: $H_j \triangleq x =^\upsilon N_j$;

[14]This approach turns out to be more effective during the activity of proof development in frameworks with advanced tactics supporting inductive types.

$$\tau_j = \upsilon_{j1} \times \cdots \times \upsilon_{jm_j} \to \sigma_j\text{:}\ H_j \triangleq (\forall y_1{:}\upsilon_{j1}.\ldots\forall y_{m_j}{:}\upsilon_{jm_j}.(R\ x\ (N_j\ y_1 \cdots y_{m_j}))).$$

The two definitions are provably equivalent. We have in fact the following:

$$\vdash_\Sigma \forall x{:}\upsilon.\forall M{:}\iota.x \in M \Leftrightarrow x \in^c M$$

Without going into the details of the proof, we only notice that the left ($\Leftarrow$) implication requires $Unsat_\iota^\upsilon$, while the right ($\Rightarrow$) direction is provable by means of $Ind^\iota$, decidability of equality over names, $Unsat_\iota^\upsilon$ and $MON_{\not\in}$.

A third possibility of deriving the monotonicity properties of both $\in_\upsilon^\iota$ and $\not\in_\upsilon^\iota$ is the following:

1. we define a relation $l{:}\iota \to nat \to o$ such that $(l\ M\ n)$ holds if and only if $M$ contains $n$ occurrences of constructors of $\iota$ (that is we define a measure of the complexity of terms);

2. we show that $l$ is preserved by renaming, i.e., $(l\ (M\ x)\ n)$ implies $(l\ (M\ y)\ n)$ for any $x$, $y$.

3. we show that for every term $M$ there is a natural $n$ such that $(l\ M\ n)$ holds (i.e., $l$ is total w.r.t. the first argument);

4. we carry out the proof of monotonicity by *complete* induction[15] (also known as *course of values* induction) on $n$, where $n$ is the natural such that $(l\ (My)\ n)$ holds (see the properties $MON_{\not\in_\upsilon^\iota}$ and $MON_{\in_\upsilon^\iota}$).

The only other axioms of the Theory of Contexts needed in the previous proof are unsaturation and the decidability of equality of names. In Section 6.2.10 of Chapter 6 the abovementioned proof technique is explained in full detail for an encoding of the Ambient Calculus.

As we anticipated in the introduction of the present chapter, according to our experience, in order to reason about the metatheory of nominal calculi, full classical logic is not strictly needed. Indeed, we could replace $DN$ in Figure 4.4 with either an axiom stating the decidability of Leibniz equality over names (LEM$_{=^\upsilon}$) or, as we noticed above, an axiom stating the decidability of occurrence predicates of names in terms (LEM$_{\not\in_\upsilon^\iota}$). We have already seen how to render in our framework $\Upsilon$, the axiom LEM$_{\not\in_\upsilon^\iota}$; LEM$_{=^\upsilon}$ instead is represented as follows:

$$\frac{-}{\Gamma \vdash_\Sigma \forall x{:}\upsilon.\forall y{:}\upsilon.x =^\upsilon y \vee x \neq^\upsilon y} \qquad (\text{LEM}_{=^\upsilon})$$

As we mentioned above LEM$_{=^\upsilon}$ derives directly from LEM$_{\not\in_\upsilon^\iota}$. For the converse, LEM$_{\not\in_\upsilon^\iota}$ can be derived by a structural induction on terms of type $\iota$ (i.e., applying $Ind^\iota$) using LEM$_{=^\upsilon}$ and the monotonicity of $\not\in_\upsilon^\iota$ and $\in_\upsilon^\iota$ in the cases involving binders.

Thus, the minimal classical flavour that $\Upsilon$ must have in order to allow metatheoretic reasoning about the representation of nominal calculi amounts to decidability of equality of names or to decidability of occurrence predicates of names in terms. However, in presenting $\Upsilon$ for simplicity we preferred to stick to full classical logic.

In this chapter we have stated the axioms for first-order ($Ind^\iota$) and higher-order ($Ind^{\upsilon\to\iota}$) induction. However, it is possible to go further, introducing induction principles for contexts

[15] In the next section we will explain why there is the need of a complete induction instead of a "traditional" one.

of arbitrary arity ($Ind^{\upsilon^n \to \iota}$, where $n > 1$). However, since their formulation in full generality is too complicated, the reader is referred to Figure 5.8 for an example regarding the encoding of the fragment of $\pi$-calculus used in Chapter 5. Higher-order induction principles play a fundamental rôle in the following result concerning the axioms schemata $\beta\_exp^{\upsilon}_{\upsilon^n \to \iota}$ and $Ext^{\upsilon}_{\upsilon^{n+1} \to \iota}$:

**Proposition 4.2** *For all $n \in \mathbb{N}$: $Ind^{\upsilon^n \to \iota}$ allows to derive $\beta\_exp^{\upsilon}_{\upsilon^n \to \iota}$ from $\beta\_exp^{\upsilon}_{\upsilon^{n+1} \to \iota}$ and (if $n > 0$) $Ext^{\upsilon}_{\upsilon^n \to \iota}$ from $Ext^{\upsilon}_{\upsilon^{n+1} \to \iota}$.*

*Proof.* By structural induction on contexts of type $\upsilon^n \to \iota$, using $Ind^{\upsilon^n \to \iota}$. Most cases are trivial; in the case of the $\nu$ constructor, we apply the axioms $\beta\_exp^{\upsilon}_{\upsilon^{n+1} \to \iota}$ and $Ext^{\upsilon}_{\upsilon^{n+1} \to \iota}$. □

It should be noticed that the derivability of $\beta\_exp^{\upsilon}_{\upsilon^n \to \iota}$ and $Ext^{\upsilon}_{\upsilon^n \to \iota}$ by structural induction ($Ind^{\upsilon^n \to \iota}$) can be carried out only in Logical Frameworks featuring an extensional syntactical equality (e.g. Isabelle/HOL). In `Coq` (whose syntactical equality is not extensional) instead, the cases involving binders fail since they require $\beta$-expansion and extensionality for contexts with higher arity (i.e., with "one additional hole")[16].

To sum up, the only properties of the original Theory of Contexts [HMS01b] that seem to be orthogonal (not derivable from the remaining) are unsaturation, $\beta$-expansion and extensionality. It follows that the cleanest refinement of the Theory of Contexts coincides with the informal presentation made in the introduction of this chapter:

- decidability of equality of names (may be omitted if the framework is classical);
- unsaturation;
- $\beta$-expansion;
- extensionality.

We chose decidability of equality of names as a primitive axiom instead of decidability of occur checking predicates because of its simplicity. Moreover, with the previous set of axioms and $Ind^{\iota}$ we can derive monotonicity of $\in^{\iota}_{\upsilon}$ and $\notin^{\iota}_{\upsilon}$ and $\mathrm{LEM}_{\notin^{\iota}_{\upsilon}}$.

### 4.5.2 Expressiveness

We already anticipated that the soundness of the Theory of Contexts will be proved in Chapter 5. However, as far the completeness is concerned, we do not have yet a result stating the expressive power of our axioms w.r.t. some known logic system.

However, in this section we prove an important result in this direction, namely, the derivability of the higher-order induction principle $Ind^{\upsilon \to \iota}$ by means of the complete induction principle on natural numbers, $Ind^{\iota}$ and the axioms of the Theory of Contexts.

In order to spell out all the details, we will consider the encoding of untyped $\lambda$-calculus in `Coq`; hence, the following formal development will be expressed in the metalanguage of `Coq`, namely, `Gallina` (see Section 2.3.2). The complete source code of the proof is gathered in Appendix A.

[16] However, if the object language does not include any binders among its constructors, both $\beta$-expansion and extensionality are derivable using $Ind^{\upsilon^n \to \iota}$.

### Encoding of syntax

Since we want to use both inductive definitions and HOAS, we represent variables of untyped $\lambda$-calculus by means of `Coq` metavariables of type `var`, where the latter is the type defined by the following declaration:

```
Parameter var: Set.
```

Untyped $\lambda$-terms are represented by means of the following inductive type (see Section 3.1.2):

```
Inductive tm : Set:=
    is_var: var -> tm
  | app: tm -> tm -> tm
  | lam: (var -> tm) -> tm.
```

The freshness predicate is defined following the general pattern of Section 4.15:

```
Inductive notin [x:var]: tm -> Prop:=
  notin_var: (y:var)~x=y -> (notin x (is_var y))
| notin_app: (M,N:tm)(notin x M) -> (notin x N) -> (notin x (app M N))
| notin_lam: (M:var->tm)((y:var)~x=y -> (notin x (M y))) ->
             (notin x (lam M)).
```

At this point we are ready to introduce the "measure relation" `l` already mentioned in the previous section (when we illustrated the third technique used to derive the monotonicity of the freshness predicate):

```
Inductive l: tm -> nat -> Prop:=
  l_var : (x:var)(l (is_var x) (S O))
| l_app : (M,N:tm)(n1,n2:nat)(l M n1) -> (l N n2) ->
          (l (app M N) (S (plus n1 n2)))
| l_lam : (M:var->tm)(n:nat)((y:var)(l (M y) n)) -> (l (lam M) (S n)).
```

Intuitively `(l M n)` holds if and only if `M` contains exactly `n` occurrences of constructors belonging to the type `tm`.

### The Theory of Contexts for the untyped $\lambda$-calculus

During the proof development we used the following instantiations of the axiom schemata of the Theory of Contexts:

```
Axiom dec_var: (x,y:var)x=y \/ ~x=y.

Axiom unsat: (M:tm)(Ex [x:var](notin x M)).

Axiom exp: (M:tm)(x:var)(Ex [N:var->tm](notin x (lam N)) /\ M=(N x)).

Axiom ho_exp: (M:var->tm)(x:var)
  (Ex [N:var->var->tm](notin x (lam [_:var](lam (N _)))) /\ M=(N x)).

Axiom ext: (F,G:var->tm)(x:var)
           (notin x (lam F)) -> (notin x (lam G)) ->
           (F x)=(G x) -> F=G.
```

### The formal development

The first results we need concern properties of the measure relation `l`; first of all, we show that `l` is preserved by fresh renaming:

```
Lemma L_RW: (n:nat)(M:tm)(l M n) -> (x:var)(N:var->tm)(notin x (lam N)) ->
            M=(N x) -> (y:var)(l (N y) n).
```

The proof technique used is a complete induction on `n`. We notice that complete induction on natural numbers is trivially derivable from the induction principle `nat_ind` automatically provided by `Coq` on type `nat` (see Appendix A for the details and the proof script). The reason for using such a principle is that it allows to apply the inductive hypothesis to any term structurally smaller than that of the current hypothesis, not only to the immediate subterm of the latter, which is instead the only possibility offered by the induction principle `tm_ind` provided by `Coq`. Hence, we can "mimick" a complete induction principle on the structure of terms by means of a complete induction on the number of constructors' occurrences of terms. This is fundamental in proving renaming results like `L_RW` since in the cases involving binders, there is the need to apply the inductive hypothesis two times before concluding the case. Indeed, the first application is carried out only to replace all the occurrences of the generic variable introduced by the `l_lam` rule. Indeed, being generic, such a variable is not generally fresh and this fact is in conflict with the `notin` judgment present in the inductive hypothesis. A glance at the relative `Coq` session will make the argument clear:

```
n : nat
n0 : nat
H : (m:nat)
     (lt m n0)
     ->(M:tm)
        (l M m)
        ->(x:var; N:(var->tm))
           (notin x (lam N))->M=(N x)->(y:var)(l (N y) m)
M : tm
H0 : (l M n0)
x : var
N : var->tm
H1 : (notin x (lam N))
y : var
M0 : var->tm
n1 : nat
H5 : (S n1)=n0
H3 : (y:var)(l (M0 y) n1)
x0 : var->var->tm
H7 : (notin x (lam [_:var](lam (x0 _))))
H8 : M0=(x0 x)
H4 : (lam (x0 x))=M
H2 : (lam (x0 x))=(N x)
H6 : N=([_:var](lam (x0 _)))
============================
 (l (lam (x0 y)) (S n1))
```

Here we are considering the case relative to the binder `lam`; hence, we must apply rule `l_lam` (`Apply l_lam; Intro.`) getting the following proof environment:

```
n : nat
n0 : nat
H : (m:nat)
     (lt m n0)
     ->(M:tm)
        (l M m)
        ->(x:var; N:(var->tm))
           (notin x (lam N))->M=(N x)->(y:var)(l (N y) m)
M : tm
H0 : (l M n0)
x : var
N : var->tm
H1 : (notin x (lam N))
y : var
M0 : var->tm
n1 : nat
H5 : (S n1)=n0
H3 : (y:var)(l (M0 y) n1)
x0 : var->var->tm
H7 : (notin x (lam [_:var](lam (x0 _))))
H8 : M0=(x0 x)
H4 : (lam (x0 x))=M
H2 : (lam (x0 x))=(N x)
H6 : N=([_:var](lam (x0 _)))
y0 : var
============================
 (l (x0 y y0) n1)
```

Naïvely applying the inductive hypothesis `H` in order to replace `y` with `x` does not work since, among the new subgoals, we have to prove `(notin x (lam [_:var](x0 _ y0)))` and this is not possible since `y0`, being generic, could be equal to `x`. The right approach consists of replacing `y0` with a new fresh variable (obtained by means of `unsat`) and then replacing `y` with `x`. These operations amount to applying two times the inductive hypothesis.

Once `L_RW` is derived, we can prove the totality of `l` w.r.t. the first argument by means of a structural induction on it:

```
Lemma L_TOT: (M:tm)(Ex [n:nat](l M n)).
```

Now, we have all the results we need in order to derive the following lemma again by a complete induction on `n` (notice the generic variable of the schematic judgment `((y:var)(P [x:var](M x y)))`):

```
Lemma PRE_HO_TM_IND: (P:(var->tm)->Prop)
                     ((x:var)(P [_:var](is_var x))) ->
                     (P is_var) ->
                     ((M,N:var->tm)(P M) -> (P N) ->
                      (P [x:var](app (M x) (N x)))
```

```
                    ) ->
                    ((M:var->var->tm)((y:var)(P [x:var](M x y))) ->
                     (P [x:var](lam (M x)))
                    ) ->
                    (n:nat)(M:tm)(l M n) ->
                    (N:var->tm)(x:var)(notin x (lam N)) ->
                    (N x)=M -> (P N).
```

The main result, i.e., the higher-order induction principle for terms of type `var->tm` can be obtained as a straightforward corollary of `PRE_HO_TM_IND`:

```
Lemma HO_TM_IND: (P:(var->tm)->Prop)
                 ((x:var)(P [_:var](is_var x))) ->
                 (P is_var) ->
                 ((M,N:var->tm)(P M) -> (P N) ->
                  (P [x:var](app (M x) (N x)))
                 ) ->
                 ((M:var->var->tm)((y:var)(P [x:var](M x y))) ->
                  (P [x:var](lam (M x)))
                 ) ->
                 (M:var->tm)(P M).
```

The axioms of $\beta$-expansion and extensionality played a fundamental rôle in proving lemmata `L_RW` and `PRE_HO_TM_IND` by "transferring" structural information from terms of type `tm` to contexts of type `var->tm`. This fact is explained in more detail in Section 6.3.1.

The whole approach can be adapted (changing the definition of the measure relation `l`) for deriving higher-order induction principles for terms of type `var -> var -> tm`, `var -> var -> var -> tm` and so on. For instance, the measure relation for unary contexts of type `var->tm` is the following:

```
Inductive ho_l  : (var->tm)->nat->Prop :=
      ho_l_var1 : (ho_l [_:var](is_var _) (S O))
    | ho_l_var2 : (x:var)(ho_l [_:var](is_var x) (S O))
    | ho_l_app  : (M,N:var->tm; n1,n2:nat)
                  (ho_l M n1)->(ho_l N n2) ->
                  (ho_l [_:var](app (M _) (N _)) (S (plus n1 n2)))
    | ho_l_lam  : (M:var->var->tm)(n:nat)
                  ((y:var)(ho_l [_:var](M _ y) n)) ->
                  (ho_l [_:var](lam (M _)) (S n)).
```

## 4.6 Related work

Recently, there has been a growing interest in studying Higher-Order Abstract Syntax or theories allowing a smooth treatment of names and binders. In this section we will briefly describe some related work, referring the interested reader to the bibliography for further details.

**The Theory of Contexts and Isabelle/HOL.** The Theory of Contexts can be used in many different logical frameworks in order to reason about higher-order abstract syntax.

A HOAS-based encoding of the syntax of $\pi$-calculus processes in Isabelle/HOL is given in [RHB01]. For types of the form $\upsilon^n \to \iota$, inductively defined well-formedness predicates delineate members that correspond to terms with free names in the object syntax. Relativised versions of the axioms of the Theory of Contexts can then be proved by induction on the definition of these well-formedness predicates.

In particular, this allows for the axioms to be proved within the theory, i.e., no non-standard interpretation of the logic is required to establish soundness. On the other hand, for each term in question one first has to assert well-formedness which in view of its defining rules is rather cumbersome from the point of view of the burden imposed on the user. It should be noticed that well-formedness predicates cannot be dropped, even if one may want to simply declare the properties of the Theory of Contexts as axioms (not deriving them). Indeed, Isabelle/HOL validates the Axiom of Unique Choice and, as a consequence, the class of functional terms modeling syntactical contexts must be restricted.

**The Nominal Logic.** A metalanguage for reasoning about languages with binders, based on the Frænkel-Mostowski permutation model of set theory, has been proposed in [GP99] and later expanded with the name of *Nominal Logic* in [Pit01b]. This logic features a special quantifier $\text{И}$ for expressing freshness of names. The intuitive meaning of $\text{И}y.p$ is "$p$ holds for $y$ a fresh name". $\text{И}$ resembles both $\forall$ and $\exists$, as it satisfies the rules:

$$\frac{\Gamma, y\#\vec{x} \vdash p}{\Gamma \vdash \text{И}y.p} \qquad \frac{\Gamma \vdash \text{И}y.p \quad \Gamma, p, y\#\vec{x} \vdash q}{\Gamma \vdash q}$$

where $\vec{x}$ is the "support" of $p$. In the Theory of Contexts, $\text{И}y.p$ and $y\#\vec{x}$ can be encoded as follows:

$$\text{И}y.p \triangleq \forall y{:}\upsilon.y \notin^{\upsilon\to o} (\lambda y{:}\upsilon.p) \Rightarrow p \qquad y\#\vec{x} \triangleq y \notin^{o} p$$

Rules, corresponding to the ones above, can then be easily derived using the Theory of Contexts. Correspondingly, suitable adaptations of our Theory of Contexts are validated in the FM.

The abstraction $(x.a)$ and instantiation $(a@x)$ operators are taken as primitives in FM. The *fresh* operator, on the other hand, cannot be encoded at the level of terms.

The main difference with our approach is that in our formulation processes with free names are modeled as functions $\upsilon \to \iota$, whereas in [GP99] they are modeled as equivalence classes of name-process pairs. In a nutshell one can say that our approach works in the standard setting of higher-order logic and type theory, allowing one to take advantage of the machinery of an existing framework like the `Coq` system, whereas a remarkable work had to be done to embed FM-sets theory in Isabelle (see [Gab00]). On the other hand, FM has the advantage that axioms about nominal calculi can be derived from more primitive concepts so that it would more easily carry over to different settings.

**Meta-metalogics.** In the approaches we discussed so far, the logical level belongs to the same metalanguage which is used for the representation of the syntax. A different perspective is to add explicitly an extra logical level for reasoning over metalogics. One of these meta-metalogic is $FO\lambda^{\Delta\mathrm{N}}$ [MM01], a higher-order intuitionistic logic extended with definitions and higher order quantification over simply typed $\lambda$-terms. Induction on types is recovered from induction on natural numbers via appropriate notions of measure.

# 5

# A functorial model for the Theory of Contexts

## 5.1 Introduction

This chapter, whose material is taken from [BHH$^+$01], is the heart of the present thesis since in the following we will prove the *consistency* of the Theory of Contexts (Theorem 5.2). More precisely, we give a model of the system $\Upsilon$ (a Classical Higher Order Logic, extended with the axioms of the Theory of Contexts, over a simple theory of types *à la Church* [Chu40]) introduced in the previous chapter. As an example of encoding, we give the interpretation of datatypes of processes and names of $\pi$-calculus. Moreover, we prove that suitable structural *induction* and *recursion principles* over contexts are validated by this model.

In order to achieve these results, we have to resort to rather sophisticated mathematical tools, such as a *tripos* over *functor categories*, like in [Hof99]. Datatypes are interpreted as (covariant) presheaves over the category of variable substitutions, while predicates are interpreted (as certain subpresheaves) in the category of presheaves over *injective* variable substitutions. As we remarked in the previous chapter (Section 4.4), the Theory of Contexts contradicts the Axiom of Unique Choice, thus making essential the use of triposes, as opposed to plain topos logic (where AC! is always validated).

Despite its complexity, the material contained in this chapter should be accessible also to non categorically minded people. Indeed, we tried to work out in full detail the novelty of the approach introduced in [Hof99] for reasoning about systems in HOAS using presheaves to model types, natural transformations to model terms, and a tripos for interpreting predicates. One of the crucial tools that we introduce to this end is a notion of *forcing* which allows us to streamline the computation of the truth value of a proposition. Our hope is that this methodology should be useful also for reasoning about other models for HOAS based on functor categories.

The idea of using functor categories for dealing with HOAS has been recently proposed also by other authors [FPT99, FT01]. Another (apparently different) solution, based on the Frænkel-Mostowski permutation model of set theory, has been presented in [GP99, GP01]; a first-order axiomatization of this model, called *Nominal Logic*, has been presented in [Pit01b]. However, as the authors of that work point out, this model could be described in a topos-theoretical setting; thus the underlying categorical structures of these approaches are strongly related.

This chapter is organized as follows. In Section 5.2 we introduce the object language we take as example, namely a small fragment of the $\pi$-calculus. In Section 5.3 we encode our example language in the framework $\Upsilon$ introduced in the previous chapter, instantiating the machinery of non-occurrence predicates and the axioms of the Theory of Contexts for our $\pi$-calculus signature. In order to help the reader not familiar with category theory, in Appendix B we give the minimum categorical definitions and notions needed in order to understand the following material. The construction of the functorial model $\mathcal{U}$ is worked out in Section 5.4 by introducing the ambient categories $\check{\mathcal{V}}$ and $\check{\mathcal{I}}$ and giving the interpretation of types, environments, typing judgments and logical judgments. The main result of the present work is carried out in Section 5.5 by means of a suitable notion of forcing. In Sections 5.6 and 5.7 we show how our model can also accommodate the useful notions of (possibly higher-order) recursion and induction. The last sections are devoted to giving some deep insights into the main categorical concepts used throughout the chapter. More precisely, in Section 5.8 we illustrate the connections with tripos theory, which allow for a more concise and elegant account of the results proved in the previous sections. A comparison of our approach with similar works in the literature is given in Section 5.9. Longer proofs are gathered in Appendix C.

## 5.2 The object language

The object language we focus on is a fragment of $\pi$-calculus, which is a process algebra introduced in [MPW92]. The $\pi$-calculus is a good example of a nominal calculus since its formalisation and fundamental design choices are strictly tied to the notion of *name*. Actually, the act of *naming* allows to naturally explain concurrency because it implies the independence of the *namer* and the *named* as coexisting entities running in parallel. Moreover, the notion of communication is tightly coupled with the concept of name (or address, port, channel etc.).

It is worthwhile noticing that, even if the fragment we have chosen lacks the computational expressiveness of the original system since it features neither synchronization nor mobility of processes, it highlights the problematic issues of reasoning about names in higher-order abstract syntax.

**Syntax.** There are two basic syntactical entities:

- *Names:* the set $\mathcal{N}$ is an infinite set of names, ranged over by $x$, $y$, $\ldots$;
- *Processes:* the set *Proc*, ranged over by $P$, $Q$, is defined by the following abstract syntax, where the operators are listed in decreasing order of precedence:

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

The *restriction* operator $(\nu x)$ binds the occurrences of $y$ in $(\nu y)P$. Thus, for each process $P$ we can define in the standard way the sets of its *free names* $\mathit{fn}(P)$, *bound names* $\mathit{bn}(P)$ and *names* $n(P) \triangleq \mathit{fn}(P) \cup \mathit{bn}(P)$. Let $X \subset \mathcal{N}$ a finite set of names; $\mathit{Proc}_X$ denotes the set $\{P \subset \mathit{Proc} \mid \mathit{fn}(P) \subseteq X\}$. Processes are taken up to $\alpha$-equivalence. Capture-avoiding substitution of a single name $y$ in place of $x$ in $P$ is denoted by $P[y/x]$. A *(process) context* is a process with a (possibly repeated) hole.

$$\frac{-}{\tau.P \longrightarrow P} \quad \text{(TAU)} \qquad \frac{P \longrightarrow P'}{(\nu y)P \longrightarrow (\nu y)P'} \quad \text{(RES)}$$

$$\frac{P \longrightarrow P'}{P|Q \longrightarrow P'|Q} \quad (\text{PAR}_1) \qquad \frac{P \longrightarrow P'}{[x \neq y]P \longrightarrow P'} x \neq y \quad \text{(MISMATCH)}$$

$$\frac{Q \longrightarrow Q'}{P|Q \longrightarrow P|Q'} \quad (\text{PAR}_2)$$

Figure 5.1: Operational semantics.

This fragment of $\pi$-calculus has been chosen by striving for simplicity in order to highlight the problematic issues of reasoning about names in higher-order abstract syntax. Thus, labels and communication primitives have been dropped, because their formalization would only introduce many obscure technical details without any substantial change in the treatment of names. On the other hand, the formalization of the mismatch operator comes "for free" because it requires only some judgments, which are needed anyway, about the free (non-)occurrence of names in processes. The $\tau$ prefix is needed in order to have a non-trivial theory of strong bisimulation: without the $\tau$, all processes would be strongly bisimilar to 0. For a formalization of the full $\pi$-calculus, see [HMS01b].

**Operational semantics.** The operational semantics of $\pi$-calculus is the relation $\longrightarrow$ which is the smallest relation over processes satisfying the rules in Figure 5.1.

**Bisimilarity.** The notion of bisimilarity is a common tool introduced in process algebras in order to define the notion of equivalence between processes. For our fragment of the $\pi$-calculus this notion can be formulated as follows:

**Definition 5.1 (Bisimilarity)** *A binary relation $S$ on processes is a* simulation *iff, for all $P, Q$ processes, if $P\ S\ Q$ and $P \longrightarrow P'$ then for some $Q'$, $Q \longrightarrow Q'$ and $P'\ S\ Q'$. $S$ is a* bisimulation *if both $S$ and $S^{-1}$ are simulations.*

*The* bisimilarity *is the binary relation $\dot\sim$ defined by*

$$P \dot\sim Q \iff \exists S.S \textit{ bisimulation and } (P\ S\ Q)\,.$$

It is well-known that bisimilarity can be defined as the greatest fixed point of a suitable monotonic operator over subsets of *Proc* $\times$ *Proc* (see [MPW92]).

The following lemmata (adapted from [MPW92] to the $\pi$-calculus fragment used in this chapter) deal directly with the notions of name and substitution:

**Lemma 3** If $P \longrightarrow P'$, then for all $y \notin fn(P)$: $P[y/x] \longrightarrow P'[y/x]$.

**Lemma 6** If $P \dot\sim Q$, then for all $y \notin fn(P, Q)$: $P[y/x] \dot\sim Q[y/x]$.

They can be regarded as instances of a more general property which, in a sense, states that the choice of particular names is not important as long as we are able to distinguish among them. This kind of property turns out to be fundamental in developing the metatheory of any nominal calculus [MPW92]. In this chapter we will not deal with the metaproperties of $\longrightarrow$ and $\dot\sim$; we refer the interested reader to [HMS01b].

$$
\begin{array}{ll}
0 : \iota & \tau : \iota \to \iota \\
| : \iota \to \iota \to \iota & [\cdot \neq \cdot]\cdot : \upsilon \to \upsilon \to \iota \to \iota \\
\nu : (\upsilon \to \iota) \to \iota &
\end{array}
$$

Figure 5.2: The signature $\Sigma$.

## 5.3 Encoding the $\pi$-calculus fragment in $\Upsilon$

In this section we briefly give the encoding of our example language in the system $\Upsilon$. Hence, we have to give the signature and instantiate the occurrence-checking predicates and the axioms and rules of the Theory of Contexts for our particular case.

The *signature* $\Sigma$ is given in Figure 5.2. The occurrence-checking predicates, the operational semantics and the bisimulation are defined impredicatively using higher-order quantifications, as in Figure 5.3.

Finally, the instantiations of the axioms at the heart of the Theory of Context are given in Figure 5.4.

The signature given so far allows for an adequate encoding of the object language introduced in Section 5.2. The corresponding terms in long $\beta\eta$-normal form are defined as follows, using infix notation:

$$x ::= x_1 \mid \ldots \mid x_n \qquad P ::= 0 \mid \tau P \mid P|Q \mid [x_1 \neq x_2]P \mid \nu\lambda y{:}\upsilon.P$$

We denote the set of such normal forms by $Proc_X$. Let $\Sigma^\iota$ be the subsignature of $\Sigma$ consisting only of the process constructors, i.e., $0$, $\tau$, $|$, $[\cdot \neq \cdot]$ and $\nu$:

**Proposition 5.1** *There is a bijective correspondence between terms of the object language with names in $X = \{x_1, \ldots, x_n\}$ and the normal forms of type $\iota$ in the signature $\Sigma^\iota$ and in the environment $\Gamma_X \triangleq \{x_1 : \upsilon, \ldots, x_n : \upsilon\}$.*

We omit the proof which follows the standard argument by induction on the syntax of terms and on the derivation of the typing judgment already depicted in Theorem 4.1.

## 5.4 The construction of model $\mathcal{U}$

Following the idea of [Hof99], we will define the interpretation of types and environments as set-valued functors from the category of finite sets of names and functions. The meaning of a term depends on the set of names which can be associated to its free variables. The functor interpreting a type, therefore, gives the set of possible values for every set of names, while its action on a function between two sets of names corresponds to the capture-avoiding substitution of names in terms. The meaning of a well-typed term is then the interpretation of its typing judgment, which is a natural transformation from the meaning of the environment to the meaning of the type. Naturality ensures that this interpretation is compatible with all possible substitutions of names for interpreting free variables.

Given a set of names for interpreting free variables, the meaning of a formula is the set of names substitutions under which it is verified. Intuitively, a valid proposition must be satisfied under all injective substitutions, since these keep distinct the meaning given to

$$
\begin{aligned}
T_{\notin} &: (\upsilon \to \iota \to o) \to (\upsilon \to \iota \to o)\\
&\triangleq \lambda R{:}\upsilon \to \iota \to o.\lambda x{:}\upsilon.\lambda P{:}\iota.P = 0 \vee\\
&\quad (\exists Q{:}\iota.P = \tau.Q \wedge (R\ x\ Q)) \vee\\
&\quad (\exists P_1{:}\iota.\exists P_2{:}\iota.P = P_1|P_2 \wedge (R\ x\ P_1) \wedge (R\ x\ P_2)) \vee\\
&\quad (\exists Q{:}\iota.\exists y{:}\upsilon.\exists z{:}\upsilon.P = [y \neq z]Q \wedge \neg x =^{\upsilon} y \wedge \neg x =^{\upsilon} z \wedge (R\ x\ Q)) \vee\\
&\quad (\exists Q{:}\upsilon \to \iota.P = \nu Q \wedge (\forall y{:}\upsilon.\neg x =^{\upsilon} y \Rightarrow (R\ x\ (Q\ y))))\\
\notin &\triangleq \lambda x{:}\upsilon.\lambda P{:}\iota.\forall p{:}\upsilon \to \iota \to o.(\forall y{:}\upsilon.\forall Q{:}\iota.(T_{\notin}\ p\ y\ Q) \Rightarrow (p\ y\ Q)) \Rightarrow (p\ x\ P)\\
\in &\triangleq \lambda x{:}\upsilon.\lambda P{:}\iota.\neg(x \notin P)\\
\notin^n &\triangleq \lambda x{:}\upsilon.\lambda P{:}\upsilon^n \to \iota.x \notin (\nu\lambda x_1{:}\upsilon.\dots\nu\lambda x_{n-1}{:}\upsilon.\nu(P\ x_1 \dots x_{n-1})) \quad (n \geq 1)
\end{aligned}
$$

$$
\begin{aligned}
T_{\longrightarrow} &: (\iota \to \iota \to o) \to (\iota \to \iota \to o)\\
&\triangleq \lambda R{:}\iota \to \iota \to o.\lambda P{:}\iota.\lambda Q{:}\iota.P = \tau.Q \vee\\
&\quad (\exists P_1{:}\iota.\exists Q_1{:}\iota.\exists S{:}\iota.P = P_1|S \wedge Q = Q_1|S \wedge (R\ P_1\ Q_1)) \vee\\
&\quad (\exists P_2{:}\iota.\exists Q_2{:}\iota.\exists S{:}\iota.P = S|P_2 \wedge Q = S|Q_2 \wedge (R\ P_2\ Q_2)) \vee\\
&\quad (\exists P'{:}\iota.\exists x{:}\upsilon.\exists y{:}\upsilon.P = [x \neq y]P' \wedge \neg x =^{\upsilon} y \wedge (R\ P'\ Q)) \vee\\
&\quad (\exists P'{:}\upsilon \to \iota.\exists Q'{:}\upsilon \to \iota.P = \nu P' \wedge Q = \nu Q' \wedge (\forall x{:}\upsilon.x \notin P' \Rightarrow (R\ (P'\ x)\ (Q'\ x))))\\
\longrightarrow &\triangleq \lambda P{:}\iota.\lambda Q{:}\iota.\forall p{:}\iota \to \iota \to o.(\forall P'{:}\iota.\forall Q'{:}\iota.(T_{\longrightarrow}\ p\ P'\ Q') \Rightarrow (p\ P'\ Q')) \Rightarrow (p\ P\ Q)
\end{aligned}
$$

$$
\begin{aligned}
T_{\dot\sim} &: (\iota \to \iota \to o) \to (\iota \to \iota \to o)\\
&\triangleq \lambda S{:}\iota \to \iota \to o.\lambda P{:}\iota.\lambda Q{:}\iota.\\
&\quad (\forall P'{:}\iota.(P \longrightarrow P') \Rightarrow \exists Q'{:}\iota.(Q \longrightarrow Q') \wedge (S\ P'\ Q')) \wedge\\
&\quad (\forall Q'{:}\iota.(Q \longrightarrow Q') \Rightarrow \exists P'{:}\iota.(P \longrightarrow P') \wedge (S\ P'\ Q'))\\
\dot\sim &\triangleq \lambda P{:}\iota.\lambda Q{:}\iota.\exists R{:}\iota \to \iota \to o.(\forall P'{:}\iota.\forall Q'{:}\iota.(R\ P'\ Q') \Rightarrow (T_{\dot\sim}\ R\ P'\ Q')) \wedge (R\ P\ Q)
\end{aligned}
$$

Figure 5.3: Logical abbreviations.

$$\frac{\Gamma \vdash_{\Sigma} P : \iota}{\Gamma \vdash_{\Sigma} \exists x{:}\upsilon.x \notin P} \qquad (Unsat^{\upsilon}_{\iota})$$

$$\frac{\Gamma \vdash_{\Sigma} P : \upsilon^{n+1} \to \iota \quad \Gamma \vdash_{\Sigma} Q : \upsilon^{n+1} \to \iota \quad \Gamma \vdash_{\Sigma} x : \upsilon}{\Gamma \vdash_{\Sigma} x \notin^{n+1} P \Rightarrow x \notin^{n+1} Q \Rightarrow (P\ x) =^{\upsilon^n \to \iota} (Q\ x) \Rightarrow P =^{\upsilon^{n+1} \to \iota} Q} \qquad (Ext^{\upsilon^{n+1} \to \iota})$$

$$\frac{\Gamma \vdash_{\Sigma} P : \upsilon^n \to \iota \quad \Gamma \vdash_{\Sigma} x : \upsilon}{\Gamma \vdash_{\Sigma} \exists Q{:}\upsilon^{n+1} \to \iota.x \notin^{n+1} Q \wedge P =^{\upsilon^n \to \iota} (Q\ x)} \qquad (\beta\_exp^{\upsilon^n \to \iota})$$

Figure 5.4: Axioms for the Theory of Contexts.

different variables. Therefore a proposition is valid if, for all sets of names, its interpretation contains at least all injective substitutions.

We will proceed as follows: in Section 5.4.1 we introduce the base categories $\check{\mathcal{V}}$ and $\check{\mathcal{I}}$, which will be used in Section 5.4.2 for interpreting the types of $\Upsilon$. The interpretation of environments and terms will be given in Sections 5.4.3 and 5.4.4, respectively. Finally in Section 5.4.5 we will give the interpretation of the logical judgment.

### 5.4.1 The ambient categories $\check{\mathcal{V}}$ and $\check{\mathcal{I}}$

In this section we introduce the categories we will use to build the model and we state some useful properties. We will mainly work in $\check{\mathcal{V}} \triangleq \mathcal{S}et^{\mathcal{V}}$, where $\mathcal{V}$ is the category whose objects are finite sets of variables, ranged over by $X, Y, Z, \ldots$, and whose morphisms are functions between them.

The intended meaning of morphisms is that of *variable substitutions*. We will use the fact that $\mathcal{V}$ has coproducts, given by disjoint union, and also that, by Yoneda Lemma[1], for all $F \in \check{\mathcal{V}}$, $\check{\mathcal{V}}(\mathbf{1}, F) \cong F_\emptyset$.

Though the abovementioned category $\check{\mathcal{V}}$ would suffice to interpret basic datatypes, in order to obtain a consistent model for our extra logical axioms ($Unsat^{v}_{\iota}$, $Ext^{v^{n+1}\to\iota}$ and $\beta\_exp^{v^n\to\iota}$), we must interpret the type of propositions $o$ in a non-standard way. Indeed, using the plain topos logic of $\check{\mathcal{V}}$ is not sufficient since it is well known that it validates the Axiom of Unique Choice which is inconsistent w.r.t. the properties of the Theory of Contexts (see Section 4.4). Hence, following [Hof99], we introduce the auxiliary notion of predicate over a given type exploiting the subcategory of $\mathcal{V}$ whose objects are the same of $\mathcal{V}$ and morphisms are injective functions. We will denote this category with $\mathcal{I}$.

The following proposition is an instance of a general result on subcategories (see [Mac71], § 10.3). It will be fundamental in the construction of the model since it "builds a bridge" between the two ambient categories we are considering, i.e., $\check{\mathcal{V}}$ and $\check{\mathcal{I}}$.

**Proposition 5.2** *There is an adjunction $((\_)^r, (\_)^*, \phi)$ from $\check{\mathcal{V}}$ to $\check{\mathcal{I}}$ with $(\_)^r$ the restriction to $\check{\mathcal{I}}$ of functors in $\check{\mathcal{V}}$ and the identity on morphisms*[2]*, $(\_)^*$ and $\phi$ defined as follows:*

$(\_)^*$: *for $G \in \check{\mathcal{I}}$, $G^* : \mathcal{V} \longrightarrow \mathcal{S}et$ is the functor whose action is*

$$G^*_X \triangleq \check{\mathcal{I}}(\mathcal{V}(X, \_)^r, G), \qquad (G^*_f(t))_Z(h) \triangleq t_Z(h \circ f),$$

*and for $s \in \check{\mathcal{V}}(F, G)$, $s^* \in \check{\mathcal{V}}(F^*, G^*)$ is the natural transformation defined by*

$$(s^*_X(m))_Y(f) \triangleq s_Y(m_Y(f)),$$

$\phi$: *for all $F \in \check{\mathcal{V}}$, $G \in \check{\mathcal{I}}$, $\phi_{FG} : \check{\mathcal{V}}(F, G^*) \cong \check{\mathcal{I}}(F^r, G)$ is defined by $(\phi_{FG}(\alpha))_X(x) \triangleq (\alpha_X(x))_X(\mathrm{id}_X)$, for all $\alpha \in \check{\mathcal{V}}(F, G^*)$, $X \in \mathcal{V}$ and $x \in F_X$.*

[1] The reader is referred to Theorem B.1 for the formal statement. In the following we will denote the Yoneda functor by $\check{\mathcal{Y}}$.

[2] More precisely, the *restriction* functor $(\_)^r : \check{\mathcal{V}} \longrightarrow \check{\mathcal{I}}$ is defined on objects by:

$$\begin{array}{rcl} F^r_X & \triangleq & F_X \\ F^r_h & \triangleq & F_h : F_X \longrightarrow F_Y \ \text{(for } h \in \mathcal{I}(X,Y)\text{)} \end{array}$$

and on morphisms $t : F \longrightarrow G$ by:

$$t^r \triangleq t$$

*Proof.* See Appendix C.0.4. □

In the following it will be useful to have the explicit definition of the inverse $\psi$ of $\phi$: for $F \in \check{\mathcal{V}}$, $G \in \check{\mathcal{I}}$, $\alpha \in \check{\mathcal{I}}(F^r, G)$, $X, Y \in \mathcal{V}$, $x \in F_X$ and $g \in \mathcal{V}(X, Y)$:

$$((\psi_{FG}(\alpha))_X(x))_Y(g) \triangleq \alpha_Y(F_g(x)) .$$

### 5.4.2 Interpreting types

#### Variables

The interpretation of *variables* is the functor $[\![v]\!] \triangleq Var : \mathcal{V} \longrightarrow \mathcal{S}et$ defined by $Var_X \triangleq X$ and, for $h \in \mathcal{V}(X,Y), x \in X$, $Var_h(x) \triangleq h(x)$. In other words, $Var$ is simply the embedding of $\mathcal{V}$ into $\mathcal{S}et$. Note that it is isomorphic to the representable functor $\check{\mathcal{Y}}(\{\star\})$.

#### Processes

The interpretation $[\![\iota]\!]$ of *processes* is given by the functor $Proc$, which is defined by extending the previous definition $Proc_X$ (denoting the set of processes with free names in $X$) with the action on morphisms. Given $h : X \longrightarrow Y$, we define $Proc_h \triangleq \sigma$, where $\sigma : Proc_X \longrightarrow Proc_Y$ is the substitution function which replaces every $X$-indeterminate $x$ in $t \in Proc_X$ with $h(x)$, yielding a term of $Proc_Y$. For this reason, sometimes in the following we will denote $Proc_h(t)$ by $t[h]$. This notation can be extended to any type, i.e., for all $A \in \check{\mathcal{V}}$, $h \in \mathcal{V}(X,Y)$, $a \in A_X$: $a[h] \triangleq A_h(a)$.

Notice that $Proc$ is not representable; indeed if this were the case, then there would be a finite set of variables $Z$ such that $[\![\iota]\!] \cong \check{\mathcal{Y}}(Z) \triangleq \mathcal{V}(Z, \_)$. From this we could infer that $[\![\iota]\!]_X \triangleq Proc_X \cong \check{\mathcal{Y}}(Z)_X \triangleq \mathcal{V}(Z, X)$, i.e., that the set of processes with free variables included in $X$ would be isomorphic to the set of finite substitutions with domain $Z$ and codomain $X$. This is clearly absurd since the cardinality of the latter set is finite and precisely $|Z| \cdot |X|$, while the cardinality of $Proc_X$ is infinite (since it is inhabited by the following succession of processes: $0,\ 0|0,\ 0|0|0, \ldots$).

#### Propositions

As anticipated, we cannot interpret propositions in the standard way. Instead we will proceed as follows:

1. a functor $\mathbf{Pred}_{\check{\mathcal{I}}} : \check{\mathcal{I}}^{op} \longrightarrow \mathcal{S}et$ with suitable properties is introduced.

2. $\mathbf{Pred}_{\check{\mathcal{I}}}$ is extended to a functor $\mathbf{Pred} : \check{\mathcal{V}}^{op} \longrightarrow \mathcal{S}et$ by means of Proposition 5.2; the adjunction ensures that the properties of $\mathbf{Pred}_{\check{\mathcal{I}}}$ we are interested in are transferred to $\mathbf{Pred}$. In particular $\mathbf{Pred}$ is representable.

3. $Prop : \mathcal{V} \longrightarrow \mathcal{S}et$ is defined as the functor representing $\mathbf{Pred}$.

The whole construction is inspired by results related to the notion of *tripos* [Pit99]. Indeed, the properties of $\mathbf{Pred}$ we are interested in essentially amount to the conditions ensuring that $\mathbf{Pred}_{\check{\mathcal{I}}}$ is a tripos on $\check{\mathcal{I}}$, so that we can interpret Higher Order Logic. However, to keep the construction of the model as elementary as possible, in this section we will not refer to tripos theory, but we will just introduce the notions needed to carry out a direct verification that our construction indeed yields a model of $\Upsilon$. In Section 5.8 we will briefly discuss how our results can be set in the general setting of tripos theory.

First of all we introduce $\mathbf{Pred}_{\check{\mathcal{I}}}$, which assigns to every functor $F \in \check{\mathcal{I}}$, a Boolean algebra of *predicates*. For this purpose we recall the following definition:

**Definition 5.2** *Given a functor $F : \mathcal{I} \longrightarrow \mathcal{S}et$, a* subfunctor *of $F$ is a $\mathcal{I}$-indexed family of sets $\{P_X\}_{X \in \mathcal{I}}$ such that*

$$\textit{for } X \in \mathcal{I} : P_X \subseteq F_X \tag{5.1}$$

$$\textit{for } h \in \mathcal{I}(X,Y), \textit{if } f \in P_X \textit{ then } F_h(f) \in P_Y. \tag{5.2}$$

*We will say that a subfunctor $P$ is* closed *if it satisfies the following*[3]*:*

$$\textit{for all } X, Y \in \mathcal{I} \textit{ and } f \in F_X,\ \textit{if } F_h(f) \in P_Y \textit{ for some } h \in \mathcal{I}(X,Y),\ \textit{then } f \in P_X \tag{5.3}$$

We will denote a subfunctor $P$ of $F$ by $P \rightarrowtail F$. With the usual abuse of language, we will identify subfunctors of $F$ with the *subobjects* of $F$.

Now let $\mathbf{Pred}_{\check{\mathcal{I}}} : \check{\mathcal{I}}^{op} \longrightarrow \mathcal{S}et$ be defined as follows:

- for $F \in \check{\mathcal{I}}^{op}$, $\mathbf{Pred}_{\check{\mathcal{I}}}(F) \triangleq \{P \in \check{\mathcal{I}} \mid P \rightarrowtail F, P \text{ is closed}\}$;
- for $\alpha \in \check{\mathcal{I}}^{op}(F,G)$ and $P \in \mathbf{Pred}_{\check{\mathcal{I}}}(G)$, $\mathbf{Pred}_{\check{\mathcal{I}}}(\alpha)(P)$ is the subfunctor of $F$ such that $(\mathbf{Pred}_{\check{\mathcal{I}}}(\alpha)(P))_X \triangleq \alpha_X^{-1}(P_X)$, and $(\mathbf{Pred}_{\check{\mathcal{I}}}(\alpha)(P))_f \triangleq F_f$.

It is a standard result that the previous conditions indeed define a functor, and moreover that the following holds:

**Proposition 5.3** *For all $F \in \check{\mathcal{I}}$, $\mathbf{Pred}_{\check{\mathcal{I}}}(F)$ is a boolean algebra w.r.t. the operations:*

$$\begin{array}{lll} 0_X \triangleq \emptyset & (U \vee V)_X \triangleq U_X \cup V_X & \\ 1_X \triangleq F_X & (U \wedge V)_X \triangleq U_X \cap V_X & (\overline{U})_X \triangleq \{f \in F_X \mid f \notin U_X\} \end{array}$$

*and moreover for all $\alpha \in \check{\mathcal{I}}^{op}(F,G)$, $\mathbf{Pred}_{\check{\mathcal{I}}}(\alpha)$ preserves all boolean operations.*

*Proof.* See Appendix C.0.5. $\square$

In the following, we will denote by $\leq$ the order naturally arising from the operations of the algebra.

Now let $\Omega \in \check{\mathcal{I}}$ be the functor defined by $\Omega_X \triangleq \mathbf{Pred}_{\check{\mathcal{I}}}(\mathcal{I}(X,\_))$ and $\Omega_f \triangleq \mathbf{Pred}_{\check{\mathcal{I}}}(\_ \circ f)$. (The notation is reminiscent of the fact that this is the subobject classifier in the topos of $\neg\neg$-sheaves over $\mathcal{I}$.) Then:

**Proposition 5.4** $\mathbf{Pred}_{\check{\mathcal{I}}}$ *and $\check{\mathcal{I}}(\_,\Omega)$ are naturally isomorphic, so $\mathbf{Pred}_{\check{\mathcal{I}}}$ is representable.*

*Proof.* See Appendix C.0.6. $\square$

We recall here the definition of the isomorphism of the previous proposition, since it will be useful in the rest of the chapter: $\chi^{\check{\mathcal{I}}} : \mathbf{Pred}_{\check{\mathcal{I}}} \longrightarrow \check{\mathcal{I}}(\_,\Omega)$ and $\kappa^{\check{\mathcal{I}}} : \check{\mathcal{I}}(\_,\Omega) \longrightarrow \mathbf{Pred}_{\check{\mathcal{I}}}$ are defined by:

$$\begin{array}{rcl} (\chi^{\check{\mathcal{I}}}_F(U))_X(t) & \triangleq & \{\{f \in \mathcal{I}(X,Y) \mid F_f(t) \in U_Y\}\}_{Y \in \mathcal{I}} \\ \kappa^{\check{\mathcal{I}}}_F(m) & \triangleq & \left\{\{f \in F_X \mid m_X(f) = \check{\mathcal{Y}}_{\check{\mathcal{I}}}(X)\}\right\}_{X \in \mathcal{I}}. \end{array}$$

Now let us proceed to define the functor $\mathbf{Pred} : \check{\mathcal{V}}^{op} \longrightarrow \mathcal{S}et$ by setting $\mathbf{Pred}(F) \triangleq \mathbf{Pred}_{\check{\mathcal{I}}}(F^r)$ and $\mathbf{Pred}(\alpha) \triangleq \mathbf{Pred}_{\check{\mathcal{I}}}(\alpha^r)$. By Propositions 5.2 and 5.4 we have the following natural isomorphisms:

[3] A we will see in Section 5.8, closed subfunctors of $F$ are precisely the double negation closed predicates in the topos logic of $\check{\mathcal{I}}$.

- for all $f \in \check{\mathcal{V}}$, $\quad \mathbf{Pred}_{\check{\mathcal{I}}}(F^r) \xrightarrow[\chi^{\check{\mathcal{I}}}_{F^r}]{\sim} \check{\mathcal{I}}(F^r, \Omega) \xrightarrow[\psi_{F,\Omega}]{\sim} \check{\mathcal{V}}(F, \Omega^*)$

- for all $X \in \mathcal{V}$, $\quad \gamma_X \triangleq \kappa^{\check{\mathcal{I}}}_{\mathcal{V}(X,\_)^r} : (\Omega^*)_X = \check{\mathcal{I}}(\mathcal{V}(X,\_)^r, \Omega) \longrightarrow \mathbf{Pred}_{\check{\mathcal{I}}}(\mathcal{V}(X,\_)^r)$.

Let *Prop* be defined by

$$Prop_X \triangleq \mathbf{Pred}(\mathcal{V}(X,\_)) \qquad Prop_f \triangleq \mathbf{Pred}(\mathcal{V}(f,\_)) = f \circ \_.$$

Then we obtain natural isomorphisms $\chi{:}\mathbf{Pred} \longrightarrow \check{\mathcal{V}}(\_, Prop)$ and $\kappa{:}\check{\mathcal{V}}(\_, Prop) \longrightarrow \mathbf{Pred}$ given by

$$\begin{array}{rcl}(\chi_F(U))_X(t) & \triangleq & (\gamma \circ \psi_{F,\Omega}(\chi^{\check{\mathcal{I}}}_{F^r}(U)))_X(t) = \{\{g \in \mathcal{V}(X,Y) \mid F_g(t) \in U_Y\}\}_{Y \in \mathcal{V}}, \\ \kappa_F(m) & \triangleq & \kappa^{\check{\mathcal{I}}}_{F^r}(\phi_{F,\Omega}(\gamma^{-1} \circ m)) = \{\{f \in F_X \mid m_X(f) \geq \mathcal{I}(X,\_)\}\}_{X \in \mathcal{V}}.\end{array}$$

Now we can define the interpretation of the type of propositions as $[\![o]\!] \triangleq Prop$, i.e. the object representing **Pred**.

#### Simple Types

The interpretation of $[\![\sigma \to \sigma']\!]$ is given by the functor $[\![\sigma]\!] \Rightarrow [\![\sigma']\!]$ where $\Rightarrow$ is the exponential in $\check{\mathcal{V}}$. For example the interpretation of $\iota$ schemata over $\upsilon$, is given by the functor $[\![\upsilon]\!] \Rightarrow [\![\iota]\!]$ and hence since $[\![\upsilon]\!]$ is representable, by Proposition B.2 we have $([\![\upsilon]\!] \Rightarrow [\![\iota]\!])_X = ([\![\iota]\!]^{\{x\}})_X = [\![\iota]\!]_{X \uplus \{x\}}$.

### 5.4.3 Interpreting environments

The interpretation $[\![\Gamma]\!]$ of an environment $\Gamma \triangleq \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ is given by the functor $\prod_{i=1}^n [\![\sigma_i]\!]$, so:

$$[\![\Gamma]\!]_X \triangleq \prod_{i=1}^n [\![\sigma_i]\!]_X \qquad [\![\Gamma]\!]_f \triangleq [\![\sigma_1]\!]_f \times \cdots \times [\![\sigma_n]\!]_f.$$

### 5.4.4 Interpreting the typing judgment of terms

Typing judgments of the form $\Gamma \vdash M : \sigma$ will be interpreted as suitable natural transformations with domain $[\![\Gamma]\!]$ and codomain $[\![\sigma]\!]$. We shall give the interpretation of the typing judgment by induction on the depth of the derivation of $\Gamma \vdash M : \sigma$. In general, this make sense if there exists at most one derivation for each typing judgment (this fact can be easily verified proceeding by induction on the depth of the derivation of the typing judgment itself).

**Rule VAR:** $[\![x_1 : \sigma_1, \ldots, x_i : \sigma_i, \ldots, x_n : \sigma_n \vdash_\Sigma x_i : \sigma_i]\!] \triangleq \pi_i : \prod_{j=1}^n [\![\sigma_j]\!] \longrightarrow [\![\sigma_i]\!]$

**Rule CONST:** for interpreting the judgments involving constants in $\Sigma$ we introduce the following natural transformations (naturality is trivial to prove):

$$\begin{aligned} nil : \mathbf{1} &\longrightarrow Proc \\ nil_X : \mathbf{1}_X &\longrightarrow Proc_X \\ * &\longmapsto 0 \end{aligned} \qquad \begin{aligned} mismatch : Var \times Var \times Proc &\longrightarrow Proc \\ mismatch_X : X \times X \times Proc_X &\longrightarrow Proc_X \\ \langle x, y, P\rangle &\longmapsto [x \neq y]P \end{aligned}$$

$$\begin{aligned} tau : Proc &\longrightarrow Proc \\ tau_X : Proc_X &\longrightarrow Proc_X \\ P &\longmapsto \tau.P \end{aligned} \qquad \begin{aligned} par : Proc \times Proc &\longrightarrow Proc \\ par_X : Proc_X \times Proc_X &\longrightarrow Proc_X \\ \langle P, Q\rangle &\longmapsto P \mid Q \end{aligned}$$

$$\begin{aligned} new : Var \Rightarrow Proc &\longrightarrow Proc \\ new_X : \check{\mathcal{V}}(Var \times \mathcal{V}(X, \_), Proc) &\longrightarrow Proc_X \\ \alpha &\longmapsto (\nu x)(\alpha_{X \uplus \{x\}}(\langle x, in_X\rangle)) \end{aligned}$$

where $in_X : X \longrightarrow X \uplus \{x\}$ is the left injection.

Now, we can interpret judgments of kind $\Gamma \vdash_\Sigma c : \sigma$ for $c : \sigma \in \Sigma$. Let $!_{[\![\Gamma]\!]}$ be the unique morphism from $[\![\Gamma]\!]$ to $\mathbf{1}$; then:

- $[\![\Gamma \vdash_\Sigma 0 : \iota]\!] \triangleq nil \circ !_{[\![\Gamma]\!]}$, i.e., the constant natural transformation always picking the term $0 \in Proc_X$ for all $X$ and $\eta \in [\![\Gamma]\!]_X$:

$$\begin{aligned} [\![\Gamma \vdash_\Sigma 0 : \iota]\!]_X : [\![\Gamma]\!]_X &\longrightarrow Proc_X \\ \eta &\longmapsto 0 \end{aligned}$$

- $[\![\Gamma \vdash_\Sigma \tau : \iota \to \iota]\!] \triangleq \gamma_\tau(tau) \circ\ !_{[\![\Gamma]\!]}$, where $\gamma_\tau : \check{\mathcal{V}}(Proc, Proc) \longrightarrow \check{\mathcal{V}}(\mathbf{1}, Proc \Rightarrow Proc)$ is the natural isomorphism given from cartesian closedness of $\check{\mathcal{V}}$, hence:

$$\begin{aligned} ([\![\Gamma \vdash_\Sigma \tau : \iota \to \iota]\!]_X(\eta))_Y : Proc_Y \times \mathcal{V}(X, Y) &\longrightarrow Proc_Y \\ \langle P, f\rangle &\longmapsto tau_Y(P) \end{aligned}$$

- $[\![\Gamma \vdash_\Sigma \mid : \iota \to \iota \to \iota]\!] \triangleq \gamma_|(par) \circ\ !_{[\![\Gamma]\!]}$, where $\gamma_| : \check{\mathcal{V}}(Proc \times Proc, Proc) \longrightarrow \check{\mathcal{V}}(\mathbf{1}, Proc \Rightarrow Proc \Rightarrow Proc)$ is the natural isomorphism given from cartesian closedness of $\check{\mathcal{V}}$, hence:

$$\begin{aligned} ([\![\Gamma \vdash_\Sigma \mid : \iota \to \iota \to \iota]\!]_X(\eta))_Y : Proc_Y \times \mathcal{V}(X, Y) &\longrightarrow Proc \times \mathcal{V}(Y, \_) \\ \langle P, f\rangle &\longmapsto m_Y(P), \end{aligned}$$

  where

$$\begin{aligned} (m_Y(P))_Z : Proc_Z \times \mathcal{V}(Y, Z) &\longrightarrow Proc_Z \\ \langle Proc_g(P), Q\rangle &\longmapsto par_Z(\langle Proc_g(P), Q\rangle) \end{aligned}$$

- $[\![\Gamma \vdash_\Sigma [\cdot \neq \cdot]\cdot : \upsilon \to \upsilon \to \iota \to \iota]\!] \triangleq \gamma_{[\cdot\neq\cdot]\cdot}(mismatch) \circ\ !_{[\![\Gamma]\!]}$, where $\gamma_{[\cdot\neq\cdot]\cdot} : \check{\mathcal{V}}(Var \times Var \times Proc, Proc) \longrightarrow \check{\mathcal{V}}(\mathbf{1}, Var \Rightarrow Var \Rightarrow Proc \Rightarrow Proc)$ is the natural isomorphism given from cartesian closedness of $\check{\mathcal{V}}$, hence:

$$\begin{aligned} (([\![\Gamma \vdash_\Sigma [\cdot \neq \cdot]\cdot : \upsilon \to \upsilon \to \iota \to \iota]\!]_X(\eta))_Y : Proc_Y \times \mathcal{V}(X, Y) &\longrightarrow Var \times \mathcal{V}(Y, \_) \\ \langle a, f\rangle &\longmapsto m_Y(a), \end{aligned}$$

where

$$\begin{aligned}(m_Y(a))_Z : Var_Z \times \mathcal{V}(Y,Z) &\longrightarrow Proc_Z \times \mathcal{V}(Z,\_)\\ \langle b,g\rangle &\longmapsto n_Z(\langle g(a),b\rangle)\end{aligned}$$

and

$$\begin{aligned}(n_Z(\langle g(a),b\rangle))_U : Proc_Z \times \mathcal{V}(Z,U) &\longrightarrow Proc_U\\ \langle P,h\rangle &\longmapsto mismatch_U(\langle h(g(a)),h(b),P\rangle)\end{aligned}$$

- $[\![\Gamma \vdash_\Sigma \nu : (\upsilon \to \iota) \to \iota]\!] \triangleq \gamma_\nu(new)\circ\, !_{[\![\Gamma]\!]}$, where $\gamma_\nu : \check{\mathcal{V}}(Var \Rightarrow Proc, Proc) \longrightarrow \check{\mathcal{V}}(\mathbf{1}, (Var \Rightarrow Proc) \Rightarrow Proc)$ is the natural isomorphism given from cartesian closedness of $\check{\mathcal{V}}$, hence:

$$\begin{aligned}([\![\Gamma \vdash_\Sigma \nu{:}(\upsilon \to \iota) \to \iota]\!]_X(\eta))_Y : (Var \Rightarrow Proc)_Y \times \mathcal{V}(X,Y) &\longrightarrow Var \times \mathcal{V}(Y,\_)\\ \langle P,f\rangle &\longmapsto new_Y(P).\end{aligned}$$

**Rule APP:** given $t_1 = [\![\Gamma \vdash_\Sigma M : \sigma' \to \sigma]\!] : [\![\Gamma]\!] \longrightarrow ([\![\sigma']\!] \Rightarrow [\![\sigma]\!])$ and $t_2 = [\![\Gamma \vdash_\Sigma N : \sigma']\!] : [\![\Gamma]\!] \longrightarrow [\![\sigma']\!]$, we define

$$[\![\Gamma \vdash_\Sigma MN : \sigma]\!] \triangleq ev_{[\![\sigma]\!],[\![\sigma']\!]} \circ \langle t_1, t_2\rangle : [\![\Gamma]\!] \longrightarrow [\![\sigma]\!],$$

**Rule ABS:** given $t = [\![\Gamma, x:\sigma \vdash_\Sigma M : \sigma']\!] : [\![\Gamma]\!] \times [\![\sigma]\!] \longrightarrow [\![\sigma']\!]$, we define

$$[\![\Gamma \vdash_\Sigma \lambda x{:}\sigma.M : \sigma \to \sigma']\!] \triangleq \ulcorner t \urcorner,$$

where $\ulcorner t \urcorner : [\![\Gamma]\!] \longrightarrow ([\![\sigma]\!] \Rightarrow [\![\sigma']\!])$ is the exponential transpose of $t$ (Appendix B)

**Rule $\Rightarrow$:** $[\![\Gamma \vdash_\Sigma p \Rightarrow q : o]\!] = imp \circ \langle [\![\Gamma \vdash_\Sigma p : o]\!], [\![\Gamma \vdash_\Sigma q : o]\!]\rangle$, where

$$\begin{aligned}imp_X :\ Prop_X \times Prop_X &\longrightarrow Prop_X\\ \langle U,V\rangle &\longmapsto \overline{U} \vee V.\end{aligned}$$

**Rule $\forall$:** $[\![\Gamma \vdash_\Sigma \forall_\sigma p : o]\!] = forall_\sigma \circ [\![\Gamma \vdash_\Sigma p : \sigma \to o]\!]$, where

$$\begin{aligned}(forall_\sigma)_X :\ ([\![\sigma]\!] \Rightarrow Prop)_X &\longrightarrow Prop_X\\ m &\longmapsto \forall_\pi(\kappa_{[\![\sigma]\!]\times\check{\mathcal{Y}}(X)}(m))\end{aligned}$$

and $m$ is a natural transformation from $[\![\sigma]\!]\times\check{\mathcal{Y}}(X)$ to $Prop$ (remember that $([\![\sigma]\!] \Rightarrow Prop)_X \triangleq \check{\mathcal{V}}([\![\sigma]\!] \times \check{\mathcal{Y}}(X), Prop))$, $\pi : [\![\sigma]\!] \times \check{\mathcal{Y}}(X) \longrightarrow \check{\mathcal{Y}}(X)$ is the projection and, for $F \in \mathbf{Pred}([\![\sigma]\!] \times \check{\mathcal{Y}}(X))$,

$$\forall_\pi(F) \triangleq \{\{f \in \mathcal{V}(X,Y) \mid \forall g \in \mathcal{I}(Y,Z).\pi_Z^{-1}(g \circ f) \subseteq F_Z\}\}_{Y\in\mathcal{V}}.$$

More explicitly

$$(forall_\sigma)_X(m) = \left\{u \in \mathcal{V}(X,Y) \,|\, \forall g \in \mathcal{I}(Y,Z).\forall t \in [\![\sigma]\!]_Z.\langle t, g\circ u\rangle \in \kappa_{[\![\sigma]\!]\times\check{\mathcal{Y}}(X)}(m)_Z\right\}_{Y\in\mathcal{V}}$$

*Remark.* Notice that, if $[\![\Gamma \vdash_\Sigma M : \sigma]\!]$ is defined and $x \notin dom(\Gamma)$, then $[\![\Gamma, x{:}\sigma' \vdash_\Sigma M : \sigma]\!]$ is the following natural transformation:

$$\begin{aligned}([\![\Gamma, x{:}\sigma' \vdash_\Sigma M : \sigma]\!])_X : [\![\Gamma]\!]_X \times [\![\sigma']\!]_X &\longrightarrow [\![\sigma]\!]_X\\ \langle \eta, \eta_x\rangle &\longmapsto [\![\Gamma \vdash_\Sigma M : \sigma]\!]_X(\eta)\end{aligned}$$

This means that the model $\Upsilon$ admits the weakening rule (i.e., it is sound).

We end this section by making explicit the interpretations of processes which derive immediately from these definitions [Pit00]. Let $\Gamma$ be an environment, $x, y$ be variables, and $P, Q$ be terms. Then:

- $[\![\Gamma \vdash_\Sigma 0 : \iota]\!] = nil \circ !_{[\![\Gamma]\!]}$,
- $[\![\Gamma \vdash_\Sigma \tau P : \iota]\!] = tau \circ [\![\Gamma \vdash_\Sigma P : \iota]\!]$,
- $[\![\Gamma \vdash_\Sigma P|Q : \iota]\!] = par \circ \langle [\![\Gamma \vdash_\Sigma P : \iota]\!], [\![\Gamma \vdash_\Sigma Q : \iota]\!] \rangle$,
- $[\![\Gamma \vdash_\Sigma [x \neq y]P : \iota]\!] = mismatch \circ \langle [\![\Gamma \vdash_\Sigma x : \upsilon]\!], [\![\Gamma \vdash_\Sigma y : \upsilon]\!], [\![\Gamma \vdash_\Sigma P : \iota]\!] \rangle$,
- $[\![\Gamma \vdash_\Sigma \nu\lambda x{:}\upsilon.P : \iota]\!] = new \circ [\![\Gamma \vdash_\Sigma \lambda x{:}\upsilon.P : \upsilon \to \iota]\!]$.

### 5.4.5 Interpreting logical judgments

As we said before, intuitively a proposition is valid iff it is verified under all injective substitutions of names. In our model, this means that the interpretation of a valid proposition contains all injective substitutions. More formally, let $[\![\Gamma \vdash_\Sigma p : o]\!]^{\upharpoonleft}$ and $\top$ be defined by

$$\begin{array}{rcccl}
[\![\Gamma \vdash_\Sigma p : o]\!]^{\upharpoonleft} & : & [\![\Gamma]\!] & \longrightarrow & Prop \\
[\![\Gamma \vdash_\Sigma p : o]\!]^{\upharpoonleft}_X & : & [\![\Gamma]\!]_X & \longrightarrow & Prop_X \\
 & & \eta & \longmapsto & [\![\Gamma \vdash_\Sigma p : o]\!]_X(\eta) \wedge \mathcal{I}(X, _) \\
\top & : & \mathbf{1} & \longrightarrow & Prop \\
\top_X & : & \mathbf{1}_X & \longrightarrow & Prop_X \\
 & & * & \longmapsto & \mathcal{I}(X, _)
\end{array}$$

Then we give the following definition:

**Definition 5.3 (Validity)** *We say that* $\Gamma \vdash_\Sigma p$ holds *in* $\mathcal{U}$ *if* $[\![\Gamma \vdash_\Sigma p : o]\!]^{\upharpoonleft}$ *is the constant natural transformation*

$$\begin{aligned}
(True_\Gamma)_X : [\![\Gamma]\!]_X &\longrightarrow Prop_X \\
\eta &\longmapsto \mathcal{I}(X, _)
\end{aligned}$$

This is equivalent to saying that $[\![\Gamma \vdash_\Sigma p : o]\!]^{\upharpoonleft} = \top \circ !_{[\![\Gamma]\!]}$:

$$\begin{tikzcd}
\kappa_{[\![\Gamma]\!]}([\![\Gamma \vdash_\Sigma p : o]\!]) \arrow[rr, "!_{\kappa_{[\![\Gamma]\!]}([\![\Gamma \vdash_\Sigma p:o]\!])}"] \arrow[d, tail, "\overline{[\![\Gamma \vdash_\Sigma p:o]\!]^{\upharpoonleft}}"'] & & \mathbf{1} \arrow[d, "\top"] \\
[\![\Gamma]\!] \arrow[urr, "!_{[\![\Gamma]\!]}" description] \arrow[rr, "[\![\Gamma \vdash_\Sigma p:o]\!]^{\upharpoonleft}"'] & & Prop
\end{tikzcd}$$

where $(\kappa_{[\![\Gamma]\!]}([\![\Gamma \vdash_\Sigma p : o]\!]), \overline{[\![\Gamma \vdash_\Sigma p : o]\!]^{\upharpoonleft}}, !_{\kappa_{[\![\Gamma]\!]}([\![\Gamma \vdash_\Sigma p:o]\!])})$ is the pullback of $(Prop, [\![\Gamma \vdash_\Sigma p : o]\!]^{\upharpoonleft}, !_{[\![\Gamma]\!]})$. Notice that in this case we have $\kappa_{[\![\Gamma]\!]}([\![\Gamma \vdash_\Sigma p : o]\!]) = 1 \in \mathbf{Pred}([\![\Gamma]\!])$. One should also note that $\kappa_{[\![\Gamma]\!]}([\![\Gamma \vdash_\Sigma p : o]\!]) = \kappa_{[\![\Gamma]\!]}([\![\Gamma \vdash_\Sigma p : o]\!]^{\upharpoonleft})$; indeed we have the following:

$$\begin{aligned}
\kappa_{[\![\Gamma]\!]}([\![\Gamma \vdash_\Sigma p : o]\!]) &\triangleq \{\{f \in [\![\Gamma]\!]_X \mid \mathcal{I}(X, _) \leq [\![\Gamma \vdash_\Sigma p : o]\!]_X(f)\}\}_{X \in \mathcal{V}} \\
&= \{\{f \in [\![\Gamma]\!]_X \mid \mathcal{I}(X, _) \leq [\![\Gamma \vdash_\Sigma p : o]\!]_X(f) \wedge \mathcal{I}(X, _)\}\}_{X \in \mathcal{V}} \\
&\triangleq \kappa_{[\![\Gamma]\!]}([\![\Gamma \vdash_\Sigma p : o]\!]^{\upharpoonleft})
\end{aligned}$$

## 5.5 $\mathcal{U}$ is a model of $\Upsilon$

In this section we verify that the model defined in Section 5.4 validates the axioms and rules of the framework $\Upsilon$. In order to be able to streamline the computation of the truth value of a judgment $\Gamma \vdash_\Sigma p$ in the model $\mathcal{U}$, in Section 5.5.1 we introduce an appropriate notion of *forcing*. By means of this useful tool, in Section 5.5.2 we will give a characterisation of Leibniz equality; finally, in Sections 5.5.3 and 5.5.4 we will verify that $\mathcal{U}$ is a model of Classical Higher-Order Logic and of the Theory of Contexts, respectively.

### 5.5.1 Forcing

**Definition 5.4** *Forcing judgments are statements of the shape*

$$X \Vdash_{F,\eta} U$$

*for $X \in \mathcal{V}$, $F \in \check{\mathcal{V}}$, $U \in \mathbf{Pred}(F)$, and $\eta \in F_X$. The intended meaning of $X \Vdash_{F,\eta} U$ is that $\eta \in U_X$.*

When $F = [\![\Gamma]\!]$, $U = \kappa_{[\![\Gamma]\!]}([\![\Gamma \vdash_\Sigma p : o]\!])$ and $\eta \in [\![\Gamma]\!]_X$, we will also write $X \Vdash_{\Gamma,\eta} p$ instead of $X \Vdash_{\Gamma,\eta} \kappa_{[\![\Gamma]\!]}([\![\Gamma \vdash_\Sigma p : o]\!])$. We will write $X \Vdash p$ to denote "for any $\Gamma$ such that $\Gamma \vdash_\Sigma p : o$, for all $\eta \in [\![\Gamma]\!]_X$: $X \Vdash_{\Gamma,\eta} p$".

Hence we can rephrase the condition for a logical judgment to be valid in terms of the forcing relation, namely

**Proposition 5.5** *The judgment $\Gamma \vdash_\Sigma p$ holds in $\mathcal{U}$ iff for all $X \in \mathcal{V}$ and for all $\eta \in [\![\Gamma]\!]_X$ we have $X \Vdash_{\Gamma,\eta} p$.*

**Lemma 5.1** *Let $P \in \mathbf{Pred}(\check{\mathcal{Y}}(X))$ such that $P \not\geq \mathcal{I}(X,\_)$, then $P_Y \cap \mathcal{I}(X,Y) = \emptyset$ for all $Y \in \mathcal{V}$.*

*Proof.* We proceed by an absurdity argument: let us suppose that $P \not\geq \mathcal{I}(X,\_)$ and there exists $Y \in \mathcal{V}$ and $f \in \mathcal{I}(X,Y)$ such that $f \in P_Y$, we will show that, given any $Z \in \mathcal{V}$ and $g \in \mathcal{I}(X,Z)$, $g \in P_Z$.

Indeed, by condition 5.3 (satisfied by predicates), we have that $\mathrm{id}_X \in P_X$ since $\check{\mathcal{Y}}(X)_f(\mathrm{id}_X) = f \circ \mathrm{id}_X = f \in P_Y$. Then, by condition 5.2 (satisfied by predicates), we have that, for all $g \in \mathcal{I}(X,Z)$, $g \in P_Z$ since $\check{\mathcal{Y}}(X)_g(\mathrm{id}_X) = g \circ \mathrm{id}_X = g \in P_Z$. □

A number of useful propositions can now be easily stated.

**Theorem 5.1** *For all $X$, $\Gamma$, $\eta \in [\![\Gamma]\!]_X$,*

1. *$X \Vdash_{\Gamma,\eta} \forall x{:}\sigma.p$ if and only if for all $Y$, $h \in \mathcal{I}(X,Y)$, and for all $a \in [\![\sigma]\!]_Y$ we have that $Y \Vdash_{(\Gamma,x:\sigma),\ \langle [\![\Gamma]\!]_h(\eta),a\rangle} p$;*
2. *$X \Vdash_{\Gamma,\eta} p \Rightarrow q$ if and only if $X \Vdash_{\Gamma,\eta} p$ implies $X \Vdash_{\Gamma,\eta} q$;*
3. *$X \Vdash_{\Gamma,\eta} PM$ iff $\langle [\![\Gamma \vdash_\Sigma M : \sigma]\!]_X(\eta), \mathrm{id}_X\rangle \in \kappa_{[\![\sigma]\!]\times\check{\mathcal{Y}}(X)}([\![\Gamma \vdash_\Sigma P : \sigma \to o]\!]_X(\eta))$, iff $([\![\Gamma \vdash_\Sigma P : \sigma \to o]\!]_X(\eta))_X(\langle [\![\Gamma \vdash_\Sigma M : \sigma]\!]_X(\eta), \mathrm{id}_X\rangle) \geq \mathcal{I}(X,\_)$.*

*Proof.* See Appendix C.0.7. □

The next theorem is the main achievement of this thesis, since it states the consistency of our model:

**Theorem 5.2 (Consistency)** *It is never the case that* $X \Vdash_{\Gamma,\eta} \bot$.

*Proof.* By definition of $\bot$ the statement we have to prove is equivalent to $X \Vdash_{\Gamma,\eta} \forall r{:}o.r$. It follows, by the first point of Theorem 5.1, that we have to show that there exist $Y$, $h \in \mathcal{I}(X,Y)$ and $a \in Prop_Y$ such that it is not the case that $Y \Vdash_{(\Gamma,r:o),\langle [\![\Gamma]\!]_h(\eta),a\rangle} r$, i.e., that $[\![\Gamma, r : o \vdash_\Sigma r : o]\!]_Y(\langle [\![\Gamma]\!]_h(\eta), a\rangle) = a \not\geq \mathcal{I}(Y, \_)$. Hence, it is sufficient to take $a = \mathbf{0}$ (i.e., the initial object of $\check{\mathcal{I}}$) to conclude the proof. □

**Corollary 5.1** *1.* $X \Vdash_{\Gamma,\eta} \neg p$ *if and only if it is not the case that* $X \Vdash_{\Gamma,\eta} p$*;*

*2.* $X \Vdash_{\Gamma,\eta} p \wedge q$ *if and only if* $X \Vdash_{\Gamma,\eta} p$ *and* $X \Vdash_{\Gamma,\eta} q$*;*

*3.* $X \Vdash_{\Gamma,\eta} p \vee q$ *if and only if* $X \Vdash_{\Gamma,\eta} p$ *or* $X \Vdash_{\Gamma,\eta} q$*;*

*4.* $X \Vdash_{\Gamma,\eta} \exists x{:}\sigma.p$ *if and only if there exist* $Y$*,* $h \in \mathcal{I}(X,Y)$ *and* $a \in [\![\sigma]\!]_Y$ *such that* $Y \Vdash_{(\Gamma,x:\sigma),\ \langle [\![\Gamma]\!]_h(\eta),a\rangle} p$*.*

*5.* $X \Vdash_{\Gamma,\eta} \forall x_1{:}\sigma_1.\dots\forall x_n{:}\sigma_n.p$ *if and only if for all* $Y$*,* $f \in \mathcal{I}(X,Y)$*,* $\eta_1 \in [\![\sigma_1]\!]_Y, \dots \eta_n \in [\![\sigma_n]\!]_Y$ *we have that* $Y \Vdash_{(\Gamma,x_1:\sigma_1,\dots,x_n:\sigma_n),\langle [\![\Gamma]\!]_f(\eta),\eta_1,\dots,\eta_n\rangle} p$*.*

*Proof.* See Appendix C.0.8. □

### 5.5.2 Characterisation of Leibniz equality

**Definition 5.5 (Separatedness)** *An object* $F \in \check{\mathcal{V}}$ *is said to be* separated *if its diagonal* $\Delta_F \leq F \times F$*, defined as* $(\Delta_F)_X \triangleq \{\langle a, a\rangle \mid a \in F_X\}$ *and* $(\Delta_F)_h \triangleq F_h \times F_h$ *, is a predicate of* $F \times F$*, i.e.,* $\Delta_F \in \mathbf{Pred}(F \times F)$*.*

This definition is equivalent to those usually given about sheaves in textbooks, in the case of sheaves for the $\neg\neg$ topology; see e.g. [MM92], p.227 and Lemma V.3.3. As will be shown below, for separated objects Leibniz equality coincides with true equality. We have the following useful result:

**Lemma 5.2** *A is separated if and only if for each injective map* $i \in \mathcal{I}(X,Y)$ *the function* $A_i : A_X \to A_Y$ *is injective.*

*Proof.* ($\Rightarrow$) By definition, if $A$ is separated, then $\Delta_A$ is a predicate of $A \times A$. Hence, by the closure condition 5.3, we have that for all $X$, $Y \in \mathcal{I}$ and $f = \langle a, b\rangle \in A_X \times A_X$, if $(A \times A)_h(f) \in (\Delta_A)_Y$ for some $h \in \mathcal{I}(X,Y)$, then $f \in (\Delta)_X$. Observing that $(A \times A)_h(f) = \langle A_h(a), A_h(b)\rangle$, $A_h(a) = A_h(b)$ (since $(A \times A)_h(f) \in (\Delta_A)_Y$) and $a = b$ (since $f \in (\Delta)_X$), we have proved that $A_h$ is injective for a generic $h \in \mathcal{I}(X,Y)$.

($\Leftarrow$) It is trivial to verify that $\Delta_A$ satisfies both condition 5.1 and condition 5.2. For the closure condition, we observe that, for all $X$, $Y \in \mathcal{I}$ and $f = \langle a, b\rangle \in A_X \times A_X$, if $(A \times A)_h(f) \in (\Delta_A)_Y$ for some $h \in \mathcal{I}(X,Y)$, then we must have $A_h(a) = A_h(b)$. At this point, since we know that $A_h$ is injective, we can deduce that $a = b$ holds, whence $f \in (\Delta_A)_X$. □

Notice that, if $i$ has a left inverse $p$, then it is obvious that $A_i$ is injective since in this case $A_p$ is a left inverse to $A_i$ by functoriality. So, to establish separatedness, it suffices to check injectivity of $A_?$ where $? : \emptyset \to X$ is the empty function. For example, the presheaf $A$ given by $A_\emptyset = \{0, 1\}$ and $A_X = \{0\}$ otherwise fails to be separated since $A_?$ is not injective.

**Lemma 5.3** *The objects Var, Proc and Prop are separated. If G is separated, so is* $F \Rightarrow G$*.*

*Proof.* If $i \in \mathcal{I}(X,Y)$ and $x \in Var_X = X$ then $Var_i(x) = i(x)$ which is clearly injective. Similarly, if $p \in Proc_X$ then $Proc_i(p) = p[i]$ which again is injective.

For $Prop$ we appeal to the above analysis and merely check that $Prop_?$ is injective. Indeed, $Prop_\emptyset$ contains exactly two elements corresponding to $\top$ and $\bot$ which are never identified.

Finally, assume $u, v \in (F \Rightarrow G)_X = \check{\mathcal{V}}(\check{\mathcal{Y}}(X) \times F, G)$, let $i : X \to Y$ be injective and assume $(F \Rightarrow G)_i(u) = (F \Rightarrow G)_i(v)$. To show $u = v$ assume a—not necessarily injective—map $f : X \to X'$ and $a \in F_{X'}$. We must show $u(f,a) = v(f,a)$. Now, we can find an injective map $j : X' \to Y'$ and arbitrary map $g : Y \to Y'$ such that $g \circ i = j \circ f$. Since $G$ is separated, it suffices to show $G_j(u(f,a)) = G_j(v(f,a))$. But, $G_j(u(f,a)) = u(j \circ f, F_j(a)) = u(g \circ i, F_j(a)) = (F \Rightarrow G)_i(u)(g, F_j(a))$ which yields the desired conclusion by assumption and symmetry. $\Box$

**Corollary 5.2** *For all types $\sigma$, $[\![\sigma]\!]$ is separated.*

**Theorem 5.3** *For all $\sigma$, $\Gamma$, $M, N$, $X$ and $\eta \in [\![\Gamma]\!]_X$:*

$$X \Vdash_{\Gamma,\eta} M =^\sigma N \quad \Longleftrightarrow \quad [\![\Gamma \vdash_\Sigma M : \sigma]\!]_X(\eta) = [\![\Gamma \vdash_\Sigma N : \sigma]\!]_X(\eta)$$

*Proof.* Let us denote by $T$ the interpretation $[\![\sigma]\!]$ and by $\Gamma'$ the environment $\Gamma, P : \sigma \to o$, for $P$ a fresh variable. By definition of $=^\sigma$ and Theorem 5.1, $X \Vdash_{\Gamma,\eta} M =^\sigma N$ holds iff

$$\begin{array}{l}\text{for all } Y, h \in \mathcal{I}(X,Y), p \in (T \Rightarrow Prop)_Y : \\ \text{if } [\![\Gamma' \vdash_\Sigma PM : o]\!]_Y(\eta[h], p) \geq \mathcal{I}(Y, \_), \text{then } [\![\Gamma' \vdash_\Sigma PN : o]\!]_Y(\eta[h], p) \geq \mathcal{I}(Y, \_)\end{array}$$

iff

$$\begin{array}{l}\text{for all } Y, h \in \mathcal{I}(X,Y), p : T \times \mathcal{V}(Y,\_) \longrightarrow Prop : \\ \text{if } ([\![\Gamma' \vdash_\Sigma P : \sigma \to o]\!]_Y(\eta[h], p))_Y([\![\Gamma' \vdash_\Sigma M : \sigma]\!]_Y(\eta[h], p), \mathrm{id}_Y) \geq \mathcal{I}(Y, \_), \\ \text{then } ([\![\Gamma' \vdash_\Sigma P : \sigma \to o]\!]_Y(\eta[h], p))_Y([\![\Gamma' \vdash_\Sigma N : \sigma]\!]_Y(\eta[h], p), \mathrm{id}_Y) \geq \mathcal{I}(Y, \_)\end{array}$$

iff

$$\begin{array}{l}\text{for all } Y, h \in \mathcal{I}(X,Y), p : T \times \mathcal{V}(Y,\_) \longrightarrow Prop : \\ \text{if } p_Y(m_Y(\eta[h]), \mathrm{id}_Y) \geq \mathcal{I}(Y,\_), \text{ then } p_Y(n_Y(\eta[h]), \mathrm{id}_Y) \geq \mathcal{I}(Y,\_).\end{array} \tag{5.4}$$

where $m, n : [\![\Gamma]\!] \longrightarrow T$ denote the natural transformations $[\![\Gamma \vdash_\Sigma M : \sigma]\!]$ and $[\![\Gamma \vdash_\Sigma N : \sigma]\!]$, respectively. We have to prove that this is equivalent to

$$m_X(\eta) = n_X(\eta). \tag{5.5}$$

(5.4 $\Rightarrow$ 5.5) By Corollary 5.3, $\Delta_T$ is a predicate of $T \times T$. Let $\delta_T : T \times T \longrightarrow Prop$ be its characteristic map, i.e. the Kronecker delta: for all $X$ and $s, t \in TX$: $(\delta_T)_X(s,t) \geq \mathcal{I}(X,\_)$ iff $s = t$.

Let $\bar{m} : \mathcal{V}(X,\_) \longrightarrow T$ be the natural transformation $\bar{m}_Z(h) \triangleq m_Z(\eta[h])$, and define $q \triangleq \delta_T \circ (\mathrm{id}_T \times \bar{m}) : T \times \mathcal{V}(X,\_) \longrightarrow Prop$. Then, for all $t \in TX$:

$$q_X(t, \mathrm{id}_X) \geq \mathcal{I}(X,\_) \iff (\delta_T)_X(t, m_X(\eta)) \geq \mathcal{I}(X,\_) \iff t = m_X(\eta)$$

Instantiating (5.4) for $Y = X$, $h = \mathrm{id}_X$ and $p = q$, we have

$$\text{if } q_X(m_X(\eta), \mathrm{id}_X) \geq \mathcal{I}(X\_) \text{ then } q_X(n_X(\eta), \mathrm{id}_X) \geq \mathcal{I}(X,\_)$$

which is equivalent to say that

$$\text{if } m_X(\eta) = m_X(\eta) \text{ then } n_X(\eta) = m_X(\eta)$$

hence the thesis.

(5.5 $\Rightarrow$ 5.4) By naturality, if $m_X(\eta) = n_X(\eta)$ then for all $Y$ and $h \in \mathcal{I}(X,Y)$, we have $m_Y(\eta[h]) = n_Y(\eta[h])$, hence the thesis. $\Box$

### 5.5.3 $\mathcal{U}$ models logical axioms and rules

**Theorem 5.4** *The model validates all logical axioms and rules; indeed if* $\Gamma \vdash_\Sigma p : o$, $\Gamma \vdash_\Sigma q : o$ *and* $\Gamma \vdash_\Sigma r : o$, *then the following holds:*

1. $\Gamma \vdash_\Sigma (p \Rightarrow q \Rightarrow r) \Rightarrow (p \Rightarrow q) \Rightarrow p \Rightarrow r$.
2. $\Gamma \vdash_\Sigma p \Rightarrow q \Rightarrow p$.
3. *If* $\Gamma \vdash_\Sigma P : \sigma \to o$ *and* $\Gamma \vdash_\Sigma M : \sigma$, *then we have* $\Gamma \vdash_\Sigma \forall_\sigma(P) \Rightarrow PM$.
4. *If* $\Gamma, x : \sigma \vdash_\Sigma M : \sigma'$ *and* $\Gamma \vdash_\Sigma N : \sigma$, *then we have* $\Gamma \vdash_\Sigma (\lambda x{:}\sigma.M)N =^{\sigma'} M[N/x]$.
5. *If* $\Gamma, x : \sigma \vdash_\Sigma M : \sigma'$, $\Gamma, x : \sigma \vdash_\Sigma N : \sigma'$, *then we have* $\Gamma \vdash_\Sigma (\forall x{:}\sigma.M =^{\sigma'} N) \Rightarrow \lambda x{:}\sigma.M =^{\sigma\to\sigma'} \lambda x{:}\sigma.N$.
6. *If* $\Gamma \vdash_\Sigma M : \sigma \to \sigma$ *and* $x \notin FV(M)$, *then we have* $\Gamma \vdash_\Sigma \lambda x{:}\sigma'.Mx =^{\sigma'\to\sigma} M$.
7. $\Gamma \vdash_\Sigma \neg\neg p \Rightarrow p$.
8. *If* $\Gamma \vdash_\Sigma p \Rightarrow q$ *and* $\Gamma \vdash_\Sigma p$, *then we have* $\Gamma \vdash_\Sigma q$.
9. *If* $\Gamma, x : \sigma \vdash_\Sigma p \Rightarrow q$, *then we have* $\Gamma \vdash_\Sigma p \Rightarrow \forall x{:}\sigma.q$.

*Proof.* See Appendix C.0.9. □

We conclude this section with a result about the $\notin$ predicate which will be useful in the following proofs.

**Theorem 5.5** *For all* $\Gamma$, $y$, $M$, $X$ *and* $\eta \in [\![\Gamma]\!]_X$, *such that* $\Gamma \vdash_\Sigma y : \upsilon$ *and* $\Gamma \vdash_\Sigma M : \iota$, *we have the following:*

$$X \Vdash_{\Gamma,\eta} y \notin M \iff [\![\Gamma \vdash_\Sigma y : \upsilon]\!]_X(\eta) \notin FV([\![\Gamma \vdash_\Sigma M : \iota]\!]_X(\eta))$$

*Proof.* See Appendix C.0.10. □

### 5.5.4 $\mathcal{U}$ models the Theory of Contexts

**Theorem 5.6** *The model* $\mathcal{U}$ *validates* $Unsat^\upsilon_\iota$: *if* $\Gamma \vdash_\Sigma P : \iota$, *then for all* $X$, $\eta \in [\![\Gamma]\!]_X$, *the following holds:* $X \Vdash_{\Gamma,\eta} \exists x{:}\upsilon.x \notin P$.

*Proof.* Applying Corollary 5.1, we deduce that $X \Vdash_{\Gamma,\eta} \exists x{:}\upsilon.x \notin P$ holds if and only if there exist $Z$, $g \in \mathcal{I}(X, Z)$, $z \in [\![\upsilon]\!]_Z \triangleq Z$ such that $Z \Vdash_{(\Gamma,x:\upsilon),\langle[\![\Gamma]\!]_g(\eta),z\rangle} x \notin P$. By Theorem 5.5, this is equivalent to prove

$$z \notin FV([\![\Gamma, x{:}\upsilon \vdash_\Sigma P{:}\iota]\!]_Z(\langle[\![\Gamma]\!]_g(\eta), z\rangle)) = FV([\![\Gamma \vdash_\Sigma P{:}\iota]\!]_Z([\![\Gamma]\!]_g(\eta))).$$

Hence it is sufficient to take $Z \triangleq X \cup \{n\}$ where $n \notin X$ (which surely exists since $X$ is a finite set, while we have at disposal an infinite set of names), $z \triangleq n$ and $g \triangleq \mathrm{id}_X$. □

**Theorem 5.7** *The model* $\mathcal{U}$ *validates* $Ext^{\upsilon\to\iota}$: *if* $\Gamma \vdash_\Sigma P : \upsilon \to \iota$, $\Gamma \vdash_\Sigma Q : \upsilon \to \iota$ *and* $\Gamma \vdash_\Sigma x : \upsilon$, *then for all* $X$, $\eta \in [\![\Gamma]\!]_X$, *the following holds:*

$$X \Vdash_{\Gamma,\eta} x \notin^1 P \Rightarrow x \notin^1 Q \Rightarrow (P\ x) =^\iota (Q\ x) \Rightarrow P = Q.$$

*Proof.* By Theorem 5.1, we have to prove $X \Vdash_{\Gamma,\eta} P =^{\upsilon\to\iota} Q$, knowing that $X \Vdash_{\Gamma,\eta} x \not\in^1 P$, $X \Vdash_{\Gamma,\eta} x \not\in^1 Q$ and $X \Vdash_{\Gamma,\eta} (P\ x) =^\iota (Q\ x)$ hold. The latter statement, by Theorem 5.3, is equivalent to say that $[\![\Gamma \vdash_\Sigma (P\ x) : \iota]\!]_X(\eta) = [\![\Gamma \vdash_\Sigma (Q\ x) : \iota]\!]_X(\eta)$, where $[\![\Gamma \vdash_\Sigma (P\ x) : \iota]\!]_X(\eta)$ and $[\![\Gamma \vdash_\Sigma (Q\ x) : \iota]\!]_X(\eta)$ belong to $Proc_X$; hence, restricting both processes on $\eta_x \triangleq [\![\Gamma \vdash_\Sigma x : \upsilon]\!]$, we preserve the equality relation, i.e., the following holds:

$$(\nu\eta_x)([\![\Gamma \vdash_\Sigma (P\ x) : \iota]\!]_X(\eta)) = (\nu\eta_x)([\![\Gamma \vdash_\Sigma (Q\ x) : \iota]\!]_X(\eta)). \tag{5.6}$$

Now we observe that

$$\begin{array}{rcl}[\![\Gamma \vdash_\Sigma (P\ x) : \iota]\!]_X(\eta) & = & (ev_{Proc,Var})_X(\langle [\![\Gamma \vdash_\Sigma P : \upsilon \to \iota]\!]_X(\eta), [\![\Gamma \vdash_\Sigma x : \upsilon]\!]_X(\eta)\rangle) \\ & = & ([\![\Gamma \vdash_\Sigma P : \upsilon \to \iota]\!]_X(\eta))_X(\langle \eta_x, \mathrm{id}_X\rangle).\end{array}$$

By a similar argument we also have

$$[\![\Gamma \vdash_\Sigma (Q\ x) : \iota]\!]_X(\eta) = ([\![\Gamma \vdash_\Sigma Q : \upsilon \to \iota]\!]_X(\eta))_X(\langle \eta_x, \mathrm{id}_X\rangle).$$

Moreover, from equation 5.6 and the latter statements, the following holds:

$$\begin{array}{c}(\nu\eta_x)([\![\Gamma \vdash_\Sigma P : \upsilon \to \iota]\!]_X(\eta))_X(\langle \eta_x, \mathrm{id}_X\rangle) = new([\![\Gamma \vdash_\Sigma P : \upsilon \to \iota]\!]_X(\eta)) \\ (\nu\eta_x)([\![\Gamma \vdash_\Sigma Q : \upsilon \to \iota]\!]_X(\eta))_X(\langle \eta_x, \mathrm{id}_X\rangle) = new([\![\Delta \vdash_\Sigma Q : \upsilon \to \iota]\!]_X(\eta)).\end{array}$$

Hence, from the injectivity of $new$, we deduce the validity of

$$[\![\Gamma \vdash_\Sigma P : \upsilon \to \iota]\!]_X(\eta) = [\![\Gamma \vdash_\Sigma Q : \upsilon \to \iota]\!]_X(\eta),$$

which is equivalent to the thesis by Theorem 5.3. □

**Theorem 5.8** *The model $\mathcal{U}$ validates $\beta\_exp^\iota$: if $\Gamma \vdash_\Sigma P : \iota$ and $\Gamma \vdash_\Sigma x : \upsilon$, then for all $X$, $\eta \in [\![\Gamma]\!]_X$, we have that $X \Vdash_{\Gamma,\eta} \exists Q{:}\upsilon \to \iota . x \not\in^1 Q \wedge P =^\iota (Q\ x)$ holds.*

*Proof.* By Corollary 5.1, we just have to prove that there exist $Z$, $g \in \mathcal{I}(X,Z)$, $\eta_Q \in (Var \Rightarrow Proc)_Z$ such that $Z \Vdash_{\Delta,\mu} x \not\in^1 Q$ and $Z \Vdash_{\Delta,\mu} P =^\iota (Q\ x)$ hold (where $\Delta \triangleq \Gamma, Q : \upsilon \to \iota$ and $\mu \triangleq \langle [\![\Gamma]\!]_g(\eta), \eta_Q\rangle$). Hence we choose $Z \triangleq X$, $g \triangleq \mathrm{id}_X$ and $\eta_Q \triangleq [\![\Gamma \setminus \{x : \upsilon\} \vdash_\Sigma \lambda x.P : \upsilon \to \iota]\!]_X(\eta')$ (where $\eta' \triangleq \eta\restriction_{dom(\Gamma\setminus\{x:\upsilon\})}$). In order to prove the first forcing statement, we observe that it is equivalent, by definition of $\not\in^1$, to $X \Vdash_{\Delta,\mu_X} x \not\in \nu Q$ (where $\mu_X \triangleq \langle \eta, \eta_Q\rangle$). By Theorem 5.5, this is equivalent to prove $\eta_x \not\in FV([\![\Delta \vdash_\Sigma \nu Q : \iota]\!]_X(\mu_X))$. Hence we may easily conclude since the following holds:

$$\begin{aligned}[\![\Delta \vdash_\Sigma \nu Q : \iota]\!]_X(\mu_X) &= new_X([\![\Delta \vdash_\Sigma Q : \upsilon \to \iota]\!]_X(\mu_X)) \\ &= (\nu\eta_x)(([\![\Delta \vdash_\Sigma Q : \upsilon \to \iota]\!]_X(\mu_X))_X(\langle \eta_x, \mathrm{id}_X\rangle)) \\ &= (\nu\eta_x)([\![\Gamma \setminus \{x : \upsilon\} \vdash_\Sigma \lambda x.P : \upsilon \to \iota]\!]_X(\eta'))_X(\langle \eta_x, \mathrm{id}_X\rangle)) \\ &= (\nu\eta_x)([\![\Gamma \vdash_\Sigma P : \iota]\!]_X(\eta)).\end{aligned}$$

Referring to the proof of $X \Vdash_{\Delta,\mu_X} P =^\iota (Q\ x)$, we observe that this statement holds if and only if $[\![\Delta \vdash_\Sigma P : \iota]\!]_X(\mu_X) = [\![\Delta \vdash_\Sigma (Q\ x) : \iota]\!]_X(\mu_X)$ holds. Then we have that $[\![\Delta \vdash_\Sigma P : \iota]\!]_X(\mu_X) = [\![\Gamma \vdash_\Sigma P : \iota]\!]_X(\eta)$, hence we can conclude since the following holds:

$$\begin{aligned}[\![\Delta \vdash_\Sigma (Q\ x) : \iota]\!]_X(\mu_X) &= (ev_{Proc,Var})_X(\langle [\![\Delta \vdash_\Sigma Q : \upsilon \to \iota]\!]_X(\mu_X), [\![\Delta \vdash_\Sigma x : \upsilon]\!]_X(\mu_X)\rangle) \\ &= (ev_{Proc,Var})_X(\langle [\![\Gamma \setminus \{x : \upsilon\} \vdash_\Sigma \lambda x{:}\upsilon.P : \upsilon \to \iota]\!]_X(\eta'), \eta_x\rangle) \\ &= ([\![\Gamma \setminus \{x : \upsilon\} \vdash_\Sigma \lambda x{:}\upsilon.P : \upsilon \to \iota]\!]_Y(\eta'))_X(\langle \eta_x, \mathrm{id}_X\rangle) \\ &= [\![\Gamma \vdash_\Sigma P : \iota]\!]_X(\eta).\Box\end{aligned}$$

□

$$\frac{\Gamma \vdash_\Sigma f_1 : \sigma \quad \Gamma \vdash_\Sigma f_2 : \sigma \to \sigma \quad \Gamma \vdash_\Sigma f_3 : \sigma \to \sigma \to \sigma \quad \Gamma \vdash_\Sigma f_4 : \upsilon \to \upsilon \to \sigma \to \sigma \quad \Gamma \vdash_\Sigma f_5 : (\upsilon \to \sigma) \to \sigma}{\Gamma \vdash_\Sigma (R\ 0) =^\sigma f_1} \quad (Rec^\iota_\sigma\_red_1)$$

$$\frac{\Gamma \vdash_\Sigma f_1 : \sigma \quad \Gamma \vdash_\Sigma f_2 : \sigma \to \sigma \quad \Gamma \vdash_\Sigma f_3 : \sigma \to \sigma \to \sigma \quad \Gamma \vdash_\Sigma f_4 : \upsilon \to \upsilon \to \sigma \to \sigma \quad \Gamma \vdash_\Sigma f_5 : (\upsilon \to \sigma) \to \sigma}{\Gamma \vdash_\Sigma \forall P{:}\iota.(R\ \tau.P) =^\sigma (f_2\ (R\ P))} \quad (Rec^\iota_\sigma\_red_2)$$

$$\frac{\Gamma \vdash_\Sigma f_1 : \sigma \quad \Gamma \vdash_\Sigma f_2 : \sigma \to \sigma \quad \Gamma \vdash_\Sigma f_3 : \sigma \to \sigma \to \sigma \quad \Gamma \vdash_\Sigma f_4 : \upsilon \to \upsilon \to \sigma \to \sigma \quad \Gamma \vdash_\Sigma f_5 : (\upsilon \to \sigma) \to \sigma}{\Gamma \vdash_\Sigma \forall P{:}\iota.\forall Q{:}\iota.(R\ P|Q) =^\sigma (f_3\ (R\ P)\ (R\ Q))} \quad (Rec^\iota_\sigma\_red_3)$$

$$\frac{\Gamma \vdash_\Sigma f_1 : \sigma \quad \Gamma \vdash_\Sigma f_2 : \sigma \to \sigma \quad \Gamma \vdash_\Sigma f_3 : \sigma \to \sigma \to \sigma \quad \Gamma \vdash_\Sigma f_4 : \upsilon \to \upsilon \to \sigma \to \sigma \quad \Gamma \vdash_\Sigma f_5 : (\upsilon \to \sigma) \to \sigma}{\Gamma \vdash_\Sigma \forall x{:}\upsilon.\forall y{:}\upsilon.\forall P{:}\iota.(R\ [x \neq y]P) =^\sigma (f_4\ x\ y\ (R\ P))} \quad (Rec^\iota_\sigma\_red_4)$$

$$\frac{\Gamma \vdash_\Sigma f_1 : \sigma \quad \Gamma \vdash_\Sigma f_2 : \sigma \to \sigma \quad \Gamma \vdash_\Sigma f_3 : \sigma \to \sigma \to \sigma \quad \Gamma \vdash_\Sigma f_4 : \upsilon \to \upsilon \to \sigma \to \sigma \quad \Gamma \vdash_\Sigma f_5 : (\upsilon \to \sigma) \to \sigma}{\Gamma \vdash_\Sigma \forall P{:}\upsilon \to \iota.(R\ \nu P) =^\sigma (f_5\ \lambda x{:}\upsilon.(R\ (P\ x)))} \quad (Rec^\iota_\sigma\_red_5)$$

where $R$ is a typographic shorthand for $(Rec^\iota_\sigma\ f_1\ f_2\ f_3\ f_4\ f_5)$;

Figure 5.5: Reduction rules for first-order recursion.

## 5.6 Recursion

The model $\mathcal{U}$ is expressive enough to justify also *recursion* and *induction* principles, even higher-order ones.

### 5.6.1 First-order recursion

In order to be able to define recursive functions over $\iota$, we extend the signature $\Sigma$ with a *recursor operator* $Rec^\iota_\sigma$ for any type $\sigma$:

$$Rec^\iota_\sigma : \sigma \to (\sigma \to \sigma) \to (\sigma \to \sigma \to \sigma) \to (\upsilon \to \upsilon \to \sigma \to \sigma) \to ((\upsilon \to \sigma) \to \sigma) \to \iota \to \sigma$$

The intended reduction rules for recursors are given in Figure 5.5.

In order to interpret the constant $Rec^\iota_\sigma$ we will show that $(Proc, \alpha)$ is an initial algebra for the functor $T : \check{\mathcal{V}} \longrightarrow \check{\mathcal{V}}$ defined on objects by

$$TF \triangleq \mathbf{1} + F + (F \times F) + (Var \times Var \times F) + (Var \Rightarrow F),$$

on morphisms (at each stage $X \in \mathcal{V}$) $h : F \longrightarrow G$ by

$$\begin{array}{rcl} (Th)_X : (TF)_X & \longrightarrow & (TG)_X \\ in_1(*) & \longmapsto & in_1(*) \\ in_2(a) & \longmapsto & in_2(h_X(a)) \\ in_3(\langle a, b \rangle) & \longmapsto & in_3(\langle h_X(a), h_X(b) \rangle) \\ in_4(\langle x, y, a \rangle) & \longmapsto & in_4(\langle x, y, h_X(a) \rangle) \\ in_5(a) & \longmapsto & in_5(\gamma_{G,X}(h_{X \uplus \{x\}}(a_{X \uplus \{x\}}(\langle x, in_X \rangle)))) \end{array}$$

where $\gamma_{G,X} : G_{X \uplus \{x\}} \longrightarrow (Var \Rightarrow G)_X$ is the isomorphism given by Proposition B.2 and $\alpha : TProc \longrightarrow Proc$ is the *natural term forming operation* at each stage $X \in \mathcal{V}$:

$$\begin{array}{lcl}
\alpha_X(in_1(*)) & \triangleq & 0 \\
\alpha_X(in_2(P)) & \triangleq & \tau.P \\
\alpha_X(in_3(\langle P_1, P_2\rangle)) & \triangleq & P_1 | P_2 \\
\alpha_X(in_4(\langle x, y, P\rangle)) & \triangleq & [x \neq y]P \\
\alpha_X(in_5(P)) & \triangleq & (\nu x)P_{X \uplus \{x\}}(\langle x, in_X\rangle)
\end{array}$$

**Proposition 5.6** *$(Proc, \alpha)$ is an initial $T$-algebra.*

*Proof.* Let $(B, \beta)$ be an arbitrary $T$-algebra; then there is a unique homomorphism $f : (Proc, \alpha) \longrightarrow (B, \beta)$ of $T$-algebras such that $f \circ \alpha = \beta \circ Tf$. Given $f$, in order to prove the latter equality we must consider each component $f_X$ for $X \in \mathcal{V}$. We define $f$ by recursion as follows:

$$\begin{array}{lcll}
f_X(0) & \triangleq & \beta_X(in_1(*)) & \\
f_X(\tau.P) & \triangleq & \beta_X(in_2(f_X(P))) & \\
f_X(P_1|P_2) & \triangleq & \beta_X(in_3(\langle f_X(P_1), f_X(P_2)\rangle)) & \\
f_X([x \neq y]P) & \triangleq & \beta_X(in_4(\langle x, y, f_X(P)\rangle)) & \\
f_X((\nu x)P)) & \triangleq & \beta_X(in_5(\gamma_{B,X}(f_{X \uplus \{x\}}(P)))) & \qquad (P \in Proc_{X \uplus \{x\}})
\end{array}$$

where $\gamma_{B,X} : B_{X \uplus \{x\}} \longrightarrow (Var \Rightarrow B)_X$ is the isomorphism given by Proposition B.2. Then we can easily check that, for each $t \in (TProc)_X$, we have $f_X(\alpha_X(t)) = \beta_X((Tf)_X(t))$:

$$\begin{aligned}
f_X(\alpha_X(in_1(*))) &= f_X(0) \triangleq \beta_X(in_1(*)) = \beta_X((Tf)_X(in_1(*))) \\
f_X(\alpha_X(in_2(P))) &= f_X(\tau.P) \triangleq \beta_X(in_2(f_X(P))) = \beta_X((Tf)_X(in_2(P))) \\
f_X(\alpha_X(in_3(\langle P_1, P_2\rangle))) &= f_X(P_1|P_2) \triangleq \beta_X(in_3(\langle f_X(P_1), f_X(P_2)\rangle)) \\
&= \beta_X((Tf)_X(in_3(\langle P_1, P_2\rangle))) \\
f_X(\alpha_X(in_4(\langle x, y, P\rangle))) &= f_X([x \neq y]P) \triangleq \beta_X(in_4(\langle x, y, f_X(P)\rangle)) \\
&= \beta_X((Tf)_X(in_4(\langle x, y, P\rangle))) \\
f_X(\alpha_X(in_5(P))) &= f_X((\nu x)P_{X \uplus \{x\}}(x, in_X)) \\
&\triangleq \beta_X(in_5(\gamma_{B,X}(f_{X \uplus \{x\}}(P_{X \uplus \{x\}}(x, in_X))))) \\
&= \beta_X((Tf)_X(in_5(P)))
\end{aligned}$$

The uniqueness of $f$ follows by its definition. Indeed, if there would be another homomorphism $g : (Proc, \alpha) \longrightarrow (B, \beta)$ such that $g \circ \alpha = \beta \circ Tg$, then rewriting the previous equality and simplifying it according to the definition of $\alpha$ we would obtain exactly the definition of $f$. □

Using this result we can interpret the recursor $Rec^\sigma_\iota$ as follows. Let $A \triangleq [\![\sigma]\!]$, $G \triangleq [\![\Gamma]\!]$ and $\Gamma \vdash R : Proc \to \sigma$, where $R \triangleq (Rec^\sigma_\iota\ f_1\ f_2\ f_3\ f_4\ f_5)$. Let $g_i$ be the meaning of $f_i$, as follows:

$$\begin{array}{ll}
g_1 = [\![\Gamma \vdash f_1 : \sigma]\!] & : G \longrightarrow A \\
g_2 = [\![\Gamma \vdash f_2 : \sigma \to \sigma]\!] & : G \longrightarrow A \Rightarrow A \\
g_3 = [\![\Gamma \vdash f_3 : \sigma \to \sigma \to \sigma]\!] & : G \longrightarrow A \Rightarrow A \Rightarrow A \\
g_4 = [\![\Gamma \vdash f_4 : \upsilon \to \upsilon \to \sigma \to \sigma]\!] & : G \longrightarrow Var \Rightarrow Var \Rightarrow A \Rightarrow A \\
g_5 = [\![\Gamma \vdash f_5 : (\upsilon \to \sigma) \to \sigma]\!] & : G \longrightarrow (Var \Rightarrow A) \Rightarrow A
\end{array}$$

We define a natural transformation $m : T(G{\Rightarrow}A) \longrightarrow G{\Rightarrow}A$, that is, for $X \in \mathcal{V}$,

$$m_X : \mathbf{1}_X + (G{\Rightarrow}A)_X + (G{\Rightarrow}A)_X \times (G{\Rightarrow}A)_X + \mathit{Var}_X \times \mathit{Var}_X \times (G{\Rightarrow}A)_X + (\mathit{Var}{\Rightarrow}G{\Rightarrow}A)_X \\ \longrightarrow (G{\Rightarrow}A)_X$$

by cases as follows, bearing in mind that $(G{\Rightarrow}A)_X = \check{\mathcal{V}}(G \times \mathcal{V}(X, \_), A)$: for $Y$ stage, $\eta \in G_Y$ and $h \in \mathcal{V}(X, Y)$,

$$\begin{aligned}
(m_X(in_1(*)))_Y(\eta, h) &\triangleq g_{1Y}(\eta) \\
(m_X(in_2(r)))_Y(\eta, h) &\triangleq (g_{2Y}(\eta))_Y(r_Y(\eta, h), \mathrm{id}_Y) \\
(m_X(in_3(\langle r_1, r_2\rangle)))_Y(\eta, h) &\triangleq ((g_{3Y}(\eta))_Y(r_{1Y}(\eta, h), \mathrm{id}_Y))_Y(r_{2Y}(\eta, h), \mathrm{id}_Y) \\
(m_X(in_4(\langle x, y, r\rangle)))_Y(\eta, h) &\triangleq (((g_{4Y}(\eta))_Y(h(x), \mathrm{id}_Y))_Y(h(y), \mathrm{id}_Y))_Y(r_Y(\eta, h), \mathrm{id}_Y)) \\
(m_X(in_5(r)))_Y(\eta, h) &\triangleq (g_{5Y}(\eta))_Y(r', \mathrm{id}_Y) \\
&\quad \text{where } \ r' : \mathit{Var} \times \mathcal{V}(Y, _) \longrightarrow A \\
&\qquad\qquad r'_Z : Z \times \mathcal{V}(Y, Z) \longrightarrow A_Z \\
&\qquad\qquad\quad \langle z, k\rangle \longmapsto (r_Z(z, k \circ h))_Z(\eta[h], \mathrm{id}_Z)
\end{aligned}$$

Thus, $(G{\Rightarrow}A, m)$ is a $T$-algebra; therefore, there exists a unique natural transformation $\bar{m} : \mathit{Proc} \longrightarrow G{\Rightarrow}A$ such that $m \circ T\bar{m} = \bar{m} \circ \alpha$. By using a standard argument of cartesian closed categories, $\bar{m}$ can be converted into the morphism $G \longrightarrow \mathit{Proc}{\Rightarrow}A$ we need. More explicitly, we interpret $\Gamma \vdash R : \mathit{Proc} \to \sigma$ as follows:

$$\begin{aligned}
[\![\Gamma \vdash R : \mathit{Proc} \to \sigma]\!] &: G \longrightarrow \mathit{Proc}{\Rightarrow}A \\
[\![\Gamma \vdash R : \mathit{Proc} \to \sigma]\!]_X &: G_X \longrightarrow (\mathit{Proc}{\Rightarrow}A)_X \\
[\![\Gamma \vdash R : \mathit{Proc} \to \sigma]\!]_X(\eta) &: \mathit{Proc} \times \mathcal{V}(X, _) \longrightarrow A \\
[\![\Gamma \vdash R : \mathit{Proc} \to \sigma]\!]_X(\eta)_Y &: \mathit{Proc}_Y \times \mathcal{V}(X, Y) \longrightarrow A_Y \\
&\quad \langle P, h\rangle \longmapsto (\bar{m}_Y(P))_Y(\eta[h], \mathrm{id}_Y)
\end{aligned}$$

We can now prove the soundness of the recursion principles.

**Theorem 5.9** *The model $\mathcal{U}$ validates $\mathit{Rec}^{\iota}_{\sigma}\_\mathit{red}_i$, for $i = 1 \dots 5$.*

*Proof.* See Appendix C.0.11. □

### 5.6.2 Higher-order recursion

First-order recursion rules can be generalized to higher-order processes, i.e. terms with holes. Indeed, the initial algebra over *Proc* can be readily "lifted" to the types $\mathit{Var} \Rightarrow \mathit{Proc}$, $\mathit{Var} \Rightarrow \mathit{Var} \Rightarrow \mathit{Proc}, \dots$. Let us consider the functor $T' : \check{\mathcal{V}} \longrightarrow \check{\mathcal{V}}$ defined on objects by

$$T'_F \triangleq \mathbf{1} + F + F \times F + (\mathit{Var} \Rightarrow \mathit{Var}) \times (\mathit{Var} \Rightarrow \mathit{Var}) \times F + (\mathit{Var} \Rightarrow F),$$

and on morphisms in the obvious way. Then, the following holds:

**Proposition 5.7** *$\mathit{Var} \Rightarrow \mathit{Proc}$ has an initial $T'$-algebra structure, which is isomorphic to $\mathit{Var} \Rightarrow \alpha$.*

*Proof.* Let $G : \check{\mathcal{V}} \longrightarrow \check{\mathcal{V}}$ be the functor $G(F) \triangleq \mathit{Var} \Rightarrow F$. $G$ has a right adjoint, namely the functor $R : \check{\mathcal{V}} \longrightarrow \check{\mathcal{V}}$ defined on objects by

$$R(F)_X = \check{\mathcal{V}}(\mathit{Var} \Rightarrow \check{\mathcal{Y}}(X), F) \qquad R(F)_h = \_ \circ (\mathit{Var} \Rightarrow \check{\mathcal{Y}}(h)) \quad (h \in \mathcal{V}(X,Y))$$

and on natural transformations $t \in \check{\mathcal{V}}(F, F')$ by $R(t) : \mathit{Var} \Rightarrow F \longrightarrow \mathit{Var} \Rightarrow F'$, $R(t)_X(f) = t \circ f$ for $f \in \check{\mathcal{V}}(\mathit{Var} \Rightarrow \check{\mathcal{Y}}(X), F)$. Hence, by Theorem B.1, we need only to show that $T' \circ G \cong G \circ T$. Given any functor $F$ in $\check{\mathcal{V}}$, we have that

$$\begin{aligned}
(T' \circ G)(F) =& T'(\mathit{Var} \Rightarrow F) = \mathbf{1} + \mathit{Var} \Rightarrow F + (\mathit{Var} \Rightarrow F) \times (\mathit{Var} \Rightarrow F) + \\
& + (\mathit{Var} \Rightarrow \mathit{Var}) \times (\mathit{Var} \Rightarrow \mathit{Var}) \times (\mathit{Var} \Rightarrow F) + \mathit{Var} \Rightarrow (\mathit{Var} \Rightarrow F) \\
\cong& \mathit{Var} \Rightarrow \mathbf{1} + \mathit{Var} \Rightarrow F + \mathit{Var} \Rightarrow (F \times F) + \\
& + \mathit{Var} \Rightarrow (\mathit{Var} \times \mathit{Var} \times F) + (\mathit{Var} \Rightarrow (\mathit{Var} \Rightarrow F)) \\
\cong& \mathit{Var} \Rightarrow (\mathbf{1} + F + F \times F + \mathit{Var} \times \mathit{Var} \times F + \mathit{Var} \Rightarrow F) \\
=& \mathit{Var} \Rightarrow TF = (G \circ T)_F
\end{aligned}$$

and similarly for the morphism part. □

We can elaborate the functor $T'$ a step further, by noticing that

$$\mathit{Var} \Rightarrow \mathit{Var} \cong \mathit{Var} + \mathbf{1}.$$

Indeed, for all $X$, we have $(\mathit{Var} \Rightarrow \mathit{Var})_X = \mathit{Var}_{X \uplus \{x\}} = X \uplus \{x\} \cong X + 1 = (\mathit{Var} + \mathbf{1})_X$. Thus we can rewrite $T'$ as follows:

$$T'_F \triangleq \mathbf{1} + F + F \times F + \underbrace{\mathit{Var} \times \mathit{Var} \times F + \mathit{Var} \times F + \mathit{Var} \times F + F}_{\cong (\mathit{Var} \Rightarrow \mathit{Var}) \times (\mathit{Var} \Rightarrow \mathit{Var}) \times F} + \mathit{Var} \Rightarrow F \tag{5.7}$$

and Proposition 5.7 still holds, that is, $\mathit{Var} \Rightarrow \mathit{Proc}$ is a $T'$-algebra. The intuitive meaning of the four cases arising from an abstraction $\lambda x.[y \neq z]P$ corresponds to the four situations when none, one or both $y, z$ are exactly $x$, and hence are bound by the abstraction.

This argument can be generalized to an arbitrary number of "holes", so that all types $\mathit{Var}^n \Rightarrow \mathit{Proc}$ have an initial algebra structure for a suitable functor. In fact, it is easy to see that for all $n$:

$$\mathit{Var}^n \Rightarrow \mathit{Var} \cong \mathit{Var} + \underbrace{\mathbf{1} + \cdots + \mathbf{1}}_{n \text{ times}}$$

Hence we can generalize (5.7) at any number of holes, as follows:

$$\begin{aligned}
T^{(n)}_F \triangleq\ & \mathbf{1} + F + F \times F + \mathit{Var} \times \mathit{Var} \times F + \\
& + \underbrace{\mathit{Var} \times F + \cdots + \mathit{Var} \times F}_{2n \text{ times}} + \underbrace{F + \cdots + F}_{n^2 \text{ times}} + \mathit{Var} \Rightarrow F
\end{aligned} \tag{5.8}$$

Correspondingly, Proposition 5.7 can be generalized as follows:

**Theorem 5.10** *For all $n$, $\mathit{Var}^n \Rightarrow \mathit{Proc}$ has an initial $T^{(n)}$-algebra structure.*

From the definition of $T^{(n)}$ we can derive immediately that the recursor over higher-order terms with $n$ holes (i.e., contexts with $n$ free variables) for type $\sigma$ has the following type:

$$\begin{aligned}
\mathit{Rec}_\sigma^{\upsilon^n \to \iota} :\ & \sigma \to (\sigma \to \sigma) \to (\sigma \to \sigma \to \sigma) \to (\upsilon \to \upsilon \to \sigma \to \sigma) \to \\
& \underbrace{(\upsilon \to \sigma \to \sigma) \to \cdots \to (\upsilon \to \sigma \to \sigma)}_{2n \text{ times}} \to \underbrace{(\sigma \to \sigma) \to \cdots \to (\sigma \to \sigma)}_{n^2 \text{ times}} \to \\
& ((\upsilon \to \sigma) \to \sigma) \to (\upsilon^n \to \iota) \to \sigma
\end{aligned}$$

$$\frac{H}{\Gamma \vdash_\Sigma (R\ \lambda\vec{x}^{\upsilon}.0) =^\sigma f_1} \qquad (Rec_\sigma^{\upsilon^n\to\iota}\_red_1)$$

$$\frac{H}{\Gamma \vdash_\Sigma \forall P{:}\upsilon^n \to \iota.(R\ \lambda\vec{x}{:}\upsilon.\tau.(P\ \vec{x})) =^\sigma (f_2\ (R\ P))} \qquad (Rec_\sigma^{\upsilon^n\to\iota}\_red_2)$$

$$\frac{H}{\Gamma \vdash_\Sigma \forall P{:}\upsilon^n \to \iota.\forall Q{:}\upsilon^n \to \iota.(R\ \lambda\vec{x}{:}\upsilon.(P\ \vec{x})|(Q\ \vec{x})) =^\sigma (f_3\ (R\ P)\ (R\ Q))} \qquad (Rec_\sigma^{\upsilon^n\to\iota}\_red_3)$$

$$\frac{H}{\Gamma \vdash_\Sigma \forall y{:}\upsilon.\forall z{:}\upsilon.\forall P{:}\upsilon^n \to \iota.(R\ \lambda\vec{x}{:}\upsilon.[y \neq z](P\ \vec{x})) =^\sigma (f_{41}\ y\ z\ (R\ P))} \qquad (Rec_\sigma^{\upsilon^n\to\iota}\_red_{41})$$

$$\frac{H}{\Gamma \vdash_\Sigma \forall y{:}\upsilon.\forall P{:}\upsilon^n \to \iota.(R\ \lambda\vec{x}{:}\upsilon.[y \neq x_j](P\ \vec{x})) =^\sigma (f_{42j}\ y\ (R\ P))}\ j = 1\ldots n \qquad (Rec_\sigma^{\upsilon^n\to\iota}\_red_{42j})$$

$$\frac{H}{\Gamma \vdash_\Sigma \forall z{:}\upsilon.\forall P{:}\upsilon^n \to \iota.(R\ \lambda\vec{x}{:}\upsilon.[x_i \neq z](P\ \vec{x})) =^\sigma (f_{43i}\ z\ (R\ P))}\ i = 1\ldots n \qquad (Rec_\sigma^{\upsilon^n\to\iota}\_red_{43i})$$

$$\frac{H}{\Gamma \vdash_\Sigma \forall P{:}\upsilon^n \to \iota.(R\ \lambda\vec{x}{:}\upsilon.[x_i \neq x_j](P\ \vec{x})) =^\sigma (f_{44ij}\ (R\ P))}\ i,j = 1\ldots n \quad (Rec_\sigma^{\upsilon^n\to\iota}\_red_{44ij})$$

$$\frac{H}{\Gamma \vdash_\Sigma \forall P{:}\upsilon^{n+1} \to \iota.(R\ \lambda\vec{x}{:}\upsilon.(\nu\lambda y{:}\upsilon(P\ y\ \vec{x}))) =^\sigma (f_5\ \lambda y{:}\upsilon.(R\ (P\ y)))} \qquad (Rec_\sigma^{\upsilon^n\to\iota}\_red_5)$$

where $H$ is a typographic shorthand for the following hypotheses

$$\begin{array}{llll} \Gamma \vdash f_1 : \sigma & \Gamma \vdash f_2 : \sigma \to \sigma & \Gamma \vdash f_3 : \sigma \to \sigma \to \sigma & \Gamma \vdash f_{41} : \upsilon \to \upsilon \to \sigma \to \sigma \\ \Gamma \vdash f_{42j} : \upsilon \to \sigma \to \sigma & \Gamma \vdash f_{43i} : \upsilon \to \sigma \to \sigma & \Gamma \vdash f_{44ij} : \sigma \to \sigma & (i,j = 1\ldots n) \end{array}$$

and $R$ is a typographic shorthand for

$$(Rec_\sigma^{\upsilon^n\to\iota}\ f_1\ f_2\ f_3\ f_{41}\ f_{421}\ldots f_{42n}\ f_{431}\ldots f_{43n}\ f_{4411}\ldots f_{44nn}\ f_5)$$

Figure 5.6: Reduction rules for higher-order recursion.

The reduction rules for higher-order recursion are in Figure 5.6. Notice that when $n = 0$, these rules degenerate in those for first-order terms (Figure 5.5).

**Theorem 5.11** *For all $n$, the model $\mathcal{U}$ validates all axioms in Figure 5.6.*

*Proof.* A straightforward generalization of the proof of Theorem 5.9. □

## 5.7 Induction

In order to allow for (structural) inductive arguments, we need to extend the theory introduced so far with *induction principles*. In this section we consider induction principles both over plain terms (i.e., of type $\iota$) and over higher-order terms (i.e., of type $\upsilon^n \to \iota$ with $n > 0$).

### 5.7.1 First-order induction

The first-order induction principle we consider is presented in Figure 5.7.

$$\frac{\Gamma \vdash_\Sigma R : \iota \to o}{\begin{array}{l}\Gamma \vdash_\Sigma (R\ 0) \Rightarrow (\forall P{:}\iota.(R\ P) \Rightarrow (R\ \tau.P)) \Rightarrow \\ \qquad (\forall P{:}\iota.(R\ P) \Rightarrow \forall Q{:}\iota.(R\ Q) \Rightarrow (R\ P|Q)) \Rightarrow \\ \qquad (\forall y{:}\upsilon.\forall z{:}\upsilon.\forall P{:}\iota.(R\ P) \Rightarrow (R\ [y \neq z]P)) \Rightarrow \\ \qquad (\forall P{:}\upsilon \to \iota.(\forall x{:}\upsilon.(R\ (P\ x))) \Rightarrow (R\ \nu P)) \Rightarrow \\ \qquad \forall P{:}\iota.(R\ P)\end{array}} \quad (Ind^\iota)$$

Figure 5.7: First-order induction principle.

Since the model $\mathcal{U}$ does not support the "proposition-as-types, proofs-as-$\lambda$-terms" interpretation, induction principles do not derive automatically from the recursion principles in Section 5.6. A problem we have to deal with, is the presence of parameters, represented by the environment $\Gamma$. Actually, induction with parameters in $\check{\mathcal{V}}$ can be recovered from the initial algebra property in a simple slice category defined from $\check{\mathcal{V}}$ (Definition B.4). In fact the signature functor $T$ in $\check{\mathcal{V}}$ can be "transferred" in this category, so that it has an initial algebra corresponding to the initial algebra in $\check{\mathcal{V}}$ [Jac95].

For the sake of simplicity, and without loss of generality, in the following we consider $\Gamma = R : \iota \to o$, where $R$ is the predicate over terms in the induction principle.

We proceed as follows. We will work in the simple slice category $\check{\mathcal{V}}/\!\!/G$, where $G \triangleq Proc \Rightarrow Prop$; over this category we will consider the functor $T/\!\!/G$ where $T : \check{\mathcal{V}} \longrightarrow \check{\mathcal{V}}$ is the signature functor defined in Section 5.6. We will prove that $(Proc, G^*(\alpha))$ is an initial $T/\!\!/G$-algebra. Then, the soundness of the induction principle will derive from a usual argument in the category $\check{\mathcal{V}}/\!\!/G$.

In order to prove the main statement, we need the following two results:

**Proposition 5.8** *The functor $T$ is strong.*

*Proof.* We define the strength of $T$ as follows:

$$(st_{A,B})_X : A_X \times (TB)_X \longrightarrow (TA \times B)_X$$

$$\begin{aligned}
(st_{A,B})_X(\langle a, in_1(*)\rangle) &\triangleq in_1(*)\\
(st_{A,B})_X(\langle a, in_2(b)\rangle) &\triangleq in_2(\langle a, b\rangle)\\
(st_{A,B})_X(\langle a, in_3(\langle b_1, b_2\rangle)\rangle) &\triangleq in_3(\langle a, b_1, a, b_2\rangle)\\
(st_{A,B})_X(\langle a, in_4(\langle x, y, b\rangle)\rangle) &\triangleq in_4(\langle x, y, a, b\rangle)\\
(st_{A,B})_X(\langle a, in_5(b)\rangle) &\triangleq in_5(\overline{b}_a)
\end{aligned}$$

where $\overline{b}_a \in \check{\mathcal{V}}(Var \times \mathcal{V}(X, \_), A \times B)$ is the natural transformation such that $(\overline{b}_a)_Y(\langle y, g\rangle) \triangleq \langle A_g(a), b_Y(\langle y, g\rangle)\rangle$.

The commutativity of the two diagrams of Definition B.5 is proved by cases over $b$ (see Appendix C.0.12). □

**Proposition 5.9** *For every $G \in \check{\mathcal{V}}$, $(Proc, G^*(\alpha))$ is an initial $T/\!\!/G$-algebra.*

*Proof.* Since $\alpha \in \check{\mathcal{V}}(TProc, Proc)$, $G^*(\alpha) \in \check{\mathcal{V}}/\!\!/G((T/\!\!/G)_{Proc}, Proc)$, i.e., $(Proc, G^*(\alpha))$ is a $T/\!\!/G$-algebra. It remains to show that, given any other $T/\!\!/G$-algebra $(B, \beta)$, there is a unique morphism $f$ from *Proc* to $B$ such that the following diagram in $\check{\mathcal{V}}/\!\!/G$ commutes:

$$\begin{tikzcd}
(T/\!\!/G)Proc \arrow[r, "G^*(\alpha)"] \arrow[d, "(T/\!\!/G)_f"'] & Proc \arrow[d, "f"] \\
(T/\!\!/G)_B \arrow[r, "\beta"'] & B
\end{tikzcd}$$

Notice that the same diagram can be read in $\check{\mathcal{V}}$ as follows:

$$\begin{tikzcd}
G \times TProc \arrow[r, "\mathrm{id}_G \times \alpha"] \arrow[d, "\langle \pi, Tf \circ st_{G,Proc} \rangle"'] & G \times Proc \arrow[d, "f"] \\
G \times TB \arrow[r, "\beta"'] & B
\end{tikzcd}$$

We define $f$ as follows:

$$\begin{aligned}
f_X(\langle g, 0 \rangle) &\triangleq \beta_X(\langle g, in_1(*) \rangle) \\
f_X(\langle g, \tau.P \rangle) &\triangleq \beta_X(\langle g, in_2(f_X(\langle g, P \rangle)) \rangle) \\
f_X(\langle g, P_1|P_2 \rangle) &\triangleq \beta_X(\langle g, in_3(\langle f_X(\langle g, P_1 \rangle), f_X(\langle g, P_2 \rangle) \rangle) \rangle) \\
f_X(\langle g, [x \neq y]P \rangle) &\triangleq \beta_X(\langle g, in_4(\langle x, y, f_X(\langle g, P \rangle) \rangle) \rangle) \\
f_X(\langle g, (\nu x)P \rangle) &\triangleq \beta_X(\langle g, in_5(\gamma_{B,X}(f_{X \uplus \{x\}}(\langle G_{in_X}(g), P \rangle))) \rangle)
\end{aligned}$$

Commutativity of the previous diagram is proved by cases on $P$ (Appendix C.0.13).

The uniqueness of $f$ follows by its definition. Given any other homomorphism $g : (Proc, G^*(\alpha)) \longrightarrow (B, \beta)$ such that $g \bullet G^*(\alpha) = \beta \bullet (T/\!\!/G)_g$, rewriting the previous equality and simplifying it according to the definitions of $G^*(\alpha)$ and $st_{G,Proc}$ we obtain exactly the definition by cases of $f$. $\square$

Now we have the necessary tools for proving our goal:

**Theorem 5.12** *The model $\mathcal{U}$ validates $Ind^\iota$, i.e., the following holds:*

$$\begin{aligned}
\emptyset \vdash_\Sigma \ \forall R{:}\iota \to o.((R\ 0) \Rightarrow (\forall P{:}\iota.(R\ P) \Rightarrow (R\ \tau.P)) \Rightarrow \\
(\forall P{:}\iota.(R\ P) \Rightarrow \forall Q{:}\iota.(R\ Q) \Rightarrow (R\ P|Q)) \Rightarrow \\
(\forall y{:}\upsilon.\forall z{:}\upsilon.\forall P{:}\iota.(R\ P) \Rightarrow (R\ [y \neq z]P)) \Rightarrow \\
(\forall P{:}\upsilon \to \iota.(\forall x{:}\upsilon.(R\ (P\ x))) \Rightarrow (R\ \nu P)) \Rightarrow \\
\forall P{:}\iota.(R\ P))
\end{aligned}$$

*Proof.* By Proposition 5.5, we have to prove that for all $X \in \mathcal{V}$ the following judgment holds:

$$\begin{aligned}
X \Vdash_{\emptyset,*} \ \forall R{:}\iota \to o.((R\ 0) \Rightarrow (\forall P{:}\iota.(R\ P) \Rightarrow (R\ \sigma.P)) \Rightarrow \\
(\forall P{:}\iota.(R\ P) \Rightarrow \forall Q{:}\iota.(R\ Q) \Rightarrow (R\ P|Q)) \Rightarrow \\
(\forall y{:}\upsilon.\forall z{:}\upsilon.\forall P{:}\iota.(R\ P) \Rightarrow (R\ [y \neq z]P)) \Rightarrow \\
(\forall P{:}\upsilon \to \iota.(\forall x{:}\upsilon.(R\ (P\ x))) \Rightarrow (R\ \nu P)) \Rightarrow \\
\forall P{:}\iota.(R\ P))
\end{aligned}$$

By Theorem 5.1, this is equivalent to prove that, under the following assumptions

$$\begin{aligned}
&Y \Vdash_{R:\iota\to o,\eta_R} (R\ 0), \\
&Y \Vdash_{R:\iota\to o,\eta_R} (\forall P{:}\iota.(R\ P) \Rightarrow (R\ \tau.P)), \\
&Y \Vdash_{R:\iota\to o,\eta_R} (\forall P{:}\iota.(R\ P) \Rightarrow \forall Q{:}\iota.(R\ Q) \Rightarrow (R\ P|Q)), \\
&Y \Vdash_{R:\iota\to o,\eta_R} (\forall y{:}\upsilon.\forall z{:}\upsilon.\forall P{:}\iota.(R\ P) \Rightarrow (R\ [y \neq z]P)), \\
&Y \Vdash_{R:\iota\to o,\eta_R} (\forall P{:}\upsilon \to \iota.(\forall x{:}\upsilon.(R\ (P\ x))) \Rightarrow (R\ \nu P)),
\end{aligned}$$

we have that for all $Z \in \mathcal{V}$, for all $f \in \mathcal{I}(Y,Z)$ and for all $\eta_P \in Proc_Z$, the judgment $Z \Vdash_{(R:\iota\to o,P:\iota),\langle (Proc\Rightarrow Prop)_f(\eta_R),\eta_P\rangle} (R\ P)$ holds. This fact amounts to say that the following equation must hold:

$$p \triangleq [\![R:\iota\to o, P:\iota \vdash_\Sigma (R\ P):o]\!]^\upharpoonright = \top\circ !_{[\![(R:\iota\to o,P:\iota)]\!]}. \tag{5.9}$$

Consider the following pullback in $\check{\mathcal{V}}/\!\!/G$:

$$\begin{tikzcd} U \arrow[r, "G^*(!_U)"] \arrow[d, "h"'] & \mathbf{1} \arrow[d, "G^*(\top)"] \\ Proc \arrow[r, "p"'] & Prop \end{tikzcd}$$

where $G \triangleq [\![R:\iota\to o]\!]$. Then, from the assumptions above, we have that the following diagram in $\check{\mathcal{V}}/\!\!/G$ commutes (see Appendix C.0.14):

$$\begin{tikzcd} TU \arrow[r, dotted, "\beta"'] \arrow[rr, bend left, "G^*(!_{TU})"] \arrow[d, "T/\!\!/G(h)"'] & U \arrow[r, "G^*(!_U)"'] \arrow[d, "h"] & \mathbf{1} \arrow[d, "G^*(\top)"] \\ TProc \arrow[r, "G^*(\alpha)"'] & Proc \arrow[r, "p"'] & Prop \end{tikzcd}$$

Let $\beta : TU \to U$ be the unique map defined by the universal property of the pullback. Then, $(U,\beta)$ is a $T/\!\!/G$-algebra; therefore, by initiality of *Proc* (existential part) there is a map $h' \in \check{\mathcal{V}}/\!\!/G(Proc, U)$. Moreover, again by initiality of *Proc* (unicity part) we have $h \bullet h' = G^*(\mathrm{id}_{Proc})$. Hence we have the following:

$$p = p \bullet G^*(\mathrm{id}_{Proc}) = p \bullet h \bullet h' = G^*(\top) \bullet G^*(!_U) \bullet h'$$

Translating the equation in terms of the composition in the category $\check{\mathcal{V}}$, we get

$$p = G^*(\top) \circ \langle \pi, G^*(!_U) \circ \langle \pi, h'\rangle\rangle = \top\circ !_U \circ h' = \top\circ !_{G\times Proc}$$

i.e., the thesis (5.9). □

## 5.7.2 Higher-order induction

As in the case of recursion, also the induction principle can be generalized to higher-order processes. The higher-order induction principle is given in Figure 5.8. Notice that in the case of $n = 0$, this rule degenerates in that for first-order terms introduced above.

The proofs of the validity of higher-order induction principles follow the same pattern of the first-order case, exploiting the initiality of the corresponding higher-order initial algebra (see Section 5.6) and the following result which extends Lemma 7.8 of [Jac95] to the exponentiation of functors in the category $\check{\mathcal{V}}$:

**Lemma 5.4** *If $T : \check{\mathcal{V}} \longrightarrow \check{\mathcal{V}}$ is strong, then the functor $Var \Rightarrow T$ (whose action on objects is $A \mapsto Var \Rightarrow TA$) is strong as well.*

$$
\frac{\Gamma \vdash_\Sigma R : (\upsilon^n \to \iota) \to o}{\begin{array}{l}\Gamma \vdash_\Sigma (R\ \lambda\vec{x}{:}\upsilon.0) \Rightarrow (\forall P{:}\upsilon^n \to \iota.(R\ P) \Rightarrow (R\ \lambda\vec{x}{:}\upsilon.(\tau.(P\ \vec{x})))) \Rightarrow \\ \qquad (\forall P{:}\upsilon^n \to \iota.(R\ P) \Rightarrow \forall Q{:}\upsilon^n \to \iota.(R\ Q) \Rightarrow (R\ \lambda\vec{x}{:}\upsilon.(P\ \vec{x})|(Q\ \vec{x}))) \Rightarrow \\ \qquad (\forall P{:}\upsilon^n \to \iota.\forall y{:}\upsilon.\forall z{:}\upsilon.(R\ P) \Rightarrow (R\ \lambda\vec{x}{:}\upsilon.[y \neq z](P\ \vec{x}))) \Rightarrow \\ \qquad (\forall P{:}\upsilon^n \to \iota.\forall z{:}\upsilon.(R\ P) \Rightarrow \bigwedge_{i=1}^{n}(R\ \lambda\vec{x}{:}\upsilon.[x_i \neq z](P\ \vec{x}))) \Rightarrow \\ \qquad (\forall P{:}\upsilon^n \to \iota.\forall y{:}\upsilon.(R\ P) \Rightarrow \bigwedge_{j=1}^{n}(R\ \lambda\vec{x}{:}\upsilon.[y \neq x_j](P\ \vec{x}))) \Rightarrow \\ \qquad (\forall P{:}\upsilon^n \to \iota.(R\ P) \Rightarrow \bigwedge_{i,j=1}^{n}(R\ \lambda\vec{x}{:}\upsilon.[x_i \neq x_j](P\ \vec{x}))) \Rightarrow \\ \qquad (\forall P{:}\upsilon^{n+1} \to \iota.(\forall y{:}\upsilon.(R\ \lambda\vec{x}{:}\upsilon.(P\ \vec{x}\ y))) \Rightarrow (R\ \lambda\vec{x}{:}\upsilon.\nu(P\ \vec{x}))) \Rightarrow \\ \qquad \forall P{:}\upsilon^n \to \iota.(R\ P)\end{array}} \quad (Ind^{\upsilon^n \to \iota})
$$

Figure 5.8: Higher-order induction principle.

*Proof.* Let $st$ be the strength of $T$; we define (up-to suitable isomorphisms) the strength $st'$ for $Var \Rightarrow T$ as $(st'_{A,B})_X \triangleq (st_{A,B})_{X \uplus \{x\}} \circ (A_{in} \times id)$, where $X \xhookrightarrow{in} X \uplus \{x\}$ is the obvious injection. More explicitly, for $A, B \in \check{\mathcal{V}}$ and $X \in Var$:

$$
\begin{array}{rlll}
(st'_{A,B})_X : & A_X \times (TB)_{X \uplus \{x\}} & \longrightarrow & (TA \times B)_{X \uplus \{x\}} \\
& \langle a, b \rangle & \longmapsto & (st_{A,B})_{X \uplus \{x\}}(A_{in}(a), b)
\end{array}
$$

It is easy to check that $st'$ is a strength for the functor $Var \Rightarrow T$, that is, the following diagrams commute:

$$
\begin{tikzcd}
A_X \times (TB)_{X \uplus \{x\}} \arrow[r, "A_{in\,X} \times id"] \arrow[rrd, "\pi'"'] & A_{X \uplus \{x\}} \times (TB)_{X \uplus \{x\}} \arrow[r, "(st_{A,B})_{X \uplus \{x\}}"] \arrow[rd, "\pi'"] & (TA \times B)_{X \uplus \{x\}} \arrow[d, "(T\pi')_{X \uplus \{x\}}"] \\
& & (TB)_{X \uplus \{x\}}
\end{tikzcd}
$$

$$
\begin{tikzcd}
A_X \times (B_X \times (TC)_{X \uplus \{x\}}) \arrow[r, "A_{in} \times (B_{in} \times id)"] \arrow[dd, "\sim"] & A_{X \uplus \{x\}} \times (B \times (TC))_{X \uplus \{x\}} \arrow[r, "id \times st_{B,C}"] \arrow[dd, "\sim"] & A_{X \uplus \{x\}} \times ((TB \times C)_{X \uplus \{x\}}) \arrow[d, "st_{A,B \times C}"] \\
& & (TA \times (B \times C))_{X \uplus \{x\}} \arrow[d, "\sim"] \\
(A_X \times B_X) \times (TC)_{X \uplus \{x\}} \arrow[r, "(A \times B)_{in} \times id"] & (A \times B)_{X \uplus \{x\}} \times (TC)_{X \uplus \{x\}} \arrow[r, "st_{A \times B,C}"] & (T(A \times B) \times C)_{X \uplus \{x\}}
\end{tikzcd}
$$

In the latter diagram, the left square is the naturality of the associativity isomorphism of product, and the right part is the property of the strength $st$. □

An alternative way of validating higher-order induction principles is to exploit the result of Section 4.5.2. Indeed, since we already proved the validity of the axioms of the Theory of Contexts and of the first-order induction principle, the validity of higher-order induction principles is automatically[4] granted since they can be derived using the proof technique illustrated in Section 4.5.2.

[4] Obviously, we need to add the type of natural numbers and the related induction principle, but this can be done in a straightforward way.

## 5.8 Connections with tripos theory

As anticipated, the constructions and the results carried out in this chapter can be reread from the point of view of *tripos theory*. This can be useful in order to relate our work with other recent research on the use of functor categories to model the notions of variable binding and freshness. Moreover, it is very funny to see how a work, which required pages and pages of complex handmade proofs, can be obtained in less than three pages by means of general, although quite esoteric, properties.

In the following we suppose the reader is familiar with the notions of topos [Joh77, MM92, Bel88], Lawvere-Tierney topology and sheaf (see, *e.g.*, [Joh77, MM92]). More precisely, in the following we will use the following results:

1. Given a topos $\mathcal{E}$, there is a (contravariant) functor $Sub : \mathcal{E}^{op} \longrightarrow \mathcal{S}et$ associating to every $X \in \mathcal{E}$ the set of its subobjects, and to every arrow $f \in \mathcal{E}(X, Y)$ the function $Sub(f) : Sub(Y) \longrightarrow Sub(X)$ defined by $Sub(f)([m]) = [f^{-1}(m)]$ for any monic $m$ with codomain $Y$. Moreover, we have that the subobject classifier $\Omega$ of $\mathcal{E}$ is a representing object for $Sub$, i.e., $Sub \cong \mathcal{E}(\_, \Omega)$. In general the partially ordered set $Sub(X)$ is a Heyting algebra and the function $Sub(f)$ is a Heyting algebra morphism. A topos is said *Boolean* if, for every $X \in \mathcal{E}$, the Heyting algebra $Sub(X)$ is a Boolean algebra (in this case $Sub(f)$ is a morphism of Boolean algebras).

2. Given a Lawvere-Tierney topology $j$ on the topos $\mathcal{E}$, the subobject classifier, denoted by $\Omega_j$, in the topos of $j$-sheaves $Sh_j\ \mathcal{E}$ is the equalizer of $\mathrm{id}_\Omega$ and $j$. Actually, $\Omega_j$ classifies the $j$-closed monomorphisms, and the subsheaves of a sheaf are exactly its closed subobjects ([MM92], §V.2 Theorem 2). Moreover the inclusion functor $I : Sh_j\ \mathcal{E} \longrightarrow \mathcal{E}$ has a left adjoint $\mathbf{a} : E \longrightarrow Sh_j\ \mathcal{E}$ preserving finite limits[5] ([MM92], §V.3 Theorem 1). These two facts imply that there is an isomorphism between $j$-closed subobjects of $X$ and subsheaves of $\mathbf{a}(X)$.

3. If *false* is the characteristic map of the unique arrow $\mathbf{0} \longrightarrow \mathbf{1}$ and $\neg$ is the characteristic map of *false*, the morphism $\neg \circ \neg : \Omega \longrightarrow \Omega$ is a Lawvere-Tierney topology on $\mathcal{E}$ and $Sh_{\neg\neg}\ \mathcal{E}$ is a Boolean topos ([MM92], §VI.1 Theorem 3).

4. Finally if $\mathcal{E} \triangleq \mathcal{S}et^{\mathcal{C}}$ for some small category $\mathcal{C}$, then the functor $\Omega$ defined by $\Omega_X = Sub(\mathcal{E}(X, \_))$ and $\Omega_f(F) = Sub(\mathcal{E}(X, f))(F)$ is the subobject classifier of $\mathcal{E}$; so the subobject classifier $\Omega_{\neg\neg}$ in the topos of $\neg\neg$-sheaves is given by $\Omega_{\neg\neg}(X) = \{F \mid F \text{ is a subobject } \neg\neg\text{-closed of } \mathcal{E}(X, \_)\}$ on objects and the restriction of $\Omega$ on morphisms.

Now, we show how these notions and results are related to the properties of $\mathbf{Pred}_{\check{\mathcal{I}}}$. First, notice that the closure condition (5.3) in the definition of $\mathbf{Pred}_{\check{\mathcal{I}}}$ is exactly the request that a subfunctor is closed w.r.t. the $\neg\neg$-topology. Indeed, following the proof of § VI.1 Lemma 4 in [MM92], the verification is straightforward: by using twice the following description of $\neg U$, for $U \rightarrowtail A$ mono in $\check{\mathcal{I}}$:

$$(\neg U)_X = \{a \mid a \in A_X \text{ and, for all } h : X \longrightarrow Y, A_h(a) \notin U_Y\}$$

[5] The functor $\mathbf{a}$ is called the *associated sheaf functor* or the *sheafification functor*; for any $E \in \mathcal{E}$, $\mathbf{a}(E)$ is called the sheaf associated to $E$ or the sheafification of $E$.

one obtains

$$\neg(\neg U)_X = \{x \mid x \in F_X \text{ and, for all } f : X \longrightarrow Y, \text{ there exists } Z \in \check{\mathcal{I}} \text{ and } g \in \check{\mathcal{I}}(Y, Z) \text{ such that } A_{g\circ f}(x) \in U_Z\}.$$

As a consequence, $\mathbf{Pred}_{\check{\mathcal{I}}}$ is the functor $Sub$ in the topos of $\neg\neg$-sheaves of $\check{\mathcal{I}}$. This immediately implies Proposition 5.3.

The previous remarks allow one to conclude that $\Omega \in \check{\mathcal{I}}$ is precisely the subobject classifier in the topos $Sh_{\neg\neg}(\check{\mathcal{I}})$. Whence, Proposition 5.4 follows by observing that $\Omega$ is a representing object for $Sub$ (see the first point at the beginning of this section). We remark that, actually, $\Omega_X$ is a two-element set.

In order to define **Pred** we must resort to tripos theory. In the literature there are several slightly different definitions of tripos(see, e.g., [HJP80, Pit81, vO91, Jac99]); the following one, is good for our purposes:

**Definition 5.6** *Let $\mathcal{C}$ be a category with finite products. A $\mathcal{C}$-*tripos *is a functor $\mathcal{P} : \mathcal{C}^{op} \longrightarrow \mathcal{S}et$ such that*

*i) for each $A \in \mathcal{C}$, $\mathcal{P}(A)$ is a Heyting algebra*

*ii) for each $f \in \mathcal{C}(A, B)$*

*(a) $\mathcal{P}(f)$ is a homomorphism of Heyting algebras*

*(b) $\mathcal{P}(f)$ has left and right adjoints $\exists f$ and $\forall f$ which satisfy the Beck-Chevalley condition: if*

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle h} \\ C & \xrightarrow{k} & D \end{array}$$

*is a pullback square then $\exists f \circ \mathcal{P}(k) = \mathcal{P}(g) \circ \exists h$ (and hence also the dual condition for $\forall$ holds)*

*iii) $\mathcal{P}$, when regarded as a set-valued functor, is representable, i.e., there is an object $Prop \in \mathcal{C}$ such that for all $A$: $\mathcal{P}(A) \cong \mathcal{C}(A, Prop)$.*

Since our final goal is to show that **Pred** is a tripos on $\check{\mathcal{V}}$, we need the following results:

**Proposition 5.10 ([Pit81], Example 1.3 (i))** *If $\mathcal{E}$ is a topos, the functor $Sub : \mathcal{E}^{op} \longrightarrow \mathcal{S}et$ carries the structure of a tripos.*

**Proposition 5.11 ([vO91], Prop. 1.4)** *If $\mathcal{C}, \mathcal{D}$ are categories with finite products, $F \dashv G : \mathcal{C} \longrightarrow \mathcal{D}$, $F$ preserves products and $\mathbf{Pred}_{\mathcal{D}}$ is a tripos on $D$, then the functor $\mathbf{Pred}_{\mathcal{C}}$ defined by*

$$\begin{array}{rrcl} \mathbf{Pred}_{\mathcal{C}} : & \mathcal{C}^{op} & \longrightarrow & \mathcal{S}et \\ & X & \longmapsto & \mathbf{Pred}_{\mathcal{D}}(F(X)) \\ & (X \xrightarrow{f} Y) & \longmapsto & \mathbf{Pred}_{\mathcal{D}}(F(f)) \end{array}$$

*is a tripos on $C$.*

As anticipated, we can now prove the following:

**Proposition 5.12** **Pred** *is a tripos on* $\check{\mathcal{V}}$.

*Proof.* Consider the adjunctions $\mathbf{a} \dashv I : \check{\mathcal{I}} \longrightarrow Sh_{\neg\neg}\, \check{\mathcal{I}}$ and $(\_)^r \dashv (\_)^* : \check{\mathcal{V}} \longrightarrow \check{\mathcal{I}}$, where $\mathbf{a}$ and $(\_)^r$ preserve products. By Proposition 5.11 and the fact that $\neg\neg\text{-}Sub(F) \cong Sub_{Sh_{\neg\neg}(\check{\mathcal{I}})}(\mathbf{a}(F))$, the functor $\mathbf{Pred}_{\check{\mathcal{I}}}$ is a tripos. Another application of Proposition 5.11 immediately shows that **Pred** is a tripos on $\check{\mathcal{V}}$. □

A fundamental property of triposes is that, if $\mathcal{C}$ models some metalanguage then a $\mathcal{C}$-tripos models intuitionistic higher order logic over that metalanguage. This means that there is a type for propositions, term formers for implication and universal quantification. We can therefore interpret the logical judgment $\Gamma \vdash \phi$ which intuitively states that the proposition $\phi$, involving variables from $\Gamma$, holds. A standard result states that intuitionistic logic is sound w.r.t. this semantics [Pit81].

From the abovementioned results, it follows that all intuitionistic theorems hold in $\mathcal{U}$ (remember that the interpretation of logical propositions has been defined in terms of *Prop* which is the representing object of **Pred**). Moreover, since $\mathbf{Pred}(F)$ is a Boolean algebra, the logic of **Pred** is the full higher-order classical logic. This is a consequence of the fact that we consider only the $\neg\neg$-subobjects. In other words, although we work in $\check{\mathcal{V}}$, our logical propositions ultimately live in $Sh_{\neg\neg}\, \check{\mathcal{I}}$.

## 5.9 Related work

The application of functor categories in the semantics of programming languages goes back to the early '80s, when "variable" sets (i.e., objects of $\check{\mathcal{C}}$) were recognized as a useful tool to model variability of memory allocation in Algol-like languages [Rey81, Ole85]. An important step towards the generalization of this approach has been the *monad for allocation* over the category $\check{\mathcal{C}}$ [Mog89]. More recently, presheaf models have been extensively used for interpreting concurrency and mobility [Sta96, FMS96, CSW97].

Recently, the use of functor categories as a semantics for HOAS has been advocated in [FPT99, Hof99], the latter being the basis for the present work. At the same time, an alternative approach based on Frænkel-Mostowski set theory has been presented in [GP99]. Here we briefly illustrate the connections between these models.

In [Hof99], functor categories are used for formally justifying several logical principles which have been previously proposed for reasoning about HOAS. In particular, metalanguage types are interpreted as suitable objects of $\mathcal{S}et^{\mathcal{C}^{op}}$, where the index category $\mathcal{C}$ depends on the nature of the metalanguage. A key feature of this approach (which we have exploited in Proposition B.2) is that the interpretation of types which appear in negative position in the types of syntactic constructors must be representable. This allows to apply the following property:

$$\text{for } X \in \mathcal{C} \text{ and } F \in \mathcal{S}et^{\mathcal{C}^{op}} : (\mathcal{Y}(X) \Rightarrow F)_Y \cong F_{X \times Y}\,.$$

For instance, in the case of untyped $\lambda$-calculus, whose syntax is defined by

$$\begin{aligned} \text{Inductive } tm : Set := {}& isvar : var \to tm \mid app : var \times var \to tm \\ & \mid lam : (var \to tm) \to tm. \end{aligned}$$

the higher-order type $var \to tm$ is interpreted as the functor $Var \Rightarrow Tm$. Since $Var \cong \mathcal{Y}(\{x\})$, we have that $(Var \Rightarrow Tm)_X \cong Tm_{X \uplus \{x\}}$. In other words, functions over variables correspond exactly to terms with an extra variable, which can be seen as the "hole" of a particular context such that the act of filling it resembles a capture-avoiding substitution rather than a, possibly capturing, textual one. Thus, the interpretation of $tm$ is the initial algebra of the functor $T(A) = Var + A \times A + Var \Rightarrow A$.

In order to interpret predicates, on the other hand, one cannot use the plain topos $\mathcal{S}et^{\mathcal{C}^{op}}$, because induction principles over higher-order types contradict the Axiom of Unique Choice. The solution originally conceived in [Hof99], and which has been fully developed in this work, is to resort to some tripos over the category of types.

Covariant presheaves are instead adopted in [FPT99] where a general methodology is developed in order to associate to every *binding signature* a category of models which gives a notion of initial algebra semantics. The models are presheaves which are both algebras for the signature functor and monoids with respect to the substitution. The choice of $\mathcal{F}$ (that is, the skeleton of the category of finite sets and functions) as the index category is motivated by the operations which are allowed on environments: names swapping, contraction and weakening. Indeed, the closure by composition of these operations generates exactly all the functions between finite cardinals. A key feature of the category $\mathcal{S}et^{\mathcal{F}}$ is that it has a type constructor

$$\delta : \mathcal{S}et^{\mathcal{F}} \to \mathcal{S}et^{\mathcal{F}} \qquad (\delta A)_X = A_{X+1} \quad (\delta A)_h = A_{h+\mathrm{id}_1}$$

which is used for interpreting higher-order types like $var \to tm$ of the previous example. Thus, the interpretation of $tm$ is the initial algebra of the functor $T(A) = Var + A \times A + \delta A$. Clearly, $\delta$ corresponds to the functor $Var \Rightarrow \_$ in Hofmann's approach, via the isomorphism previously described.

However, since $\check{\mathcal{F}}$ alone is proposed as a framework for higher-order abstract syntax, this work is fine for the purely algebraic aspect, i.e., terms and equations; as we have seen, in order to *reason* about HOAS, $\check{\mathcal{F}}$ alone is inadequate since for example equality of names cannot be expressed. The value of [FPT99] is to have placed inductive types like *Proc* in $\check{\mathcal{V}}$ in the context of universal algebra.

A different perspective is taken in [GP99], where a metalogic for specifying and reasoning about formal systems with name binders is introduced using as a semantic basis the *Frænkel-Mostowski* permutation model of set theory with atoms. Terms over a signature with binders are interpreted as an inductive element of the universe of FM-sets. In other words, the quotient with respect to $\alpha$-equivalence is applied only to the interpretation of binding constructors, instead of being applied to the whole initial algebra (like in the case of a pure first-order syntax approach). Processes with free names are modeled as equivalence classes of name-process pairs, obtained by the *atom abstraction* operation: for each variable $x$ and FM-set $A$, there is a FM-set $x.A$, the *atom abstraction* of $A$ over $x$, defined by

$$x.A \triangleq \{(y, (y\ x) \cdot A) \mid y \in Var \wedge (y = x \vee y \# A)\}$$

where $(y\ x) \cdot A$ is the *variable transposition*, and $y \# A$ is the *apartness* (which intuitively means that $y$ does not occur "free" in $A$). Then, the FM-set of *abstractions* over variables of elements of $A$ is defined by

$$[Var]A \triangleq \{b \mid \exists x \in Var.\exists a \in A.(b = x.a \wedge x \# A)\} .$$

Thus, if $A$ is the interpretation of some syntactic class, $[Var]A$ gives the interpretation of contexts of type $A$. In our running example, the interpretation of $tm$ is the FM-set defined as the least fixed point of the (FM-set valued) function $F_\alpha(A) = Var + A \times A + [Var]A$.

As a result, the usual arguments about least fixed points allow for deriving induction and recursion principles over the higher-order abstract syntax.

In order to highlight the close connection between this latter approach and the previous ones, notice that the universe of sets used in [GP99] is the category of the perm($Var$)-sets with finite support and equivariant functions. This is very closely related to the so-called *Schanuel topos*[6], which is isomorphic to the category of sheaves over $\mathcal{I}^{op}$ for the $\neg\neg$-topology ([MM92], § III.9). As we have noticed in Section 5.8, this category is exactly the topos we have used in our model for the interpretation of logical judgments. Both $Sh_{\neg\neg}(\check{\mathcal{I}})$ and the Schanuel topos embed in $\check{\mathcal{I}}$, which is related to $\check{\mathcal{V}}$ by the adjunction of Proposition 5.2. The reason we could not use $Sh_{\neg\neg}(\check{\mathcal{I}})$ to interpret terms and functions and resorted to $\check{\mathcal{V}}$ instead was precisely that otherwise processes would not be an inductive type with $\upsilon \to \iota$ (as opposed to equivalence classes of pairs) among the argument types of the constructors.

A final remark is about the peculiar behaviour of the interpretation of abstraction and instantiation in [GP99]. In our approach both can be rendered naturally using the features of the metalanguage: the first as $\lambda$-abstraction, the latter as application. On the other hand, notice that instantiation $A@x$ in FM is only partially defined, *i.e.*, when $x$ is not in the support of $A$, *i.e.*, the free variables of $A$. Actually, the abstraction set of FM has a clear correspondent in our categorical setting. Recall that in our model, the type constructor "$\upsilon \to \_$" is interpreted exactly as the exponentiation $Var \Rightarrow \_ : \check{\mathcal{V}} \longrightarrow \check{\mathcal{V}}$ (Section 5.4.2). The corresponding operation in $\check{\mathcal{I}}$ via the adjunction (*i.e.*, the restriction) is not the usual exponentiation of $\check{\mathcal{I}}$, but only a certain arrow $Var \multimap \_ : \check{\mathcal{I}} \longrightarrow \check{\mathcal{I}}$ which is the right adjunct of a suitable tensor product. This arrow corresponds exactly to the exponent in the Schanuel topos, and ultimately it corresponds to $[Var]\_$ of FM. Using such a somewhat "linear" exponentiation has the advantage that, like in any topos, the Axiom of Unique Choice holds and thus it can be consistently assumed.

[6]Indeed, the Schanuel topos can be defined so that its objects are the elements with empty support of the universe of FM-sets. In order to get at FM-sets with non-empty support within the Schanuel topos, one must consider internal families of subobjects.

# 6

# Case studies

This chapter is devoted to the presentation of two case studies on the applicability of the Theory of Contexts to the formal development of the metatheory of nominal calculi. More precisely, the former concerns the well-known untyped $\lambda$-calculus, while the second deals with the Ambient Calculus and the related modal logic introduced in [CG00] and further extended in [CG01]. Both case studies follow the research line started with the formal development of the metatheory of the $\pi$-calculus in [HMS01b] and continued in [Mic01a] with the formal development of the theory of capture-avoiding substitution for a HOAS-based encoding of untyped $\lambda$-calculus. The main motivation in pursuing this investigation of the applicability of the Theory of Contexts to "real" and complex case studies is the hope to get some hints on the expressive power of its axioms. Indeed, despite the fact that we have proved in Chapter 5 their soundness, we have not yet a completeness result w.r.t. some known theory.

## 6.1 $\alpha$-equivalence for the untyped $\lambda$-calculus

In this section we present a case study concerning the formalization of $\alpha$-equivalence for the untyped $\lambda$-calculus. Despite the fact that in every textbook or introductory course on $\lambda$-calculus, the $\alpha$-equivalence relation is presented as a trivial or natural notion (often without an explicit axiomatization, but only a short definition in words followed by some examples[1]), its formalization in a type theory-based logical framework reveals many subtleties. Actually, $\alpha$-equivalence can be defined rigorously in more than one way; for instance, besides the "conventional" one [Bar81], there is the variant used by McKinna-Pollack [MP99], and Gabbay-Pitts' alternative, based on the notion of *variable transposition* [GP99, GP01]. It is therefore natural to show formally that these three definitions are really equivalent. Moreover, the development of the metatheory, namely the proofs of the reflexivity and transitivity of $\alpha$-equivalence, requires to show a non-trivial invariance property with respect to variable renamings. It is interesting to notice that such a property plays the same rôle of similar renaming results needed in the formal development of the metatheory of well known nominal calculi (namely, Lemma 6 of [MPW92] part II for the $\pi$-calculus and Lemma 2-3 of [CG01] for the ambient calculus). In other words, it seems that nominal calculi enjoy a common

[1] Commonly, $\alpha$-equivalence is presented as a *convention* allowing to identify $\lambda$-terms which differ only by a change of bound variables with fresh ones.

property stating that we can freely replace a name/variable in a proof with a fresh one. This corresponds to the notion of *equivariance* (see [Pit01b]).

### 6.1.1 Encoding the untyped $\lambda$-calculus

We recall that the syntax of untyped $\lambda$-calculus is defined by the following grammar:

$$\Lambda : \qquad M ::= x \mid MN \mid \lambda x.M,$$

where $x \in \mathcal{V}$ ($\mathcal{V}$ is an infinite set of variables). Obviously, since we want to encode $\alpha$-equivalence of $\lambda$-terms and reason about its (meta)properties, we cannot use a HOAS-based approach for representing the binder $\lambda$. Otherwise $\alpha$-equivalence would be automatically granted by the metalanguage and could not be accessed at the object level.

In the following we will denote by $\Sigma_\Lambda$ the signature of the present case study; starting from an empty one, we will progressively add the needed constants. So, we proceed by introducing the first constant, namely, a type `var` representing the set of variables:

```
Parameter var: Set.
```

More precisely, since the set `var` has no constructors, variables of the object language will be represented by variables of `Coq` of type `var`. It is worth noticing that, even if we are going to give a plain first-order encoding of $\lambda$-terms, we cannot take `var` as an inductive type if we want to take advantage of the axioms of the Theory of Contexts. Indeed, as we notice in Chapter 4 (Section 4.4), those axioms, namely, the extensionality law, would give rise to inconsistencies in presence of an inductive encoding of the type of variables.

The set of $\lambda$-terms is represented by the following inductive type:

```
Inductive tm : Set :=
  is_var : var -> tm | app : tm  -> tm -> tm | lam : var -> tm -> tm.
```

In the following, for $X \triangleq \{x_1, \ldots, x_n\} \subset \mathcal{V}$ finite, we will denote by $\Lambda_X$ the set $\{M \mid M \in \Lambda, v(M) \subseteq X\}$, where $v(M)$ is the set of all the variables occurring (free, binding, or bound) in $M$. Moreover, we will denote by $\Gamma_X$ the typing environment given by:

$$\{\mathtt{x1} : \mathtt{var}, \ldots, \mathtt{xn} : \mathtt{var}\} \cup \{\mathtt{dij} : \tilde{}(\mathtt{xi} = \mathtt{xj}) \mid \mathtt{1} \leq \mathtt{i} < \mathtt{j} \leq \mathtt{n}\}$$

Finally $\mathtt{tm}_X$ will represent the canonical forms `M` (i.e. $\beta\eta$-head normal forms) of type `tm` such that $\Gamma_X \vdash_{\Sigma_\Lambda} \mathtt{M} : \mathtt{tm}$. As we can see, there is a one-to-one correspondence between the productions of the grammar defining the syntax of the object language and the constructors of type `tm`. This fact is formalized by the encoding and decoding functions depicted in Figure 6.1 and by the following adequacy result:

**Proposition 6.1** *For each $X \subset \mathcal{V}$ finite, $\epsilon^\Lambda_X$ is a compositional bijection between $\Lambda_X$ and* $\mathtt{tm}_X$.

*Proof.* Standard, using the definitions in Figure 6.1 and proceeding by induction on the structure of $\lambda$-terms and of canonical forms of type `tm`. $\square$

$$\begin{array}{rcl} \epsilon^{\Lambda}_{X} & : & \Lambda_X \longrightarrow \mathtt{tm}_X \\ \epsilon^{\Lambda}_{X}(x) & \triangleq & (\mathtt{is\_var\ x}) \\ \epsilon^{\Lambda}_{X}(MN) & \triangleq & (\mathtt{app}\ \epsilon^{\Lambda}_{X}(M)\ \epsilon^{\Lambda}_{X}(N)) \\ \epsilon^{\Lambda}_{X}(\lambda x.M) & \triangleq & (\mathtt{lam\ x}\ \epsilon^{\Lambda}_{X}(M)) \end{array} \qquad \begin{array}{rcl} \delta^{\Lambda}_{X} & : & \mathtt{tm}_X \longrightarrow \Lambda_X \\ \delta^{\Lambda}_{X}((\mathtt{is\_var\ x})) & \triangleq & x \\ \delta^{\Lambda}_{X}((\mathtt{app\ M\ N})) & \triangleq & \delta^{\Lambda}_{X}(\mathtt{M})\delta^{\Lambda}_{X}(\mathtt{N}) \\ \delta^{\Lambda}_{X}((\mathtt{lam\ x\ M})) & \triangleq & \lambda x.\delta^{\Lambda}_{X}(\mathtt{M}) \end{array}$$

Figure 6.1: Encoding and decoding functions for the untyped $\lambda$-calculus.

$$\frac{x \neq y}{x \notin^{\Lambda}_{\mathcal{V}} y} \quad (\text{NOTIN\_VAR})$$

$$\frac{x \notin^{\Lambda}_{\mathcal{V}} M, \quad x \notin^{\Lambda}_{X} N}{x \notin^{\Lambda}_{\mathcal{V}} MN} \quad (\text{NOTIN\_APP})$$

$$\frac{x \notin^{\Lambda}_{\mathcal{V}} M, \quad x \neq y}{x \notin^{\Lambda}_{\mathcal{V}} \lambda y.M} \quad (\text{NOTIN\_LAM})$$

Figure 6.2: Non-occurrence predicate for the untyped $\lambda$-calculus.

### 6.1.2 The Theory of Contexts for the untyped $\lambda$-calculus

Before introducing the properties of the Theory of Contexts, we need to define an auxiliary binary predicate in order to formalize the notion of non-occurrence (neither free nor bound) of a variable in a $\lambda$-term. This notion of "freshness" is rendered by means of the following inductive predicate:

```
Inductive notin [x:var]: tm -> Prop :=
    notin_var: (y:var)~x=y -> (notin x (is_var y))
  | notin_app: (M,N:tm)(notin x M) -> (notin x N) -> (notin x (app M N))
  | notin_lam: (y:var)(M:tm)(notin x M) -> ~x=y -> (notin x (lam y M)).
```

As anticipated, the intuitive meaning of `(notin x M)` is that the variable `x` does not occur (neither free nor bound) in the `M`. It is clear that the definition of the `notin` predicate is directly driven by the inductive definition of type `tm` (one case for each constructor of `tm`). The next adequacy result states that `notin` represents the predicate $\notin^{\Lambda}_{\mathcal{V}}$ whose rules (as one would write them "on paper") are depicted in Figure 6.2.

**Proposition 6.2 (Adequacy of `notin`)** *1. (Soundness) Let $X \subset \mathcal{V}$ finite, $x \in \mathcal{V}$, if* `t` *is a canonical form such that $\Gamma_{X\cup\{x\}} \vdash_{\Sigma_\Lambda}$* `t` *: (*`notin x M`*), then we have $x \notin^{\Lambda}_{\mathcal{V}} \delta^{\Lambda}_{X}(\mathtt{M})$.*

*2. (Completeness) Let $X \subset \mathcal{V}$ finite, $x \in \mathcal{V}$, $M \in \Lambda_X$, then if $x \notin^{\Lambda}_{\mathcal{V}} M$ there is a canonical form* `t` *such that $\Gamma_{X\cup\{x\}} \vdash_{\Sigma_\Lambda}$* `t` *: (*`notin x` *$\epsilon^{\Lambda}_{X}(M)$).*

*Proof.* This result can be proved easily by means of Proposition 6.1 and the following techniques:

1. (Soundness) induction on the structure of the normal forms `t`.
2. (Completeness) induction on the structure of the derivation of the non-occurrence judgment. □

The `notin` predicate will allow us, by means of the unsaturation axiom of the Theory of Contexts, to always have at hand *fresh* variables whenever this is needed.

Even if there are no higher-order constructors in the definition of the type `tm`, all the axioms of the Theory of Contexts turned out to be useful or even necessary in the formal development illustrated in the next sections. As the type `var` is concerned we assume that the equivalence between names is decidable:

```
Axiom dec: (x,y:var)x=y \/ ~x=y.
```

Obviously, in the case of a logical framework implementing classical logic this axiom is an instantiation of the law of excluded middle and does not need to be explicitly assumed.

The second axiom about the type `var` is the unsaturation, but here we prefer to assume a "general" form which is independent from the particular syntax of the object language. Hence, we introduce a type encoding (finite) lists of variables:

```
Inductive var_list: Set :=
  empty: var_list | cons : var -> var_list -> var_list.
```

The next step is to define a predicate `notin_list` allowing to express the non occurrence of a variable in a given list:

```
Inductive notin_list [x:var]: var_list -> Prop :=
    notin_empty: (notin_list x empty)
  | notin_cons : (y:var)(l:var_list)~x=y -> (notin_list x l) ->
                 (notin_list x (cons y l)).
```

At this point we can introduce the following general axiom of unsaturation:

```
Axiom unsat_list: (l:var_list)(Ex [x:var](notin_list x l)).
```

It simply states that, for every list of variables, we can always pick a new variable not occurring in it. The usual unsaturation axiom of the Theory of Contexts can be derived from `unsat_list` by means of the following result:

```
Lemma TM_VAR_LIST: (M:tm)
                   (Ex [l:var_list](x:var)(notin_list x l) -> (notin x M)).
```

The previous lemma states that, for every term `M`, there is a list of variables such that if a given variable does not occur in the list, then it cannot occur in `M` either. The proof is an easy induction on the structure of `M`. As anticipated, we can now derive the unsaturation property for terms of type `tm`:

```
Lemma UNSAT: (M:tm)(Ex [x:var](notin x M)).
```

The lack of higher-order constructors in `tm` allows us to derive the axiom of $\beta$-expansion for plain terms by means of an easy structural induction on `M`:

```
Lemma EXP: (M:tm)(x:var)(Ex [N:var->tm](notin_context x N) /\ M=(N x)).
```

The decidability of occur checking is also derivable by means of an easy structural induction over `M`:

```
Lemma NOTIN_DEC: (M:tm)(x:var)(notin x M) \/ (isin x M).
```

where `(isin x M)` holds if and only if the variable `x` occurs in `M`. For the sake of completeness, we also give the inductive definition of the `isin` predicate:

```
Inductive isin [x:var]: tm -> Prop:=
    isin_var: (isin x (is_var x))
  | isin_app: (M,N:tm)(isin x M) \/ (isin x N) -> (isin x (app M N))
  | isin_lam: (y:var)(M:tm)x=y \/ (isin x M) -> (isin x (lam y M)).
```

Like in the case of the `dec` axiom, in a classical logical framework this result could be obtained as a special case of the law of excluded middle, provided that `(isin x M)` is equivalent to `(notin x M)` (this is not difficult to prove).

On the other hand, the extensionality and monotonicity axioms still have to be postulated:

```
Axiom ext: (F,G:var->tm)(x:var)(notin_context x F)->(notin_context x G)->
           (F x)=(G x) -> F=G.

Axiom notin_mono: (M:var->tm)(x,y:var)(notin x (M y))->(notin_context x M).
```

where `notin_context` is the following abbreviation:

```
Definition notin_context:=[x:var][F:var->tm]((y:var)~x=y->(notin x (F y))).
```

Interestingly, it is possible to derive the following result by structural induction on `A` (using `ext` and `EXP`):

```
Lemma PRE_NOTIN_MONO: (A:tm)(M:var->tm)(z:var)(notin_context z M) -> A=(M z)
                      -> (x,y:var)(notin x (M y)) -> (notin_context x M).
```

However, the previous lemma is not sufficient in order to derive the monotonicity; indeed, in order to achieve such a result, we should be able to provide a term `A` and a variable `z` not occurring in the context `M` such that `A=(M x)`, while our unsaturation result works only for terms of type `tm`, not for functional terms of type `var->tm`. Ironically enough, the lack of a higher order constructor of the type `tm` is the main reason for this failure. Actually, in the case of a HOAS-based encoding the type of `lam` would be `(var->tm)->tm` and we could prove the unsaturation result for contexts by eliminating `UNSAT` over the first order term `(lam M)`.

In order to give an idea of the expressive power of the Theory of Contexts, we mention here that with the signature given so far, it is possible to derive a higher-order induction principle. More precisely, we have the following result:

```
Lemma HO_TM_IND: (P:(var->tm)->Prop)
                 ((x:var)(P [_:var](is_var x))) ->
                 (P is_var) ->
                 ((M,N:var->tm)(P M)->(P N)->(P [x:var](app (M x) (N x))))->
                 ((M:var->tm)(P M) -> (P [x:var](lam x (M x)))) ->
                 ((y:var)(M:var->tm)(P M) -> (P [x:var](lam y (M x)))) ->
                 (M:var->tm)(P M).
```

The proof technique is the same used in the lemma `PRE_NOTIN_MONO`: we first prove a weaker result, i.e.:

```
Lemma PRE_HO_TM_IND: (M:tm)(x:var)(N:var->tm)
                     (notin_context x N) -> M=(N x) ->
                     (P:(var->tm)->Prop)
                     ((x:var)(P [_:var](is_var x))) -> (P is_var) ->
                     ((p,q:var->tm)(P p) -> (P q) ->
                      (P [x:var](app (p x) (q x)))) ->
                     ((p:var->tm)(P p) -> (P [x:var](lam x (p x)))) ->
                     ((y:var)(p:var->tm)(P p) -> (P [x:var](lam y (p x))))->
                     (P N).
```

by structural induction on `M`. The $\beta$-expansion and extensionality axioms (resp. `EXP` and `ext`) play a fundamental role in the proof; indeed, the structural induction on terms of type `tm` automatically generated by the `Coq` proof assistant provides only information related to the structure of `M`, but in order to conclude we must know the structure of the context `N`. This important step is accomplished in the following way: since we know by hypothesis that `M=(N x)`, we can expand `M` into `(M' x)`, such that `(notin_context x M')` holds. It follows that `(N x)=(M' x)`, whence, by means of extensionality, we obtain `N=M'` which provides the information about the structure of `N` we were looking for. A practical example will make things more clear: let us suppose we are working on the case where `M=(app A B)`. So, we expand `M` w.r.t. the variable `x`, but this means expanding both `A` and `B` w.r.t. the variable `x` obtaining two contexts `A'` and `B'` such that `A=(A' x)`, `(notin_context x A')`, `B=(B' x)` and `(notin_context x B')`. Hence, we obtain that `M=(app (A' x) (B' x))` holds and consequently `(N x)=(app (A' x) (B' x))`. It follows by extensionality that `N=([_:var](app (A' _) (B' _)))`, whence we know that the context `N` is an application context.

Now, in order to obtain `HO_TM_IND`, we need to provide a term `A` and a variable `z` not occurring in the context `M` such that `A=(M x)`. This time we can do it because the higher-order unsaturation can be derived from `UNSAT` and `notin_mono`:

```
Lemma HO_UNSAT: (M:var->tm)(Ex [x:var](notin_context x M)).
```

Hence, in order to conclude, we can choose the variable, say `z`, provided by eliminating `HO_UNSAT` over `M` and the term `(M z)`.

### 6.1.3 Encoding the $\alpha$-equivalence relation (I)

We take as a starting point the following definition taken from [Bar81]:

> **Definition 6.1** *(i) A* change of bound variables *in $M$ is the replacement of a part $\lambda x.N$ of $M$ by $\lambda y.(N[x := y])$ where $y$ does not occur (at all) in $N$. (Because $y$ is fresh there is no danger in the substitution $N[x := y]$).*
>
> *(ii) $M$ is $\alpha$-congruent with $N$, notation $M \equiv_\alpha N$, if $N$ results from $M$ by a series of changes of bound variables.*

It is clear that a fresh renaming mechanism lies at the very heart of the notion of $\alpha$-equivalence. Hence, we must first approach the problem of representing the mechanism of changing a bound variable with a fresh one in a $\lambda$-term. This task is accomplished by introducing the following inductive predicate:

```
Inductive change_var [x,y:var]: tm -> tm -> Prop :=
    change_var_var1: (change_var x y (is_var x) (is_var y))
  | change_var_var2: (z:var)~x=z -> (change_var x y (is_var z) (is_var z))
```

$$\frac{-}{x[x:=y]=y} \quad \text{(CV\_VAR1)}$$

$$\frac{x \neq z}{z[x:=y]=z} \quad \text{(CV\_VAR2)}$$

$$\frac{M[x:=y]=M' \quad N[x:=y]=N'}{(MN)[x:=y]=M'N'} \quad \text{(CV\_APP)}$$

$$\frac{M[x:=y]=M'}{(\lambda x.M)[x:=y]=\lambda y.M'} \quad \text{(CV\_LAM1)}$$

$$\frac{M[x:=y]=M' \quad x \neq z}{(\lambda z.M)[x:=y]=\lambda z.M'} \quad \text{(CV\_LAM2)}$$

Figure 6.3: Changing a variable in a $\lambda$-term.

```
| change_var_app:  (R,S,R',S':tm)
                   (change_var x y R R') -> (change_var x y S S') ->
                   (change_var x y (app R S) (app R' S'))
| change_var_lam1: (M,M':tm)(change_var x y M M') ->
                   (change_var x y (lam x M) (lam y M'))
| change_var_lam2: (M,M':tm)(z:var)~x=z -> (change_var x y M M') ->
                   (change_var x y (lam z M) (lam z M')).
```

Intuitively, `(change_var x y M N)` holds if and only if the term `N` is the result of replacing each occurrence (free or bound) of `x` with `y` in `M`. Whence, if `M` $\triangleq \epsilon_X^\Lambda(M)$ and `N` $\triangleq \epsilon_X^\Lambda(N)$, then `(change_var x y M N)` represents the change of variable $N = M[x := y]$. More formally, the following result states the adequacy of `change_var` w.r.t. the predicate whose inference rules are depicted in Figure 6.3.

**Proposition 6.3 (Adequacy of `change_var`)** *1. (Soundness) Let $X \subset \mathcal{V}$ finite, $x, y \in \mathcal{V}$, if `t` is a canonical form such that $\Gamma_{X\cup\{x,y\}} \vdash_{\Sigma_\Lambda}$ `t` : (`change_var x y M N`), then we have $\delta_X^\Lambda$(`M`)$[x := y] = \delta_X^\Lambda$(`N`).*

*2. (Completeness) Let $X \subset \mathcal{V}$ finite, $x, y \in \mathcal{V}$, $M, N \in \Lambda_X$, then if $M[x := y] = N$ there is a canonical form `t` such that*

$$\Gamma_{X\cup\{x,y\}} \vdash_{\Sigma_\Lambda} \texttt{t} : (\texttt{change\_var x y } \epsilon_X^\Lambda(M)\ \epsilon_X^\Lambda(N)).$$

*Proof.* Also this result can be proved trivially by the same technique specified in Proposition 6.2. □

Obviously, changing a variable "blindly" (i.e., without checking if the new one is fresh) by means of `change_var` could yield a problem of *capturing* occurrences of variables which were free before performing the replacement. An example will make things clear: consider the term `(lam x (app x y))`$= \epsilon_{\{x,y\}}^\Lambda(\lambda x.xy)$, then we can derive `(change_var y x (lam x (app x y)) (lam x (app x x)))` which corresponds to the following equation "on the paper":

$$(\lambda x.xy)[y := x] = \lambda x.xx$$

It is clear that the free occurrence of $y$ in $\lambda x.xy$ has been *captured* once replaced by $x$. As a consequence the $\lambda$-terms $\lambda x.xy$ and $\lambda x.xx$ are not $\alpha$-equivalent. Hence, in order to avoid such a danger, we must be sure that, when we are going to replace a variable in a $\lambda$-term, the new one is fresh. This is where our auxiliary predicate `notin` comes into play, i.e., we are now ready to introduce the inductive predicate encoding the notion of $\alpha$-equivalence:

```
Inductive alphaBar: tm -> tm -> Prop:=
    alphaBar_var   : (x:var)(alphaBar (is_var x) (is_var x))
  | alphaBar_app   : (M,M',N,N':tm)(alphaBar M M') -> (alphaBar N N') ->
                     (alphaBar (app M N) (app M' N'))
  | alphaBar_lam1 : (x:var)(M,N:tm)(alphaBar M N) ->
                     (alphaBar (lam x M) (lam x N))
  | alphaBar_lam2 : (x,y:var)(M,N:tm)(notin y M) ->
                     (change_var x y M N) -> (alphaBar (lam x M) (lam y N))
  | alphaBar_trans: (M,N,R:tm)
                     (alphaBar M R) -> (alphaBar R N) -> (alphaBar M N).
```

The first three constructors are congruence rules, the fourth one is the *change of bound variables* rule, while the last one is transitivity (recall that two terms can differ by a series of changes of bound variables).

**Proposition 6.4 (Adequacy of `alphaBar`)** *1. (Soundness) Let $X \subset \mathcal{V}$ finite, if* `t` *is a canonical form such that $\Gamma_X \vdash_{\Sigma_\Lambda}$* `t` : (`alphaBar M N`)*, then we have $\delta_X$*(`M`) $\equiv_\alpha$ *$\delta_X$*(`N`)*.*

*2. (Completeness) Let $X \subset \mathcal{V}$ finite, $M, N \in \Lambda_X$, then if $M \equiv_\alpha N$ there is a canonical form* `t` *such that $\Gamma_X \vdash_{\Sigma_\Lambda}$* `t` : (`alphaBar` $\epsilon_X(M)$ $\epsilon_X(N)$)*.*

*Proof.* Again, this result easily follows applying the same technique used in the previous adequacy propositions. □

Thanks to the previous adequacy theorem, `alphaBar` can be regarded as a specification which the subsequent definitions we are going to investigate must fulfill in order to faithfully represent the notion of $\alpha$-equivalence.

### 6.1.4 Encoding of $\alpha$-equivalence (II)

The presence of rule `alphaBar_trans` in our first encoding of $\alpha$-equivalence is rather problematic from the point of view of a computer assisted formal development. Indeed, if we need to invert an hypothesis of type `(alphaBar M N)` during a proof in order to acquire some information on the structure of the arguments `M` and `N`, among the other subgoals we obtain a case where our initial hypothesis has been replaced by two new hypotheses `(alphaBar M R)` and `(alphaBar R N)` for a generic `R`. Hence, no useful information is provided and we are stuck, since inverting again would yield another subcase of this kind and so on. In order to overcome this kind of difficulty, we need an alternative axiomatization of $\alpha$-equivalence where all the rules enjoy the *subformula property.* This is the approach taken in [MP99]; *mutatis mutandis* their alternative formulation gives rise to the following inductive predicate:

```
Inductive alphaMKP: tm -> tm -> Prop:=
    alphaMKP_var: (x:var)(alphaMKP (is_var x) (is_var x))
  | alphaMKP_app: (M,M',N,N':tm)(alphaMKP M M') -> (alphaMKP N N') ->
                  (alphaMKP (app M N) (app M' N'))
```

```
| alphaMKP_lam: (x,y,z:var)(M,M',N,N':tm)
                (notin z M) -> (notin z N) ->
                (change_var x z M M') -> (change_var y z N N') ->
                (alphaMKP M' N') -> (alphaMKP (lam x M) (lam y N)).
```

The first two constructors (alphaMKP_var and alphaMKP_app) do not deserve any comments since they represent congruence rules, while the alphaMKP_lam constructor is not completely trivial. It states that, in order to conclude that the terms $\lambda x.M$ and $\lambda y.N$ are equivalent, we must pick a fresh name $z$ (not occurring at all in $M$ and $N$) and prove that $M[x := z]$ is $\alpha$-equivalent to $N[y := z]$. That is, two terms are $\alpha$-equivalent if and only if they differ by a series of changes of bound variables (see Definition 6.1).

Before formally proving the equivalence of alphaBAR and alphaMKP we need to show that alphaMKP is preserved by the lam constructor and that it is transitive. Hence, we delay the equivalence result to Section 6.1.6, when all the necessary lemmata will have been derived.

### 6.1.5 Formal metatheory of $\alpha$-equivalence

For the reason illustrated in the previous section (impossibility of using inversion on hypotheses of type (alphaBar A B)), we stick to our second encoding in order to carry out a formal development of the metatheory of $\alpha$-equivalence. However, all the results can be easily "ported" to the first encoding thanks to the equivalence result proved in Section 6.1.6.

Naturally, the first thing one would like to prove in Coq about an encoding of the $\alpha$-equivalence is that it is indeed an equivalence, i.e., a reflexive, symmetric and transitive relation. However, the initial enthusiasm ends shortly at the first attempt to prove the reflexivity property:

```
Lemma ALPHAMKP_REFL: (A:tm)(alphaMKP A A).
```

Indeed, the "natural" approach is to proceed by structural induction on A, but when we must solve the last subgoal (related to the lam constructor), we have a non trivial problem to deal with. More precisely, the proof context is the following:

```
A : tm
v : var
t : tm
H : (alphaMKP t t)
============================
 (alphaMKP (lam v t) (lam v t))
```

Hence, in order to proceed, applying the appropriate introduction rule (alphaMKP_lam), we must supply a fresh name yielded by the unsaturation axiom. Thus, after some easy steps we get the following:

```
A : tm
v : var
t : tm
H : (alphaMKP t t)
x : var
H2 : (notin x t)
H0 : ~x=v
x0 : tm
```

```
H1 : (change_var v x t x0)
============================
 (alphaMKP (lam v t) (lam v t))
```

Now, we are ready to apply the rule alphaMKP_lam with arguments x (the fresh name which will replace v in t) and x0 (the result of replacing x for v in t). The result of this step is:

```
A : tm
v : var
t : tm
H : (alphaMKP t t)
x : var
H2 : (notin x t)
H0 : ~x=v
x0 : tm
H1 : (change_var v x t x0)
============================
 (alphaMKP x0 x0)
```

At his point we are stuck because we know that (alphaMKP t t) holds but we have to prove that (alphaMKP x0 x0) holds. Translating the problem in terms of informal proofs with "pencil and paper", this means that we must prove $A[v := x] \equiv_\alpha A[v := x]$ knowing that $A \equiv_\alpha A$, $x \neq v$ and $x \notin^{\Lambda}_{\mathcal{V}} A$. Later on, proving the transitivity of alphaMKP and the internal equivalence between alphaMKP and the second encoding alphaGP of the $\alpha$-equivalence relation, we will need a more general result stating that we can derive $A[x := y] \equiv_\alpha B[x := y]$ knowing that $A \equiv_\alpha B$, $x \neq y$ and $y \notin^{\Lambda}_{\mathcal{V}} A, B$. This is exactly the renaming result mentioned in Section 6.1; once formulated in Coq, it appears as follows:

```
Lemma ALPHAMKP_RW: (A,B:tm)(alphaMKP A B) ->
                   (x,z:var)(notin z A) -> (notin z B) ->
                   (A':tm)(change_var x z A A') ->
                   (B':tm)(change_var x z B B') -> (alphaMKP A' B').
```

In order to prove ALPHAMKP_RW, a naïve approach would be to use the induction principle for the predicate alphaMKP, automatically generated by the Coq proof assistant. However, this attempt fails when we face the case where the last rule used in the derivation of the judgment (alphaMKP A B) is alphaMKP_lam. Indeed, in order to prove this step, we need to apply the inductive hypothesis to any proof *smaller* than the induction variable. In other words, we need a *complete induction principle*; in order to derive it, we define the following predicate:

```
Inductive l: tm -> nat -> Prop:=
    l_var : (x:var)(l (is_var x) O)
  | l_app : (M,N:tm)(n1,n2:nat)
            (l M n1) -> (l N n2) -> (l (app M N) (S (plus n1 n2)))
  | l_lam : (x:var)(M:tm)(n:nat)(l M n) -> (l (lam x M) (S n)).
```

Intuitively, (l M n) holds if and only if the term M of type tm contains n occurrences of the app and lam constructors. It follows that l can be taken as a *measure* of a term of type tm; moreover, if N is a proper subterm of M and (l N n1) and (l M n2) hold, we have that (lt `n1 n2)` holds[2]. Hence, we can "mimick" a complete induction principle on the structure of terms of type `tm` proceeding by complete induction on natural numbers. Again, the latter principle is not automatically provided by `Coq`, but it can be easily derived:

```
Lemma NAT_IND: (P:nat->Prop)
               (P O) -> ((n:nat)((m:nat)(lt m n) -> (P m)) -> (P n)) ->
               (n:nat)(P n).
```

Now, we can prove the following result applying `NAT_IND`:

```
Lemma PRE_ALPHAMKP_RW: (n:nat)(A:tm)(l A n) -> (B:tm)(alphaMKP A B) ->
                       (x,z:var)~(x=z) -> (notin z A) -> (notin z B) ->
                       (A':tm)(change_var x z A A') ->
                       (B':tm)(change_var x z B B') ->
                       (alphaMKP A' B').
```

During the proof development of the previous lemma there is a wide use of the fact that the measure of a term of type `tm` is preserved by changing variables in it:

```
Lemma L_CHANGE_VAR: (A:tm)(n:nat)(l A n) ->
                    (x,y:var)(B:tm)(change_var x y A B) -> (l B n).
```

Hence, we can obtain `ALPHAMKP_RW` from `PRE_ALPHAMKP_RW` and the following lemma, stating that every term of type `tm` has a measure:

```
Lemma L_TOT: (A:tm)(Ex [n:nat](l A n)).
```

Continuing with our attempt to formalize the metatheory of the $\alpha$-equivalence relation, we have to prove that `alphaMKP` is a symmetric relation:

```
Lemma ALPHAMKP_SYM: (A,B:tm)(alphaMKP A B) -> (alphaMKP B A).
```

The proof is a straightforward structural induction on the judgment `(alphaMKP A B)`, since the introduction rules of `alphaMKP` are clearly symmetric.

As to the transitivity of `alphaMKP`, this represents another big challenge. Indeed, an attempt to prove it by induction on the derivation of the judgment of type `(alphaMKP A B)` using the principle provided by `Coq` fails when we must face the case relative to the rule `alphaMKP_lam`. The reason is that the induction hypothesis is too weak; actually, the proof context is the following:

```
A : tm
B : tm
H : (alphaMKP A B)
x : var
y : var
z : var
M : tm
M' : tm
N : tm
N' : tm
```

[2] In `Coq` `lt` (standing for *less than*) is a binary predicate on natural numbers which holds if and only if the first argument is strictly smaller than the second one.

```
H0 : (notin z M)
H1 : (notin z N)
H2 : (change_var x z M M')
H3 : (change_var y z N N')
H4 : (alphaMKP M' N')
H5 : (C:tm)(alphaMKP N' C)->(alphaMKP M' C)
C : tm
H6 : (alphaMKP (lam y N) C)
============================
 (alphaMKP (lam x M) C)
```

Whence, inverting hypothesis `H6`, we obtain:

```
A : tm
B : tm
H : (alphaMKP A B)
x : var
y : var
z : var
M : tm
M' : tm
N : tm
N' : tm
H0 : (notin z M)
H1 : (notin z N)
H2 : (change_var x z M M')
H3 : (change_var y z N N')
H4 : (alphaMKP M' N')
H5 : (C:tm)(alphaMKP N' C)->(alphaMKP M' C)
C : tm
y0 : var
z0 : var
M'0 : tm
N0 : tm
N'0 : tm
H7 : (notin z0 N)
H8 : (notin z0 N0)
H9 : (change_var y z0 N M'0)
H10 : (change_var y0 z0 N0 N'0)
H11 : (alphaMKP M'0 N'0)
============================
 (alphaMKP (lam x M) (lam y0 N0))
```

The next step would be to use the unsaturation axiom in order to obtain a name, say `u`, fresh in `M`, `N` and `N0` (since neither `z` nor `z0` enjoys this property). Then, applying rule `alphaMKP_lam` we would have to prove that the term obtained changing the variable `x` with `u` in `M` is $\alpha$-equivalent to the term obtained changing the variable `y0` with `u` in `N0`. However, the only way available to accomplish this is using the induction hypothesis `H5` which is clearly too weak. Actually, it only holds for renamings of `M` and `N` where `x` and `y` are replaced by `z`. In order to obtain the right induction hypothesis, we follow an approach pioneered in [MP99], i.e., we define another equivalence relation alphaMKP' with the appropriate rule for the lam constructor. Then, we prove its transitivity and, finally, we show that it is formally equivalent to alphaMKP. As anticipated, the definition of alphaMKP' differs from that of alphaMKP only for the rule of the lam constructor:

```
alphaMKP'_lam: (x,y:var)(M,N:tm)
               ((z:var)(M',N':tm)(notin z M) -> (notin z N) ->
                (change_var x z M M') -> (change_var y z N N') ->
                (alphaMKP' M' N')
               ) -> (alphaMKP' (lam x M) (lam y N)).
```

Obviously, the transitivity of alphaMKP' requires a renaming result similar to ALPHAMKP_RW and it is proved with the same technique.

As a pragmatic remark, we notice that in the abovementioned proofs involving the predicate l, in order to prove the subgoals of the form (lt n1 n2), we used the Coq library Omega which completely automatizes such cases, freeing the user from proving tedious auxiliary lemmata[3].

For the sake of completeness, we list here a set of properties about change_var which have been useful in proving both the abovementioned results and the rest of our formal development.

- If $B = A[x := y]$ and $C = A[x := y]$, then $B = C$:

```
Lemma CHANGE_VAR_DET: (A,B:tm)(x,y:var)(change_var x y A B) ->
                      (C:tm)(change_var x y A C) -> B=C.
```

- For every $A \in \Lambda$, $x$, $y \in \mathcal{V}$, there exists $B$ such that $B = A[x := y]$:

```
Lemma CHANGE_VAR_TOT: (A:tm)(x,y:var)(Ex [B:tm](change_var x y A B)).
```

- If $B = A[x := z]$, $y \neq z$ and $y \notin^{\Lambda}_{\mathcal{V}} A$, then $y \notin^{\Lambda}_{\mathcal{V}} B$:

```
Lemma CHANGE_VAR_NOTIN: (x,z:var)(A,B:tm)(change_var x z A B) ->
                        (y:var)~(y=z) -> (notin y A) -> (notin y B).
```

- If $B = A[x := y]$ and $x \neq y$, then $x \notin^{\Lambda}_{\mathcal{V}} B$:

```
Lemma CHANGE_VAR_NOTIN2: (x,y:var)(A,B:tm)
                         (change_var x y A B) -> ~x=y -> (notin x B).
```

- If $B = A[x := y]$, $y \notin^{\Lambda}_{\mathcal{V}} A$, and $C = B[y := z]$, then $C = A[x := z]$:

```
Lemma CHANGE_VAR_COMP: (x,y:var)(A,B:tm)
                       (change_var x y A B) -> (notin y A) ->
                       (z:var)(C:tm)(change_var y z B C) ->
                       (change_var x z A C).
```

[3]Someone could find that there are some eschatological insights into the computer assisted proof development activity since, in order to formalize the metatheory of the $\alpha$-equivalence, one resorts to use a library named $\Omega$.

- If $B = A[x := z]$, $z \notin^{\Lambda}_{\mathcal{V}} A$, and $C = A[x := y]$, then $B = C[y := z]$:

```
Lemma CHANGE_VAR_COMPINV: (x,z:var)(A,B:tm)
                          (change_var x z A B) -> (notin z A) ->
                          (y:var)(C:tm)(change_var x y A C) ->
                          (change_var z y B C).
```

- If $y \notin^{\Lambda}_{\mathcal{V}} A$ and $B = A[x := y]$, then $B = A[y := x]$:

```
Lemma CHANGE_VAR_INV: (x,y:var)(A,B:tm)(notin y A) ->
                      (change_var x y A B) -> (change_var y x B A).
```

- If $B = A[x := y]$, $x \neq z$, $y \neq z$, $x \neq v$, $C = B[z := v]$ and $D = A[z := v]$, then $C = D[x := y]$:

```
Lemma CHANGE_VAR_COMM: (x,y:var)(A,B:tm)(change_var x y A B) ->
                       (z,v:var)(C:tm)~(x=z) -> ~(y=z) -> ~(x=v) ->
                       (change_var z v B C) ->
                       (D:tm)(change_var z v A D) ->
                       (change_var x y D C).
```

- If $N = M[x := x]$, then $M = N$:

```
Lemma CHANGE_VAR_ID: (M,N:tm)(x:var)(change_var x x M N) -> M=N.
```

- If $x \notin^{\Lambda}_{\mathcal{V}} A$ and $B := A[x := y]$, then $A = B$:

```
Lemma CHANGE_VAR_ID2: (x,y:var)(A,B:tm)(notin x A) ->
                      (change_var x y A B) -> A=B.
```

- If $B = A[x := y]$, $y \notin^{\Lambda}_{\mathcal{V}} A$ and $y \notin^{\Lambda}_{\mathcal{V}} B$, then $A = B$:

```
Lemma CHANGE_VAR_ID3: (A,B:tm)(x,y:var)(change_var x y A B) ->
                      (notin y A) -> (notin y B) -> A=B.
```

### 6.1.6 Formal equivalence of `alphaBar` and `alphaMKP`

As anticipated in Section 6.1.4, we formally proved in `Coq` the equivalence of `alphaBar` and `alphaMKP`:

```
Lemma ALPHABAR_ALPHAMKP: (A,B:tm)(alphaBar A B) -> (alphaMKP A B).

Lemma ALPHAMKP_ALPHABAR: (A,B:tm)(alphaMKP A B) -> (alphaBar A B).
```

Both proofs proceed by structural induction on the derivation of the respective premises. In the former case we needed the closure under the `lam` constructor of `alphaMKP`, its reflexivity and transitivity. In the latter case instead, we used the reflexivity of `alphaBar` (which is trivial to prove). For the sake of completeness we give here the `Coq` statements of the abovementioned auxiliary lemma stating that `alphaMKP` is preserved by the `lam` constructor:

```
Lemma ALPHAMKP_LAM: (M,N:tm)(alphaMKP M N)->
                    (x:var)(alphaMKP (lam x M) (lam x N)).
```

$$\frac{x \in \mathcal{V}}{x \sim x} \qquad \text{(GP-}\alpha\text{-VAR)}$$

$$\frac{M_1 \sim M_1' \qquad M_2 \sim M_2'}{M_1 M_2 \sim M_1' M_2'} \qquad \text{(GP-}\alpha\text{-APP)}$$

$$\frac{(z\ x) \cdot M \sim (z\ y) \cdot M'}{\lambda x.M \sim \lambda y.M'} (z \text{ does not occur in } M,\ M') \quad \text{(GP-}\alpha\text{-LAM)}$$

Figure 6.4: Gabbay-Pitts alternative definition of $\alpha$-equivalence.

### 6.1.7 Encoding the $\alpha$-equivalence relation (III)

In [GP99], an alternative formulation of the $\alpha$-equivalence relation, relying on variable-transpositions, has been proposed and proved to be equivalent to the "conventional" definition on paper. In this section we will show how the HOAS-encoding approach can be fruitfully exploited in order to encode the abovementioned alternative definition; moreover, by means of the Theory of Contexts, we will formally prove the equivalence of this encoding and the one illustrated in Section 6.1.4.

In the following we will use the notation introduced in [GP99] to denote the operation of variable-transposition. Hence, $(y\ x) \cdot M$ stands for the *transposition* of all the occurrences (both free and bound) of $x$ and $y$ in the term $M$ (in other words all the occurrences of $x$ are replaced by occurrences of $y$ and vice versa).

As proved in [GP99], despite the fact that this operation is more basic than other more common notions of renaming (namely, textual and capture-avoiding substitution), it can be used to give an alternative definition of $\alpha$-equivalence. Indeed, the latter coincides with the binary relation $\sim$ inductively defined by the axioms and rules in Figure 6.4.

The operation of variable transposition can be expressed by means of HOAS in a very natural way; indeed, if $x$ and $y$ are are two distinct variables occurring in the term $M$ and $\epsilon_X^\Lambda(M) =$ `M`, then we can derive[4] that there is a context `M':var->var->tm` such that `x` and `y` do not occur in `M'` and `M=(M' x y)` holds. Then the operation $(y\ x) \cdot M$ can be simply denoted by `(M' y x)`. Moreover, if $y$ does not occur in $M$, we have that `M=(M'' x)` where both `x` and `y` do not occur in `M''`. Whence the operation $(y\ x) \cdot M$ can be denoted by `(M'' y)`, without resorting to binary contexts.

Thus, the encoding of the Gabbay-Pitts formulation of $\alpha$-equivalence is given by the following inductive predicate:

```
Inductive alphaGP: tm -> tm -> Prop:=
    alphaGP_var: (x:var)(alphaGP (is_var x) (is_var x))
  | alphaGP_app: (M,M',N,N':tm)(alphaGP M M') -> (alphaGP N N') ->
                 (alphaGP (app M N) (app M' N'))
  | alphaGP_lam: (M,N:var->tm)(x,x',y:var)
                 (notin_context x M) -> (notin_context x' N) ->
                 (notin_context y M) -> (notin_context y N) ->
                 (alphaGP (M y) (N y)) ->
                 (alphaGP (lam x (M x)) (lam x' (N x'))).
```

[4] The derivation makes use of lemma `EXP` (presented in Section 6.1.2) and of the axioms of $\beta$-expansion for unary contexts and monotonicity.

As we can see, the only differences w.r.t. the definition of `alphaMKP` are obviously in the rule involving the `lam` constructor.

### 6.1.8 Formal equivalence of `alphaMKP` and `alphaGP`

So far, if we exclude the derivation of the higher-order induction principle `HO_TM_IND` and of the lemma `PRE_NOTIN_MONO`, among the axioms of the Theory of Contexts, only the unsaturation property (`unsat_list`) and the decidability of the equality between names (`dec`) have been necessary during the proof development. This is not a surprising result since no higher-order constructors are present in our signature. However, the situation dramatically changes when we face the problem of formally proving the equivalence between `alphaMKP` and `alphaGP`, since the latter features an introduction rule where variable-transposition is modeled by means of functional application.

The equivalence is given by the following results:

```
Lemma ALPHAMKP_ALPHAGP: (A,B:tm)(alphaMKP A B) -> (alphaGP A B).

Lemma ALPHAGP_ALPHAMKP: (A,B:tm)(alphaGP A B) -> (alphaMKP A B).
```

Both proofs are carried out by induction on the derivation of the premise (`(alphaMKP A B)` for the former and `(alphaGP A B)` for the latter). The only interesting cases are related to the introduction rules for the `lam` constructor: in the proof of `ALPHAMKP` the key property to prove in order to conclude is the following (an easy induction on `M` with the aid of the expansion and extensionality axioms):

```
Lemma CHANGE_VAR_RW: (M,N:tm)(x,y:var)
                     (change_var x y M N) ->
                     (M':var->tm)(notin_context x M') ->
                     M=(M' x)->N=(M' y).
```

In the proof of `ALPHAGP`, also the renaming result `ALPHAMKP_RW` is necessary since the judgments `(notin_context y M)` and `(notin_context y N)` do not necessarily imply `(notin y (M x))` and `(notin y (N x'))` (while the vice versa is true by the monotonicity axiom). Hence, we cannot use the name `y` provided by the induction hypothesis, but we must pick a completely fresh variable in order to conclude.

Lemma `CHANGE_VAR_RW` can be regarded as the formal link between first-order implementations of substitution (in this case a change of variables) and the higher-order paradigm which delegates this fundamental mechanism to the underlying metalanguage.

From the main result of this section and the previously proved equivalence of `alphaBar` and `alphaMKP` we can conclude that all the alternative formulations we investigated are indeed formally equivalent. In our opinion, the moral of this case study is that an HOAS-based approach towards substitution of variables for variables turns out to be useful even when there are no binders in the object language (or for some reason they cannot be encoded by means of the binder of the metalanguage). Indeed, `alphaGP`, as an inductive definition, appears to be more clean and elegant than both `alphaBar` and `alphaMKP`; moreover, the former does not depend on an auxiliary relation (`alphaBar` and `alphaMKP`, instead, depend on `change_var`).

We conclude this section noticing that in [GP01] (an extended and revised version of [GP99]) the introduction rule of $\sim$ regarding the binder $\lambda$ is enriched by a new side condition, whence rule (GP-$\alpha$-LAM) in Figure 6.4 should appear as follows:

$$\frac{(z\ x) \cdot M \sim (z\ y) \cdot M'}{\lambda x.M \sim \lambda y.M'} (z \text{ does not occur in } M, M',\ z \neq x, y)$$

However, in our formal development we did not need to introduce such a new condition in order to carry out the proofs. Indeed, if we assume that $z$ does not occur in $M$,$M'$, then, in the case that $x$ (resp. $y$) occurs in $M$ (resp. $M'$), it is automatically verified that $z$ is distinct from $x$, $y$. Otherwise (i.e. if $x$, resp. $y$, does not occur in $M$, resp. $M'$), the side condition becomes superfluous.

## 6.2 Ambients

In this section we will focus on the encoding of the *ambient calculus* [CG98], a process algebra extending the $\pi$-calculus with primitives well-suited for expressing and handling *ambients*, i.e., *bounded places* where a computation can happen. Since ambients can be nested and moved as a whole, they seem perfectly suited for representing administrative domains and to model *mobility*. The access to a given ambient is controlled by its name, which can be passed from agent to agent in order to provide access capabilities.

### 6.2.1 Syntax

The basic syntactic categories of the Ambient Calculus are names, capabilities and processes (or agents): capabilities are defined by the following grammar:

$$\begin{array}{lcll} M & ::= & \text{Capabilities} & \\ & & n & \text{name} \\ & & in\ M & \text{can enter into } M \\ & & out\ M & \text{can exit out of } M \\ & & open\ M & \text{can open } M \\ & & \varepsilon & \text{null} \\ & & M.M' & \text{path} \end{array}$$

Processes features two new constructors (w.r.t. the original $\pi$-calculus), namely, the ambient operator ($M[P]$) and the capability action ($M.P$):

$$\begin{array}{lcll} P, Q, R & ::= & \text{Processes} & \\ & & (\nu n)P & \text{restriction} \\ & & \mathbf{0} & \text{void} \\ & & P|Q & \text{composition} \\ & & !P & \text{replication} \\ & & M[P] & \text{ambient} \\ & & M.P & \text{capability action} \\ & & (n).P & \text{input action} \\ & & \langle M \rangle & \text{output action} \end{array}$$

There are no binders among the capability constructors, while processes provide the usual binders, i.e., restriction and input action. Hence, the set of free names of a capability $M$ ($fn(M)$) and of a process $P$ ($fn(P)$) are defined as usual (see Figure 6.5). In the following, using the notation introduced in [CG01], we will denote by $P\{n \leftarrow M\}$, the capture avoiding

$$
\begin{array}{lll}
fn(n) & \triangleq & \{n\} \\
fn(in\ M) & \triangleq & fn(M) \\
fn(out\ M) & \triangleq & fn(M) \\
fn(open\ M) & \triangleq & fn(M) \\
fn(\varepsilon) & \triangleq & \emptyset \\
fn(M.M') & \triangleq & fn(M) \cup fn(M')
\end{array}
\qquad
\begin{array}{lll}
fn((\nu n)P & \triangleq & fn(P) \setminus \{n\} \\
fn(\mathbf{0}) & \triangleq & \emptyset \\
fn(P|Q) & \triangleq & fn(P) \cup fn(Q) \\
fn(!P) & \triangleq & fn(P) \\
fn(M[P]) & \triangleq & fn(M) \cup fn(P) \\
fn(M.P) & \triangleq & fn(M) \cup fn(P) \\
fn((n).P) & \triangleq & fn(P) \setminus \{n\} \\
fn(\langle M \rangle) & \triangleq & fn(M)
\end{array}
$$

Figure 6.5: Free names.

substitution of the capability $M$ for the free occurrences of $n$ in $P$. It is assumed that processes are identified up to $\alpha$-conversion, that is, the identities $(\nu n)P = (\nu m)P\{n \leftarrow m\}$ and $(n).P = (m).P\{n \leftarrow m\}$ hold (where $m \notin fn(P)$ or $m = n$).

An ambient is denoted by $n[P]$ ($n$ is the name of the ambient) where P may contain other ambients as well (i.e. ambients can be nested to form a hierarchical structure). Operations changing such a structure are potentially dangerous (e.g. they can mean crossing a firewall, entering into a restricted domain where sensitive data are stored); whence, they are regulated by means of capabilities. For instance, the capability *in* $n$ allows to enter into the ambient named $n$, while *out* $n$ allows to exit out of $n$ and *open* $n$ allows to open $n$. It should be noted that, given $n$, possessing a capability on it does not allow one to retrieve the name $n$ itself.

Following the notation adopted in [CG01] we will denote by $\Lambda$ the sort of names, and by $\Pi$ the syntactic category of processes. Moreover, capabilities will be indicated by $\zeta$.

### 6.2.2 Structural Congruence

The Ambient Calculus comes equipped with a relation of Structural Congruence ($\equiv$); the latter makes the reduction system (see Section 6.2.3) simpler by identifying expressions differing only by elementary (syntactic) rearrangements without any computational meaning. The rules defining such a relation are listed in Figure 6.6. As noted in [Dal00], almost all axioms and rules in Figure 6.6 have an equivalent among those defining the structural congruence for the $\pi$-calculus. The most notable differences (if we obviously exclude the cases involving the new constructors) are in two axioms involving replication, namely, (Struct Repl Par) and (Struct Repl Repl).

### 6.2.3 Reduction System

The operational semantics of processes is given by means of a reduction system, an idea going back to the *Chemical Abstract Machine* by Berry and Boudol [BB92] and adopted later in several presentations of the $\pi$-calculus (e.g. [Mil93]) as an alternative to the labelled transition system semantics.

While Structural Congruence can be seen as a relation describing the evolution of processes in space, the Reduction System allows to represent the evolution in time, i.e, it allows to formalize the dynamic behaviour of processes. The reduction rules for the Ambient Calculus are depicted in Figure 6.7. The first three axioms in Figure 6.7 deal with the three fundamental capabilities (*in*, *out* and *open*) allowing to express mobility of ambients, while the fourth one represents local communication (that is the possibility of transmitting a capability to other agents). Among the structural rules, (Red Amb), establishing that any

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

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

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

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

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

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

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

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

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

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

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

Figure 6.6: Structural Congruence.

reduction of $P$ becomes a reduction of $n[P]$, emphasizes the fact that $P$ is running (active) in the surrounding ambient even if the latter is moving. Finally, rule (Red $\equiv$) allows to take care of spatial rearrangements of processes during reductions.

### 6.2.4 The Logic

In [CG00], Cardelli and Gordon introduce a modal logic having the Ambient Calculus as a model in order to naturally express and reason about properties of mobile computations. The logic is then extended in [CG01] with new quantifiers and operators allowing to soundly reason about properties involving restricted names (that is names bound by the $\nu$ operator).

Logical formulæ are defined by the following grammar (where $\eta$ can be a name $n$ or a variable $x$):

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

Keeping in mind that there are no name binders and only one variable binder (universal quantification), the set of free names $fn(\mathcal{A})$ and free variables $fv(\mathcal{A})$ of a formula $\mathcal{A}$ are

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

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

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

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

$$\frac{P \to Q}{(\nu n)P \to (\nu n)Q} \qquad \text{(Red Res)}$$

$$\frac{P \to Q}{P|R \to Q|R} \qquad \text{(Red Par)}$$

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

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

Figure 6.7: Reduction System.

defined by the clauses in Figure 6.8. 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 $\mu$ for another name or variable $\eta$ (variables can range only over names). As in the case of processes, formulæ are identified up to $\alpha$-conversion, that is the identity $\forall x.\mathcal{A} = \forall y.\mathcal{A}\{x \leftarrow y\}$ is assumed (where $y \notin fv(\mathcal{A})$ or $y = x$). Following the notation adopted in [CG01], we will denote by $\vartheta$ the sort of variables and by $\Phi$ the syntactic category of formulæ.

Intuitively, the constructors $\mathbf{T}$, $\neg$ and $\vee$ provide propositional logic (negation is classical), universal quantification gives predicate logic, while the remaining ones are related to the process constructors.

More formally, the relation between processes and formulæ is established by the *satisfaction relation* ($\models$): $P \models \mathcal{A}$ means that the process $P$ satisfies the closed formula $\mathcal{A}$ as specified by the rules in Figure 6.9. In particular, the satisfaction for the temporal modality is defined by means of the reduction relation ($\to^*$ is the reflexive and transitive closure of $\to$). As to the spatial modality instead, its satisfaction is given by means of the relation $\downarrow$. Intuitively, $P \downarrow P'$ means that $P$ contains $P'$ within exactly one level of nesting: more formally 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 $\downarrow$.

### 6.2.5 Encoding of Syntax

In the following we will denote by $\Sigma_A$ the signature of our encoding of the Ambient Calculus. The first constant we introduce in $\Sigma_A$ is the one encoding the type of names:

```
Parameter name: Set.
```

Hence, names of the Ambient Calculus will be represented by variables of `Coq` of type `name`. Next, we encode the syntactic category of capabilities by means of the following inductive

$$
\begin{array}{lll}
fn(\mathbf{T}) & \triangleq & \emptyset \\
fn(\neg\mathcal{A}) & \triangleq & fn(\mathcal{A}) \\
fn(\mathcal{A} \vee \mathcal{B}) & \triangleq & fn(\mathcal{A}) \cup fn(\mathcal{B}) \\
fn(\mathbf{0}) & \triangleq & \emptyset \\
fn(\mathcal{A}|\mathcal{B}) & \triangleq & fn(\mathcal{A}) \cup fn(\mathcal{B}) \\
fn(\mathcal{A} \triangleright \mathcal{B}) & \triangleq & fn(\mathcal{A}) \cup fn(\mathcal{B}) \\
fn(\eta[\mathcal{A}]) & \triangleq & \begin{cases} \{n\} \cup fn(\mathcal{A}) & \text{if } \eta = n \\ fn(\mathcal{A}) & \text{if } \eta = x \end{cases} \\
fn(\mathcal{A}@\eta) & \triangleq & \begin{cases} fn(\mathcal{A}) \cup \{n\} & \text{if } \eta = n \\ fn(\mathcal{A}) & \text{if } \eta = x \end{cases} \\
fn(\eta®\mathcal{A}) & \triangleq & \begin{cases} \{n\} \cup fn(\mathcal{A}) & \text{if } \eta = n \\ fn(\mathcal{A}) & \text{if } \eta = x \end{cases} \\
fn(\mathcal{A} \oslash \eta) & \triangleq & \begin{cases} fn(\mathcal{A}) \cup \{n\} & \text{if } \eta = n \\ fn(\mathcal{A}) & \text{if } \eta = x \end{cases} \\
fn(\Diamond\mathcal{A}) & \triangleq & fn(\mathcal{A}) \\
fn(\diamondsuit\mathcal{A}) & \triangleq & fn(\mathcal{A}) \\
fn(\forall x.\mathcal{A}) & \triangleq & fn(\mathcal{A})
\end{array}
\qquad
\begin{array}{lll}
fv(\mathbf{T}) & \triangleq & \emptyset \\
fv(\neg\mathcal{A}) & \triangleq & fv(\mathcal{A}) \\
fv(\mathcal{A} \vee \mathcal{B}) & \triangleq & fv(\mathcal{A}) \cup fv(\mathcal{B}) \\
fv(\mathbf{0}) & \triangleq & \emptyset \\
fv(\mathcal{A}|\mathcal{B}) & \triangleq & fv(\mathcal{A}) \cup fv(\mathcal{B}) \\
fv(\mathcal{A} \triangleright \mathcal{B}) & \triangleq & fv(\mathcal{A}) \cup fv(\mathcal{B}) \\
fv(\eta[\mathcal{A}]) & \triangleq & \begin{cases} fv(\mathcal{A}) & \text{if } \eta = n \\ \{x\} \cup fv(\mathcal{A}) & \text{if } \eta = x \end{cases} \\
fv(\mathcal{A}@\eta) & \triangleq & \begin{cases} fv(\mathcal{A}) & \text{if } \eta = n \\ fv(\mathcal{A}) \cup \{x\} & \text{if } \eta = x \end{cases} \\
fv(\eta®\mathcal{A}) & \triangleq & \begin{cases} fv(\mathcal{A}) & \text{if } \eta = n \\ \{x\} \cup fv(\mathcal{A}) & \text{if } \eta = x \end{cases} \\
fv(\mathcal{A} \oslash \eta) & \triangleq & \begin{cases} fv(\mathcal{A}) & \text{if } \eta = n \\ fv(\mathcal{A}) \cup \{x\} & \text{if } \eta = x \end{cases} \\
fv(\Diamond\mathcal{A}) & \triangleq & fv(\mathcal{A}) \\
fv(\diamondsuit\mathcal{A}) & \triangleq & fv(\mathcal{A}) \\
fv(\forall x.\mathcal{A}) & \triangleq & fv(\mathcal{A}) \setminus \{x\}
\end{array}
$$

Figure 6.8: Free names and free variables of formulæ.

definition:

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

In the following, for $N \triangleq \{n_1, \ldots, n_k\} \subset \Lambda$ finite, we will denote by $\zeta_X$ the set $\{M \mid M \in \zeta, fn(M) \subseteq N\}$. Moreover, we will denote by $\Gamma_N$ the typing environment given by:

$$\Gamma_N \triangleq \{\mathtt{n1} : \mathtt{name}, \ldots, \mathtt{nk} : \mathtt{name}\} \cup \{\mathtt{dij} :\tilde{}\,(\mathtt{ni} = \mathtt{nj}) \mid \mathtt{1} \leq \mathtt{i} < \mathtt{j} \leq \mathtt{k}\}$$

Finally $\mathtt{cap}_N$ will represent the canonical forms M (i.e. $\beta\eta$-head normal forms) of type cap such that $\Gamma_N \vdash_{\Sigma_A} \mathtt{M} : \mathtt{cap}$. As we can see, there is a one-to-one correspondence between the productions of the grammar defining the syntax of the object language and the constructors of type cap. This fact is formalized by the encoding and decoding functions depicted in Figure 6.10 and by the following adequacy result:

**Proposition 6.5** *For each $N \subset \Lambda$ finite, $\epsilon^{\zeta}_N$ is a compositional bijection between $\zeta_N$ and* $\mathtt{cap}_N$.

*Proof.* Standard, using the definitions in Figure 6.10 and proceeding by induction on the structure of capabilities and of canonical forms of type cap. □

So far, there are no binders; hence, the encoding of capabilities is a plain first order one. However, the syntactic category of processes features two binders, namely, restriction

| | | | |
|---|---|---|---|
| for all $P \in \Pi$, | $P \models \mathbf{T}$ | | |
| for all $P \in \Pi, \mathcal{A} \in \Phi$ | $P \models \neg\mathcal{A}$ | $\triangleq$ | $P \not\models \mathcal{A}$ |
| for all $P \in \Pi, \mathcal{A}, \mathcal{B} \in \Phi$ | $P \models \mathcal{A} \vee \mathcal{B}$ | $\triangleq$ | $P \models \mathcal{A}$ or $P \models \mathcal{B}$ |
| for all $P \in \Pi$ | $P \models \mathbf{0}$ | $\triangleq$ | $P \equiv \mathbf{0}$ |
| for all $P \in \Pi, \mathcal{A}, \mathcal{B} \in \Phi$ | $P \models \mathcal{A}\|\mathcal{B}$ | $\triangleq$ | there exist $P', P'' \in \Pi$ such that $P \equiv P'\|P''$, $P' \models \mathcal{A}$ and $P'' \models \mathcal{B}$ |
| for all $P \in \Pi, \mathcal{A}, \mathcal{B} \in \Phi$ | $P \models \mathcal{A} \triangleright \mathcal{B}$ | $\triangleq$ | for all $P' \in \Pi, P' \models \mathcal{A}$ implies $P\|P' \models \mathcal{B}$ |
| for all $P \in \Pi, n \in \Lambda, \mathcal{A} \in \Phi$ | $P \models n[\mathcal{A}]$ | $\triangleq$ | there exists $P' \in \Pi$ such that $P \equiv n[P']$ and $P' \models \mathcal{A}$ |
| for all $P \in \Pi, \mathcal{A} \in \Phi, n \in \Lambda$ | $P \models \mathcal{A}@n$ | $\triangleq$ | $n[P] \models \mathcal{A}$ |
| for all $P \in \Pi, n \in \Lambda, \mathcal{A} \in \Phi$ | $P \models n®\mathcal{A}$ | $\triangleq$ | there exists $P' \in \Pi$ such that $P \equiv (\nu n)P'$ and $P' \models \mathcal{A}$ |
| for all $P \in \Pi, \mathcal{A} \in \Phi, n \in \Lambda$ | $P \models \mathcal{A} \oslash n$ | $\triangleq$ | $(\nu n)P \models \mathcal{A}$ |
| for all $P \in \Pi, \mathcal{A} \in \Phi$ | $P \models \Diamond\mathcal{A}$ | $\triangleq$ | there exists $P' \in \Pi$ such that $P \rightarrow^* P'$ and $P' \models \mathcal{A}$ |
| for all $P \in \Pi, \mathcal{A} \in \Phi$ | $P \models \diamond\!\!\!\cdot\,\mathcal{A}$ | $\triangleq$ | there exists $P' \in \Pi$ such that $P \downarrow^* P'$ and $P' \models \mathcal{A}$ |
| for all $P \in \Pi, x \in \vartheta, \mathcal{A} \in \Phi$ | $P \models \forall x.\mathcal{A}$ | $\triangleq$ | for all $m \in \Lambda, P \models \mathcal{A}\{x \leftarrow m\}$ |

Figure 6.9: Satisfaction.

and input action: in this case we want to take full advantage by adopting a HOAS-based 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 `nu` and `in_act` take as arguments functions of type `name->proc`, $\alpha$-conversion and capture-avoiding substitution of names for names are automatically delegated to the meta-language, freeing the user from thinking about renaming of bound variables and name clashes in general. As in the case of the capabilities encoding, for $N \subset \Lambda$ finite, we will denote by $\Pi_N$ the set $\{P \mid P \in \Pi, fn(P) \subseteq N\}$. Moreover, $\texttt{proc}_N$ will represent the canonical forms `P` (i.e. $\beta\eta$-head normal forms) of type `proc` such that $\Gamma_N \vdash_{\Sigma_A} \texttt{P} : \texttt{proc}$. Since the encoding follows the HOAS principles, in order to define the decoding function $\delta_N^\Pi$ (see Figure 6.11), we need a map *fresh* : $\mathcal{P}_{<\omega}(\Lambda) \longrightarrow \Lambda$. The latter can be viewed as a "fresh name selection" function, i.e., for every $N \subset \Lambda$ finite ($N \in \mathcal{P}_{<\omega}(\Lambda)$), $\mathit{fresh}(N) \notin N$. If $\Lambda$ is enumerable, a possible definition for *fresh* is $\mathit{fresh}(N) = n_{max\{i|n_i \in N\}+1}$, where $(n_i)_i$ is a given enumeration of $\Lambda$. The following result establishes the adequacy of our encoding:

$$
\begin{array}{rcl}
\epsilon^{\zeta}_{N} & : & \zeta_N \longrightarrow \texttt{cap}_N \\
\epsilon^{\zeta}_{N}(n) & \triangleq & (\texttt{name2cap n}) \\
\epsilon^{\zeta}_{N}(in\ M) & \triangleq & (\texttt{in\_cap}\ \epsilon^{\zeta}_{N}(M)) \\
\epsilon^{\zeta}_{N}(out\ M) & \triangleq & (\texttt{out\_cap}\ \epsilon^{\zeta}_{N}(M)) \\
\epsilon^{\zeta}_{N}(open\ M) & \triangleq & (\texttt{open}\ \epsilon^{\zeta}_{N}(M)) \\
\epsilon^{\zeta}_{N}(\varepsilon) & \triangleq & \texttt{eps} \\
\epsilon^{\zeta}_{N}(M.M') & \triangleq & (\texttt{path}\ \epsilon^{\zeta}_{N}(M)\ \epsilon^{\zeta}_{N}(M'))
\end{array}
\qquad
\begin{array}{rcl}
\delta^{\zeta}_{N} & : & \texttt{cap}_N \longrightarrow \zeta_N \\
\delta^{\zeta}_{N}((\texttt{name2cap n})) & \triangleq & n \\
\delta^{\zeta}_{N}((\texttt{in\_cap M})) & \triangleq & in\ \delta^{\zeta}_{N}(\texttt{M}) \\
\delta^{\zeta}_{N}((\texttt{out\_cap M})) & \triangleq & out\ \delta^{\zeta}_{N}(\texttt{M}) \\
\delta^{\zeta}_{N}((\texttt{open M})) & \triangleq & open\ \delta^{\zeta}_{N}(\texttt{M}) \\
\delta^{\zeta}_{N}(\texttt{eps}) & \triangleq & \varepsilon \\
\delta^{\zeta}_{N}((\texttt{path M M}')) & \triangleq & \delta^{\zeta}_{N}(\texttt{M}).\delta^{\zeta}_{N}(\texttt{M}')
\end{array}
$$

Figure 6.10: Encoding and decoding functions for the syntactic category of capabilities.

$$
\begin{array}{rcl}
\epsilon^{\Pi}_{N} & : & \Pi_N \longrightarrow \texttt{proc}_N \\
\epsilon^{\Pi}_{N}((\nu n)P) & \triangleq & (\texttt{nu}\ [\texttt{n}:\texttt{name}]\ \epsilon^{\Pi}_{N,n}(P)) \\
\epsilon^{\Pi}_{N}(\mathbf{0}) & \triangleq & \texttt{nil} \\
\epsilon^{\Pi}_{N}(P|Q) & \triangleq & (\texttt{par}\ \epsilon^{\Pi}_{N}(P)\ \epsilon^{\Pi}_{N}(Q)) \\
\epsilon^{\Pi}_{N}(!P) & \triangleq & (\texttt{bang}\ \epsilon^{\Pi}_{N}(P)) \\
\epsilon^{\Pi}_{N}(M[P]) & \triangleq & (\texttt{ambient}\ \epsilon^{\zeta}_{N}(M)\ \epsilon^{\Pi}_{N}(P)) \\
\epsilon^{\Pi}_{N}(M.P) & \triangleq & (\texttt{cap\_act}\ \epsilon^{\zeta}_{N}(M)\ \epsilon^{\Pi}_{N}(P)) \\
\epsilon^{\Pi}_{N}((n).P) & \triangleq & (\texttt{in\_act}\ [\texttt{n}:\texttt{name}]\ \epsilon^{\Pi}_{N,n}(P)) \\
\epsilon^{\Pi}_{N}(\langle M\rangle) & \triangleq & (\texttt{out\_act}\ \epsilon^{\zeta}_{N}(M))
\end{array}
\qquad
\begin{array}{rcl}
\delta^{\Pi}_{N} & : & \texttt{proc}_N \longrightarrow \Pi_N \\
\delta^{\Pi}_{N}((\texttt{nu P})) & \triangleq & (\nu m)\delta^{\Pi}_{N\cup\{z\}}((\texttt{P m})), \\
 & & m = fresh(N) \\
\delta^{\Pi}_{N}(\texttt{nil}) & \triangleq & \mathbf{0} \\
\delta^{\Pi}_{N}((\texttt{par P Q})) & \triangleq & \delta^{\Pi}_{N}(\texttt{P})|\delta^{\Pi}_{N}(\texttt{Q}) \\
\delta^{\Pi}_{N}((\texttt{bang P})) & \triangleq & !\delta^{\Pi}_{N}(\texttt{P}) \\
\delta^{\Pi}_{N}((\texttt{ambient M P})) & \triangleq & [\delta^{\zeta}_{N}(\texttt{M})]\delta^{\Pi}_{N}(\texttt{P}) \\
\delta^{\Pi}_{N}((\texttt{cap\_act M P})) & \triangleq & \delta^{\zeta}_{N}(\texttt{M}).\delta^{\Pi}_{N}(\texttt{P}) \\
\delta^{\Pi}_{N}((\texttt{in\_act P})) & \triangleq & (m).\delta^{\Pi}_{N\cup\{z\}}((\texttt{P m})), \\
 & & m = fresh(N) \\
\delta^{\Pi}_{N}((\texttt{out\_act M})) & \triangleq & \langle\delta^{\zeta}_{N}(\texttt{M})\rangle
\end{array}
$$

Figure 6.11: Encoding and decoding functions for the syntactic category of processes.

**Proposition 6.6** *For each $N \subset \Lambda$ finite, $\epsilon^{\Pi}_{N}$ is a compositional bijection between $\Pi_N$ and* $\texttt{proc}_N$.

*Proof.* Standard, using the definitions in Figure 6.11 and proceeding by induction on the structure of processes and of canonical forms of type `proc`. □

The notion of "freshness" is enforced within the encoding by non-occurrence predicates which follow the general pattern described in Definition 4.15. We prefer to introduce immediately those for capabilities and processes, since we will need them to encode Structural Congruence and the Reduction System in the next Sections.

- Non-occurrence predicate for capabilities:

```
Inductive notin_cap [m:name]: cap -> Prop :=
  notin_cap_name : (n:name)~m=n -> (notin_cap m (name2cap n))
| notin_cap_in   : (M:cap)(notin_cap m M) -> (notin_cap m (in_cap M))
| notin_cap_out  : (M:cap)(notin_cap m M) -> (notin_cap m (out_cap M))
| notin_cap_open : (M:cap)(notin_cap m M) -> (notin_cap m (open M))
| notin_cap_eps  : (notin_cap m eps)
| notin_cap_path : (M,N:cap)(notin_cap m M) -> (notin_cap m N) ->
                   (notin_cap m (path M N)).
```

- Non-occurrence predicate for processes:

```
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_nil  : (notin_proc m nil)
| notin_proc_par  : (P,Q:proc)
                    (notin_proc m P) -> (notin_proc m Q) ->
                    (notin_proc m (par P Q))
| notin_proc_bang : (P:proc)(notin_proc m P)->(notin_proc m (bang P))
| notin_proc_amb  : (M:cap)(P:proc)
                    (notin_cap m M) -> (notin_proc m P) ->
                    (notin_proc m (ambient M P))
| notin_proc_act  : (M:cap)(P:proc)
                    (notin_cap m M) -> (notin_proc m P) ->
                    (notin_proc m (cap_act M P))
| notin_proc_in   : (P:name->proc)
                    ((n:name)~m=n -> (notin_proc m (P n))) ->
                    (notin_proc m (in_act P))
| notin_proc_out  : (M:cap)(notin_cap m M)->(notin_proc m (out_act M)).
```

As usual, `(notin_cap n M)` intuitively means that the name `n` does not occur in `M`. Analogously, `(notin_proc n P)` holds if and only if the name `n` does not occur in `P`. Since, we adopted a HOAS-based encoding, bound names are automatically kept different from free ones by the metalanguage $\alpha$-conversion mechanism. Hence, we can say that `notin_proc` allows to capture either non-occurrence or non fresh occurrence judgments.

Beside freshness predicates it is sometimes useful to speak about names freely occurring in a capability or a process. Hence, we introduce the following inductive definitions which will be used in Section 6.2.10 and in the formal development illustrated in Section 6.2.11:

- Free occurrence predicate for capabilities:

```
Inductive isin_cap [m:name]: cap -> Prop :=
  isin_cap_name : (isin_cap m (name2cap m))
| isin_cap_in   : (M:cap)(isin_cap m M) -> (isin_cap m (in_cap M))
| isin_cap_out  : (M:cap)(isin_cap m M) -> (isin_cap m (out_cap M))
| isin_cap_open : (M:cap)(isin_cap m M) -> (isin_cap m (open M))
| isin_cap_path : (M,N:cap)(isin_cap m M) \/ (isin_cap m N) ->
                  (isin_cap m (path M N)).
```

- Free occurrence predicate for processes:

```
Inductive isin_proc [m:name]: proc -> Prop :=
  isin_proc_nu   : (P:name->proc)((n:name)(isin_proc m (P n))) ->
                   (isin_proc m (nu P))
| isin_proc_par  : (P,Q:proc)(isin_proc m P) \/ (isin_proc m Q) ->
                   (isin_proc m (par P Q))
| isin_proc_bang : (P:proc)(isin_proc m P) -> (isin_proc m (bang P))
| isin_proc_amb  : (M:cap)(P:proc)
                   (isin_cap m M) \/ (isin_proc m P) ->
                   (isin_proc m (ambient M P))
```

```
   | isin_proc_act  : (M:cap)(P:proc)
                      (isin_cap m M) \/ (isin_proc m P) ->
                      (isin_proc m (cap_act M P))
   | isin_proc_in   : (P:name->proc)((n:name)(isin_proc m (P n))) ->
                      (isin_proc m (in_act P))
   | isin_proc_out  : (M:cap)(isin_cap m M) -> (isin_proc m (out_act M)).
```

Intuitively, (isin_cap x M) (resp. (isin_proc x P)) holds if and only if the name x occurs free in M (resp. P).

Since every process $P$ "lives" in an environment together with other processes, bound names in $P$ must be different not only from the remaining names of $P$, but also from all the other names occurring in the remaining processes of the environment. It follows that in schematic judgments, where a functional term (representing the scope of a binder) is applied to a generic name, the latter must be assumed "conveniently" fresh. For instance, if we are proving some property involving a process (nu P) and we need to "tear off" the nu binder in order to analyze the body P, we can apply P to a generic name n and consider the resulting plain term (P n) provided that n does not occur free in (nu P). Moreover, if the property we are proving involves some other names not occurring in (nu P), we must ensure that n is fresh w.r.t. them too. In order to keep trace of this kind of names and ensure freshness we use an inductive type representing (finite) lists of names:

```
Inductive Nlist : Set :=
   empty : Nlist
 | cons  : name -> Nlist -> Nlist.
```

Non occurrence of a given name in an element of type Nlist is trivially rendered by means of the following predicate:

```
Inductive Nlist_notin [x:name] : Nlist -> Prop :=
   Nlist_notin_empty  : (Nlist_notin x empty)
 | Nlist_notin_cons   : (y:name)(l:Nlist)~x=y -> (Nlist_notin x l) ->
                        (Nlist_notin x (cons y l)).
```

### 6.2.6 Encoding of Structural Congruence

The Structural Congruence relation is represented by the following inductive predicate:

```
Inductive struct_eq: proc -> proc -> Prop :=
  struct_refl      : (P:proc)(struct_eq P P)
| struct_symm      : (P,Q:proc)(struct_eq P Q) -> (struct_eq Q P)
| struct_trans     : (P,Q,R:proc)(struct_eq P Q) -> (struct_eq Q R) ->
                     (struct_eq P R)
| struct_res       : (P,Q:name->proc)(l:Nlist)
                     ((n:name)(Nlist_notin n l) ->
                              (notin_proc n (nu P)) ->
                              (notin_proc n (nu Q)) ->
                              (struct_eq (P n) (Q n))
                     ) -> (struct_eq (nu P) (nu Q))
| struct_par       : (P,Q:proc)(struct_eq P Q) ->
                     (R:proc)(struct_eq (par P R) (par Q R))
```

```
| struct_repl      : (P,Q:proc)(struct_eq P Q) ->
                               (struct_eq (bang P) (bang Q))
| struct_amb       : (P,Q:proc)(struct_eq P Q) ->
                     (n:name)(struct_eq (ambient (name2cap n) P)
                                        (ambient (name2cap n) Q)
                             )
| struct_action    : (P,Q:proc)(struct_eq P Q) ->
                     (M:cap)(struct_eq (cap_act M P) (cap_act M Q))
| struct_input     : (P,Q:name->proc)(l:Nlist)
                     ((n:name)(Nlist_notin n l) ->
                              (notin_proc n (nu P)) ->
                              (notin_proc n (nu Q)) ->
                              (struct_eq (P n) (Q n))
                     ) -> (struct_eq (in_act P) (in_act Q))
| struct_epsilon   : (P:proc)(struct_eq (cap_act eps P) P)
| struct_path      : (M,M':cap)(P:proc)
                     (struct_eq (cap_act (path M M') P)
                                (cap_act M (cap_act M' P))
                     )
| struct_res_res   : (P:name->name->proc)
                     (struct_eq (nu [n:name](nu [m:name](P n m)))
                                (nu [n:name](nu [m:name](P m n)))
                     )
| struct_res_zero  : (struct_eq (nu [n:name]nil) nil)
| struct_res_par   : (P:proc)(Q:name->proc)
                     (struct_eq (nu [n:name](par P (Q n))) (par P (nu Q)))
| struct_res_amb   : (m:name)(P:name->proc)
                     (struct_eq (nu [n:name](ambient (name2cap m) (P n)))
                                            (ambient (name2cap m) (nu P)))
| struct_par_zero  : (P:proc)(struct_eq (par P nil) P)
| struct_par_comm  : (P,Q:proc)(struct_eq (par P Q) (par Q P))
| struct_par_assoc : (P,Q,R:proc)
                     (struct_eq (par (par P Q) R) (par P (par Q R)))
| struct_repl_zero : (struct_eq (bang nil) nil)
| struct_repl_par  : (P,Q:proc)
                     (struct_eq (bang (par P Q)) (par (bang P) (bang Q)))
| struct_repl_copy : (P:proc)(struct_eq (bang P) (par P (bang P)))
| struct_repl_repl : (P:proc)(struct_eq (bang P) (bang (bang P))).
```

Every constructor corresponds to a rule in Figure 6.6; as far as (Struct Res) and (Struct Input) are concerned, the premises are encoded by means of a schematic judgment where the local name `n` is assumed fresh w.r.t. both `(nu P)` and `(nu Q)` and to a list of names `l` which allows to avoid name clashes with external names (as outlined at the end of the previous section).

**Proposition 6.7 (Adequacy of `struct_eq`)** *Let $N \subset \Lambda$ finite, $P, Q \in \Pi_N$,*

1. *(Soundness) if there exists* `t` *canonical such that* $\Gamma_N \vdash_{\Sigma_A} \mathtt{t} : (\mathtt{struct\_eq\ P\ Q})$*, then we have* $\delta^{\Pi}_N(\mathtt{P}) \equiv \delta^{\Pi}_N(\mathtt{Q})$.

2. *(Completeness) if $P \equiv Q$ holds, then there is a canonical form* `t` *such that* $\Gamma_N \vdash_{\Sigma_A}$ `t` : (`struct_eq` $\epsilon^{\Pi}_N(P)$ $\epsilon^{\Pi}_N(Q)$).

*Proof.* Easily proved by induction on the structure of the normal forms (Soundness), and induction on the structure of the derivation of the Structural Congruence judgment (Completeness). □

### 6.2.7 Encoding of the Reduction System

Before introducing the encoding of the reduction system whose rules are depicted in Figure 6.7, we need to represent substitution of capabilities for names since in the axiom (Red Comm) $(n).P|\langle M\rangle$ reduces to $P\{n \leftarrow M\}$. Since a capability is in general a compound term and not simply a name, we cannot delegate the substitution to the metalanguage if we want to be able to use the axioms of the Theory of Contexts. Indeed, in order to delegate the substitution to the metalanguage, we would need to represent $P\{n \leftarrow M\}$ with `(P' M)` where `P':cap->proc`, `P` $\triangleq \epsilon^{\Pi}_X(P)$ and `P=(P' (name2cap n))`. Hence, during the proof development we could not apply $\beta$-expansion and extensionality in order to elaborate on the structure of `P'`.

Since processes are also built on top of capabilities, we need to represent the substitution of capabilities for names both in capabilities and in processes. However, since the type `name` is not inductive, we cannot define in `Coq` the abovementioned substitution as a recursive function as proposed in [Hof99] for the untyped $\lambda$-calculus, but only as a functional relation. Hence, we introduce the following inductive predicates:

```
Inductive subst_cap [M:cap]: (name->cap) -> cap -> Prop :=
  subst_cap_name : (subst_cap M name2cap M)
| subst_cap_void : (n:name)(subst_cap M [_:name](name2cap n) (name2cap n))
| subst_cap_in   : (N:name->cap)(N':cap)(subst_cap M N N') ->
                   (subst_cap M [n:name](in_cap (N n)) (in_cap N'))
| subst_cap_out  : (N:name->cap)(N':cap)(subst_cap M N N') ->
                   (subst_cap M [n:name](out_cap (N n)) (out_cap N'))
| subst_cap_open : (N:name->cap)(N':cap)(subst_cap M N N') ->
                   (subst_cap M [x:name](open (N x)) (open N'))
| subst_cap_eps  : (subst_cap M [_:name]eps eps)
| subst_cap_path : (N,O:name->cap)(N',O':cap)(subst_cap M N N') ->
                   (subst_cap M O O') ->
                   (subst_cap M [x:name](path (N x) (O x)) (path N' O')).

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_nil  : (subst_proc M [_:name]nil nil)
| 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'))
| subst_proc_bang : (P:name->proc)(P':proc)(subst_proc M P P') ->
                    (subst_proc M [x:name](bang (P x)) (bang P'))
| subst_proc_amb  : (N:name->cap)(P:name->proc)(N':cap)(P':proc)
```

```
                    (subst_cap M N N') -> (subst_proc M P P') ->
                    (subst_proc M [x:name](ambient (N x) (P x))
                                 (ambient N' P'))
| subst_proc_cap  : (N:name->cap)(P:name->proc)(N':cap)(P':proc)
                    (subst_cap M N N') -> (subst_proc M P P') ->
                    (subst_proc M [x:name](cap_act (N x) (P x))
                                 (cap_act N' P'))
| subst_proc_in   : (P:name->name->proc)(P':name->proc)
                    ((y:name)(subst_proc M [x:name](P x y) (P' y))) ->
                    (subst_proc M [x:name](in_act (P x)) (in_act P'))
| subst_proc_out  : (N:name->cap)(N':cap)(subst_cap M N N') ->
                    (subst_proc M [x:name](out_act (N x)) (out_act N')).
```

Intuitively, (subst_cap M N N') (resp. subst_proc M P P') holds if and only if the capability N' (resp. process P') is the result of "filling the hole" of the context N (resp. P) with M. The adequacy of subst_cap and subst_proc w.r.t. the capture-avoiding substitutions defined in Figure 6.12 is guaranteed by the following result:

**Proposition 6.8 (Adequacy of substitution)** *Let $L \subset \Lambda$ finite, $M, N' \in \zeta_L$, $N \in \zeta_{L \uplus \{n\}}$, $P' \in \Pi_L$, $P \in \Pi_{L \uplus \{n\}}$, then:*

- $N\{n \leftarrow M\} = N'$ *iff* $\Gamma_L \vdash_{\Sigma_A} \mathtt{t} : (\mathtt{subst\_cap}\ \epsilon^{\zeta}_{L}(M)\ [\mathtt{n} : \mathtt{name}]\epsilon^{\zeta}_{L,n}(N)\ \epsilon^{\zeta}_{L}(N'))$,
- $P\{n \leftarrow M\} = P'$ *iff* $\Gamma_L \vdash_{\Sigma_A} \mathtt{t} : (\mathtt{subst\_proc}\ \epsilon^{\zeta}_{L}(M)\ [\mathtt{n} : \mathtt{name}]\epsilon^{\Pi}_{L,n}(P)\ \epsilon^{\Pi}_{L}(P'))$.

*Proof.* The completeness ($\Rightarrow$) is obtained by structural induction on the derivation of $N\{n \leftarrow M\} = N'$ ($P\{n \leftarrow M\} = P'$), while the soundness ($\Leftarrow$) follows by means of structural induction on the canonical form t. □

The reduction system is then encoded by means of the following inductive predicate:

```
Inductive red: proc -> proc -> Prop :=
  red_in   : (m,n:name)(P,Q,R:proc)
  (red (par (ambient (name2cap n) (par (cap_act (in_cap (name2cap m)) P) Q))
            (ambient (name2cap m) R))
       (ambient (name2cap m) (par (ambient (name2cap n) (par P Q)) R))
  )
| red_out  : (m,n:name)(P,Q,R:proc)
  (red (ambient (name2cap m)
                (par (ambient (name2cap n)
                              (par (cap_act (out_cap (name2cap m)) P) Q)) R))
       (par (ambient (name2cap n) (par P Q)) (ambient (name2cap m) R))
  )
| red_open : (n:name)(P,Q:proc)
  (red (par (cap_act (open (name2cap n)) P) (ambient (name2cap n) Q))
       (par P Q))
| red_comm : (P:name->proc)(M:cap)(P':proc)
             (subst_proc M P P') -> (red (par (in_act P) (out_act M)) P')
| red_res  : (P,Q:name->proc)(l:Nlist)
             ((n:name)(Nlist_notin n l) -> (notin_proc n (nu P)) ->
                      (notin_proc n (nu Q)) -> (red (P n) (Q n))
             ) -> (red (nu P) (nu Q))
```

$$n\{n \leftarrow M\} = M$$
$$m\{n \leftarrow M\} = m \quad (m \neq n)$$
$$(in\ N)\{n \leftarrow M\} = in\ N\{n \leftarrow M\}$$
$$(out\ N)\{n \leftarrow M\} = out\ N\{n \leftarrow M\}$$
$$(open\ N)\{n \leftarrow M\} = open\ N\{n \leftarrow M\}$$
$$\varepsilon\{n \leftarrow M\} = \varepsilon$$
$$N.N'\{n \leftarrow M\} = N\{n \leftarrow M\}.N'\{n \leftarrow M\}$$

$$((\nu m)P)\{n \leftarrow M\} = \begin{cases} (\nu m)P\{n \leftarrow M\} & \text{if } m \neq n \text{ and } (m \notin fn(M) \text{ or } n \notin fn(P)) \\ (\nu z)P\{m \leftarrow z\}\{n \leftarrow M\} & \text{where } z \text{ fresh, otherwise} \end{cases}$$
$$\mathbf{0}\{n \leftarrow M\} = \mathbf{0}$$
$$(P|Q)\{n \leftarrow M\} = P\{n \leftarrow M\}|Q\{n \leftarrow M\}$$
$$(N[P])\{n \leftarrow M\} = N\{n \leftarrow M\}[P\{n \leftarrow M\}]$$
$$(N.P)\{n \leftarrow M\} = N\{n \leftarrow M\}.P\{n \leftarrow M\}$$
$$((m).P)\{n \leftarrow M\} = \begin{cases} (m).P\{n \leftarrow M\} & \text{if } m \neq n \text{ and } (m \notin fn(M) \text{ or } n \notin fn(P)) \\ (z).P\{m \leftarrow z\}\{n \leftarrow M\} & \text{where } z \text{ fresh, otherwise} \end{cases}$$
$$\langle N\rangle\{n \leftarrow M\} = \langle N\{n \leftarrow M\}\rangle$$

Figure 6.12: Capture-avoiding Substitution of capabilities for names.

```
| red_par   : (P,Q:proc)(red P Q) -> (R:proc)(red (par P R) (par Q R))
| red_amb   : (P,Q:proc)(red P Q) -> (n:name)
              (red (ambient (name2cap n) P) (ambient (name2cap n) Q))
| red_struct_eq : (P,Q:proc)(red P Q) -> (P':proc)(struct_eq P P') ->
                  (Q':proc)(struct_eq Q Q') -> (red P' Q').
```

As in the case of the encoding of rules (Struct Res) and (Struct Input) in the previous section, the rule (Red Res) is represented by encoding its premise by means of a schematic judgment where the local name `n` is assumed fresh w.r.t. both `(nu P)` and `(nu Q)` and to a list of names `l` which allows to avoid name clashes with external names. It should be noticed that, having implemented the substitution of capabilities for names by means of `subst_proc` which takes a higher-order term as the second argument, allows us to avoid the use of a schematic judgment in the rule `red_comm`. Indeed, if `cap->proc->proc` would be the type of `subst_proc`, `red_comm` would necessarily have the following shape:

```
(P:name->proc)(M:cap)(P':proc)
((n:name)(subst_proc M (P n) P')) -> (red (par (in_act P) (out_act M)) P')
```

**Proposition 6.9 (Adequacy of `red`)** *Let $N \subset \Lambda$ finite, $P, Q \in \Pi_N$,*

1. *(Soundness) if there exists* `t` *canonical such that* $\Gamma_N \vdash_{\Sigma_A}$ `t` : (`red P Q`), *then we have* $\delta_N^{\Pi}(\mathtt{P}) \rightarrow \delta_N^{\Pi}(\mathtt{Q})$.
2. *(Completeness) if $P \rightarrow Q$ holds, there is a canonical form* `t` *such that we can derive* $\Gamma_N \vdash_{\Sigma_A}$ `t` : (`red` $\epsilon_N^{\Pi}(P)$ $\epsilon_N^{\Pi}(Q)$).

*Proof.* Easily proved by induction on the structure of the normal forms (Soundness), and induction on the structure of the derivation of the reduction judgment (Completeness). □

### 6.2.8 Encoding of formulæ

In Section 6.2.4 we recalled the syntax of formulæ which required an additional sort of variables in order to have a universal quantifier and reasoning about open formulæ(that is formulæwith free variables). 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 $\oslash$ since they can take as one of their arguments either a name or a variable (see the second example of Section 4.2). Moreover, the universal quantifier would be encoded by means of a constructor of type `(var->form)->form`; this fact would prevents us from taking advantage of the possibility of encoding 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, despite an HOAS-based encoding approach, we could delegate to the metalanguage only $\alpha$-conversion of formulæ.

The solution we propose instead will encode variables of the object language by means of `Coq` variables of type `name`; however, the latter, differently from those encoding names, will not be assumed to be distinct. 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. This approach is not peculiar to HOAS-encodings; indeed, if we think of the first-order logic encoding given in [HHP93], we see that the same pattern is followed in representing variables of the object language. Actually, the latter are not represented through a separate type, but as variables of the metalanguage whose type is the one encoding individuals (i.e. the elements they stand for). In our case there will be no risk of confusing variables representing names with variables representing variables of the object language. Indeed, only the former come equipped with inequality judgments that allow one to think of them as constants, rather than variables[5].

The inductive type encoding the syntactic category of formulæ is the following:

```
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 chapter we will denote by $\Gamma_X$ the typing environment $\{\texttt{x1} : \texttt{name}, \ldots, \texttt{xn} : \texttt{name}\}$. Encoding and decoding functions

[5] In `Coq` a set of names $n_1, \ldots, n_k$ could be introduced by the statement `Parameter n1,...,nk:name` together with the statements `Axiom dij:~(ni=nj)` for $1 \leq i < j \leq k$, $i \neq j$.

$$
\begin{array}{rcl}
\epsilon^{\Phi}_{N,X} & : & \Phi_{N,X} \longrightarrow \texttt{form}_{N,X} \\
\epsilon^{\Phi}_{N,X}(\mathbf{T}) & \triangleq & \texttt{T} \\
\epsilon^{\Phi}_{N,X}(\neg\mathcal{A}) & \triangleq & (\texttt{neg}\ \epsilon^{\Phi}_{N,X}(\mathcal{A})) \\
\epsilon^{\Phi}_{N,X}(\mathcal{A} \vee \mathcal{B}) & \triangleq & (\texttt{Or}\ \epsilon^{\Phi}_{N,X}(\mathcal{A})\ \epsilon^{\Phi}_{N,X}(\mathcal{B})) \\
\epsilon^{\Phi}_{N,X}(\mathcal{A}|\mathcal{B}) & \triangleq & (\texttt{comp}\ \epsilon^{\Phi}_{N,X}(\mathcal{A})\ \epsilon^{\Phi}_{N,X}(\mathcal{B})) \\
\epsilon^{\Phi}_{N,X}(\mathcal{A} \rhd \mathcal{B}) & \triangleq & (\texttt{comp\_adj}\ \epsilon^{\Phi}_{N,X}(\mathcal{A})\ \epsilon^{\Phi}_{N,X}(\mathcal{B})) \\
\epsilon^{\Phi}_{N,X}(\eta[\mathcal{A}]) & \triangleq & \begin{cases} (\texttt{loc n}\ \epsilon^{\Phi}_{N,X}(\mathcal{A})) & \text{if } \eta = n \in N \\ (\texttt{loc x}\ \epsilon^{\Phi}_{N,X}(\mathcal{A})) & \text{if } \eta = x \in X \end{cases} \\
\epsilon^{\Phi}_{N,X}(\mathcal{A}@\eta) & \triangleq & \begin{cases} (\texttt{loc\_adj}\ \epsilon^{\Phi}_{N,X}(\mathcal{A})\ \texttt{n}) & \text{if } \eta = n \in N \\ (\texttt{loc\_adj}\ \epsilon^{\Phi}_{N,X}(\mathcal{A})\ \texttt{x}) & \text{if } \eta = x \in X \end{cases} \\
\epsilon^{\Phi}_{N,X}(\eta®\mathcal{A}) & \triangleq & \begin{cases} (\texttt{rev n}\ \epsilon^{\Phi}_{N,X}(\mathcal{A})) & \text{if } \eta = n \in N \\ (\texttt{rev x}\ \epsilon^{\Phi}_{N,X}(\mathcal{A})) & \text{if } \eta = x \in X \end{cases} \\
\epsilon^{\Phi}_{N,X}(\mathcal{A} \oslash \eta) & \triangleq & \begin{cases} (\texttt{rev\_adj}\ \epsilon^{\Phi}_{N,X}(\mathcal{A})\ \texttt{n}) & \text{if } \eta = n \in N \\ (\texttt{rev\_adj}\ \epsilon^{\Phi}_{N,X}(\mathcal{A})\ \texttt{x}) & \text{if } \eta = x \in X \end{cases} \\
\epsilon^{\Phi}_{N,X}(\Diamond\mathcal{A}) & \triangleq & (\texttt{sometime}\ \epsilon^{\Phi}_{N,X}(\mathcal{A})) \\
\epsilon^{\Phi}_{N,X}(\diamondsuit\mathcal{A}) & \triangleq & (\texttt{somewhere}\ \epsilon^{\Phi}_{N,X}(\mathcal{A})) \\
\epsilon^{\Phi}_{N,X}(\forall x.\mathcal{A}) & \triangleq & (\texttt{forall}\ [\texttt{x}:\texttt{name}]\ \epsilon^{\Phi}_{N,X,x}(\mathcal{A}))
\end{array}
$$

Figure 6.13: Encoding map of formulæ.

between formulæ with free names in $N$ ($\Phi_{N,X}$) and free variables in $X$ and canonical forms `t` of type `form` such that $\Gamma_N, \Gamma_X \vdash_{\Sigma_A} \texttt{t} : \texttt{form}$ ($\texttt{form}_{N,X}$) are illustrated in Figures 6.13 and 6.14.

Similarly to the *fresh* function we used in defining the decoding map $\delta^{\Pi}_{N}$ in Section 6.2.5, $\mathit{fresh}_{\Phi} : \mathcal{P}_{<\omega}(\Lambda \cup \vartheta) \longrightarrow \Lambda \cup \vartheta$ such that for every $A \subset \Lambda \cup \vartheta$ finite, $\mathit{fresh}_{\Phi}(A) \notin A$. Again, if $\Lambda \cup \vartheta$ is enumerable, a possible definition for $\mathit{fresh}_{\Phi}$ is $\mathit{fresh}_{\Phi}(A) = n_{max\{i|n_i \in A\}+1}$, where $(n_i)_i$ is a given enumeration of $\Lambda \cup \vartheta$.

The adequacy of the encoding is given by the following result:

**Proposition 6.10** *For each $N \subset \Lambda$ finite, $X \subset \vartheta$, $\epsilon^{\Phi}_{N,X}$ is a compositional bijection between $\Phi_{N,X}$ and $\texttt{form}_{N,X}$.*

*Proof.* Standard, using the definitions in Figures 6.13 and 6.14 and proceeding by induction on the structure of formulæ and of canonical forms of type `form`. □

The freshness predicate for terms of type `form` is inductively defined as follows:

```
Inductive notin_form [m:name] : form->Prop :=
    notin_T         : (notin_form m T)
  | notin_neg       : (F:form)(notin_form m F) -> (notin_form m (neg F))
  | notin_Or        : (F,G:form)(notin_form m F) -> (notin_form m G) ->
                      (notin_form m (Or F G))
  | notin_zero      : (notin_form m zero)
  | notin_comp      : (F,G:form)(notin_form m F) -> (notin_form m G) ->
                      (notin_form m (comp F G))
  | notin_comp_adj  : (F,G:form)(notin_form m F) -> (notin_form m G) ->
                      (notin_form m (comp_adj F G))
```

$$
\begin{array}{rcl}
\delta^{\Phi}_{N,X} & : & \mathtt{form}_{N,X} \longrightarrow \Phi_{N,X} \\
\delta^{\Phi}_{N,X}(\mathtt{T}) & \triangleq & \mathbf{T} \\
\delta^{\Phi}_{N,X}(\mathtt{(neg\ A)}) & \triangleq & \neg\delta^{\Phi}_{N,X}(\mathtt{A}) \\
\delta^{\Phi}_{N,X}(\mathtt{(Or\ A\ B)}) & \triangleq & \delta^{\Phi}_{N,X}(\mathtt{A}) \vee \delta^{\Phi}_{N,X}(\mathtt{B}) \\
\delta^{\Phi}_{N,X}(\mathtt{zero}) & \triangleq & \mathbf{0} \\
\delta^{\Phi}_{N,X}(\mathtt{(comp\ A\ B)}) & \triangleq & \delta^{\Phi}_{N,X}(\mathtt{A})|\delta^{\Phi}_{N,X}(\mathtt{B}) \\
\delta^{\Phi}_{N,X}(\mathtt{(comp\_adj\ A\ B)}) & \triangleq & \delta^{\Phi}_{N,X}(\mathtt{A}) \triangleright \delta^{\Phi}_{N,X}(\mathtt{B}) \\
\delta^{\Phi}_{N,X}(\mathtt{(loc\ n\ A)}) & \triangleq & n[\delta^{\Phi}_{N,X}(\mathtt{A})] \quad \text{if } \mathtt{n} : \mathtt{name} \in \Gamma_N \\
\delta^{\Phi}_{N,X}(\mathtt{(loc\ x\ A)}) & \triangleq & x[\delta^{\Phi}_{N,X}(\mathtt{A})] \quad \text{if } \mathtt{x} : \mathtt{name} \in \Gamma_X \\
\delta^{\Phi}_{N,X}(\mathtt{(loc\_adj\ A\ n)}) & \triangleq & \delta^{\Phi}_{N,X}(\mathtt{A})@n \quad \text{if } \mathtt{n} : \mathtt{name} \in \Gamma_N \\
\delta^{\Phi}_{N,X}(\mathtt{(loc\_adj\ A\ x)}) & \triangleq & \delta^{\Phi}_{N,X}(\mathtt{A})@x \quad \text{if } \mathtt{x} : \mathtt{name} \in \Gamma_X \\
\delta^{\Phi}_{N,X}(\mathtt{(rev\ n\ A)}) & \triangleq & n®\delta^{\Phi}_{N,X}(\mathtt{A}) \quad \text{if } \mathtt{n} : \mathtt{name} \in \Gamma_N \\
\delta^{\Phi}_{N,X}(\mathtt{(rev\ x\ A)}) & \triangleq & x®\delta^{\Phi}_{N,X}(\mathtt{A}) \quad \text{if } \mathtt{x} : \mathtt{name} \in \Gamma_X \\
\delta^{\Phi}_{N,X}(\mathtt{(rev\_adj\ A\ n)}) & \triangleq & \delta^{\Phi}_{N,X}(\mathtt{A}) \oslash n \quad \text{if } \mathtt{n} : \mathtt{name} \in \Gamma_N \\
\delta^{\Phi}_{N,X}(\mathtt{(rev\_adj\ A\ x)}) & \triangleq & \delta^{\Phi}_{N,X}(\mathtt{A}) \oslash x \quad \text{if } \mathtt{x} : \mathtt{name} \in \Gamma_X \\
\delta^{\Phi}_{N,X}(\mathtt{(forall\ A)}) & \triangleq & \forall z.\delta^{\Phi}_{N,X,z}(\mathtt{(A\ z)}) \quad \text{where } z = \mathit{fresh}_{\Phi}(N \cup X)
\end{array}
$$

Figure 6.14: Decoding map of formulæ.

```
  | notin_loc       : (F:form)(n:name)~m=n -> (notin_form m F) ->
                      (notin_form m (loc n F))
  | notin_loc_adj   : (F:form)(n:name)~m=n -> (notin_form m F) ->
                      (notin_form m (loc_adj F n))
  | notin_rev       : (F:form)(n:name)~m=n -> (notin_form m F) ->
                      (notin_form m (rev n F))
  | notin_rev_adj   : (F:form)(n:name)~m=n -> (notin_form m F) ->
                      (notin_form m (rev_adj F n))
  | notin_sometime  : (F:form)(notin_form m F) ->
                              (notin_form m (sometime F))
  | notin_somewhere : (F:form)(notin_form m F) ->
                              (notin_form m (somewhere F))
  | notin_forall    : (F:name->form)
                      ((n:name)~m=n -> (notin_form m (F n))) ->
                      (notin_form m (forall F)).
```

### 6.2.9 Encoding of Satisfaction

Before introducing the encoding of the satisfaction relation, we need to represent the relations $\downarrow$ and $\downarrow^*$ (see Section 6.2.4) and $\rightarrow^*$ (see Section 6.2.7). The nesting relation is represented by the following definition:

```
Definition nest:= [P:proc][P':proc]
  (Ex [n:name]
      (Ex [P'':proc](struct_eq P (par (ambient (name2cap n) P') P'')))).
```

Being defined in terms of `struct_eq`, its adequacy is a direct consequence of Propositions 6.5, 6.6 and 6.7. Reflexive and transitive closures of $\rightarrow$ and $\downarrow$ are encoded by the following inductive predicates:

```
Inductive red_star: proc -> proc -> Prop :=
  base_red  : (P,Q:proc)(red P Q) -> (red_star P Q)
| red_refl  : (P:proc)(red_star P P)
| red_trans : (P,Q,R:proc)
              (red_star P Q) -> (red_star Q R) -> (red_star P R).

Inductive nest_star: proc -> proc -> Prop :=
  base_nest  : (P,Q:proc)(nest P Q) -> (nest_star P Q)
| nest_refl  : (P:proc)(nest_star P P)
| nest_trans : (P,Q,R:proc)
               (nest_star P Q) -> (nest_star Q R) -> (nest_star P R).
```

The adequacy of `red_star` follows from Propositions 6.6 and 6.9 while the adequacy of `nest_star` is a consequence of Proposition 6.6 and the adequacy of `nest`.

Coming to the representation of the satisfaction relation, we cannot use an inductive definition since the introduction rule for $\neg$ and $\triangleright$ do not satisfy the positivity constraints imposed by the `Coq` type system on inductive constructors. Indeed, the inductive predicate encoding satisfaction 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))
...
```

It is clear that the occurrence `(sat P' A)` in `sat_comp_adj` is negative. There is also a negative occurrence of `sat` in `sat_neg` since `~(sat P A)` is an abbreviation for `(sat P A) -> False`. Hence, `Coq` complains about the previous inductive definition rejecting it.

Hence, the encoding approach we adopt is to axiomatize explicitly the satisfaction rules by means of `Axiom` statements like one would do in the Edinburgh Logical Framework [HHP93, AHMP92]:

```
Parameter sat : proc -> form -> Prop.

Axiom sat_T: (P:proc)(sat P T).

Axiom sat_neg: (P:proc)(A:form)~(sat P A) <-> (sat P (neg A)).

Axiom sat_Or: (P:proc)(A,B:form)((sat P A) \/ (sat P B)) <-> (sat P
(Or A B)).

Axiom sat_zero: (P:proc)(struct_eq P nil) <-> (sat P zero).

Axiom sat_comp: (P:proc)(A,B:form)
                (Ex [P':proc](Ex [P'':proc](struct_eq P (par P' P''))
                                        /\ (sat P' A) /\ (sat P'' B)))
                <-> (sat P (comp A B)).
```

```
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)).

Axiom sat_loc: (P:proc)(n:name)(A:form)
               (Ex [P':proc](struct_eq P (ambient (name2cap n) P')) /\
                            (sat P' A))
               <-> (sat P (loc n A)).

Axiom sat_loc_adj: (P:proc)(A:form)(n:name)
             (sat (ambient (name2cap n) P) A) <-> (sat P (loc_adj A n)).

Axiom sat_rev: (P:proc)(n:name)(A:form)
           ((notin_proc n P) /\ (Ex [P':name->proc](struct_eq P (nu P'))
                             /\ (sat (P' n) A)))
           <-> (sat P (rev n A)).

Axiom sat_rev_adj: (P:proc)(n:name)(A:form)
           (Ex [P':name->proc]P=(P' n)
                            /\ (notin_proc n (nu P')) /\ (sat (nu P') A))
           <-> (sat P (rev_adj A n)).

Axiom sat_sometime: (P:proc)(A:form)
                    (Ex [P':proc](red_star P P') /\ (sat P' A))
                    <-> (sat P (sometime A)).

Axiom sat_somewhere: (P:proc)(A:form)
                     (Ex [P':proc](nest_star P P') /\ (sat P' A))
                     <-> (sat P (somewhere A)).

Axiom sat_forall: (P:proc)(A:name->form)
                  ((m:name)(sat P (A m))) <-> (sat P (forall A)).
```

Notice that the symbol `<->` in the previous axioms represents $\triangleq$ in Figure 6.9. The adequacy of `sat` is given by the following proposition:

**Proposition 6.11 (Adequacy of `sat`)** *Let $N \subset \Lambda$ finite, $P \in \Pi_N$, $\mathcal{A} \in \Phi_{N,\emptyset}$,*

1. *(Soundness) if there exists* `t` *canonical such that* $\Gamma_N \vdash_{\Sigma_A}$ `t` : (`sat P A`)*, then we have* $\delta^{\Pi}_N(\mathtt{P}) \models \delta^{\Phi}_{N,\emptyset}(\mathtt{A})$.
2. *(Completeness) if $P \models A$ holds, there is a canonical form* `t` *such that we can derive* $\Gamma_N \vdash_{\Sigma_A}$ `t` : (`sat` $\epsilon^{\Pi}_N(P)$ $\epsilon^{\Phi}_{N,\emptyset}(\mathcal{A})$).

*Proof.* The argument is an induction on the structure of the normal form `A` (Soundness), and a structural induction on the formula $\mathcal{A}$ (Completeness). □

Notice that $\mathcal{A}$ in the previous proposition ranges only over $\Phi_{N,\emptyset}$ (closed formulæ) because in [CG01] the satisfaction relation is defined only on formulæ with no free variables.

We conclude this section by remarking that a non-inductive definition of a relation can be adequate in the activity of computer assisted proof development as long as we do not need to prove properties by structural induction on derivations. Luckily, this is the case for the properties of the satisfaction relation we proved so far. However, in order to have a better interaction with type theory based proof assistants like Coq, it would be useful to formulate a natural deduction style system for the satisfaction relation. Among the benefits of this solution, it is worth mentioning that we could use the automatic inversion tactics on instances of sat judgments instead of manually pruning inconsistent cases or manually deriving the syntactic constraints on the arguments involved.

### 6.2.10 The Theory of Contexts for the Ambient Calculus

In this section we will introduce the Theory of Contexts adapted to the encoding of the Ambient Calculus. As usual, nominal calculi require to be able to decide whether two names are equal or not[6]; whence we assume that Leibniz equality over names is decidable:

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

Since we have several distinct syntactic categories (names, capabilities, processes and formulæ), the shape of the unsaturation axiom involves all of them:

```
Axiom unsat: (l:Nlist)(M:cap)(P:proc)(F:form)
             (Ex [n:name](Nlist_notin n l) /\
                         (notin_cap n M)   /\
                         (notin_proc n P)  /\
                         (notin_form n F)
             ).
```

When needed, one can then deduce from unsat more specialized and weaker forms, e.g., unsaturation for processes, formulæ, capabilities and processes etc. Here we list the particular unsaturation properties we used in the formal development described in Section 6.2.11:

```
Lemma UNSAT_PROC_NLIST: (P:proc)(l:Nlist)
                        (Ex [n:name](notin_proc n P) /\ (Nlist_notin n l)).

Lemma UNSAT_FORM: (F:form)(Ex [x:name](notin_form x F)).

Lemma UNSAT_CAP_PROC_NLIST: (M:cap)(P:proc)(l:Nlist)
                            (Ex [n:name](notin_cap n M)  /\
                                        (notin_proc n P) /\
                                        (Nlist_notin n l)
                            ).

Lemma UNSAT_NLIST_PROC_FORM: (l:Nlist)(P:proc)(F:form)
                             (Ex [n:name](Nlist_notin n l) /\
                                         (notin_proc n P)  /\
                                         (notin_form n F)
                             ).
```

[6] Despite the fact that it is often implicitly assumed in the presentations of nominal calculi, the decidability of the equality over names is heavily used in the development of many fundamental metatheorical results (see, e.g., the proof of Lemma 6 in [MPW89] or the proof of Lemma 4-6 in [CG00]).

As far as the extensionality and the $\beta$-expansion properties are concerned, we need several instances (for plain terms, unary and binary contexts) for capabilities, processes and formulæ:
**Capabilities.** Extensionality:

```
Axiom cap_ext: (M,N:name->cap)(n:name)
               (notin_ho_cap n M) -> (notin_ho_cap n N) ->
               (M n)=(N n) -> M=N.

Axiom ho_cap_ext: (M,N:name->name->cap)(n:name)
                  (notin_ho2_cap n M) -> (notin_ho2_cap n N) ->
                  (M n)=(N n) -> M=N.
```

$\beta$-expansion:

```
Axiom cap_exp: (M:cap)(n:name)
               (Ex [M':name->cap](notin_ho_cap n M') /\ M=(M' n)).

Axiom ho_cap_exp: (M:name->cap)(n:name)
               (Ex [M':name->name->cap](notin_ho2_cap n M') /\ M=(M' n)).
```

**Processes.** Extensionality:

```
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.

Axiom ho_proc_ext: (P,Q:name->name->proc)(x:name)
                   (notin_proc x (nu [_:name](nu (P _)))) ->
                   (notin_proc x (nu [_:name](nu (Q _)))) ->
                   (P x)=(Q x) -> P=Q.
```

$\beta$-expansion:

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

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

Axiom ho2_proc_exp: (P:name->name->proc)(n:name)
               (Ex [P':name->name->name->proc]
                   (notin_proc n (nu [u:name](nu [v:name](nu (P' u v)))))
                   /\ P=(P' n)).
```

**Formulæ.** Extensionality:

```
Axiom form_ext: (F,G:name->form)(x:name)
                (notin_form x (forall F)) -> (notin_form x (forall G)) ->
                (F x)=(G x)->F=G.
```

$\beta$-expansion:

```
Axiom form_exp: (F:form)(n:name)
                (Ex [G:name->form](notin_form n (forall G)) /\ F=(G n)).

Axiom ho_form_exp: (F:name->form)(n:name)
  (Ex [G:name->name->form]
      (notin_form n (forall ([_:name](forall (G _)))))
   /\ F=(G n)).

Axiom ho2_form_exp: (F:name->name->form)(n:name)
  (Ex [G:name->name->name->form]
      (notin_form n (forall ([u:name](forall [v:name](forall (G u v))))))
   /\ F=(G n)).
```

A very useful property, used a lot in the formal development described in [HMS01b] and derivable from the Theory of Contexts, is the monotonicity of freshness predicates:

```
Lemma NOTIN_CAP_MONO  : (M:name->cap)(x,y:name)(notin_cap x (M y)) ->
                        (notin_ho_cap x M).

Lemma NOTIN_PROC_MONO : (P:name->proc)(x,y:name)(notin_proc x (P y)) ->
                        (notin_proc x (nu P)).
```

Informally, the previous lemmata state that, in order to conclude that a name `x` does not occur free in a capability context `M` (resp. a process context `P`), it is sufficient to prove that `x` does not occur free in `(M y)` (resp. `(P y)`) for a generic name `y`. The proof technique used to derive the first result is the same illustrated in Section 6.1.2 to prove `HO_TM_IND`, i.e., we first prove the following lemma which then implies (by means of the unsaturation property[7]) `NOTIN_CAP_MONO`:

```
Lemma PRE_NOTIN_CAP_MONO: (A:cap)(M:name->cap)
                          (z:name)(notin_ho_cap z M) ->
                          A=(M z) -> (x,y:name)(notin_cap x (M y)) ->
                          (notin_ho_cap x M).
```

As far as the proof of the second monotonicity result is concerned, we need a more sophisticated approach. Indeed, the presence of higher-order constructors requires a stronger induction hypotheses in order to apply it several times to terms which are not proper subterms of those involved in the current case, but that may differ by some renamings. Hence, we must define a suitable measure on processes and proceed by induction on it. The following inductive definition formalizes such a notion:

```
Inductive lnp: proc -> nat -> Prop:=
  lnp_nu    : (P:name->proc)(n:nat)((x:name)(lnp (P x) n)) ->
                                     (lnp (nu P) (S n))
```

[7]Notice that in this case we can obtain a fresh name not occurring in a capability context even if this syntactic category does not feature a higher-order constructor. Indeed, capabilities are used to build processes which have two higher-order constructors (namely, restriction and input action); hence, to yield a name not occurring in `M:name->cap` we can eliminate the unsaturation property over, e.g., `(nu [_:name](out_act (M _)))`.

```
| lnp_nil  : (lnp nil (S O))
| lnp_par  : (P,Q:proc)(n1,n2:nat)(lnp P n1) -> (lnp Q n2) ->
             (lnp (par P Q) (S (plus n1 n2)))
| lnp_bang : (P:proc)(n:nat)(lnp P n) -> (lnp (bang P) (S n))
| lnp_amb  : (P:proc)(n:nat)(lnp P n) -> (M:cap)(lnp (ambient M P) (S n))
| lnp_cap  : (P:proc)(n:nat)(lnp P n) -> (M:cap)(lnp (cap_act M P) (S n))
| lnp_in   : (P:name->proc)(n:nat)((x:name)(lnp (P x) n)) ->
             (lnp (in_act P) (S n))
| lnp_out  : (M:cap)(lnp (out_act M) (S O)).
```

Intuitively, `(lnp P n)` holds if and only if the term `P` contains exactly `n` occurrences of constructors of type `proc`. The following results state that `lnp` is preserved by arbitrary renamings and that it is possible to associate a measure to every term of type `proc`:

```
Lemma LNP_RW: (n:nat)(P:proc)(lnp P n) ->
              (x:name)(Q:name->proc)(notin_proc x (nu Q)) ->
              P=(Q x) -> (y:name)(lnp (Q y) n).

Lemma LNP_TOT: (P:proc)(Ex [n:nat](lnp P n)).
```

The first lemma requires a complete induction on `n` (complete induction on natural numbers is easily derivable in Coq[8]), while the second is proved by structural induction on `A` (`LNP_RW` is needed in the cases related to restriction and input action). At this point the following result can be derived by means of a complete induction on `n` and `LNP_RW`:

```
Lemma PRE_NOTIN_PROC_MONO: (n:nat)(A:proc)(lnp A n) ->
                           (P:name->proc)(z:name)(notin_proc z (nu P)) ->
                           A=(P z) -> (x,y:name)(notin_proc x (P y)) ->
                           (notin_proc x (nu P)).
```

Then `NOTIN_PROC_MONO` easily follows by means of the unsaturation property and `LNP_TOT`. It is important to stress that the axioms of $\beta$-expansion and extensionality over capabilities and processes (at various levels) are essential in order to "lift" the structural information available about plain terms to the level of functional terms (i.e. contexts) in the same way described in Section 6.1.2.

The same proof techniques described above can be successfully applied in order to derive the following monotonicity results about the free occurrence predicates `isin_cap` and `isin_proc`:

```
Lemma ISIN_CAP_MONO  : (M:name->cap)(x,y:name)~x=y ->
                       (isin_cap x (M y)) -> (isin_ho_cap x M).

Lemma ISIN_PROC_MONO : (P:name->proc)(x,y:name)~x=y ->
                       (isin_proc x (P y)) -> (isin_proc x (nu P)).
```

Then we can derive another useful property whose explicit axiomatization is needed when we do not work in a classical setting, namely, the decidability of occur checking:

```
Lemma NOTIN_CAP_DEC  : (M:cap)(x:name)(isin_cap x M) \/ (notin_cap x M).

Lemma NOTIN_PROC_DEC : (P:proc)(n:name)(isin_proc n P) \/ (notin_proc n P).
```

[8] See Section 6.1.5 for the definition in Coq of the complete induction principle on natural numbers.

It follows that the only classical flavour we need is the decidability of the equality over names (`dec_name`).

### 6.2.11 Formal metatheory

In this section we will describe the formal development so far accomplished about the Ambient Calculus and the related modal logic previously recalled (Sections 6.2.1 to 6.2.4). This is a work in progress (and still far from being finished); however, we already proved a set of interesting properties about fresh renamings which are at the heart of the metatheory of the Ambient Calculus and Ambient Logic.

#### Structural Congruence is preserved by Fresh Renaming

The first properties we prove are related to the Structural Congruence: the main result is that the latter is preserved by fresh renaming:

**Lemma 6.1** *For all processes $P$, $Q$ if $P \equiv Q$ holds, then for all names $n$, $m$ (where $m \notin fn(P) \cup fn(Q)$) we have that $P\{n \leftarrow m\} \equiv Q\{n \leftarrow m\}$ also holds.* □

The formalization in `Coq` of the previous property is as follows:

```
Lemma STRUCT_RW: (P,Q:name->proc)(x:name)
                 (notin_proc x (nu P)) -> (notin_proc x (nu Q)) ->
                 (struct_eq (P x) (Q x)) ->
                 (y:name)(notin_proc y (nu P)) -> (notin_proc y (nu Q)) ->
                 (struct_eq (P y) (Q y)).
```

That is, we model substitution of names for names by means of functional application. Indeed, by means of `proc_exp` a term `R` of type `proc` can always be equated to `(P x)` for a given `x` not occurring in `(nu P)`. Whence, if `R` $\triangleq \epsilon^{\Pi}_{X}(P)$, the result of the substitution $P\{n \leftarrow m\}$ formally corresponds to `(P y)`.

A preliminary result necessary in order to prove Lemma 6.1 is the following:

**Lemma 6.2** *For all processes $P$, $Q$, if $P \equiv Q$ then $fn(P) = fn(Q)$.* □

We proved this fact by the following two lemmata in `Coq`:

```
Lemma STRUCT_NOTIN : (A,B:proc)(struct_eq A B) ->
                     (x:name)((notin_proc x A) -> (notin_proc x B)) /\
                             ((notin_proc x B) -> (notin_proc x A)).

Lemma STRUCT_ISIN  : (A,B:proc)(struct_eq A B) ->
                     (x:name)((isin_proc x A)  -> (isin_proc x B)) /\
                             ((isin_proc x B)  -> (isin_proc x A)).
```

The first lemma is a straightforward structural induction on the derivation of the premise, while the second follows from the former by applying `NOTIN_PROC_DEC`.

The abovementioned results correspond to Lemma 4-3 of [CG00] (extended with the cases involving the restriction operator):

> Consider any process $P$ and names $m$, $m'$, with $m' \notin fn(P)$. For all $P'$, if $P \equiv P'$ then $m' \notin fn(P')$ and $P\{m \leftarrow m'\} \equiv P'\{m \leftarrow m'\}$. Moreover, for all $Q$, if $P\{m \leftarrow m'\} \equiv Q$ then there is a $P'$ with $P \equiv P'$, $m' \notin fn(P')$ and $Q = P'\{m \leftarrow m'\}$. □

Representing $P$ with `(P m)` and $P'$ with `(Q m)` allows us to formalize both directions of the quoted lemma by means of `RED_RW`. Indeed, `m` and `m'` can be swapped without altering the meaning of the lemma, allowing to deduce `(struct_eq (P m) (Q m))` from `(struct_eq (P m') (Q m'))`, i.e., $P \equiv P'$ from $P\{m \leftarrow m'\} \equiv Q$ where $Q = P'\{m \leftarrow m'\}$. This is similar to the formalization of the $\pi$-calculus metatheory carried out in [HMS01b] where Lemma 3' and Lemma 4 yield the same formalization. It is interesting to notice how naturally and cleanly HOAS allows one to express this kind of property about fresh renamings. The latter play a fundamental rôle in the metatheory of nominal calculi and their importance is confirmed by the criterion introduced in [Pit01b] in order to establish which properties are worthwhile. Such a criterion amounts to the notion of *equivariance property* (also known as the "fundamental assumption of Nominal Logic"), i.e., the invariance of the validity under *swapping* names.

Lemma `STRUCT_RW` allows to derive the following result which is often useful during the formal development:

```
Lemma STRUCT_RES: (P,Q:name->proc)(x:name)
                  (notin_proc x (nu P)) -> (notin_proc x (nu Q)) ->
                  (struct_eq (P x) (Q x)) -> (struct_eq (nu P) (nu Q)).
```

This essentially amounts to a simpler introduction rule (w.r.t. the constructor `struct_res`) for establishing the structural congruence of two restricted processes.

### Satisfaction is up to Structural Congruence

A crucial metatheoretical result involving the satisfaction relation is Lemma 2-1 of [CG01]:

$$(P \models \mathcal{A} \wedge P \equiv P') \Rightarrow P' \models \mathcal{A} \qquad \square$$

The corresponding formalization in `Coq` is the following:

```
Lemma SAT_UPTO_STRUCT_EQ: (A:form)(P:proc)(sat P A) ->
                          (P':proc)(struct_eq P P') -> (sat P' A).
```

The proof technique is a structural induction on `A`; lemmata `STRUCT_NOTIN` and `STRUCT_RES` are needed in the cases involving revelation (®) and revelation adjunct ($\oslash$), respectively.

### Reductions are preserved by Fresh Renaming

In this section we will show a similar result to that previously proved about Structural Congruence; more precisely we are going to formalize Lemma 4-4 of [CG00] extended with the case involving the restriction operator:

> Consider any process $P$ and names $m$, $m'$, with $m' \notin fn(P)$. For all $P'$, if $P \rightarrow P'$ then $m' \notin fn(P')$ and $P\{m \leftarrow m'\} \rightarrow P'\{m \leftarrow m'\}$. Moreover, for all $Q$, if $P\{m \leftarrow m'\} \rightarrow Q$ then there is a $P'$ with $P \rightarrow P'$, $m' \notin fn(P')$ and $Q = P'\{m \leftarrow m'\}$. $\square$

However, before proceeding to the main statement, we need to prove that substitution is also preserved by fresh renamings, since rule (Red Comm) involves substitution of capabilities for names. Indeed, we prove a stronger result, namely, substitution of capabilities for names is preserved by renamings (not necessarily fresh):

**Lemma 6.3** *For all capabilities $M$, $N$, $R$, processes $P$, $Q$ and names $m$, $n$, $n'$ (where $m \neq n, n'$),*

- *if $R = N\{m \leftarrow M\}$, then $R\{n \leftarrow n'\} = N\{n \leftarrow n'\}\{m \leftarrow M\{n \leftarrow n'\}\}$;*
- *if $Q = P\{m \leftarrow M\}$, then $Q\{n \leftarrow n'\} = P\{n \leftarrow n'\}\{m \leftarrow M\{n \leftarrow n'\}\}$.* □

The corresponding statements in Coq are the following:

```
Lemma SUBST_CAP_RW  : (M,R:name->cap)(N:name->name->cap)
                      (x:name)(notin_ho_cap x M) -> (notin_ho_cap x R) ->
                      (notin_ho2_cap x N) ->
                      (subst_cap (M x) (N x) (R x)) ->
                      (y:name)(subst_cap (M y) (N y) (R y)).

Lemma SUBST_PROC_RW : (M:name->cap)(P:name->name->proc)(Q:name->proc)
                      (x:name)(notin_ho_cap x M) ->
                      (notin_proc x (nu [_:name](nu (P _)))) ->
                      (notin_proc x (nu Q)) ->
                      (subst_proc (M x) (P x) (Q x)) ->
                      (y:name)(subst_proc (M y) (P y) (Q y)).
```

Similarly to the case of Structural Congruence, we formalize Lemma 4-4 in two steps: first of all we prove that if $P \to Q$ and $n \notin fn(P)$ then $n \notin fn(Q)$:

```
Lemma RED_NOTIN: (A,B:proc)(red A B) ->
                 (x:name)(notin_proc x A)->(notin_proc x B).
```

Then, we prove the main statement of this section:

```
Lemma RED_RW: (P,Q:name->proc)(x:name)
              (notin_proc x (nu P)) -> (notin_proc x (nu Q)) ->
              (red (P x) (Q x)) ->
              (y:name)(notin_proc y (nu P)) -> (notin_proc y (nu Q)) ->
              (red (P y) (Q y)).
```

The properties proved for `red` (namely `RED_NOTIN` and `RED_RW`) are easily extended to its reflexive and transitive closure `red_star`:

```
Lemma RED_STAR_NOTIN : (A,B:proc)(red_star A B) ->
                       (x:name)(notin_proc x A) -> (notin_proc x B).

Lemma RED_STAR_RW    : (P,Q:name->proc)(x:name)
                       (notin_proc x (nu P)) -> (notin_proc x (nu Q)) ->
                       (red_star (P x) (Q x)) ->
                       (y:name)
                       (notin_proc y (nu P)) -> (notin_proc y (nu Q)) ->
                       (red_star (P y) (Q y)).
```

### Nesting is preserved by Fresh Renaming

Continuing with our formalization, we will show that nesting ($\downarrow$) is preserved by fresh renaming in the sense of Lemma 4-5 of [CG00]:

> Consider any process $P$ and names $m$, $m'$, with $m' \notin fn(P)$. For all $P'$, if $P \downarrow P'$ then $m' \notin fn(P')$ and $P\{m \leftarrow m'\} \downarrow P'\{m \leftarrow m'\}$. Moreover, for all $Q$, if $P\{m \leftarrow m'\} \downarrow Q$ then there is a $P'$ with $P \downarrow P'$, $m' \notin fn(P')$ and $Q = P'\{m \leftarrow m'\}$. □

Like in the case of the previously proved results about Structural Congruence and the Reduction System, we first show that if $P \downarrow Q$ and $n \notin fn(P)$ then $n \notin fn(Q)$:

```
Lemma NEST_NOTIN: (A,B:proc)(nest A B) ->
                  (x:name)(notin_proc x A)->(notin_proc x B).
```

The main statement easily follows from `STRUCT_RW` since nesting is defined in terms of Structural Congruence:

```
Lemma NEST_RW: (P,Q:name->proc)(x:name)
               (notin_proc x (nu P)) -> (notin_proc x (nu Q)) ->
               (nest (P x) (Q x)) ->
               (y:name)(notin_proc y (nu P)) -> (notin_proc y (nu Q)) ->
               (nest (P y) (Q y)).
```

Like in the case of `red`, the previous two lemmata about `nest` are easily extendible to its reflexive and transitive closure `nest_star`:

```
Lemma NEST_STAR_NOTIN : (A,B:proc)(nest_star A B) ->
                        (x:name)(notin_proc x A) -> (notin_proc x B).

Lemma NEST_STAR_RW    : (P,Q:name->proc)(x:name)
                        (notin_proc x (nu P)) -> (notin_proc x (nu Q)) ->
                        (nest_star (P x) (Q x)) ->
                        (y:name)
                        (notin_proc y (nu P)) -> (notin_proc y (nu Q)) ->
                        (nest_star (P y) (Q y)).
```

### Satisfaction is preserved by Fresh Renaming

Almost all of the previous lemmata about fresh renamings are to be considered as preliminary results in order to prove a fundamental property of the Ambient Logic stating that fresh renaming preserves the satisfaction relation $\models$ (Lemma 2-3 of [CG01]):

> For all closed formulas $\mathcal{A}$, processes $P$, and names $m$, $m'$, if $m' \notin fn(P) \cup fn(\mathcal{A})$ then $P \models \mathcal{A} \Leftrightarrow P\{m \leftarrow m'\} \models \mathcal{A}\{m \leftarrow m'\}$. □

The corresponding statement in `Coq` is the following:

```
Lemma SAT_RW: (P:name->proc)(F:name->form)(x:name)
              (notin_proc x (nu P)) -> (notin_form x (forall F)) ->
              (sat (P x) (F x)) ->
              (y:name)(notin_proc y (nu P)) -> (notin_form y (forall F)) ->
              (sat (P y) (F y)).
```

Since the satisfaction relation is not representable by an inductive predicate (as we remarked in Section 6.2.9), we do not have a structural induction principle on `sat`. Hence, we mimick the proof "on paper" (which is carried out by induction on the number of symbols in the closed formula $\mathcal{A}$) proceeding by complete induction on the number of constructors occurring in a formula. More precisely, we start by introducing the following inductive relation:

```
Inductive ln: form -> nat -> Prop:=
  ln_T          : (ln T (S O))
| ln_neg        : (F:form)(n:nat)(ln F n) -> (ln (neg F) (S n))
| ln_Or         : (F,G:form)(n1,n2:nat)(ln F n1) -> (ln G n2) ->
                  (ln (Or F G) (S (plus n1 n2)))
| ln_zero       : (ln zero (S O))
| ln_comp       : (F,G:form)(n1,n2:nat)(ln F n1) -> (ln G n2) ->
                  (ln (comp F G) (S (plus n1 n2)))
| ln_comp_adj   : (F,G:form)(n1,n2:nat)(ln F n1) -> (ln G n2) ->
                  (ln (comp_adj F G) (S (plus n1 n2)))
| ln_loc        : (F:form)(x:name)(n:nat)(ln F n) -> (ln (loc x F) (S n))
| ln_loc_adj    : (F:form)(x:name)(n:nat)(ln F n) ->
                                          (ln (loc_adj F x) (S n))
| ln_rev        : (F:form)(x:name)(n:nat)(ln F n) -> (ln (rev x F) (S n))
| ln_rev_adj    : (F:form)(x:name)(n:nat)(ln F n) ->
                                          (ln (rev_adj F x) (S n))
| ln_sometime   : (F:form)(n:nat)(ln F n) -> (ln (sometime F) (S n))
| ln_somewhere  : (F:form)(n:nat)(ln F n) -> (ln (somewhere F) (S n))
| ln_forall     : (F:name->form)(n:nat)
                  ((x:name)(ln (F x) n)) -> (ln (forall F) (S n)).
```

Intuitively, `(ln A n)` holds if and only if `A` contains `n` occurrences of `form`-constructors.

Like `lnp` (the measure on processes we introduced in Section 6.2.10), `ln` is preserved by renaming (not necessarily fresh):

```
Lemma LNP_RW: (n:nat)(P:proc)(lnp P n) ->
              (x:name)(Q:name->proc)(notin_proc x (nu Q)) ->
              P=(Q x) -> (y:name)(lnp (Q y) n).
```

As for `LNP_RW` (see Section 6.2.10), the proof requires a complete induction on `n`. Then, we prove that for every term of type `form` there exists a natural number `n` such that `(ln A n)`:

```
Lemma LN_TOT: (A:form)(Ex [n:nat](ln A n)).
```

The proof is an easy structural induction on `A` (`LN_RW` is needed in the case related to the universal quantification).

Finally, we can carry out the proof of `SAT_RW` by means of a complete induction on the measure of `(F x)`, using when needed all the previously proved renaming lemmata about Structural Congruence, reductions, nesting etc.

## 6.3 Pragmatic remarks

We conclude this chapter by highlighting some pragmatic issues we found particularly interesting or problematic in dealing with metatheoretical proof developments about HOAS-based encodings.

### 6.3.1 Lifting structural information

All the renaming lemmata illustrated in the previous sections 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 subsumed by the following pattern:

$$\frac{x \notin fn(C_1[\cdot]) \cup \ldots \cup fn(C_n[\cdot]) \text{ and } \mathcal{P}(C_1[x], \ldots, C_n[x])}{\forall y \notin fn(C_1[\cdot]) \cup \ldots \cup fn(C_n[\cdot]).\mathcal{P}(C_1[y], \ldots, C_n[y])} \tag{6.1}$$

where $\mathcal{P}$ is a given $n$-ary relation (e.g., $\alpha$-equivalence, Structural Congruence, capture-avoiding substitution, reduction relation etc.) and $C_1[\cdot] \ldots, C_n[\cdot]$ are variables representing contexts over a given syntactic category. Usually, this kind of properties are proved "with pencil and paper" by carrying out a structural induction either on the derivation of the premise $\mathcal{P}(C_1[x], \ldots, C_n[x])$ 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) do not deal 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 $C_i[x]$ (throughout this chapter we have reported several examples, e.g., lemmata `PRE_HO_TM_IND` and `PRE_ALPHA_RW` in Sections 6.1.2 and 6.1.5 and lemma `PRE_NOTIN_PROC_MONO` in Section 6.2.10). In other words, we "lift" structural information to the level of functional terms; in order to achieve this goal, the axioms of $\beta$-expansion and extensionality turn out to be indispensable (indeed, this is the original motivation of their introduction in [HMS01b]). Hence, our original goal becomes the following:

$$\frac{x \notin fn(C_1[\cdot]) \cup \ldots \cup fn(C_n[\cdot]) \text{ and } T_1 = C_1[x], \ldots, T_n = C_n[x] \text{ and } \mathcal{P}(T_1, \ldots, T_n)}{\forall y \notin fn(C_1[\cdot]) \cup \ldots \cup fn(C_n[\cdot]).\mathcal{P}(C_1[y], \ldots, C_n[y])} \tag{6.2}$$

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 6.1 from 6.2 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 we can expand the latter into a context applied to $x$ yielding the equations $T_1 = T_1'[x], \ldots, T_n = T_n'[x]$ where $x \notin fn(T_1'[\cdot]) \cup \ldots \cup fn(T_n'[\cdot])$. Differently from $C_i[\cdot]$, $T_i'[\cdot]$ is not a variable, but a concrete context. Hence, by transitivity of equality we obtain the equations $C_i[x] = T_i'[x]$; whence, by means of 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 $\mathcal{P}$ and solve the subsequent subgoal by means of the inductive hypothesis.

### 6.3.2 Inversion issues

We have seen in Section 6.2.2 that the Structural Congruence relation identifies processes up to some (spatial) rearrangements, allowing to formalize a simpler reduction system. However, there is a price to pay for this simplification; indeed, the inversion tactic of Coq becomes practically useless when applied to a hypothesis of type `(struct_eq P Q)`. The reason is that `struct_eq` features two rules which do not enjoy the *subformula property* and are always applicable, namely, `struct_symm` and `struct_trans`. Hence, inverting a hypothesis of type `(struct_eq P Q)`, yields (among the other subgoals) a context where the original hypothesis has been replaced by `(struct_eq Q P)`. Thus, the complexity of the goal has not been reduced and inverting the new hypothesis arises the same problem leading us to switch *ad infinitum* back and forth between `(struct_eq P Q)` and `(struct_eq Q P)`. Luckily, so far we did not need to use inversion tactics on hypotheses about structural congruences of processes (only induction was in need).

### 6.3.3 Statistics

Differently from proofs with "pencil and paper", the activity of semiautomated proof development does not allow one to sweep anything under the rug. Thus, it is not possible to conclude a proof saying, e.g., "the remaining cases are proved by a similar argument" nor we can take for granted even elementary properties. This fact has the pleasant consequence of yielding proofs which are indeed sound and exhaustive, but on the other hand charges the user with a consistent burden. Obviously, the latter progressively increases with the complexity of the object language, since having more constructors means more cases to carry out in proofs by structural induction and/or inversion.

In order to give an idea of the increase of complexity when passing from the untyped $\lambda$-calculus to the Ambient Calculus, we end by giving in Table 6.1 some empirical data obtained from the two case studies illustrated in this chapter. All data refer to the following environment: Sun UltraSPARC IIe (64 bit) 440 MHz, 768 MB RAM (100 MHz), Coq V7.1 in native mode. As the number of proofs is concerned, we also included minor auxiliary lemmata which were necessary in order to prove the main results illustrated throughout the chapter.

| | **untyped $\lambda$-calculus** | **Ambients** |
|---:|:---:|:---:|
| Number of proofs | 64 | 56 |
| Size of source code | 70 KB | 172 KB |
| Broadest proof tree | 5 main subgoals | 22 main subgoals |
| Times of compilation | $\sim$ 1 min 13 sec | $\sim$ 7 min 24 sec |
| Maximum memory consumption | $\sim$ 38 MB | $\sim$ 188 MB |
| Size of `.vo` | 1075 KB | 6271 KB |

Table 6.1: Some statistics on the formal development so far accomplished.

# 7

# Conclusions

The results contained in this thesis are part of the achievements of an ongoing research program at the Computer Science Department of the University of Udine (started in 1992) on proof editors based on HOAS-encodings in dependent typed $\lambda$-calculi for formally reasoning about properties of program logics. So far, many case studies have been carried out: Structured and Natural Operational Semantics, Modal Logics, Dynamic Logics [Mic97], $\mu$-calculus [Mic97, Mic01b]. These experiences have yielded a deep knowledge on the subtleties and issues involved in the complex task of representing and reasoning about formal systems in type theory based logical frameworks.

One important achievement of the research carried out so far by our group is the so-called Theory of Contexts, which was originally conceived in [HMS01b] for metareasoning about a HOAS-encoding of the $\pi$-calculus. Its consistency, following an original idea of Hofmann, has been proved in Chapter 5 by means of a functorial model based on covariant presheaves. This rather complicated construction has been carried out in full detail, instead of resorting to "esoteric" categorical notions, in order to make it readable to readers unaware of the complicated notions of tripos theory. Indeed, we think that the technical machinery we have presented should be applicable also for reasoning about models with a similar structure. Hence, a reader wanting to carry out a similar construction will grasp more hints from a detailed construction than from an abstract, although interesting, discussion on the properties enjoyed by tripos structures.

The consistency of the Theory of Contexts yields an important consequence, i.e., it can be safely embedded in existing logical frameworks (as far as their logics do not entail the Axiom of Unique Choice) in order to provide a suitable environment for formally reasoning about HOAS-encodings of nominal calculi. For instance, this theory has been used fruitfully for developing the (meta)theory of several object languages in the proof assistant Coq [TCDT01]; see [HMS01b, Mic01a] for the case of $\pi$-calculus and $\lambda$-calculus, respectively. Moreover, in this thesis we presented two other complex case studies. The first one concerned the development of the metatheory of $\alpha$-equivalence for the untyped $\lambda$-calculus. Moreover, we provided the formal proof of the equivalence of three alternative formulations of $\alpha$-equivalence. Finally a HOAS-encoding of the Ambient Calculus and of the satisfaction relation of the related Modal Logic introduced in [CG01] has been presented. In the latter case a formal development of a set of fresh renaming properties lying at the heart of the metatheory of the Ambient Calculus has been formally proved.

We feel that one of the main advantages of our axiomatic approach, compared to other semantical solutions in the literature [FPT99, GP99], is that it requires a very low mathematical and logical overhead. We do not need to introduce a new abstraction and concretion operators as in [GP99], but we can continue to model abstraction with $\lambda$-abstraction and instantiation with functional application. Therefore our approach can be easily utilized in existing interactive proof assistants, e.g., Coq, without needing a redesign of the system.

As far as the problem of formally reasoning about nominal logics is concerned, we think that our approach is still more fruitful since the user is not forced to choose a specific framework, loosing a considerable amount of time in learning a new tool. Indeed, it can continue to use its preferred LF simply adding our axioms to its signatures. Indeed, the simplicity of our axiomatic approach comes from the very low mathematical and logical overhead required, since the existing machinery of the framework is exploited. However, as we pointed out in Section 4.4, we have a poor functional theory. Hence, if functional programming is the primary need, other frameworks (e.g., [Pit01a]) are to be considered.

**Future work.** At least three possible developments are stemming from this work.

1. A still open question is about the *completeness* of the Theory of Contexts. It is not clear which class of properties can be derived from our axioms; a suitable characterization is needed. The development of complex case studies may be of some help. In particular we plan to complete the formalization of the properties of the Ambient Logic introduced in [CG01] and further investigated in [CC01]. We are expecting some interesting insights from this work since Cardelli and Gordon explicitly use the Gabbay-Pitts' fresh-name quantifier ($\text{И}$) in order to talk about restricted names in a process. It would be interesting to see if the properties involving $\text{И}$ can be derived using the Theory of Contexts. Our confidence is motivated by the fact that our encoding methodology and axioms allow one to formulate and prove equivariant properties [Pit01b] in a very natural way, as it is witnessed by the formal development described in Section 6.2.11.

2. Another direction is to extend the model in order to handle more expressive metalanguages. For instance, one could take into account a theory of *dependent* and *impredicative types*. The expressive power of such a metalanguage would allow the representation and the manipulation of *proof objects*, via the usual "propositions-as-types" paradigm. An example of object theories which could be dealt with in this case are Natural Deduction-style proof systems; then, the well-known *Inversion Lemma* could be proved by induction over proof objects, using (a suitable extension of) the Theory of Contexts.

3. According to our experience, the activity of computer-assisted proof development is still at the beginning in the sense that the limits of current implementations are rather frustrating for the final user. For instance, in Coq there is no primitive support for HOAS-encodings and higher-order unifications produce some weird behaviours of many useful tactics, forcing the user to manually add the necessary equalities in the statements to prove (see e.g. all the PRE_lemmata of Chapter 6). Since HOAS and higher-order induction/recursion principles have been proved to be sound in many works, it is time to provide a better support for HOAS-encodings, e.g., automatic generation of higher-order induction/recursion principles for types of the form $\upsilon \to \iota$, where $\iota$ is inductively defined and $\upsilon$ is an open set (i.e., not inductive).

# A

# Deriving Higher-Order Induction Principles

This appendix contains the full `Coq` code formalizing the derivation of the higher-order induction principle for contexts of type `var->tm` by means of the complete induction principle on natural numbers, $Ind^{\iota}$ and the axioms of the Theory of Contexts.

```
(* We use the Coq library Omega to automatize the verification *)
(* of inequalitites on natural numbers.                          *)
Require Omega.

(* Encoding of syntax: variables and terms *)

Parameter var: Set.

Inductive tm : Set:=
    is_var: var -> tm
  | app: tm -> tm -> tm
  | lam: (var -> tm) -> tm.

(* Freshness predicate *)

Inductive notin [x:var]: tm -> Prop:=
    notin_var: (y:var)~x=y -> (notin x (is_var y))
  | notin_app: (M,N:tm)(notin x M) -> (notin x N) -> (notin x (app M N))
  | notin_lam: (M:var->tm)((y:var)~x=y -> (notin x (M y))) ->
               (notin x (lam M)).
```

In the following `Section` we derive the course of values induction principles on natural numbers (`NAT_IND`):

```
Section nat_ind_complete.

Lemma NAT_COMPLETE: (P:nat->Prop)
```

```
                    ((n:nat)((m:nat)(lt m n) -> (P m))->(P n)) ->
                    (a,b:nat)(lt b a) -> (P b).
Proof.
Do 2 Intro; Induction a; Intros;
[ Inversion_clear H0
| Inversion_clear H1;
  [ Apply H; Intros; Apply H0; Auto | Apply H0; Unfold lt; Assumption ] ].
Qed.

Lemma NAT_IND: (P:nat->Prop)
               (P O)->
               ((n:nat)((m:nat)(lt m n) -> (P m))->(P n))->
               (n:nat)(P n).
Proof.
Do 3 Intro; Induction n; Intros;
[ Assumption
| Apply H0; Intros; Inversion H2;
  [ Assumption | Apply NAT_COMPLETE with (S n0); Assumption ] ].
Qed.

End nat_ind_complete.
```

In the next `Section` we instantiate the Theory of Contexts for the encoding of untyped $\lambda$-calculus.

```
Section ToC.

(* Axiom stating the decidability of Leibniz's equality over names. *)

Axiom dec_var: (x,y:var)x=y \/ ~x=y.

(* Unsaturation *)

Axiom unsat: (M:tm)(Ex [x:var](notin x M)).

(* Expansion for plain terms and unary contexts *)

Axiom exp: (M:tm)(x:var)(Ex [N:var->tm](notin x (lam N)) /\ M=(N x)).

Axiom ho_exp: (M:var->tm)(x:var)
  (Ex [N:var->var->tm](notin x (lam [_:var](lam (N _)))) /\ M=(N x)).

(* Extensionality *)

Axiom ext: (F,G:var->tm)(x:var)
           (notin x (lam F)) -> (notin x (lam G)) ->
           (F x)=(G x) -> F=G.

End ToC.
```

Next we define the relation `l` counting the number of constructors in terms of type `tm`:

```
Inductive l: tm -> nat -> Prop:=
    l_var : (x:var)(l (is_var x) (S O))
  | l_app : (M,N:tm)(n1,n2:nat)(l M n1) -> (l N n2) ->
            (l (app M N) (S (plus n1 n2)))
  | l_lam : (M:var->tm)(n:nat)((y:var)(l (M y) n)) -> (l (lam M) (S n)).
```

The next auxiliary lemma allows to prune inconsistent cases during the proof development: it states that for every term `M` of type `tm` if `(l M n)` holds, then `n` must be greater than `O`:

```
Lemma L_S: (M:tm)(n:nat)(l M n)->(lt O n).
Proof.
Induction M; Intros;
[ Inversion_clear H; Unfold lt; Apply le_n
| Inversion_clear H1; Cut (lt O n1); [ Intro | Apply H; Assumption ];
  Cut (lt O n2); [ Intro | Apply H0; Assumption ]; Omega
| Inversion_clear H0; Elim (unsat (lam is_var)); Intros; Cut (lt O n0);
  [ Intro | Apply H with x; Auto ]; Unfold lt; Apply le_S; Assumption ].
Qed.
```

The net lemma plays a fundamental rôle in the rest of the development: it states that `l` is a relation invariant w.r.t. renamings:

```
Lemma L_RW: (n:nat)(M:tm)(l M n) ->
            (x:var)(N:var->tm)(notin x (lam N)) -> M=(N x) ->
            (y:var)(l (N y) n).
Proof.
Intro; Pattern n; Apply NAT_IND; Intros.
Cut (lt O (O)); [ Intro | Apply L_S with M; Assumption ].
Inversion_clear H2.

Inversion H0.
Elim (dec_var x x0); Intros.
Rewrite <- H5 in H3; Rewrite <- H3 in H2; Cut N=([_:var](is_var _));
[ Intro
| Apply ext with x; Try (Apply notin_lam; Intros; Apply notin_var);
  Auto ].
Rewrite H6; Apply l_var.

Rewrite <- H3 in H2; Cut N=([_:var](is_var x0));
[ Intro
| Apply ext with x; Try (Apply notin_lam; Intros; Apply notin_var);
  Auto ].
Rewrite H6; Apply l_var.

Elim (exp M0 x); Elim (exp N0 x); Intros.
Inversion_clear H7; Inversion_clear H8.
Rewrite H10 in H5; Rewrite H11 in H5; Rewrite <- H5 in H2.
Cut N=([_:var](app (x1 _) (x0 _)));
```

```
[ Intro
| Apply ext with x; Try (Apply notin_lam; Intros; Inversion_clear H7;
  Inversion_clear H9; Apply notin_app); Auto ].
Rewrite H8; Apply l_app;
[ Apply H with (x1 x) x;
  [ Rewrite <- H6; Omega
  | Rewrite <- H11; Assumption
  | Assumption
  | Trivial ]
| Apply H with (x0 x) x;
  [ Rewrite <- H6; Omega
  | Rewrite <- H10; Assumption
  | Assumption
  | Trivial ] ].

Elim (ho_exp M0 x); Intros.
Inversion_clear H6.
Rewrite H8 in H4; Rewrite <- H4 in H2.
Cut N=([_:var](lam (x0 _))); [ Intro | Apply ext with x; Auto ].
Rewrite H6; Apply l_lam; Intro.
Elim (unsat
       (app (lam [_:var](lam (x0 _)))
         (app (is_var x) (app (is_var y) (is_var y0))))); Intros.
Inversion_clear H9; Inversion_clear H11; Inversion_clear H9;
Inversion_clear H12; Inversion_clear H9; Inversion_clear H13.
Apply H with (x0 y x1) x1;
[ Rewrite <- H5; Unfold lt; Apply le_n
| Idtac
| Inversion_clear H10; Apply notin_lam; Intros;
  Cut (notin x1 (lam (x0 y))); [ Intro | Auto ]; Inversion_clear H14;
  Auto
| Trivial ].
Change (l ([_:var](x0 _ x1) y) n1) ; Apply H with (x0 x x1) x;
[ Rewrite <- H5; Unfold lt; Apply le_n
| Rewrite <- H8; Auto
| Inversion_clear H7; Apply notin_lam; Intros;
  Cut (notin x (lam (x0 y1))); [ Intro | Auto ]; Inversion_clear H14;
  Auto
| Trivial ].
Qed.
```

The relation `l` is also total in the sense that for every term `M` of type `tm` there exists a natural number `n` such that `(l M n)` holds:

```
Lemma L_TOT: (M:tm)(Ex [n:nat](l M n)).
Proof.
Induction M; Intros;
[ Split with (S O); Apply l_var
| Inversion_clear H; Inversion_clear H0; Split with (S (plus x x0));
```

```
  Apply l_app; Assumption
| Elim (unsat (lam t)); Intros; Elim (H x); Intros; Split with
  (S x0); Apply l_lam; Intro; Apply L_RW with (t x) x; Auto ].
Qed.
```

Finally, we are ready to prove the main result, i.e., the higher order induction principle over terms of functional type `var->tm`. This will be carried out in two steps: first we prove a preliminary lemma `PRE_HO_TM_IND` with all the necessary unifications added as premises in order to overcome the issues due to the inadequate treatment of higher-order unification in `Coq`. Then, lemma `HO_TM_IND` is proved as an immediate corollary of `PRE_HO_TM_IND`.

```
Lemma PRE_HO_TM_IND: (P:(var->tm)->Prop)
                     ((x:var)(P [_:var](is_var x))) ->
                     (P is_var) ->
                     ((M,N:var->tm)(P M) -> (P N) ->
                      (P [x:var](app (M x) (N x)))
                     ) ->
                     ((M:var->var->tm)((y:var)(P [x:var](M x y))) ->
                      (P [x:var](lam (M x)))
                     ) ->
                     (n:nat)(M:tm)(l M n) ->
                     (N:var->tm)(x:var)(notin x (lam N)) ->
                     (N x)=M -> (P N).
Proof.
Do 6 Intro; Pattern n; Apply NAT_IND; Intros.
Cut (lt O (O)); [ Intro | Apply L_S with M; Assumption ].
Inversion_clear H6.

Inversion H4.
Elim (dec_var x x0); Intros.
Rewrite <- H9 in H7; Rewrite <- H7 in H6; Cut N=is_var;
[ Intro
| Apply ext with x; Try (Apply notin_lam; Intros; Apply notin_var);
  Auto ].
Rewrite H10; Assumption.

Rewrite <- H7 in H6; Cut N=([_:var](is_var x0));
[ Intro
| Apply ext with x; Try (Apply notin_lam; Intros; Apply notin_var);
  Auto ].
Rewrite H10; Auto.

Elim (exp M0 x); Intros; Elim (exp N0 x); Intros.
Inversion_clear H11; Inversion_clear H12.
Rewrite H14 in H9; Rewrite H15 in H9; Rewrite <- H9 in H6.
Cut N=([x:var](app (x0 x) (x1 x)));
[ Intro
| Apply ext with x; Try (Apply notin_lam; Intros; Inversion_clear H11;
  Inversion_clear H13; Apply notin_app); Auto ].
```

```
Rewrite H12; Apply H1;
[ Apply H3 with n1 (x0 x) x;
  [ Rewrite <- H10; Omega
  | Rewrite <- H14; Assumption
  | Assumption
  | Trivial ]
| Apply H3 with n2 (x1 x) x;
  [ Rewrite <- H10; Omega
  | Rewrite <- H15; Assumption
  | Assumption
  | Trivial ] ].

Elim (ho_exp M0 x); Intros.
Inversion_clear H10.
Rewrite H12 in H8; Rewrite <- H8 in H6.
Cut N=([_:var](lam (x0 _))); [ Intro | Apply ext with x; Auto ].
Rewrite H10; Apply H2; Intro.
Elim (unsat (app (lam [_:var](lam (x0 _))) (app (is_var x) (is_var y))));
Intros.
Inversion_clear H13; Inversion_clear H15; Inversion_clear H13;
Inversion_clear H16.
Apply H3 with n1 (x0 x1 y) x1;
[ Rewrite <- H9; Unfold lt; Apply le_n
| Idtac
| Inversion_clear H14; Apply notin_lam; Intros;
  Cut (notin x1 (lam (x0 y0))); [ Intro | Auto ]; Inversion_clear H17;
  Auto
| Trivial ].
Elim (unsat
       (app (lam [_:var](lam (x0 _)))
         (app (is_var x) (app (is_var x1) (is_var y))))); Intros.
Inversion_clear H16; Inversion_clear H18; Inversion_clear H16;
Inversion_clear H19; Inversion_clear H16; Inversion_clear H20.
Apply L_RW with (x0 x1 x2) x2;
[ Idtac
| Inversion_clear H17; Apply notin_lam; Intros;
  Cut (notin x2 (lam (x0 x1))); [ Intro | Auto ]; Inversion_clear H21;
  Auto
| Trivial ].
Change (l ([_:var](x0 _ x2) x1) n1) ; Apply L_RW with (x0 x x2) x;
[ Rewrite <- H12; Auto
| Inversion_clear H11; Apply notin_lam; Intros;
  Cut (notin x (lam (x0 y0))); [ Intro | Auto ]; Inversion_clear H21;
  Auto
| Trivial ].
Qed.

Lemma HO_TM_IND: (P:(var->tm)->Prop)
                 ((x:var)(P [_:var](is_var x))) ->
```

```
                 (P is_var) ->
                 ((M,N:var->tm)(P M) -> (P N) ->
                  (P [x:var](app (M x) (N x)))
                 ) ->
                 ((M:var->var->tm)((y:var)(P [x:var](M x y))) ->
                  (P [x:var](lam (M x)))
                 ) ->
                 (M:var->tm)(P M).
Proof.
Intros; Elim (unsat (lam M)); Intros; Elim (L_TOT (M x)); Intros;
Apply PRE_HO_TM_IND with x0 (M x) x; Auto.
Qed.
```

# B

# Category-theoretical notions

In this section we recall some notions and results from category theory needed in order to understand the material in Chapter 5. Obviously, this is not intended to be a replacement for a good text on the subject (e.g. [Mac71, BW90] or, for the impatient reader, the first chapter of [Bel88]), but only a quick reference for non-categorically minded readers. Moreover, this section has the aim of fixing notation and giving more complete references to the involved topics.

Let us start with some basic notation: in the following we will write $X \in \mathcal{C}$ to mean that $X$ is an object of the category $\mathcal{C}$ and we will denote by $\mathcal{C}(X, Y)$ the family of arrows in $\mathcal{C}$ from $X$ to $Y$.

We will assume fixed a universe of sets, whose elements are called *small sets*. A category $\mathcal{C}$ is said *locally small* if for all $X, Y \in \mathcal{C}$, the family $\mathcal{C}(X, Y)$ is a small set, and *small* if, moreover, also the class of objects is a small set. In the following, we will refer to small sets simply as sets.

The following is a standard notion:

**Definition B.1** *A category $\mathcal{C}$ with terminal object and binary products is* cartesian closed *if for every $B$, $C \in \mathcal{C}$ there is an object $B \Rightarrow C$ and a morphism $ev_{C,B} : (B \Rightarrow C) \times B \longrightarrow C$ such that for each morphism $f : A \times B \longrightarrow C$ there is a unique morphism $\ulcorner f \urcorner : A \longrightarrow B \Rightarrow C$, the exponential transpose of $f$, such that the following diagram commutes:*

$$\begin{tikzcd} A \times B \arrow[d, "\ulcorner f \urcorner \times \mathrm{id}_B"'] \arrow[dr, "f"] & \\ (B \Rightarrow C) \times B \arrow[r, "ev_{C,B}"'] & C \end{tikzcd}$$

Next, we will present some basic results about functor categories, so it is useful a quick review on some standard notions.

A functor $F : \mathcal{C} \to \mathcal{D}$ is said to be *faithful* if, for all $A, B \in \mathcal{C}$, $F$ is injective on $\mathcal{C}(A, B)$, it is said to be *full* if for each $A$, $B \in \mathcal{C}$, $F$ carries $\mathcal{C}(A, B)$ onto $\mathcal{D}(F(A), F(B))$. Finally it is an *embedding* if it is injective on objects and faithful; moreover, $F$ preserves limits if it carries limit cones into limit cones. In this case, it will in particular preserve cartesian products.

In order to improve the readability of formulas and diagrams, we may denote the application of functors in three different ways: for instance, for $F : \mathcal{C} \longrightarrow \mathcal{D}$ and $A$ object of $\mathcal{C}$, the notations "$FA$", "$F(A)$" and "$F_A$" are equivalent.

Let $\mathcal{S}et$ be the category whose objects are sets and whose morphisms are functions between sets. Given a locally small category $\mathcal{C}$, we will denote by $\check{\mathcal{C}}$ the category $\mathcal{S}et^{\mathcal{C}}$ whose objects are the functors from $\mathcal{C}$ to $\mathcal{S}et$ and whose morphisms are natural transformations between them. More precisely:

- an object $A$ of $\check{\mathcal{C}}$ consists of a family of sets $\{A_X\}_{X\in\mathcal{C}}$, together with a family of functions $\{A_f\}_{f\in\mathcal{C}(X,Y),\ X,Y\in\mathcal{C}}$ such that $A_f : A_X \longrightarrow A_Y$, $A_{\mathrm{id}_X} = \mathrm{id}_{A_X}$ and $A_{f\circ g} = A_f \circ A_g$;
- a morphism $m \in \check{\mathcal{C}}(A, B)$ is a family of functions $\{m_X\}_{X\in\mathcal{C}}$, such that $m_X : A_X \longrightarrow B_X$ and for each $f : X \longrightarrow Y$, $m_Y \circ A_f = B_f \circ m_X$.

If $\mathcal{C}$ is small, it is known that the category $\check{\mathcal{C}}$ is cartesian closed with finite products given by

$$\begin{gathered}\mathbf{1}_X \triangleq \{\star\} \text{ and } \mathbf{1}_f \triangleq \mathrm{id}_{\{\star\}} \text{ (empty product)}\\ (A\times B)_X \triangleq A_X \times B_X \text{ and } (A\times B)_f \triangleq A_f \times B_f\,,\end{gathered}$$

moreover $(A \Rightarrow B)$ is given by

$$\begin{aligned}(A \Rightarrow B)_X &\triangleq \check{\mathcal{C}}(A \times \mathcal{C}(X, \_), B)\\ (A \Rightarrow B)_f(m) &\triangleq m \circ (\mathrm{id}_A \times (\_ \circ f)), \text{ for } f : Y \longrightarrow Z \text{ and } m \in \check{\mathcal{C}}(A \times \mathcal{C}(Y, \_), B)\end{aligned}$$

and finally $ev_{C,B}$ and $\ulcorner t \urcorner : A \to B \Rightarrow C$ are given by

$$\begin{aligned}(ev_{C,B})_X(\langle m, b\rangle) &\triangleq m_X(\langle b, \mathrm{id}_X\rangle), \text{ for all } X \in \mathcal{C}, b \in B_X, \text{ and } m \in (B \Rightarrow C)_X\\ (\ulcorner t \urcorner_X(a))_Y &: B_Y \times \mathcal{V}(X, Y) \longrightarrow C_Y\\ (\ulcorner t \urcorner_X(a))_Y(b, h) &\triangleq t_Y(\langle A_h(a), b\rangle)\end{aligned}$$

Let us consider the functor $\check{\mathcal{Y}} : \mathcal{C}^{op} \longrightarrow \check{\mathcal{C}}$, defined as follows:

- for $X \in \mathcal{C}$, $\check{\mathcal{Y}}(X) : \mathcal{C} \to \mathcal{S}et$ is the Homset functor $\mathcal{C}(X, \_)$, i.e.: $\check{\mathcal{Y}}(X)_Z \triangleq \mathcal{C}(X, Z)$ and, given $f : Y \longrightarrow Z$, for all $g \in \mathcal{C}(X, Y)$, $\check{\mathcal{Y}}(X)_f(g) \triangleq f \circ g$;
- for $f : X \to Y$, $\check{\mathcal{Y}}(f) : \check{\mathcal{Y}}(X) \to \check{\mathcal{Y}}(Y)$ is the natural transformation such that, for all $Z \in \mathcal{C}$ and $g \in \mathcal{C}(Y, Z)$, $(\check{\mathcal{Y}}(f))_X(g) \triangleq g \circ f$.

Then, the following fundamental lemma holds:

**Proposition B.1 (Yoneda Lemma)** *For each $A \in \check{\mathcal{C}}$ and $X \in \mathcal{C}$ there is a bijective correspondence between $\check{\mathcal{C}}(\check{\mathcal{Y}}(X), A)$ and $A_X$, moreover the correspondence is natural in $A$ and $X$.*

We give the definition of this bijective correspondence between $\check{\mathcal{C}}(\check{\mathcal{Y}}(X), A)$ and $A_X$: $\Phi_{X,A}(m) = m_X(\mathrm{id}_X)$, for $m \in \check{\mathcal{C}}(\check{\mathcal{Y}}(X), A)$; the inverse is the natural transformation defined on $a \in A_X$ by $(\Phi^{-1}_{X,A}(a))_Z(f) \triangleq A_f(a)$, for $f \in \check{\mathcal{Y}}(X)_Z$.

An immediate and important consequence of previous result is that the category $\mathcal{C}^{op}$ fully embeds in $\check{\mathcal{C}}$ by means of $\check{\mathcal{Y}}$, which is called, therefore, *Yoneda embedding*.

When an object in $\check{\mathcal{C}}$ is isomorphic to an object in the image of $\check{\mathcal{Y}}$ it is said to be *representable*. Notice, for example, that, if $\mathcal{C}$ has an initial object $\mathbf{0}$, then the terminal object $\mathbf{1}$ is representable since $\mathbf{1} \cong \check{\mathcal{Y}}(\mathbf{0})$.

Another useful notion to recall is the concept of adjunction; for our purposes the following definition suffices.

**Definition B.2** *Given categories $\mathcal{C}$ and $\mathcal{D}$, an* adjunction *from $C$ to $D$ is a triple $(F, G, \phi)$, where $F, G$ are functors, $F : \mathcal{C} \longrightarrow \mathcal{D}$, $G : \mathcal{D} \longrightarrow \mathcal{C}$ and $\phi$ is a function which maps every $A \in \mathcal{C}$ and $B \in \mathcal{D}$ to a bijection $\phi_{A,B} : \mathcal{C}(A, G_B) \cong \mathcal{D}(F_A, B)$, natural in $A$ and $B$, i.e., the following hold:*

- *$\phi_{AB}(h \circ f) = \phi_{A'B}(h) \circ F(f)$ for every $A' \in \mathcal{C}$, $f : A \longrightarrow A'$ and $h : A' \longrightarrow G(B)$;*
- *$\phi_{AB}(G(g) \circ h) = g \circ \phi_{AB'}(h)$ for every $B' \in \mathcal{D}$, $g : B' \longrightarrow B$ and $h : A \longrightarrow G(B')$.*

$F$ and $G$ are respectively called the left and the right adjoint of the adjunction and this is denoted by $F \dashv G$ or $G \vdash F$.

We will use the known property that a functor $F : \mathcal{C} \longrightarrow \mathcal{D}$ with a right (left) adjoint preserves colimits (limits). For the proof see, e.g., [Mac71].

Now we introduce some notions and a result about algebras of functors.

**Definition B.3** *Given a functor $T : \mathcal{C} \longrightarrow \mathcal{C}$, a $T$-*algebra *is a pair $\langle A, \alpha \rangle$, with $A \in \mathcal{C}$ and $\phi : TA \longrightarrow A$ morphism of $\mathcal{C}$. A $T$-algebra morphism from $\langle A, \alpha \rangle$ to $\langle B, \beta \rangle$ is an arrow $f \in C(A, B)$ such that the following diagram commutes:*

$$\begin{tikzcd} TA \arrow[r, "Tf"] \arrow[d, "\alpha"'] & TB \arrow[d, "\beta"] \\ A \arrow[r, "f"] & B \end{tikzcd}$$

$T$-algebras and $T$-algebra morphisms form a category, whose initial object, if it exists, is said an initial $T$-algebra.

**Theorem B.1 ([Hof99])** *Let $\mathcal{C}, \mathcal{D}$ be two categories and $F : \mathcal{C} \longrightarrow \mathcal{D}$ be a functor with a right adjoint $F^*$. Let $T : \mathcal{C} \longrightarrow \mathcal{C}$ and $T' : \mathcal{D} \longrightarrow \mathcal{D}$ be two functors such that $T' \circ F \cong F \circ T$ for some natural isomorphism $\phi : T' \circ F \longrightarrow F \circ T$. If $(A, a : TA \to A)$ is an initial $T$-algebra in $\mathcal{C}$, then $(F_A, F_a \circ \phi_A : T'(F_A) \to F_A)$ is an initial $T'$-algebra in $\mathcal{D}$.*

*Proof.* The adjoint pair $F \dashv F^*$ can be lifted to a pair of adjoint functors between the categories of $T$- and $T'$- algebras. Since any functor with a right adjoint preserve colimits and the initial object is a colimit, then the initial object of the former category is preserved in the latter. □

Another useful technique for building initial algebras is based on the notions of *simple slice category* and *strong functor*. We recall here the basic definitions and related properties from [Jac95].

**Definition B.4** *Given a category $\mathcal{C}$ with binary products and $G \in \mathcal{C}$, the* simple slice category *$\mathcal{C}/\!\!/G$ is defined as follows:*

1. *$Obj(\mathcal{C}/\!\!/G) \triangleq Obj(\mathcal{C})$,*
2. *$\mathcal{C}/\!\!/G(A, B) \triangleq \mathcal{C}(G \times A, B)$,*

3. *the identity map on $A$ in $\mathcal{C}/\!\!/G$ is the second projection $\pi' : G \times A \longrightarrow A$ in $\mathcal{C}$,*

4. *the composition of $f : A \longrightarrow B$ and $g : B \longrightarrow C$ is defined as follows:*

$$g \bullet f \triangleq g \circ \langle \pi, f \rangle : G \times A \longrightarrow G \times B \longrightarrow C,$$

*where $\bullet$ denotes the composition in $\mathcal{C}/\!\!/G$ and $\circ$ the composition in $\mathcal{C}$.*

Given $G \in \mathcal{C}$, there is a functor $G^* : \mathcal{C} \longrightarrow \mathcal{C}/\!\!/G$ defined as follows:

1. $G^*(A) \triangleq A$ for every $A \in \mathcal{C}$,
2. $G^*(f) \triangleq f \circ \pi'$ for every $f \in \mathcal{C}(A, B)$.

**Definition B.5** *An endofunctor $T : \mathcal{C} \longrightarrow \mathcal{C}$ on a category $\mathcal{C}$ with finite products is called* strong *if it comes equipped with a natural transformation, called* strength, *with components $st_{A,B} : A \times TB \longrightarrow TA \times B$ making the following two diagrams commute:*

$$\begin{tikzcd}
A \times TB \arrow[r, "st"] \arrow[dr, "\pi'"'] & TA \times B \arrow[d, "T\pi'"] \\
& TB
\end{tikzcd}
\qquad
\begin{tikzcd}
A \times (C \times TB) \arrow[r, "id \times st"] \arrow[d, "\beta"'] & A \times TC \times B \arrow[r, "st"] & TA \times (C \times B) \arrow[d, "T\beta"] \\
(A \times C) \times TB \arrow[rr, "st"] & & T(A \times C) \times B
\end{tikzcd}$$

*where $\beta$ is the obvious isomorphism $\langle\langle \pi, \pi \circ \pi' \rangle, \pi' \circ \pi' \rangle$.*

As proved in [Jac95], if $T$ is a strong functor, we can define, for every $A \in \mathcal{C}$, a functor $T/\!\!/A : \mathcal{C}/\!\!/A \longrightarrow \mathcal{C}/\!\!/A$ as follows:

- $(T/\!\!/A)_B \triangleq TB$,
- $(T/\!\!/A)_f \triangleq Tf \circ st_{A,B}$ (for every $f \in \mathcal{C}/\!\!/A(B, C)$).

It turns out that also this new functor is strong.

Finally we give a result proved in [Hof99], fundamental to the construction of the model carried out in Chapter 5. Let $\mathcal{C}$ have finite coproducts and let $\uplus$ be a choice for them; for all $A \in \check{\mathcal{C}}$, $X \in \mathcal{C}$, let $A^X$ denote the functor defined by $A^X_Y \triangleq A_{X \uplus Y}$ and $A^X_f \triangleq A_{\langle \mathrm{id}_X, f \rangle}$ (it is easy to verify that this indeed defines a functor). Then one has

**Proposition B.2** $\check{\mathcal{Y}}(X) \Rightarrow A \cong A^X$.

*Proof.*
$$\begin{array}{lll}
(\check{\mathcal{Y}}(X) \Rightarrow A)_Y & = \check{\mathcal{C}}(\check{\mathcal{Y}}(X) \times \check{\mathcal{Y}}(Y), A) & \text{by definition of } \Rightarrow \\
& \cong \check{\mathcal{C}}(\check{\mathcal{Y}}(X \uplus Y), A) & \text{since } \check{\mathcal{Y}} \text{ preserves products} \\
& \cong A_{X \uplus Y} & \text{by Yoneda Lemma} \\
& \cong A^X_Y & \text{by definition of } A^X.
\end{array}$$
$\square$

# C

# Longer proofs

### C.0.4 Proof of Proposition 5.2

First of all, we prove that for all $F \in \check{\mathcal{V}}$, $G \in \check{\mathcal{I}}$, $\phi_{FG}$ is a bijection. We introduce the inverse $\psi_{FG}$ of $\phi_{FG}$ as the function such that for $F \in \check{\mathcal{V}}$, $G \in \check{\mathcal{I}}$, $\beta \in \check{\mathcal{I}}(F^r, G)$, $X, Y \in \mathcal{V}$, $x \in F_X$ and $g \in \mathcal{V}(X, Y))$ is defined by:

$$((\psi_{FG}(\beta))_X(x))_Y(g) \triangleq \beta_Y(F_g(x))\,.$$

Now, we will prove that $(\psi_{FG} \circ \phi_{FG})(\alpha) = \alpha$ for every $\alpha : F \longrightarrow G^*$:

$$\begin{array}{rcl}(\psi_{FG} \circ \phi_{FG})(\alpha) & = & \psi_{FG}(\phi_{FG}(\alpha)) \\ & = & \psi_{FG}(\{(\phi_{FG}(\alpha))_X : x \mapsto (\alpha_X(x))_X(id_X)\}_{X \in \mathcal{V}})\end{array}$$

At this point we can verify that the natural transformation we have obtained is equal to $\alpha$; indeed, for every $X, Y \in \mathcal{X}$, $x \in F_X$, and $g \in \mathcal{V}(X, Y)$ we have:

$$\begin{array}{rcl}(\alpha_X(x))_Y(g) & = & ((\psi_{FG}(\{(\phi_{FG}(\alpha))_X : x \mapsto (\alpha_X(x))_X(id_X)\}_{X \in \mathcal{V}}))_X(x))_Y(g) \\ & = & (\{(\phi_{FG}(\alpha))_X : x \mapsto (\alpha_X(x))_X(id_X)\}_{X \in \mathcal{V}})_Y(F_g(x)) \\ & = & (\alpha_Y(F_g(x)))_Y(id_Y) \\ & = & (\alpha_X(x))_Y(g) \text{ (by naturality of } \alpha)\end{array}$$

Similarly, we can prove that $(\phi_{FG} \circ \psi_{FG})(\beta) = \beta$ for every $\beta : F^r \longrightarrow G$:

$$\begin{array}{rcl}(\phi_{FG} \circ \psi_{FG})(\beta) & = & \phi_{FG}(\psi_{FG}(\beta)) \\ & = & \phi_{FG}(\{(\psi_{FG}(\beta))_X : x \mapsto \gamma\}_{X \in \mathcal{V}})\end{array}$$

where $\gamma = \{((\psi_{FG}(\beta))_X(x))_Y : g \mapsto \beta_Y(F_g(x))\}_{Y \in \mathcal{V}}$. At this point we can verify that the natural transformation we have obtained is equal to $\beta$; indeed, for every $X \in \mathcal{X}$, $x \in F_X$ we have:

$$\begin{array}{rcl}\beta_X(x) & = & (\phi_{FG}(\{(\psi_{FG}(\beta))_X : x \mapsto \gamma\}_{X \in \mathcal{V}}))_X(x) \\ & = & (\gamma)_X(id_X) \\ & = & \beta_X(F_{id_X}(x)) \\ & = & \beta_X(id_{F_X}(x)) \\ & = & \beta_X(x)\end{array}$$

Looking at Definition B.2, it remains to prove that for all $F \in \check{\mathcal{V}}$, $G \in \check{\mathcal{I}}$, $\phi_{FG}$ the following hold:

1. $\phi_{FG}(h \circ f) = \phi_{F'G}(h) \circ f^r$ for every $F' \in \check{\mathcal{V}}$, $f : F \longrightarrow F'$ and $h : F' \longrightarrow B^*$;

2. $\phi_{FG}(g^* \circ h) = g \circ \phi_{FG'}(h)$ for every $G' \in \check{\mathcal{I}}$, $g : G' \longrightarrow G$ and $h : F \longrightarrow G'^*$.

So, let $F' \in \check{\mathcal{V}}$, $f : F \longrightarrow F'$ and $h : F' \longrightarrow G^*$, then we have (by definition of $\phi_{FG}$) that, for all $X \in \mathcal{V}$ and $x \in F_X$, $(\phi_{FG}(h \circ f))_X(x) = ((h \circ f)_X(x))_X(id_X) = (h_X(f_X(x)))_X(id_X)$. On the other hand, $(\phi_{F'G}(h) \circ f^r)_X(x) = (\phi_{F'G}(h))_X(f_X(x))$, but the last member of the equation is equal to $(h_X(f_X(x)))_X(id_X)$ by definition of $\phi_{F'G}$. Hence, we proved the first point, i.e., the commutativity of the following diagram:

$$\begin{array}{ccc}
\check{\mathcal{V}}(F', G^*) & \xrightarrow{\phi_{F'G}} & \check{\mathcal{I}}(F'^r, G) \\
{\scriptstyle \check{\mathcal{V}}(f, G^*)}\downarrow & & \downarrow{\scriptstyle \check{\mathcal{I}}(f^r, G)} \\
\check{\mathcal{V}}(F, G^*) & \xrightarrow{\phi_{FG}} & \check{\mathcal{I}}(F^r, G)
\end{array}$$

Now, let $G' \in \check{\mathcal{I}}$, $g : G' \longrightarrow G$ and $h : F \longrightarrow G'^*$, then we have (by definition of $\phi_{FG}$) that, for all $X \in \mathcal{V}$ and $x \in F_X$, $(\phi_{FG}(g^* \circ h))_X(x) = ((g^* \circ h)_X(x))_X(id_X) = (g^*_X(h_X(x)))_X(id_X)$. By the definition of the action of $(\_)^*$ on morphisms, we have that the last member of the equation is equal to $g_X((h_X(x))_X(id_X))$. On the other hand, $(g \circ \phi_{FG'}(h))_X(x) = g_X((\phi_{FG'}(h))_X(x)) = g_X((h_X(x))_X(id_X))$ (the last equation is obtained by definition of $\phi_{FG'}$). Hence, also the second point is proved; this amounts to the commutativity of the following diagram:

$$\begin{array}{ccc}
\check{\mathcal{V}}(F, G'^*) & \xrightarrow{\phi_{FG'}} & \check{\mathcal{I}}(F^r, G') \\
{\scriptstyle \check{\mathcal{V}}(F, g)}\downarrow & & \downarrow{\scriptstyle \check{\mathcal{I}}(F^r, g)} \\
\check{\mathcal{V}}(F, G^*) & \xrightarrow{\phi_{FG}} & \check{\mathcal{I}}(F^r, G)
\end{array}$$

### C.0.5 Proof of Proposition 5.3

For $U, V \in \mathbf{Pred}_{\check{\mathcal{I}}}(F)$, we put

$$\begin{array}{rcl}
(U \vee V)_X & \triangleq & U_X \cup V_X \\
(U \wedge V)_X & \triangleq & U_X \cap V_X \\
(\overline{U})_X & \triangleq & \{f \in F_X \mid f \notin U_X\} \\
0_X & \triangleq & \emptyset \\
1_X & \triangleq & F_X.
\end{array}$$

Now we prove that the objects defined above are indeed predicates:

$(U \vee V) \in \mathbf{Pred}(F)$:

1. since, by hypothesis, $U, V \in \mathbf{Pred}(F)$, it follows that $U_X \subseteq F_X$ and $V_X \subseteq F_X$ for $X \in \mathcal{I}$; then $(U \vee V)_X \triangleq U_X \cup V_X \subseteq F_X$;

2. given $h \in \mathcal{I}(X, Y)$ and $f \in (U \vee V)_X$, we can infer that either $f \in U_X$ or $f \in V_X$ (since $(U \vee V)_X \triangleq U_X \cup V_X$); in the former case we have that $F_h(f) \in U_Y$ by hypothesis, hence $F_h(f) \in U_Y \cup V_Y \triangleq (U \vee V)_Y$ (in the latter case we can conclude by a similar argument);

3. given $f \in F_X$ and $F_h(f) \in (U \vee V)_Y$ for some $h \in \mathcal{I}(X,Y)$, we can infer that either $F_h(f) \in U_Y$ or $F_h(f) \in V_Y$ (since $(U \vee V)_Y \triangleq U_Y \cup V_Y$); in the former case we can conclude that $f \in U_X$, hence $f \in U_X \cup V_X \triangleq (U \vee V)_X$ (in the latter case we can conclude by a similar argument).

$(U \wedge V) \in \mathbf{Pred}(F)$:

1. since, by hypothesis, $U, V \in \mathbf{Pred}(F)$, it follows that $U_X \subseteq F_X$ and $V_X \subseteq F_X$ for $X \in \mathcal{I}$; then $(U \wedge V)_X \triangleq U_X \cap V_X \subseteq F_X$;

2. given $h \in \mathcal{I}(X,Y)$ and $f \in (U \wedge V)_X$, we can infer that $f \in U_X$ and $f \in V_X$ (since $(U \wedge V)_X \triangleq U_X \cap V_X$); then, by hypothesis, $F_h(f) \in U_Y$ and $F_h(f) \in V_Y$, hence we can conclude that $F_h(f) \in (U_Y \cap V_Y) \triangleq (U \wedge V)_Y$;

3. given $f \in F_X$ and $F_h(f) \in (U \wedge V)_Y$ for some $h \in \mathcal{I}(X,Y)$, we can infer that $F_h(f) \in U_Y$ and $F_h(f) \in V_Y$ (since $(U \wedge V)_Y \triangleq U_Y \cap V_Y$); then, by hypothesis, we have that $f \in U_X$ and $f \in V_X$, hence we can conclude $f \in U_X \cap V_X \triangleq (U \wedge V)_X$.

$\overline{U} \in \mathbf{Pred}(F)$:

1. the first condition trivially holds by definition of $(\overline{U})_X$;

2. given $h \in \mathcal{I}(X,Y)$ and $f \in (\overline{U})_X$, by definition of $\overline{U}$ we have that $f \in F_X$ and $f \notin U_X$; then, exploiting the fact that $U \in \mathbf{Pred}(F)$ (precisely we use condition 5.3), we can conclude that $F_h(f) \notin U_Y$, hence $F_h(f) \in (\overline{U})_Y$;

3. given $f \in F_X$ and $F_h(f) \in (\overline{U})_Y$ for some $h \in \mathcal{I}(X,Y)$, we can infer that $F_h(f) \in F_Y$ and $F_h(f) \notin U_Y$ (by definition of $\overline{U}$); then, exploiting the fact that $U \in \mathbf{Pred}(F)$ (precisely we use condition 5.2), we can conclude that $f \notin U_X$, hence $f \in (\overline{U})_X$.

$0 \in \mathbf{Pred}(F)$:

1. we trivially have $0_X \triangleq \emptyset \subseteq F_X$ for $X \in \mathcal{I}$;

2. this condition trivially holds since the premise $f \in 0_X \triangleq \emptyset$ is false;

3. similarly to the previous case this condition is also trivially verified, since the premise $F_h(f) \in 0_Y \triangleq \emptyset$ cannot be fulfilled.

$1 \in \mathbf{Pred}(F)$:

1. we trivially have $1_X \triangleq F_X \subseteq F_X$ for $X \in \mathcal{I}$;

2. given $h \in \mathcal{I}(X,Y)$ and $f \in 1_X \triangleq F_X$, we trivially have $F_h(f) \in F_Y$ by functoriality of $F$, hence we can immediately conclude since $1_Y \triangleq F_Y$;

3. given $f \in F_X$ and $F_h(f) \in 1_Y \triangleq F_Y$ for some $h \in \mathcal{I}(X,Y)$, we have by hypothesis that $f \in F_X$, hence we can immediately conclude since $1_X \triangleq F_X$.

One can easily check that $\mathbf{Pred}(F)$ endowed with these operations can indeed be turned into a complemented distributive lattice.

### C.0.6 Proof of Proposition 5.4

Indeed, given $\eta{:}F \longrightarrow G$ and $U \in \mathbf{Pred}_{\check{\mathcal{I}}}(G)$ and $X \in \mathcal{I}$ , we have that $(\mathbf{Pred}_{\check{\mathcal{I}}}(\eta)(U))_X \triangleq \eta_X^{-1}(U_X)$, hence

$$\chi_F^{\check{\mathcal{I}}}(\mathbf{Pred}_{\check{\mathcal{I}}}(\eta)(U))_X \triangleq \lambda t \in F_X.\{f : X \longrightarrow B \mid F_f(t) \in (\mathbf{Pred}_{\check{\mathcal{I}}}(\eta)(U))_B\}_{B\in\mathcal{I}}.$$

On the other hand, we have $(\chi_G^{\check{\mathcal{I}}}(U))_X \triangleq \lambda t \in G_X.\{f : X \longrightarrow B \mid G_f(t) \in U_B\}_{B\in\mathcal{I}}$, hence

$$\begin{array}{rcl}(\check{\mathcal{I}}(\eta,\Omega)(\chi_G^{\check{\mathcal{I}}}(U)))_X & = & (\chi_G^{\check{\mathcal{I}}}(U)\circ\eta)_X\\ & = & (\chi_G^{\check{\mathcal{I}}}(U))_X \circ \eta_X\\ & \triangleq & \lambda t \in F_X.\{f : X \longrightarrow B \mid G_f(\eta_X(t)) \in U_B\}_{B\in\mathcal{I}},\end{array}$$

but, by naturality of $\eta$, we have that $G_f(\eta_X(t)) = \eta_B(F_f(t))$, hence $G_f(\eta_X(t)) \in U_B$ if and only if $F_f(t) \in \eta_B^{-1}(U_B) \triangleq (\mathbf{Pred}_{\check{\mathcal{I}}}(\eta)(U))_B$, i.e.,

$$\chi_F^{\check{\mathcal{I}}}(\mathbf{Pred}_{\check{\mathcal{I}}}(\eta)(U))_X = (\check{\mathcal{I}}(\eta,\Omega)(\chi_G^{\check{\mathcal{I}}}(U)))_X.$$

Thus, naturality of $\chi^{\check{\mathcal{I}}}$ is proved. Now, it remains to show that $\chi^{\check{\mathcal{I}}}$ is a natural isomorphism, i.e., that $\chi_F^{\check{\mathcal{I}}}$ has an inverse for each $F \in \check{\mathcal{V}}$. We will prove that this inverse indeed is $\kappa_F^{\check{\mathcal{I}}}$.

First let us verify that $\kappa_F^{\check{\mathcal{I}}}(\chi_F^{\check{\mathcal{I}}}(V)) = V$ for $V \in \mathbf{Pred}_{\check{\mathcal{I}}}(F)$ (i.e. $\kappa_F^{\check{\mathcal{I}}} \circ \chi_F^{\check{\mathcal{I}}} = \mathrm{id}_{\mathbf{Pred}_{\check{\mathcal{I}}}(F)}$):

$$\begin{array}{rcl}\kappa_F^{\check{\mathcal{I}}}(\chi_F^{\check{\mathcal{I}}}(V)) & \triangleq & \{\{f \in F_X \mid (\chi_F^{\check{\mathcal{I}}}(V))_X(f) = \check{\mathcal{Y}}_{\check{\mathcal{I}}}(X)\}\}_{X\in\mathcal{I}}\\ & \triangleq & \{\{f \in F_X \mid \{g : X \longrightarrow B \mid F_g(f) \in V_B\}_{B\in\mathcal{I}} = \mathcal{I}(X,\_)\}\}_{X\in\mathcal{I}}\\ & = & \{V_X\}_{X\in\mathcal{I}} \text{ (because of property 5.2 of predicates)}\\ & \triangleq & V\end{array}$$

Now we have to prove that $\chi_F^{\check{\mathcal{I}}}(\kappa_F^{\check{\mathcal{I}}}(m)) = m$ (i.e. $\chi_F^{\check{\mathcal{I}}} \circ \kappa_F^{\check{\mathcal{I}}} = \mathrm{id}_{\check{\mathcal{V}}(F,\Omega)}$):

$$\chi_F^{\check{\mathcal{I}}}(\kappa_F^{\check{\mathcal{I}}}(m)) \;\triangleq\; \{\lambda t \in F_X.\{f : X \longrightarrow B \mid F_f(t) \in (\kappa_F^{\check{\mathcal{I}}}(m))_B\}_{B\in\mathcal{I}}\}_{X\in\mathcal{I}}$$

but, by definition of $(\kappa_F^{\check{\mathcal{I}}}(m))_B$, we have that $F_f(t) \in (\kappa_F^{\check{\mathcal{I}}}(m))_B$ if and only if $F_f(t) \in F_B$ and $m_B(F_f(t)) = \mathcal{I}(B,\_)$, but, by naturality of $m$, it follows that $m_B(F_f(t)) = \Omega_f(m_X(t)) \triangleq \mathbf{Pred}_{\check{\mathcal{I}}}(\check{\mathcal{Y}}_{\check{\mathcal{I}}}(f))(m_X(t))$. Then

$$(\mathbf{Pred}_{\check{\mathcal{I}}}(\check{\mathcal{Y}}_{\check{\mathcal{I}}}(f))(m_X(t)))_B \triangleq (\check{\mathcal{Y}}_{\check{\mathcal{I}}}(f))_B^{-1}((m_X(t))_B) = \mathcal{I}(B,B),$$

i.e., for all $g \in \mathcal{I}(B,B)$, $(\check{\mathcal{Y}}_{\check{\mathcal{I}}}(f))_B(g) \in (m_X(t))_B$ holds. Since $(\check{\mathcal{Y}}_{\check{\mathcal{I}}}(f))_B(g) = g \circ f \triangleq \mathcal{I}(X,g)(f)$, we have, by properties 5.2 and 5.3 of predicates (remember that $m_X(t) \in \mathbf{Pred}_{\check{\mathcal{I}}}(\mathcal{I}(X,\_))$), that $m_B(F_f(t)) = \mathcal{I}(B,\_)$ if and only if $f \in (m_X(t))_B$ holds. Hence, we may conclude that $\chi_F^{\check{\mathcal{I}}}(\kappa_F^{\check{\mathcal{I}}}(m)) = \{\lambda t \in F_X.\{f : X \longrightarrow B \mid f \in (m_X(t))_B\}_{B\in\mathcal{I}}\}_{X\in\mathcal{I}}$, i.e., $\chi_F^{\check{\mathcal{I}}}(\kappa_F^{\check{\mathcal{I}}}(m)) = m$.

### C.0.7 Proof of Theorem 5.1

1. We have the following:

$$
\begin{array}{rcl}
[\![\Gamma \vdash_\Sigma \forall x{:}\sigma.p : o]\!]_X(\eta) & = & (forall_\sigma)_X([\![\Gamma \vdash_\Sigma \lambda x{:}\sigma.p : \sigma \to o]\!]_X(\eta)) \\
& = & \{u : X \longrightarrow Y \mid \forall g \in \mathcal{I}(Y,Z).\forall t \in [\![\sigma]\!]_Z. \\
& & \langle g \circ u, t\rangle \in \kappa_{[\![\Gamma]\!]}([\![\Gamma \vdash_\Sigma \lambda x{:}\sigma.p : \sigma \to o]\!]_X(\eta))_Z\}_{Y \in \mathcal{I}} \\
& = & \{u : X \longrightarrow Y \mid \forall g \in \mathcal{I}(Y,Z).\forall t \in [\![\sigma]\!]_Z. \\
& & ([\![\Gamma \vdash_\Sigma \lambda x{:}\sigma.p : \sigma \to o]\!]_X(\eta))_Z(\langle g \circ u, t\rangle) \geq \\
& & \geq \mathcal{I}(Z,\_)\}_{Y \in \mathcal{I}} \\
& = & \{u : X \longrightarrow Y \mid \forall g \in \mathcal{I}(Y,Z).\forall t \in [\![\sigma]\!]_Z. \\
& & (\lambda\langle f,b\rangle \in \mathcal{I}(X,Z) \times [\![\sigma]\!]_Z. \\
& & [\![\Gamma, x:\sigma \vdash_\Sigma p : o]\!]_Z(\langle [\![\Gamma]\!]_f(\eta), b\rangle))(\langle g \circ u, t\rangle) \\
& & \geq \mathcal{I}(Z,\_)\}_{Y \in \mathcal{I}} \\
& = & \{u : X \longrightarrow Y \mid \forall g \in \mathcal{I}(Y,Z).\forall t \in [\![\sigma]\!]_Z. \\
& & [\![\Gamma, x:\sigma \vdash_\Sigma p : o]\!]_Z(\langle [\![\Gamma]\!]_{(g \circ u)}(\eta), t\rangle) \\
& & \geq \mathcal{I}(Z,\_)\}_{Y \in \mathcal{I}}
\end{array}
$$

($\Rightarrow$) By hypothesis we have that $X \Vdash_{\Gamma,\eta} \forall x{:}\sigma.p$, i.e., $\eta \in \kappa_{[\![\Gamma]\!]}([\![\Gamma \vdash_\Sigma \forall x{:}\sigma.p : o]\!])_X$ which, in turn, is equivalent to say that $[\![\Gamma \vdash_\Sigma \forall x{:}\sigma.p : o]\!]_X(\eta) \geq \mathcal{I}(X,\_)$ holds. In particular we have that $h : \mathcal{I}(X,Y)$ belongs to $([\![\Gamma \vdash_\Sigma \forall x{:}\sigma.p : o]\!]_X(\eta))_Y$. Then, taking $g = \mathrm{id}_Y$ and $t = a$, we have that $[\![\Gamma, x : \sigma \vdash_\Sigma p : o]\!]_Y(\langle [\![\Gamma]\!]_{(\mathrm{id}_Y \circ h)}(\eta), a\rangle) = [\![\Gamma, x : \sigma \vdash_\Sigma p : o]\!]_Y(\langle [\![\Gamma]\!]_h(\eta), a\rangle) \geq \mathcal{I}(Y,\_)$, i.e., $Y \Vdash_{(\Gamma,x:\sigma),\ \langle [\![\Gamma]\!]_h(\eta),a\rangle}\ p$.

($\Leftarrow$) By hypothesis for all $Y$ and $h \in \mathcal{I}(X,Y)$, and for all $a \in [\![\sigma]\!]_Y$ we have that $Y \Vdash_{(\Gamma,x:\sigma),\ \langle [\![\Gamma]\!]_h(\eta),a\rangle}\ p$, i.e., $[\![\Gamma, x : \sigma \vdash_\Sigma p : o]\!]_Y(\langle [\![\Gamma]\!]_h(\eta), a\rangle) \geq \mathcal{I}(Y,\_)$. Then, take any $u \in \mathcal{I}(X,Y)$, $g \in \mathcal{I}(Y,Z)$ and $t \in [\![\sigma]\!]_Z$; it follows that there exists $h = g \circ u \in \mathcal{I}(X,Z)$. Hence, we can apply the hypothesis and conclude that $[\![\Gamma, x : \sigma \vdash_\Sigma p : o]\!]_Z(\langle [\![\Gamma]\!]_{(g \circ u)}(\eta), t\rangle) \geq \mathcal{I}(Z,\_)$ holds. Since the latter holds for every $Y$ and $u \in \mathcal{I}(X,Y)$, we have that $[\![\Gamma \vdash_\Sigma \forall x{:}\sigma.p : o]\!]_X(\eta) \geq \mathcal{I}(X,\_)$, i.e, $X \Vdash_{\Gamma,\eta} \forall x{:}\sigma.p$.

2. First we note that $X \Vdash_{\Gamma,\eta} p \Rightarrow q$ if and only if $\eta \in \kappa_{[\![\Gamma]\!]}([\![\Gamma \vdash_\Sigma p \Rightarrow q : o]\!])_X$, but this is equivalent to say that $[\![\Gamma \vdash_\Sigma p \Rightarrow q : o]\!]_X(\eta) \geq \mathcal{I}(X,\_)$. Then, since we have that $[\![\Gamma \vdash_\Sigma p \Rightarrow q : o]\!] = imp \circ \langle [\![\Gamma \vdash_\Sigma p : o]\!], [\![\Gamma \vdash_\Sigma q : o]\!]\rangle$, the latter condition is equivalent to say that $\overline{[\![\Gamma \vdash_\Sigma p : o]\!]_X(\eta)} \vee [\![\Gamma \vdash_\Sigma q : o]\!]_X(\eta) \geq \mathcal{I}(X,\_)$, i.e., for all $Y$ $\overline{([\![\Gamma \vdash_\Sigma p : o]\!]_X(\eta))_Y} \cup ([\![\Gamma \vdash_\Sigma q : o]\!]_X(\eta))_Y \supseteq \mathcal{I}(X,Y)$.

($\Rightarrow$) By hypothesis we have that $X \Vdash_{\Gamma,\eta} p \Rightarrow q$ and $X \Vdash_{\Gamma,\eta} p$ hold and the latter is equivalent to say that $[\![\Gamma \vdash_\Sigma p : o]\!]_X(\eta) \geq \mathcal{I}(X,\_)$, i.e., for all $Y$ $([\![\Gamma \vdash_\Sigma p : o]\!]_X(\eta))_Y \supseteq \mathcal{I}(X,Y)$. It follows that $(\overline{[\![\Gamma \vdash_\Sigma p : o]\!]_X(\eta)})_Y = \mathcal{V}(X,Y) \backslash \mathcal{I}(X,Y)$, hence, by the preliminary observation, $([\![\Gamma \vdash_\Sigma q : o]\!]_X(\eta))_Y \supseteq \mathcal{I}(X,Y)$. So we proved that $[\![\Gamma \vdash_\Sigma q : o]\!]_X(\eta) \geq \mathcal{I}(X,\_)$, i.e., that $X \Vdash_{\Gamma,\eta} q$.

($\Leftarrow$) By hypothesis we have that either $X \Vdash_{\Gamma,\eta} p$ does not hold or $X \Vdash_{\Gamma,\eta} q$ holds. In the former case for all $Y$ $([\![\Gamma \vdash_\Sigma p : o]\!]_X(\eta))_Y \not\supseteq \mathcal{I}(X,Y)$, hence $(\overline{[\![\Gamma \vdash_\Sigma p : o]\!]_X(\eta)})_Y \supseteq \mathcal{I}(X,Y)$. So, by the preliminary observation, we also have that for all $Y$ $([\![\Gamma \vdash_\Sigma p \Rightarrow q : o]\!]_X(\eta))_Y \supseteq \mathcal{I}(X,Y)$, hence $X \Vdash_{\Gamma,\eta} p \Rightarrow q$. The other case is even easier, since we have that for all $Y$ $([\![\Gamma \vdash_\Sigma q : o]\!]_X(\eta))_Y \supseteq \mathcal{I}(X,Y)$ and we can conclude again by the preliminary observation.

3. By definition, $X \Vdash_{\Gamma,\eta} PM$ if and only if $\eta \in \kappa_{[\![\Gamma]\!]}([\![\Gamma \vdash_\Sigma PM : o]\!])_X$, i.e., if and only if $[\![\Gamma \vdash_\Sigma PM : o]\!]_X(\eta) \geq \mathcal{I}(X, \_)$. Then the thesis directly follows from the following argument:

$$\begin{array}{rl} & [\![\Gamma \vdash_\Sigma PM : o]\!]_X(\eta) \\ = & (ev_{Prop,[\![\sigma]\!]} \circ \langle [\![\Gamma \vdash_\Sigma P : \sigma \to o]\!], [\![\Gamma \vdash_\Sigma M : \sigma]\!]\rangle)_X(\eta) \\ = & (ev_{Prop,[\![\sigma]\!]})_X(\langle [\![\Gamma \vdash_\Sigma P : \sigma \to o]\!]_X(\eta), [\![\Gamma \vdash_\Sigma M : \sigma]\!]_X(\eta)\rangle) \\ = & ([\![\Gamma \vdash_\Sigma P : \sigma \to o]\!]_X(\eta))_X(\langle [\![\Gamma \vdash_\Sigma M : \sigma]\!]_X(\eta), \mathrm{id}_X\rangle) \end{array}$$

### C.0.8 Proof of Corollary 5.1

1. First of all we have that $X \Vdash_{\Gamma,\eta} \neg p$ stands for $X \Vdash_{\Gamma,\eta} p \Rightarrow \bot$, which is equivalent to say (by Theorem 5.1) that $X \Vdash_{\Gamma,\eta} p$ implies $X \Vdash_{\Gamma,\eta} \bot$. Obviously, this is true if and only if $X \Vdash_{\Gamma,\eta} \bot$ or it is not the case that $X \Vdash_{\Gamma,\eta} p$.

   **($\Rightarrow$)** Since by Proposition 5.2 it is not the case that $X \Vdash_{\Gamma,\eta} \bot$, it must be not the case that $X \Vdash_{\Gamma,\eta} p$ (by the preliminary observation), i.e., the thesis.

   **($\Leftarrow$)** Since, by hypothesis, it is not the case that $X \Vdash_{\Gamma,\eta} p$, we automatically have (by the preliminary observation) that $X \Vdash_{\Gamma,\eta} \neg p$.

2. By definition of $\wedge$, the previous point and Theorem 5.1, we have:

$$\begin{array}{rll} X \Vdash_{\Gamma,\eta} p \wedge q & \text{iff} & X \Vdash_{\Gamma,\eta} \neg(p \Rightarrow \neg q) \\ & \text{iff} & \text{it is not the case that } X \Vdash_{\Gamma,\eta} p \Rightarrow \neg q \\ & \text{iff} & X \Vdash_{\Gamma,\eta} p \text{ and it is not the case that } X \Vdash_{\Gamma,\eta} \neg q \\ & \text{iff} & X \Vdash_{\Gamma,\eta} p \text{ and } X \Vdash_{\Gamma,\eta} q \end{array}$$

3. By definition of $\vee$, point 1 and Theorem 5.1, we have:

$$\begin{array}{rll} X \Vdash_{\Gamma,\eta} p \vee q & \text{iff} & X \Vdash_{\Gamma,\eta} \neg p \Rightarrow q \\ & \text{iff} & X \Vdash_{\Gamma,\eta} \neg p \text{ implies } X \Vdash_{\Gamma,\eta} q \\ & \text{iff} & \text{it is not the case that } X \Vdash_{\Gamma,\eta} \neg p \text{ or } X \Vdash_{\Gamma,\eta} q \\ & \text{iff} & X \Vdash_{\Gamma,\eta} p \text{ or } X \Vdash_{\Gamma,\eta} q \end{array}$$

4. By definition of $\exists$, point 1 and Theorem 5.1, we have:

$$\begin{array}{rll} X \Vdash_{\Gamma,\eta} \exists x{:}\sigma.p & \text{iff} & X \Vdash_{\Gamma,\eta} \neg\forall x{:}\sigma.\neg p \\ & \text{iff} & \text{it is not the case that } X \Vdash_{\Gamma,\eta} \forall x{:}\sigma.\neg p \\ & \text{iff} & \text{there are } Y,\ h \in \mathcal{I}(X,Y) \text{ and } a \in [\![\sigma]\!]_Y \text{ such that} \\ & & \text{it is not the case that } Y \Vdash_{(\Gamma,x:\sigma),\langle [\![\Gamma]\!]_h(\eta),a\rangle} \neg p \\ & \text{iff} & \text{there are } Y,\ h \in \mathcal{I}(X,Y) \text{ and } a \in [\![\sigma]\!]_Y \text{ such that} \\ & & Y \Vdash_{(\Gamma,x:\sigma),\langle [\![\Gamma]\!]_h(\eta),a\rangle} p \end{array}$$

5. The proof will proceed by induction on $n$:

   **(Base case)** $n = 1$: we have to prove that $X \Vdash_{\Gamma,\eta} \forall x_1{:}\sigma_1.p$ if and only if for all $Y$, $f \in \mathcal{I}(X,Y)$ and $\eta_1 \in [\![\sigma_1]\!]_Y$, we have that $Y \Vdash_{(\Gamma,x_1:\sigma_1),\langle [\![\Gamma]\!]_f(\eta),\eta_1\rangle} p$ holds. This is straightforward by point 1 of Theorem 5.1.

**(Inductive case)** let us suppose that the hypothesis holds for $n$; we will prove that it also holds for $n+1$. First of all we apply the point 1 of Theorem 5.1 to obtain the following: $X \Vdash_{\Gamma,\eta} \forall x_1{:}\sigma_1.\forall x_2{:}\sigma_2.\ldots\forall x_{n+1}{:}\sigma_{n+1}p$ if and only if for all $Y$, $f \in \mathcal{I}(X,Y)$, $\eta_1 \in [\![\sigma_1]\!]_Y$ $Y \Vdash_{(\Gamma,x_1:\sigma_1),\langle[\![\Gamma]\!]_f(\eta),\eta_1\rangle} \forall x_2{:}\sigma_2.\ldots\forall x_{n+1}{:}\sigma_{n+1}p$ holds. Then we may apply the inductive hypothesis to deduce that the previous forcing statement holds if and only if for all $Z$, $g \in \mathcal{I}(Y,Z)$, $\eta_2 \in [\![\sigma_2]\!]_Z, \ldots, \eta_{n+1} \in [\![\sigma_{n+1}]\!]_Z$ we have that the following holds:

$$Z \Vdash_{(\Gamma,x_1:\sigma_1,x_2:\sigma_2,\ldots,x_{n+1}:\sigma_{n+1}),\langle[\![\Gamma,x_1:\sigma_1]\!]_g(\langle[\![\Gamma]\!]_f(\eta),\eta_1\rangle),\eta_2,\ldots,\eta_{n+1}\rangle} p.$$

Then we observe that

$$[\![\Gamma,x_1:\sigma_1]\!]_g(\langle[\![\Gamma]\!]_f(\eta),\eta_1\rangle) = \langle[\![\Gamma]\!]_{g\circ f}(\eta),[\![x_1:\sigma_1]\!]_g(\eta_1)\rangle).$$

Hence we can easily conclude by taking $Z=Y$ and $g=\mathrm{id}_Y$:

$$Y \Vdash_{(\Gamma,x_1:\sigma_1,\ldots,x_{n+1}:\sigma_{n+1}),\langle[\![\Gamma]\!]_f(\eta),\eta_1,\eta_2,\ldots,\eta_{n+1}\rangle} p.$$

### C.0.9 Proof of Theorem 5.4

1. In this case we have to prove that $\Gamma \vdash_\Sigma (p \Rightarrow q \Rightarrow r) \Rightarrow (p \Rightarrow q) \Rightarrow p \Rightarrow r$ holds, i.e., that for all $X$, $\eta \in [\![\Gamma]\!]_X$ we have
$$X \Vdash_{\Gamma,\eta} (p \Rightarrow q \Rightarrow r) \Rightarrow (p \Rightarrow q) \Rightarrow p \Rightarrow r.$$
By Theorem 5.1, this is equivalent to prove that $X \Vdash_{\Gamma,\eta} (p \Rightarrow q \Rightarrow r)$, $X \Vdash_{\Gamma,\eta} (p \Rightarrow q)$ and $X \Vdash_{\Gamma,\eta} p$ imply $X \Vdash_{\Gamma,\eta} r$. Hence, applying repeatedly Theorem 5.1, we can easily deduce that $X \Vdash_{\Gamma,\eta} q$ holds from $X \Vdash_{\Gamma,\eta} (p \Rightarrow q)$, since we know that $X \Vdash_{\Gamma,\eta} p$ holds. At this point we can easily conclude, applying again Theorem 5.1, since $X \Vdash_{\Gamma,\eta} r$ derives from $X \Vdash_{\Gamma,\eta} (p \Rightarrow q \Rightarrow r)$, $X \Vdash_{\Gamma,\eta} p$ and $X \Vdash_{\Gamma,\eta} q$.

2. By definition we have to prove that for all $X$, $\eta \in [\![\Gamma]\!]_X$ we have $X \Vdash_{\Gamma,\eta} p \Rightarrow q \Rightarrow p$. By Theorem 5.1, this is equivalent to prove that $X \Vdash_{\Gamma,\eta} p$ and $X \Vdash_{\Gamma,\eta} q$ imply $X \Vdash_{\Gamma,\eta} q$. Hence the conclusion is trivial.

3. By definition we have to prove that for all $X$, $\eta \in [\![\Gamma]\!]_X$ we have $X \Vdash_{\Gamma,\eta} \forall_\sigma(P) \Rightarrow PM$. By Theorem 5.1, this is equivalent to prove that $X \Vdash_{\Gamma,\eta} \forall_\sigma(P)$ implies $X \Vdash_{\Gamma,\eta} PM$. But $X \Vdash_{\Gamma,\eta} \forall_\sigma(P)$ is equivalent to say that, for all $Y$, $f \in \mathcal{I}(X,Y)$ and $a \in [\![\sigma]\!]_Y$, $Y \Vdash_{(\Gamma,x:\sigma),\langle[\![\Gamma]\!]_f(\eta),a\rangle} Px$ holds. Hence, taking $Y \triangleq X$, $f \triangleq \mathrm{id}_X$ and $a \triangleq [\![\Gamma \vdash_\Sigma M : \sigma]\!]_X(\eta)$, we have that $X \Vdash_{(\Gamma,x:\sigma),\langle\eta,[\![\Gamma\vdash_\Sigma M:\sigma]\!]_X(\eta)\rangle} Px$ holds. By Theorem 5.1, this is equivalent to say that $([\![\Gamma,x:\sigma \vdash_\Sigma P : \sigma \to o]\!]_X(\langle\eta,[\![\Gamma \vdash_\Sigma M : \sigma]\!]_X(\eta)\rangle))_X(\langle[\![\Gamma,x : \sigma \vdash_\Sigma x : \sigma]\!]_X(\langle\eta,[\![\Gamma \vdash_\Sigma M : \sigma]\!]_X(\eta)\rangle),\mathrm{id}_X\rangle) \geq \mathcal{I}(X,\_)$. Now we observe that $([\![\Gamma,x : \sigma \vdash_\Sigma P : \sigma \to o]\!]_X(\langle\eta,[\![\Gamma \vdash_\Sigma M : \sigma]\!]_X(\eta)\rangle))_X(\langle[\![\Gamma,x : \sigma \vdash_\Sigma x : \sigma]\!]_X(\langle\eta,[\![\Gamma \vdash_\Sigma M : \sigma]\!]_X(\eta)\rangle),\mathrm{id}_X\rangle) \geq \mathcal{I}(X,\_) = ([\![\Gamma,x : \sigma \vdash_\Sigma P : \sigma \to o]\!]_X(\langle\eta,[\![\Gamma \vdash_\Sigma M : \sigma]\!]_X(\eta)\rangle))_X(\langle[\![\Gamma \vdash_\Sigma M : \sigma]\!]_X(\eta),\mathrm{id}_X\rangle) = ([\![\Gamma \vdash_\Sigma P : \sigma \to o]\!]_X(\eta))_X(\langle[\![\Gamma \vdash_\Sigma M : \sigma]\!]_X(\eta),\mathrm{id}_X\rangle)$ Hence, applying again Theorem 5.1, we have proved that $X \Vdash_{\Gamma,\eta} PM$ holds.

4. By definition we have to prove that for all $X$, $\eta \in [\![\Gamma]\!]_X$ we have $X \Vdash_{\Gamma,\eta} (\lambda x{:}\sigma.M)N =^{\sigma'} M[N/x]$. First of all we notice that the following holds:
$$\begin{aligned} [\![\Gamma \vdash_\Sigma (\lambda x{:}\sigma.M)N]\!]_X(\eta) &= \\ (ev_{[\![\sigma']\!],[\![\sigma]\!]})_X(\langle[\![\Gamma \vdash_\Sigma \lambda x{:}\sigma.M : \sigma \to \sigma']\!]_X(\eta),[\![\Gamma \vdash_\Sigma N : \sigma]\!]_X(\eta)\rangle) &= \\ ([\![\Gamma \vdash_\Sigma \lambda x{:}\sigma.M : \sigma \to \sigma']\!]_X(\eta))_X(\langle[\![\Gamma \vdash_\Sigma N : \sigma]\!]_X(\eta),\mathrm{id}_X\rangle) &= \\ [\![\Gamma,x : \sigma \vdash_\Sigma M : \sigma']\!]_X(\langle\eta,[\![\Gamma \vdash_\Sigma N : \sigma]\!]_X(\eta)\rangle) & \end{aligned}$$

Now, we can proceed by structural induction on $M$:

**($M \equiv y \neq x$)** Trivial.

**($M \equiv x$)** Trivial.

**($M \equiv PQ$)** The following holds:

$$\begin{array}{rl}
[\![\Gamma \vdash_\Sigma (PQ)[N/x] : \sigma']\!]_X(\eta) & = \\
[\![\Gamma \vdash_\Sigma P[N/x]Q[N/x] : \sigma']\!]_X(\eta) & = \\
(ev_{[\![\sigma']\!],[\![\gamma]\!]})_X(\langle[\![\Gamma \vdash_\Sigma P[N/x] : \gamma \to \sigma']\!]_X(\eta), [\![\Gamma \vdash_\Sigma Q[N/x] : \gamma]\!]_X(\eta)\rangle) & = \\
([\![\Gamma \vdash_\Sigma P[N/x] : \gamma \to \sigma']\!]_X(\eta))_X(\langle[\![\Gamma \vdash_\Sigma Q[N/x] : \gamma]\!]_X(\eta), \mathrm{id}_X\rangle) & \stackrel{\text{(I.H.)}}{=} \\
([\![\Gamma, \vdash_\Sigma (\lambda x{:}\sigma.P)N : \gamma \to \sigma']\!]_X(\eta))_X(\langle[\![\Gamma \vdash_\Sigma (\lambda x{:}\sigma.Q)N : \gamma]\!]_X(\eta), \mathrm{id}_X\rangle) &
\end{array}$$

Moreover, we have the following:

$$\begin{array}{rl}
[\![\Gamma, x : \sigma \vdash_\Sigma M : \sigma']\!]_X(\langle\eta, [\![\Gamma \vdash_\Sigma N : \sigma]\!]_X(\eta)\rangle) & = \\
[\![\Gamma, x : \sigma \vdash_\Sigma PQ : \sigma']\!]_X(\langle\eta, [\![\Gamma \vdash_\Sigma N : \sigma]\!]_X(\eta)\rangle) & = \\
(ev_{[\![\sigma']\!],[\![\gamma]\!]})_X(\langle A, B\rangle) & = \\
(A)_X(\langle B, \mathrm{id}_X\rangle) &
\end{array}$$

where $A \triangleq [\![\Gamma, x : \sigma \vdash_\Sigma P : \gamma \to \sigma']\!]_X(\langle\eta, [\![\Gamma \vdash_\Sigma N : \sigma]\!]_X(\eta)\rangle)$ and $B \triangleq [\![\Gamma, x : \sigma \vdash_\Sigma Q : \gamma]\!]_X(\langle\eta, [\![\Gamma \vdash_\Sigma N : \sigma]\!]_X(\eta)\rangle)$. Hence we may conclude since we have

$$\begin{array}{rl}
[\![\Gamma, \vdash_\Sigma (\lambda x{:}\sigma.P)N : \gamma \to \sigma']\!]_X(\eta) & = \\
[\![\Gamma, x : \sigma \vdash_\Sigma P : \gamma \to \sigma']\!]_X(\langle\eta, [\![\Gamma \vdash_\Sigma N : \sigma]\!]_X(\eta)\rangle) & = \\
A &
\end{array}$$

and

$$\begin{array}{rl}
[\![\Gamma \vdash_\Sigma (\lambda x{:}\sigma.Q)N : \gamma]\!]_X(\eta)\rangle) & = \\
[\![\Gamma, x : \sigma \vdash_\Sigma Q : \gamma]\!]_X(\langle\eta, [\![\Gamma \vdash_\Sigma N : \sigma]\!]_X(\eta)\rangle) & = \\
B &
\end{array}$$

**($M \equiv \lambda z{:}\gamma.P$ with $x \neq z$)** In this case $\sigma' \equiv \gamma \to \delta$; hence the following holds:

$$\begin{array}{rl}
[\![\Gamma \vdash_\Sigma (\lambda z{:}\gamma.P)[N/x] : \sigma']\!]_X(\eta) & = \\
[\![\Gamma \vdash_\Sigma (\lambda z{:}\gamma.P[N/x]) : \sigma']\!]_X(\eta) & = \\
\{\lambda\langle b, f\rangle \in [\![\gamma]\!]_Y \times \mathcal{V}(X, Y).[\![\Gamma, z : \gamma \vdash_\Sigma P[N/x] : \delta]\!]_Y(\mu)\}_{Y \in \mathcal{V}} & \stackrel{\text{(I.H.)}}{=} \\
\{\lambda\langle b, f\rangle \in [\![\gamma]\!]_Y \times \mathcal{V}(X, Y).[\![\Gamma, z : \gamma \vdash_\Sigma (\lambda x{:}\sigma.P)N : \delta]\!]_Y(\mu)\}_{Y \in \mathcal{V}} &
\end{array}$$

where $\mu \triangleq \langle[\![\Gamma]\!]_f(\eta), b\rangle$. Moreover, we have

$$\begin{array}{rl}
[\![\Gamma, z : \gamma \vdash_\Sigma (\lambda x{:}\sigma.P)N : \delta]\!]_Y(\mu) & = \\
(ev_{[\![\delta]\!],[\![\sigma]\!]})_Y(\langle[\![\Gamma, z : \gamma \vdash_\Sigma \lambda x{:}\sigma.P : \sigma \to \delta]\!]_Y(\mu), [\![\Gamma, z : \gamma \vdash_\Sigma N : \sigma]\!]_Y(\mu)\rangle) & = \\
([\![\Gamma, z : \gamma \vdash_\Sigma \lambda x{:}\sigma.P : \sigma \to \delta]\!]_Y(\mu))_Y(\langle[\![\Gamma, z : \gamma \vdash_\Sigma N : \sigma]\!]_Y(\mu), id_Y\rangle) & = \\
[\![\Gamma, z : \gamma, x : \sigma \vdash_\Sigma P : \delta]\!]_Y(\langle[\![\Gamma]\!]_f(\eta), b, [\![\Gamma, z : \gamma \vdash_\Sigma N : \sigma]\!]_Y(\mu)\rangle) & = \\
[\![\Gamma, x : \sigma, z : \gamma \vdash_\Sigma P : \delta]\!]_Y(\langle[\![\Gamma]\!]_f(\eta), [\![\Gamma, z : \gamma \vdash_\Sigma N : \sigma]\!]_Y(\mu), b\rangle) &
\end{array}$$

For what concerns $[\![\Gamma, x : \sigma \vdash_\Sigma M : \sigma']\!]_X(\langle \eta, [\![\Gamma \vdash_\Sigma N : \sigma]\!]_X(\eta)\rangle)$, we have the following:

$$\begin{aligned}
[\![\Gamma, x : \sigma \vdash_\Sigma M : \sigma']\!]_X(\langle \eta, [\![\Gamma \vdash_\Sigma N : \sigma]\!]_X(\eta)\rangle) &= \\
[\![\Gamma, x : \sigma \vdash_\Sigma \lambda z{:}\gamma.P : \sigma']\!]_X(\langle \eta, [\![\Gamma \vdash_\Sigma N : \sigma]\!]_X(\eta)\rangle) &= \\
\{\lambda\langle b, f\rangle \in [\![\gamma]\!]_Y \times \mathcal{V}(X,Y).m_Y(\langle [\![\Gamma, x : \sigma]\!]_f(\langle \eta, [\![\Gamma \vdash_\Sigma N : \sigma]\!]_X(\eta)\rangle), b\rangle)\}_{Y \in \mathcal{V}} &= \\
\{\lambda\langle b, f\rangle \in [\![\gamma]\!]_Y \times \mathcal{V}(X,Y).m_Y(\langle [\![\Gamma]\!]_f(\eta), [\![\Gamma \vdash_\Sigma N : \sigma]\!]_Y([\![\Gamma]\!]_f(\eta)), b\rangle)\}_{Y \in \mathcal{V}} &= \\
\{\lambda\langle b, f\rangle [\![\gamma]\!]_Y \times \in \mathcal{V}(X,Y) \times .m_Y(\langle [\![\Gamma]\!]_f(\eta), \beta, b\rangle)\}_{Y \in \mathcal{V}} &
\end{aligned}$$

where $m \triangleq [\![\Gamma, x : \sigma, z : \gamma \vdash_\Sigma P : \delta]\!]$ and $\beta \triangleq [\![\Gamma, z : \gamma \vdash_\Sigma N : \sigma]\!]_Y([\![\Gamma]\!]_f(\eta), b)$; in the fourth step we exploited the naturality of $[\![\Gamma \vdash_\Sigma N : \sigma]\!]$ since $[\![x : \sigma]\!]_f([\![\Gamma \vdash_\Sigma N : \sigma]\!]_X(\eta)) = [\![\Gamma \vdash_\Sigma N : \sigma]\!]_Y([\![\Gamma]\!]_f(\eta))$ and the weakening rule. Hence we have the thesis.

5. In this case we have to prove that for all $X$, $\eta \in [\![\Gamma]\!]_X$ we have

$$X \Vdash_{\Gamma,\eta} (\forall x{:}\sigma.M =^{\sigma'} N) \Rightarrow \lambda x{:}\sigma.M = \lambda x{:}\sigma'.N,$$

i.e., by Corollary 5.1, that $X \Vdash_{\Gamma,\eta} (\forall x{:}\sigma.M =^{\sigma'} N)$ implies

$$X \Vdash_{\Gamma,\eta} \lambda x{:}\sigma.M =^{\sigma \to \sigma'} \lambda x{:}\sigma.N.$$

First, we observe the following:

$$[\![\Gamma \vdash_\Sigma \lambda x{:}\sigma.M]\!]_X(\eta) = \{\lambda\langle b, f\rangle \in [\![\sigma]\!]_Y \times \mathcal{V}(X,Y).m_Y(\langle [\![\Gamma]\!]_f(\eta), b\rangle)\}_{Y \in \mathcal{V}},$$

where $m \triangleq [\![\Gamma, x : \sigma \vdash_\Sigma M : \sigma']\!]$. Similarly, we have:

$$[\![\Gamma \vdash_\Sigma \lambda x{:}\sigma.N]\!]_X(\eta) = \{\lambda\langle b, f\rangle \in [\![\sigma]\!]_Y \times \mathcal{V}(X,Y).n_Y(\langle [\![\Gamma]\!]_f(\eta), b\rangle)\}_{Y \in \mathcal{V}},$$

where $n \triangleq [\![\Gamma, x : \sigma \vdash_\Sigma N : \sigma']\!]$. Hence, in order to conclude, it is sufficient to show that $m = n$, i.e., that, for every $Y \in \mathcal{V}$, $f \in \mathcal{V}(X,Y)$ and $b \in [\![\sigma]\!]_Y$, $[\![\Gamma, x : \sigma \vdash_\Sigma M : \sigma']\!](\langle [\![\Gamma]\!]_f(\eta), b\rangle) = [\![\Gamma, x : \sigma \vdash_\Sigma N : \sigma']\!]_Y(\langle [\![\Gamma]\!]_f(\eta), b\rangle)$. Then, we observe that our hypothesis is equivalent (by Theorem 5.1) to say that for all $Y \in \mathcal{V}$, $h \in \mathcal{I}(X,Y)$, $\eta_x \in [\![\sigma]\!]_Y$, $Y \Vdash_{(\Gamma,x:\sigma),\langle [\![\Gamma]\!]_h(\eta),\eta_x\rangle} M =^{\sigma'} N$ holds. We observe that we can write $f = p \circ e$ where $p : Z \longrightarrow Y$ is surjective and $e : X \longrightarrow Z$ is injective[1]. Since every surjective map has a right inverse, we have that $[\![\sigma]\!]_p$ is surjective too; hence, there exists $b' \in [\![\sigma]\!]_Z$ such that $[\![\sigma]\!]_p(b') = b$. By naturality of $[\![\Gamma, x : \sigma \vdash_\Sigma M : \sigma']\!]$, we have $[\![\Gamma, x : \sigma \vdash_\Sigma M : \sigma']\!]_Y(\langle [\![\Gamma]\!]_f(\eta), b\rangle) = [\![\sigma']\!]_p([\![\Gamma, x : \sigma \vdash_\Sigma M : \sigma']\!]_Z(\langle [\![\Gamma]\!]_e(\eta), b'\rangle))$. By hypothesis ($e$ is injective), we have $[\![\Gamma, x : \sigma \vdash_\Sigma M : \sigma']\!]_Z(\langle [\![\Gamma]\!]_e(\eta), b'\rangle) = [\![\Gamma, x : \sigma \vdash_\Sigma N : \sigma']\!]_Z(\langle [\![\Gamma]\!]_e(\eta), b'\rangle)$, whence the thesis.

6. By Theorem 5.3 we have to show that for all $X$, $\eta \in [\![\Gamma]\!]_X$ the following holds:

$$[\![\Gamma \vdash_\Sigma \lambda x{:}\sigma.Mx : \sigma \to \sigma']\!]_X(\eta) = [\![\Gamma \vdash_\Sigma M : \sigma \to \sigma']\!]_X(\eta).$$

Since the members of the latter equation are natural transformations between the functors $[\![\sigma]\!] \times \mathcal{V}(X, \_)$ and $[\![\sigma']\!]$, the thesis is equivalent to prove that the following holds for every $Y$, $f \in \mathcal{V}(X,Y)$ and $b \in [\![\sigma]\!]_Y$:

$$([\![\Gamma \vdash_\Sigma \lambda x{:}\sigma.Mx : \sigma \to \sigma']\!]_X(\eta))_Y(\langle b, f\rangle) = ([\![\Gamma \vdash_\Sigma M : \sigma \to \sigma']\!]_X(\eta))_Y(\langle b, f\rangle).$$

[1] This can be done for every $f \in \mathcal{V}(X,Y)$ by putting $Z \triangleq X \uplus (Y \setminus Im(f))$.

Indeed, we have:

$$\begin{array}{rl}
(\llbracket \Gamma \vdash_\Sigma \lambda x{:}\sigma.Mx : \sigma \to \sigma' \rrbracket_X(\eta))_Y(\langle b, f\rangle) & = \\
\llbracket \Gamma, x : \sigma \vdash_\Sigma Mx : \sigma' \rrbracket_Y(\langle \llbracket \Gamma \rrbracket_f(\eta), b\rangle) & = \\
(ev_{\llbracket \sigma' \rrbracket, \llbracket \sigma \rrbracket})_Y(\langle \llbracket \Gamma, x : \sigma \vdash_\Sigma M : \sigma \to \sigma' \rrbracket_Y(\langle \llbracket \Gamma \rrbracket_f(\eta), b\rangle), b\rangle) & = \\
(\llbracket \Gamma, x : \sigma \vdash_\Sigma M : \sigma \to \sigma' \rrbracket_Y(\langle \llbracket \Gamma \rrbracket_f(\eta), b\rangle))_Y(\langle b, \mathrm{id}_Y\rangle) & = \\
(\llbracket \Gamma \vdash_\Sigma M : \sigma \to \sigma' \rrbracket_Y(\llbracket \Gamma \rrbracket_f(\eta)))_Y(\langle b, \mathrm{id}_Y\rangle) & = \\
((\llbracket \sigma \rrbracket \Rightarrow \llbracket \sigma' \rrbracket)_f(\llbracket \Gamma \vdash_\Sigma M : \sigma \to \sigma' \rrbracket_X(\eta)))_Y(\langle b, \mathrm{id}_Y\rangle) & = \\
(\llbracket \Gamma \vdash_\Sigma M : \sigma \to \sigma' \rrbracket_X(\eta))_Y(\langle b, f\rangle). &
\end{array}$$

7. We have to show that for all $X$, $\eta \in \llbracket \Gamma \rrbracket_X$ $X \Vdash_{\Gamma,\eta} \neg\neg p \Rightarrow p$ holds. By Theorem 5.1, this is equivalent to prove that $X \Vdash_{\Gamma,\eta} \neg\neg p$ implies $X \Vdash_{\Gamma,\eta} p$. By Corollary 5.1, the premise means that it is not the case that $X \Vdash_{\Gamma,\eta} \neg p$ holds. Applying again the same corollary, we have that it is not the case that $X \Vdash_{\Gamma,\eta} p$ does not hold, i.e., the thesis.

8. In this case the thesis follows directly from Theorem 5.1.

9. By Theorem 5.1, the premise is equivalent to say that for all $X$ and $\eta \in \llbracket \Gamma, x : \sigma \rrbracket_X$ $X \Vdash_{(\Gamma,x:\sigma),\eta} p$ implies $X \Vdash_{(\Gamma,x:\sigma),\eta} q$. To prove that the thesis holds it suffices to show, by Theorem 5.1, that for all $Y$ and $\mu \in \llbracket \Gamma \rrbracket_Y$ $Y \Vdash_{\Gamma,\mu} p$ implies $Y \Vdash_{\Gamma,\mu} \forall x{:}\sigma.q$. The latter, again by Theorem 5.1, is equivalent to show that for all $Z$, $f \in \mathcal{I}(Y, Z)$ and $a \in \llbracket \sigma \rrbracket_Z$ $Z \Vdash_{(\Gamma,x:\sigma),\langle \llbracket \Gamma \rrbracket_f(\mu),a\rangle} q$ holds. From the validity of $Y \Vdash_{\Gamma,\mu} p$, by the monotonicity of forcing, we can deduce that, for all $Z$ and $f \in \mathcal{I}(Y, Z)$, $Z \Vdash_{\Gamma,\llbracket \Gamma \rrbracket_f(\mu)} p$ holds. By the weakening rule, we also have that, for all $a \in \llbracket \sigma \rrbracket_Z$, $Z \Vdash_{(\Gamma,x:\sigma),\langle \llbracket \Gamma \rrbracket_f(\mu),a\rangle} p$ holds. Hence we can apply the premise to conclude that $Z \Vdash_{(\Gamma,x:\sigma),\langle \llbracket \Gamma \rrbracket_f(\mu),a\rangle} q$ holds.

### C.0.10 Proof of Theorem 5.5

**($\Rightarrow$)** By structural induction on the derivation of $\Gamma \vdash_\Sigma M : \iota$:

**($\Gamma \vdash_\Sigma 0 : \iota$)** Since we have $\llbracket \Gamma \vdash_\Sigma 0 : \iota \rrbracket_X(\eta) = 0$, we can easily conclude observing that $FV(0) = \emptyset$.

**($\Gamma \vdash_\Sigma \tau.P : \iota$)** Hence the previous derivation step yields $\Gamma \vdash_\Sigma P : \iota$. By inductive hypothesis we have that $\llbracket \Gamma \vdash_\Sigma y : \upsilon \rrbracket_X(\eta) \notin FV(\llbracket \Gamma \vdash_\Sigma P : \iota \rrbracket_X(\eta))$. Hence we can deduce that $\llbracket \Gamma \vdash_\Sigma y : \upsilon \rrbracket_X(\eta) \notin FV(\tau.\llbracket \Gamma \vdash_\Sigma P : \iota \rrbracket_X(\eta))$. The thesis is an easy consequence observing the following:

$$\begin{array}{rcl}
\tau.\llbracket \Gamma \vdash_\Sigma P : \iota \rrbracket_X(\eta) & = & tau(\llbracket \Gamma \vdash_\Sigma P : \iota \rrbracket_X(\eta)) \\
& = & (tau \circ \llbracket \Gamma \vdash_\Sigma P : \iota \rrbracket)_X(\eta) \\
& \triangleq & \llbracket \Gamma \vdash_\Sigma \tau.P : \iota \rrbracket_X(\eta).
\end{array}$$

**($\Gamma \vdash_\Sigma P \mid Q : \iota$)** Hence the previous derivation step yields $\Gamma \vdash_\Sigma P_1 : \iota$ and $\Gamma \vdash_\Sigma P_2 : \iota$. By inductive hypothesis we have $\llbracket \Gamma \vdash_\Sigma y : \upsilon \rrbracket_X(\eta) \notin FV(\llbracket \Gamma \vdash_\Sigma P : \iota \rrbracket_X(\eta))$ and $\llbracket \Gamma \vdash_\Sigma y : \upsilon \rrbracket_X(\eta) \notin FV(\llbracket \Gamma \vdash_\Sigma Q : \iota \rrbracket_X(\eta))$. Hence we can deduce that $\llbracket \Gamma \vdash_\Sigma y : \upsilon \rrbracket_X(\eta) \notin FV(\llbracket \Gamma \vdash_\Sigma P : \iota \rrbracket_X(\eta) \mid \llbracket \Gamma \vdash_\Sigma Q : \iota \rrbracket_X(\eta))$. The thesis is an easy consequence observing the following:

$$\begin{array}{rl}
\llbracket \Gamma \vdash_\Sigma P : \iota \rrbracket_X(\eta) \mid \llbracket \Gamma \vdash_\Sigma Q : \iota \rrbracket_X(\eta) & = \\
par(\langle \llbracket \Gamma \vdash_\Sigma P : \iota \rrbracket_X(\eta), \llbracket \Gamma \vdash_\Sigma Q : \iota \rrbracket_X(\eta)\rangle) & = \\
(par \circ \langle \llbracket \Gamma \vdash_\Sigma P : \iota \rrbracket, \llbracket \Gamma \vdash_\Sigma Q : \iota \rrbracket\rangle)_X(\eta) & \triangleq \\
\llbracket \Gamma \vdash_\Sigma P \mid Q : \iota \rrbracket_X(\eta). &
\end{array}$$

($\Gamma \vdash_\Sigma [u \neq v]P : \iota$) Hence the previous derivation step yields $\Gamma \vdash_\Sigma P : \iota$. By inductive hypothesis $[\![\Gamma \vdash_\Sigma y : \upsilon]\!]_X(\eta) \notin FV([\![\Gamma \vdash_\Sigma P : \iota]\!]_X(\eta))$; moreover, $[\![\Gamma \vdash_\Sigma y : \upsilon]\!]_X(\eta) \neq [\![\Gamma \vdash_\Sigma u : \upsilon]\!]_X(\eta)$ and $[\![\Gamma \vdash_\Sigma y : \upsilon]\!]_X(\eta) \neq [\![\Gamma \vdash_\Sigma v : \upsilon]\!]_X(\eta)$. Hence we can deduce that $[\![\Gamma \vdash_\Sigma y : \upsilon]\!]_X(\eta) \notin FV([\![\Gamma \vdash_\Sigma P : \iota]\!]_X(\eta)) \cup \{[\![\Gamma \vdash_\Sigma u : \upsilon]\!]_X(\eta), [\![\Gamma \vdash_\Sigma v : \upsilon]\!]_X(\eta)\}$. The thesis is an easy consequence observing the following:

$$\begin{aligned}
[[\![\Gamma \vdash_\Sigma u : \upsilon]\!]_X(\eta) \neq [\![\Gamma \vdash_\Sigma v : \upsilon]\!]_X(\eta)][\![\Gamma \vdash_\Sigma P : \iota]\!]_X(\eta) &= \\
mismatch(\langle [\![\Gamma \vdash_\Sigma u : \upsilon]\!]_X(\eta), [\![\Gamma \vdash_\Sigma v : \upsilon]\!]_X(\eta), [\![\Gamma \vdash_\Sigma P : \iota]\!]_X(\eta)\rangle) &= \\
(mismatch \circ \langle [\![\Gamma \vdash_\Sigma u : \upsilon]\!], [\![\Gamma \vdash_\Sigma v : \upsilon]\!], [\![\Gamma \vdash_\Sigma P : \iota]\!]\rangle)_X(\eta) &\triangleq \\
[\![\Gamma \vdash_\Sigma [u \neq v]P : \iota]\!]_X(\eta). &
\end{aligned}$$

($\Gamma \vdash_\Sigma \nu\lambda x{:}\upsilon.P : \iota$) Hence a preceding derivation step yields $\Gamma, x : \upsilon \vdash_\Sigma P : \iota$. By inductive hypothesis $[\![\Gamma \vdash_\Sigma y : \upsilon]\!]_X(\eta) \notin FV([\![\Gamma, x : \upsilon \vdash_\Sigma P : \iota]\!]_X(\langle \eta, \eta_x\rangle))$ for all $\eta_x \neq [\![\Gamma \vdash_\Sigma y : \upsilon]\!]_X(\eta)$. Hence we can deduce that $[\![\Gamma \vdash_\Sigma y : \upsilon]\!]_X(\eta) \notin FV((\nu\eta_x)([\![\Gamma, x : \upsilon \vdash_\Sigma P : \iota]\!]_X(\langle \eta, \eta_x\rangle)))$, where $\eta_x \in [\![x : \upsilon]\!]_X$. Again, the thesis is a direct consequence of the following:

$$\begin{aligned}
(\nu\eta_x)([\![\Gamma, x : \upsilon \vdash_\Sigma P : \iota]\!]_X(\langle \eta, \eta_x\rangle)) &= \\
(\nu\eta_x)([\![\Gamma \vdash_\Sigma \lambda x{:}\upsilon.P : \upsilon \to \iota]\!]_X(\eta))_{X \uplus \{x\}}(\langle \eta_x, \mathrm{id}_X\rangle)) &= \\
new([\![\Gamma \vdash_\Sigma \lambda x{:}\upsilon.P : \upsilon \to \iota]\!]_X(\eta)) &= \\
(new \circ [\![\Gamma \vdash_\Sigma \lambda x{:}\upsilon.P : \upsilon \to \iota]\!])_X(\eta) &\triangleq \\
[\![\Gamma \vdash_\Sigma \nu\lambda x{:}\upsilon.P : \iota]\!]_X(\eta) &
\end{aligned}$$

($\Leftarrow$) **Preliminary observation**: first of all we recall that $y \notin M$ is an abbreviation for

$$\forall p{:}\upsilon \to \iota \to o.(\forall z{:}\upsilon.\forall Q{:}\iota.(T_{\notin}\ p\ z\ Q) \Rightarrow (p\ z\ Q)) \Rightarrow (p\ y\ M).$$

Hence, by point 1 of Theorem 5.1, in order to prove that $X \Vdash_{\Gamma,\eta} y \notin M$, we must show that for all $Y$, $f \in \mathcal{I}(X, Y)$ and $\eta_p \in [\![\upsilon \to \iota \to o]\!]_Y = (Var \Rightarrow Proc \Rightarrow Prop)_Y$,

$$Y \Vdash_{(\Gamma, p:\upsilon\to\iota\to o),\langle [\![\Gamma]\!]_f(\eta), \eta_p\rangle} (\forall z{:}\upsilon.\forall Q{:}\iota.(T_{\notin}\ p\ z\ Q) \Rightarrow (p\ z\ Q)) \Rightarrow\ (p\ y\ M)$$

holds, i.e., by point 2 of Theorem 5.1, if and only if

$$Y \Vdash_{(\Gamma, p:\upsilon\to\iota\to o),\langle [\![\Gamma]\!]_f(\eta), \eta_p\rangle} \forall z{:}\upsilon.\ \forall Q{:}\iota.(T_{\notin}\ p\ z\ Q) \Rightarrow (p\ z\ Q)$$

implies

$$Y \Vdash_{(\Gamma, p:\upsilon\to\iota\to o),\langle [\![\Gamma]\!]_f(\eta), \eta_p\rangle} (p\ y\ M).$$

So, we suppose that the premise is true and we show that the consequence also holds; by point 5 of Corollary 5.1, we can deduce that the premise is true if and only if for all $Z$, $g \in \mathcal{I}(Y, Z)$, $\eta_z \in Var_Z \triangleq Z$ and $\eta_Q \in Proc_Z$, $Z \Vdash_{\Delta,\mu} (T_{\notin}\ p\ z\ Q) \Rightarrow (p\ z\ Q)$ holds, where $\Delta \triangleq (\Gamma, p : \upsilon \to \iota \to o, z : \upsilon, Q : \iota)$ and $\mu \triangleq \langle [\![\Gamma]\!]_{g\circ f}(\eta), [\![p : \upsilon \to \iota \to o]\!]_g(\eta_p), \eta_z, \eta_Q\rangle$. In particular, taking $Z \triangleq Y$, $g \triangleq \mathrm{id}_Y$, $\eta_z \triangleq [\![\Gamma, p : \upsilon \to \iota \to o \vdash_\Sigma y : \upsilon]\!]_Y(\langle [\![\Gamma]\!]_f(\eta), \eta_p\rangle)$ and $\eta_Q \triangleq [\![\Gamma, p : \upsilon \to \iota \to o \vdash_\Sigma M : \iota]\!]_Y(\langle [\![\Gamma]\!]_f(\eta), \eta_p\rangle)$, we have that the following holds:

$$Y \Vdash_{(\Gamma, p:\upsilon\to\iota\to o, z:\upsilon, Q:\iota),\langle [\![\Gamma]\!]_f(\eta), \eta_p, \eta_z, \eta_Q\rangle} (T_{\notin}\ p\ z\ Q) \Rightarrow (p\ z\ Q)$$

This is equivalent, by Theorem 5.1, to say that

$$Y \Vdash_{(\Gamma, p:\upsilon\to\iota\to o, z:\upsilon, Q:\iota),\langle [\![\Gamma]\!]_f(\eta), \eta_p, \eta_z, \eta_Q\rangle} (T_{\notin}\ p\ z\ Q)$$

implies

$$Y \Vdash_{(\Gamma,p:\upsilon\to\iota\to o,z:\upsilon,Q:\iota),\langle[\![\Gamma]\!]_f(\eta),\eta_p,\eta_z,\eta_Q\rangle} (p\ z\ Q).$$

Since $[\![\Gamma,p:\upsilon\to\iota\to o,z:\upsilon,Q:\iota\vdash_\Sigma (p\ z\ Q)]\!]_Y(\langle[\![\Gamma]\!]_f(\eta),\eta_p,\eta_z,\eta_Q\rangle) = [\![\Gamma,p:\upsilon\to\iota\to o\vdash_\Sigma (p\ y\ M)]\!]_Y(\langle[\![\Gamma]\!]_f(\eta),\eta_p\rangle)$, to conclude, it suffices to prove that $Y \Vdash_{(\Gamma,p:\upsilon\to\iota\to o,z:\upsilon,Q:\iota),\langle[\![\Gamma]\!]_f(\eta),\eta_p,\eta_z,\eta_Q\rangle} (T_{\notin}\ p\ z\ Q)$ holds.

By definition of $T_{\notin}$, $(T_{\notin}\ p\ z\ Q)$ is the following $\lambda$-term:

$$\begin{array}{l}
Q = 0\ \vee \\
(\exists P{:}\iota.Q = \sigma.P \wedge (p\ z\ P))\ \vee \\
(\exists P_1{:}\iota.\exists P_2{:}\iota.Q = P_1 \mid P_2 \wedge (p\ z\ P_1) \wedge (p\ z\ P_2))\ \vee \\
(\exists P{:}\iota.\exists y{:}\upsilon.\exists u{:}\upsilon.Q = [y \neq u]P \wedge \neg z =^{\upsilon} y \wedge \neg z =^{\upsilon} u \wedge (p\ z\ P))\ \vee \\
(\exists P{:}\upsilon\to\iota.Q = \nu P \wedge (\forall y{:}\upsilon.\neg z =^{\upsilon} y \Rightarrow (p\ z\ (P\ y))))
\end{array}$$

Hence (by Corollary 5.1), to prove the premise, it suffices to show that one of the disjunctions holds. At this point we can proceed by structural induction on the derivation of $\Gamma\vdash_\Sigma M:\iota$:

**($\Gamma\vdash_\Sigma 0:\iota$)** Since $M\equiv 0$, we can immediately conclude by the preliminary observation, since $\eta_Q$ was chosen as $[\![\Gamma,p:\upsilon\to\iota\to o\vdash_\Sigma M:\iota]\!]_Y(\langle[\![\Gamma]\!]_f(\eta),\eta_p\rangle)$, whence

$$Y \Vdash_{(\Gamma,p:\upsilon\to\iota\to o,z:\upsilon,Q:\iota),\langle[\![\Gamma]\!]_f(\eta),\eta_p,\eta_z,\eta_Q\rangle} Q = 0.$$

**($\Gamma\vdash_\Sigma \tau.P:\iota$)** By inductive hypothesis, we know that $X \Vdash_{\Gamma,\eta} y\notin P$ holds. Hence, by an argument similar to that used in the preliminary observation, we have that

$$Y \Vdash_{(\Gamma,p:\upsilon\to\iota\to o),\langle[\![\Gamma]\!]_f(\eta),\eta_p\rangle} \forall z{:}\upsilon.\ \forall Q{:}\iota.(T_{\notin}\ p\ z\ Q) \Rightarrow (p\ z\ Q)$$

implies

$$Y \Vdash_{(\Gamma,p:\upsilon\to\iota\to o),\langle[\![\Gamma]\!]_f(\eta),\eta_p\rangle} (p\ y\ P).$$

But, since the premise is true (by the preliminary observation), we have that $Y \Vdash_{(\Gamma,p:\upsilon\to\iota\to o),\langle[\![\Gamma]\!]_f(\eta),\eta_p\rangle} (p\ y\ P)$ holds. At this point we may easily conclude observing that the second disjunction holds (remember that $\eta_z \triangleq [\![\Gamma,p:\upsilon\to\iota\to o\vdash_\Sigma y:\upsilon]\!]_Y(\langle[\![\Gamma]\!]_f(\eta),\eta_p\rangle)$ and $\eta_Q \triangleq [\![\Gamma,p:\upsilon\to\iota\to o\vdash_\Sigma M:\iota]\!]_Y(\langle[\![\Gamma]\!]_f(\eta),\eta_p\rangle)$, where $M\equiv\sigma.P$).

**($\Gamma\vdash_\Sigma P_1 \mid P_2:\iota$)** $X \Vdash_{\Gamma,\eta} y\notin P_1$ and $X \Vdash_{\Gamma,\eta} y\notin P_2$ hold by inductive hypothesis. Hence, like in the previous case, we can deduce that

$$Y \Vdash_{(\Gamma,p:\upsilon\to\iota\to o),\langle[\![\Gamma]\!]_f(\eta),\eta_p\rangle} (p\ y\ P_1)$$

and

$$Y \Vdash_{(\Gamma,p:\upsilon\to\iota\to o),\langle[\![\Gamma]\!]_f(\eta),\eta_p\rangle} (p\ y\ P_2)$$

hold. At this point we may easily conclude observing that the third disjunction holds (remember that $\eta_z \triangleq [\![\Gamma,p:\upsilon\to\iota\to o\vdash_\Sigma y:\upsilon]\!]_Y(\langle[\![\Gamma]\!]_f(\eta),\eta_p\rangle)$ and $\eta_Q \triangleq [\![\Gamma,p:\upsilon\to\iota\to o\vdash_\Sigma M:\iota]\!]_Y(\langle[\![\Gamma]\!]_f(\eta),\eta_p\rangle)$, where $M\equiv P_1\mid P_2$).

**($\Gamma\vdash_\Sigma [u\neq v]P:\iota$)** By inductive hypothesis we know that $X \Vdash_{\Gamma,\eta} y\notin P$. Hence, as in the previous cases, we can deduce that

$$Y \Vdash_{(\Gamma,p:\upsilon\to\iota\to o),\langle[\![\Gamma]\!]_f(\eta),\eta_p\rangle} (p\ y\ P)$$

holds. Moreover from the hypothesis that $[\![\Gamma \vdash_\Sigma y : v]\!]_X(\eta) \notin FV([\![\Gamma \vdash_\Sigma M : \iota]\!]_X(\eta))$ we have that $[\![\Gamma \vdash_\Sigma y : v]\!]_X(\eta) \neq [\![\Gamma \vdash_\Sigma u : v]\!]_X(\eta)$ and $[\![\Gamma \vdash_\Sigma y : v]\!]_X(\eta) \neq [\![\Gamma \vdash_\Sigma v : v]\!]_X(\eta)$ and consequently that the statements $X \Vdash_{\Gamma,\eta} y =^v u$ and $X \Vdash_{\Gamma,\eta} y =^v v$ do not hold. By Corollary 5.1 this is equivalent to say that $X \Vdash_{\Gamma,\eta} \neg y =^v u$ and $X \Vdash_{\Gamma,\eta} \neg y =^v v$ hold. Whence, by the weakening rule and the monotonicity of forcing, we have that

$$Y \Vdash_{(\Gamma,p:v\to\iota\to o),\langle[\![\Gamma]\!]_f(\eta),\eta_p\rangle} \neg y =^v u$$

and

$$Y \Vdash_{(\Gamma,p:v\to\iota\to o),\langle[\![\Gamma]\!]_f(\eta),\eta_p\rangle} \neg y =^v v$$

hold. Again, we may easily conclude by the preliminary observation since the fourth disjunction holds (remember that $\eta_z \triangleq [\![\Gamma, p : v \to \iota \to o \vdash_\Sigma y : v]\!]_Y(\langle[\![\Gamma]\!]_f(\eta), \eta_p\rangle)$ and $\eta_Q \triangleq [\![\Gamma, p : v \to \iota \to o \vdash_\Sigma M : \iota]\!]_Y(\langle[\![\Gamma]\!]_f(\eta), \eta_p\rangle)$, where $M \equiv [u \neq v]P$).

**($\Gamma \vdash_\Sigma \nu\lambda x{:}v.P$)** Since we know that $[\![\Gamma \vdash_\Sigma y : v]\!]_X(\eta) \notin FV([\![\Gamma \vdash_\Sigma \nu\lambda x{:}v.P]\!]_X(\eta))$ and $[\![\Gamma \vdash_\Sigma \nu\lambda x{:}v.P]\!]_X(\eta) \triangleq (\nu\eta_x)([\![\Gamma, x : v \vdash_\Sigma P : \iota]\!]_X(\langle\eta,\eta_x\rangle))$, by inductive hypothesis we deduce that $X \Vdash_{(\Gamma,x:v),\langle\eta,\eta_x\rangle} y \notin P$ holds for all $\eta_x \neq [\![\Gamma \vdash_\Sigma y : v]\!]_X(\eta)$; hence, proceeding as in the previous cases and applying the weakening rule, we have that

$$Y \Vdash_{(\Gamma,x:v,p:v\to\iota\to o),\langle[\![\Gamma]\!]_f(\eta),f(\eta_x),\eta_p\rangle} (p\ y\ P)$$

holds. Moreover we have that $Y \Vdash_{(\Gamma,x:v,p:v\to\iota\to o),\langle[\![\Gamma]\!]_f(\eta),f(\eta_x),\eta_p\rangle} \neg y =^v x$. At this point we may easily conclude by the preliminary observation since the fifth disjunction holds.

### C.0.11 Proof of Theorem 5.9

We will see only the base case (rule $Rec^\iota_\sigma\_red_1$) and the case of higher-order constructor (rule $Rec^\iota_\sigma\_red_5$), the others being similar.

Let $G \triangleq [\![\Gamma]\!]$, $A \triangleq [\![\sigma]\!]$, and

$$\begin{aligned}
g_1 &= [\![\Gamma \vdash f_1 : \sigma]\!] &&: G \longrightarrow A\\
g_2 &= [\![\Gamma \vdash f_2 : \sigma \to \sigma]\!] &&: G \longrightarrow A \Rightarrow A\\
g_3 &= [\![\Gamma \vdash f_3 : \sigma \to \sigma \to \sigma]\!] &&: G \longrightarrow A \Rightarrow A \Rightarrow A\\
g_4 &= [\![\Gamma \vdash f_4 : v \to v \to \sigma \to \sigma]\!] &&: G \longrightarrow Var \Rightarrow Var \Rightarrow A \Rightarrow A\\
g_5 &= [\![\Gamma \vdash f_5 : (v \to \sigma) \to \sigma]\!] &&: G \longrightarrow (Var \Rightarrow A) \Rightarrow A
\end{aligned}$$

For proving the soundness of $Rec^\iota_\sigma\_red_1$ and $Rec^\iota_\sigma\_red_5$, we have to prove that for all $X$ and $\eta \in [\![\Gamma]\!]_X$ the following properties hold:

$$X \Vdash_{\Gamma,\eta} (R\ 0) =^\sigma f_1 \tag{C.1}$$

$$X \Vdash_{\Gamma,\eta} \forall P{:}v \to \iota.(R\ \nu P) =^\sigma (f_5\ \lambda x{:}v.(R\ (P\ x))) \tag{C.2}$$

where $R$ is a syntactic shorthand for $(Rec^\iota_\sigma\ f_1\ f_2\ f_3\ f_4\ f_5)$.

We prove equivalence (C.1). By Theorem 5.3, this is equivalent to prove that

$$[\![\Gamma \vdash_\Sigma (R\ 0) : \sigma]\!]_X(\eta) = [\![\Gamma \vdash_\Sigma f_1 : \sigma]\!]_X(\eta)$$

In fact, the following equalities hold, where $[\![R]\!]$ is a syntactic shorthand for the interpretation of $R$, and $m : T(G \Rightarrow A) \longrightarrow G \Rightarrow A$, $\bar{m} : \mathit{Proc} \longrightarrow G \Rightarrow A$ are the natural transformations used in the interpretation of $R$ above:

$$\begin{array}{lll} [\![\Gamma \vdash_\Sigma (R\ 0) : \sigma]\!]_X(\eta) & = ev_X(\langle [\![ [\![R]\!]_X(\eta), \Gamma \vdash_\Sigma 0 : \iota]\!]_X(\eta)\rangle) & \\ & = ([\![R]\!]_X(\eta))_X(\langle [\![\Gamma \vdash_\Sigma 0 : \iota]\!]_X(\eta), \mathrm{id}_X\rangle) & \\ & = (\bar{m}_X(0))_X(\eta, \mathrm{id}_X) & \text{by definition of } [\![R]\!] \text{ and since} \\ & & [\![\Gamma \vdash_\Sigma 0 : \iota]\!]_X(\eta) = 0 \\ & = ((\bar{m} \circ \alpha)_X(in_1(*)))_X(\eta, \mathrm{id}_X) & \text{since } \alpha_X(in_1(*)) = 0 \\ & = ((m \circ T\bar{m})_X(in_1(*)))_X(\eta, \mathrm{id}_X) & \text{by the initial algebra property} \\ & = (m_X(in_1(*)))_X(\eta, \mathrm{id}_X) & \text{since } (T\bar{m})_X(in_1(*)) = in_1(*) \\ & = g_{1X}(\eta) & \text{by definition of } m \\ & = [\![\Gamma \vdash_\Sigma f_1 : \sigma]\!]_X(\eta) & \end{array}$$

We prove equivalence (C.2). By Theorem 5.3, this is equivalent to prove that for all $Y$ stage, $h \in \mathcal{I}(X, Y)$, $p \in (\mathit{Var} \Rightarrow \mathit{Proc})_Y$:

$$[\![\Gamma, P : \upsilon \to \iota \vdash_\Sigma (R\ \nu P) : \sigma]\!]_Y(\eta[h], p) = [\![\Gamma, P : \iota \vdash_\Sigma (f_5\ \lambda x.(R\ (P\ x))) : \sigma]\!]_Y(\eta[h], p)$$

In fact, the following equalities hold:

$$\begin{array}{ll} [\![\Gamma, P : \upsilon \to \iota \vdash_\Sigma (R\ \nu P) : \sigma]\!]_Y(\eta[h], p) = & \\ = ev_Y(\langle [\![R]\!]_Y(\eta[h], p), [\![\Gamma, P : \upsilon \to \iota \vdash_\Sigma \nu P : \iota]\!]_Y(\eta[h], p)\rangle) & \\ = ([\![R]\!]_Y(\eta[h], p))_Y(\langle [\![\Gamma, P : \upsilon \to \iota \vdash_\Sigma \nu P : \iota]\!]_Y(\eta[h], p), \mathrm{id}_Y\rangle) & \\ = (\bar{m}_Y(\nu\lambda x.p)_Y(\eta, \mathrm{id}_Y) & \text{by definition of } [\![R]\!] \text{ and since} \\ & [\![\Gamma, P{:}\upsilon{\to}\iota \vdash_\Sigma \nu P : \iota]\!]_Y(\eta[h], p) = \nu\lambda x.p \\ = ((\bar{m} \circ \alpha)_Y(in_5(p)))_Y(\eta[h], \mathrm{id}_Y) & \text{since } \alpha_Y(in_5(p)) = \nu\lambda x.p \\ = ((m \circ T\bar{m})_Y(in_5(p)))_Y(\eta[h], \mathrm{id}_Y) & \text{by the initial algebra property} \\ = \ldots & \end{array}$$

Now, it is not hard to see that $(T\bar{m})_Y(in_5(p)) = in_5(\bar{m} \circ p)$, where $\bar{m} \circ p : \mathit{Var} \times \mathcal{V}(Y, \_) \longrightarrow G \Rightarrow A$; thus, let $r' \in (\mathit{Var} \Rightarrow A)_Y$ be the natural transformation defined as

$$\begin{array}{rl} r' : & \mathit{Var} \times \mathcal{V}(Y, \_) \longrightarrow A \\ r'_Z : & Z \times \mathcal{V}(Y, Z) \longrightarrow A_Z \\ & \langle z, k\rangle \longmapsto ((\bar{m} \circ p)_Z(z, k))_Z(\eta[k \circ h], \mathrm{id}_Z) \end{array}$$

We have then

$$\begin{array}{ll} \ldots = (m_Y(in_5(\bar{m} \circ p))_Y(\eta[h], \mathrm{id}_Y) & \\ = (g_{5Y}(\eta[h]))_Y(\langle r', \mathrm{id}_Y\rangle) & \text{by definition of } m \\ = ev_Y(\langle [\![\Gamma \vdash_\Sigma f_5 : (\upsilon \to \sigma) \to \sigma]\!]_Y(\eta[h], p), r'\rangle) & \\ = ev_Y(\langle [\![\Gamma \vdash_\Sigma f_5 : (\upsilon \to \sigma) \to \sigma]\!]_Y(\eta[h], p), & (*) \\ \qquad [\![\Gamma, P : \upsilon \to \iota \vdash_\Sigma \lambda x{:}\upsilon.(R\ (P\ x)) : \upsilon \to \sigma]\!]_Y(\eta[h], p)\rangle) & \\ = [\![\Gamma \vdash_\Sigma (f_5\ \lambda x{:}\upsilon.(R\ (P\ x)) : \sigma]\!]_Y(\eta[h], p) & \end{array}$$

The equality $(*)$ holds because

$$[\![\Gamma, P : \upsilon \to \iota \vdash_\Sigma \lambda x{:}\upsilon.(R\ (P\ x)) : \upsilon \to \sigma]\!]_Y(\eta[h], p) = r'.$$

Indeed, for all stage $Z$, $z \in Z$, $k \in \mathcal{V}(Y, Z)$, and let $\eta' \triangleq \langle \eta[k \circ h], p[k], z\rangle$:

$$
\begin{aligned}
&(\llbracket \Gamma, P : \upsilon \to \iota \vdash_\Sigma \lambda x{:}\upsilon.(R\ (P\ x)) : \upsilon \to \sigma \rrbracket_Y(\eta[h], p))_Z\,(z, k) = \\
&= \llbracket \Gamma, P : \upsilon \to \iota, x : \upsilon \vdash_\Sigma (R\ (P\ x)) : \sigma \rrbracket_Z(\eta') \\
&= (\llbracket R \rrbracket_Z(\eta'))_Z(\langle \llbracket \Gamma, P : \upsilon \to \iota, x : \upsilon \vdash_\Sigma (P\ x) : \iota \rrbracket_Z(\eta'), \mathrm{id}_Z\rangle) \\
&= (\bar{m}_Z(\llbracket \Gamma, P : \upsilon \to \iota, x : \upsilon \vdash_\Sigma (P\ x) : \iota \rrbracket_Z(\eta')))_Z(\eta', \mathrm{id}_Z) \\
&= (\bar{m}_Z(p[k]_Z(z, \mathrm{id}_Z)))_Z(\eta', \mathrm{id}_Z) \\
&= ((\bar{m} \circ p[k])_Z(z, \mathrm{id}_Z))_Z(\eta', \mathrm{id}_Z) \\
&= ((\bar{m} \circ p)_Z(z, k))_Z(\eta', \mathrm{id}_Z) = r'_Z(z, k)
\end{aligned}
$$

### C.0.12 Proof of Proposition 5.8

Let us check that the first diagram of Definition B.5 commutes, i.e., that for every $A, B \in \check{\mathcal{V}}$, $X \in \mathcal{V}$, $a \in A_X$ and $b \in (TB)_X$ we have

$$(T\pi')_X((st_{A,B})_X(\langle a, b\rangle)) = \pi'_X(\langle a, b\rangle) = b$$

This is proved by cases over $b$:

**($b = in_1(*)$)** $(T\pi')_X((st_{A,B})_X(\langle a, in_1(*)\rangle)) = (T\pi')_X(in_1(*)) \triangleq in_1(*)$.

**($b = in_2(b')$)**

$$
\begin{aligned}
& (T\pi')_X((st_{A,B})_X(\langle a, in_2(b')\rangle)) \\
= \;& (T\pi')_X(in_2(a, b')) \\
\triangleq \;& in_2(\pi'(\langle a, b'\rangle)) = in_2(b')
\end{aligned}
$$

**($b = in_3(\langle b', b''\rangle)$)**

$$
\begin{aligned}
& (T\pi')_X((st_{A,B})_X(\langle a, in_3(\langle b', b''\rangle)\rangle)) \\
= \;& (T\pi')_X(in_3(\langle a, b', a, b''\rangle)) \\
\triangleq \;& in_3(\langle \pi'(\langle a, b'\rangle), \pi'(\langle a, b''\rangle)\rangle) \\
= \;& in_3(\langle b', b''\rangle)
\end{aligned}
$$

**($b = in_4(\langle x, y, b'\rangle)$)**

$$
\begin{aligned}
& (T\pi')_X((st_{A,B})_X(\langle a, in_4(\langle x, y, b'\rangle)\rangle)) \\
= \;& (T\pi')_X(in_4(\langle x, y, a, b'\rangle)) \\
\triangleq \;& in_4(\langle x, y, \pi'(\langle a, b'\rangle)\rangle) \\
= \;& in_3(\langle x, y, b'\rangle)
\end{aligned}
$$

**($b = in_5(b')$)**

$$
\begin{aligned}
& (T\pi')_X((st_{A,B})_X(\langle a, in_5(b')\rangle)) \\
= \;& (T\pi')_X(in_5(\overline{b}_a)) \\
\triangleq \;& in_5(\gamma_{B,X}(\pi'(\langle G_{in_X}(g), b_{X \uplus \{x\}}(x, in_X)\rangle))) \\
= \;& in_5(\gamma_{B,X}(b_{X \uplus \{x\}}(x, in_X))) = in_5(b)
\end{aligned}
$$

For what concerns the commutativity of the second diagram of Definition B.5, we have to show that for every $A, B, C \in \check{\mathcal{V}}$, $X \in \mathcal{V}$, $a \in A_X$, $b \in (TB)_X$ and $c \in C_X$ we have

$$(T\beta)_X((st_{A,C\times B})_X((\mathrm{id}_A \times st_{C,B})_X(\langle a, \langle c, b\rangle\rangle))) = (st_{A\times C,B})_X(\beta_X(\langle a, \langle c, b\rangle\rangle))$$

where $\beta \triangleq \langle\langle \pi, \pi \circ \pi'\rangle, \pi' \circ \pi'\rangle$; it follows that the second member can be simplified to $(st_{A\times C,B})_X(\langle\langle a, c\rangle, b\rangle)$. In order to prove the thesis, we proceed again by cases on $b$:

**($b = in_1(*)$)**

$$\begin{array}{rl} & (T\beta)_X((st_{A,C\times B})_X((\mathrm{id}_A \times st_{C,B})_X(\langle a, \langle c, in_1(*)\rangle\rangle))) \\ = & (T\beta)_X((st_{A,C\times B})_X(\langle a, in_1(*)\rangle)) \\ = & (T\beta)_X(in_1(*)) \\ = & in_1(*) \\ = & (st_{A\times C,B})_X(\langle\langle a, c\rangle, in_1(*)\rangle) \end{array}$$

**($b = in_2(b')$)**

$$\begin{array}{rl} & (T\beta)_X((st_{A,C\times B})_X((\mathrm{id}_A \times st_{C,B})_X(\langle a, \langle c, in_2(b')\rangle\rangle))) \\ = & (T\beta)_X((st_{A,C\times B})_X(\langle a, in_2(\langle c, b'\rangle)\rangle)) \\ = & (T\beta)_X(in_2(\langle a, \langle c, b'\rangle\rangle)) \\ = & in_2(\beta_X(\langle a, \langle c, b'\rangle\rangle)) \\ = & in_2(\langle\langle a, c\rangle, b'\rangle) \\ = & (st_{A\times C,B})_X(\langle\langle a, c\rangle, in_2(b')\rangle) \end{array}$$

**($b = in_3(\langle b', b''\rangle)$)**

$$\begin{array}{rl} & (T\beta)_X((st_{A,C\times B})_X((\mathrm{id}_A \times st_{C,B})_X(\langle a, \langle c, in_3(\langle b', b''\rangle)\rangle\rangle))) \\ = & (T\beta)_X((st_{A,C\times B})_X(\langle a, in_3(\langle c, b', c, b''\rangle)\rangle)) \\ = & (T\beta)_X(in_3(\langle a, \langle c, b', c, b''\rangle\rangle)) \\ = & in_3(\beta_X(\langle a, \langle c, b'\rangle, a, \langle c, b''\rangle\rangle)) \\ = & in_3(\langle\langle a, c\rangle, b', \langle a, c\rangle, b''\rangle) \\ = & (st_{A\times C,B})_X(\langle\langle a, c\rangle, in_3(\langle b', b''\rangle)\rangle) \end{array}$$

**($b = in_4(\langle x, y, b'\rangle)$)**

$$\begin{array}{rl} & (T\beta)_X((st_{A,C\times B})_X((\mathrm{id}_A \times st_{C,B})_X(\langle a, \langle c, in_4(\langle x, y, b'\rangle)\rangle\rangle))) \\ = & (T\beta)_X((st_{A,C\times B})_X(\langle a, in_4(\langle x, y, c, b'\rangle)\rangle)) \\ = & (T\beta)_X(in_4(\langle x, y, a, \langle c, b'\rangle\rangle)) \\ = & in_4(\langle x, y, \beta_X(\langle a, \langle c, b'\rangle\rangle)\rangle) \\ = & in_4(\langle x, y, \langle a, c\rangle, b'\rangle) \\ = & (st_{A\times C,B})_X(\langle\langle a, c\rangle, in_4(\langle x, y, b'\rangle)\rangle) \end{array}$$

**($b = in_5(b')$)**

$$\begin{array}{rl} & (T\beta)_X((st_{A,C\times B})_X((\mathrm{id}_A \times st_{C,B})_X(\langle a, \langle c, in_5(b')\rangle\rangle))) \\ = & (T\beta)_X((st_{A,C\times B})_X(\langle a, in_5(\overline{b'}_c)\rangle)) \\ = & (T\beta)_X(in_5(\overline{(\overline{b'}_c)}_a)) \\ = & in_5(\gamma_{(A\times C)\times B,X}(\beta_{X\uplus\{x\}}(\langle A_{in_X}(a), \langle C_{in_X}(c), b'_{X\uplus\{x\}}(\langle x, in_X\rangle)\rangle\rangle))) \\ = & in_5(\gamma_{(A\times C)\times B,X}(\langle A_{in_X}(a), C_{in_X}(c)\rangle, b'_{X\uplus\{x\}}(\langle x, in_X\rangle))) \\ = & in_5(\overline{b'}_{\langle a,c\rangle}) \\ = & (st_{A\times C,B})_X(\langle\langle a, c\rangle, in_5(b')\rangle) \end{array}$$

### C.0.13 Proof of Proposition 5.9

In order to prove the commutativity of the diagram we must show that, for every $X \in \mathcal{V}$, $g \in G_X$ and $P \in (TProc)_X$, we have $f_X((\mathrm{id}_G \times \alpha)_X(\langle g, P\rangle)) = \beta_X((\langle \pi, Tf \circ st_{G,Proc}\rangle)_X(\langle g, P\rangle))$. First of all we notice that the second member of the previous equation can be simplified to $\beta_X(\langle g, (Tf)_X((st_{G,Proc})_X(\langle g, P\rangle))\rangle)$, then we proceed by cases on $P$:

**($P = in_1(*)$)** we have $f_X((\mathrm{id}_G \times \alpha)_X(\langle g, in_1(*)\rangle)) = f_X(\langle g, 0\rangle) \triangleq \beta_X(\langle g, in_1(*\ )\rangle)$, whence the thesis since

$$\begin{aligned} & \beta_X(\langle g, (Tf)_X((st_{G,Proc})_X(\langle g, in_1(*)\rangle))\rangle) \\ = \ & \beta_X(\langle g, (Tf)_X(in_1(*))\rangle) \\ = \ & \beta_X(\langle g, in_1(*)\rangle) \end{aligned}$$

**($P = in_2(P')$)** we have $f_X((\mathrm{id}_G \times \alpha)_X(\langle g, in_2(P')\rangle)) = f_X(\langle g, \tau.P'\rangle) \triangleq \beta_X(\langle g, in_2(f_X(\langle g, P'\rangle))\rangle)$, whence the thesis since

$$\begin{aligned} & \beta_X(\langle g, (Tf)_X((st_{G,Proc})_X(\langle g,\ in_2(P')\rangle))\rangle) \\ = \ & \beta_X(\langle g, (Tf)_X(in_2(\langle g, P'\rangle))\rangle) \\ = \ & \beta_X(\langle g, in_2(f_X(\langle g, P'\rangle))\rangle) \end{aligned}$$

**($P = in_3(\langle P', P''\rangle)$)** we have

$$\begin{aligned} & f_X((\mathrm{id}_G \times \alpha)_X(\langle g, in_3(\langle P', P''\rangle)\rangle)) \\ = \ & f_X(\langle g, P'|P''\rangle) \\ \triangleq \ & \beta_X(\langle g, in_3(\langle f_X(\langle g, P'\rangle), f_X(\langle g, P''\rangle)\rangle)\rangle), \end{aligned}$$

whence the thesis since

$$\begin{aligned} & \beta_X(\langle g, (Tf)_X((st_{G,Proc})_X(\langle g, in_3(\langle P', P''\rangle)\rangle))\rangle) \\ = \ & \beta_X(\langle g, (Tf)_X(in_3(\langle g, P', g, P''\rangle))\rangle) \\ = \ & \beta_X(\langle g, in_3(\langle f_X(\langle g, P'\rangle), f_X(\langle g, P''\rangle)\rangle)\rangle) \end{aligned}$$

**($P = in_4(\langle x, y, P'\rangle)$)** we have $f_X((\mathrm{id}_G \times \alpha)_X(\langle g, in_4(\langle x, y, P'\rangle)\rangle)) = f_X(\langle g, [x \neq y]P'\rangle) \triangleq \beta_X(\langle g, in_4(\langle x, y, f_X(\langle g, P'\rangle)\rangle)\rangle)$, whence the thesis since

$$\begin{aligned} & \beta_X(\langle g, (Tf)_X((st_{G,Proc})_X(\langle g, in_4(\langle x, y, P'\rangle)\rangle))\rangle) \\ = \ & \beta_X(\langle g, (Tf)_X(in_4(\langle x, y, g, P'\rangle))\rangle) \\ = \ & \beta_X(\langle g, in_4(\langle x, y, f_X(\langle g, P'\rangle)\rangle)\rangle) \end{aligned}$$

**($P = in_5(P')$)** we have $f_X((\mathrm{id}_G \times \alpha)_X(\langle g, in_5(P')\rangle)) = f_X(\langle g, (\nu x)P'_{X \uplus \{x\}}(x, in_X)\rangle) \triangleq \beta_X(\langle g, in_5(\gamma_{B,X}(f_{X\uplus\{x\}}(\langle G_{in_X}(g), P'\rangle)))\rangle)$, whence the thesis since

$$\begin{aligned} & \beta_X(\langle g, (Tf)_X((st_{G,Proc})_X(\langle g, in_5(P')\rangle))\rangle) \\ = \ & \beta_X(\langle g, (Tf)_X(in_5(\overline{P'}_g))\rangle) \\ = \ & \beta_X(\langle g, in_5(\gamma_{B,X}(f_{X\uplus\{x\}}(P'_{X\uplus\{x\}}(\langle x, in_X\rangle))))\rangle) \end{aligned}$$

### C.0.14 Proof of Theorem 5.12

Suppose that

$$Y \Vdash_{R:\iota\to o, \eta_R} (R\ 0), \tag{C.3}$$
$$Y \Vdash_{R:\iota\to o, \eta_R} (\forall P{:}\iota.(R\ P) \Rightarrow (R\ \tau.P)), \tag{C.4}$$
$$Y \Vdash_{R:\iota\to o, \eta_R} (\forall P{:}\iota.(R\ P) \Rightarrow \forall Q{:}\iota.(R\ Q) \Rightarrow (R\ P|Q)), \tag{C.5}$$
$$Y \Vdash_{R:\iota\to o, \eta_R} (\forall y{:}\upsilon.\forall z{:}\upsilon.\forall P{:}\iota.(R\ P) \Rightarrow (R\ [y \neq z]P)), \tag{C.6}$$
$$Y \Vdash_{R:\iota\to o, \eta_R} (\forall P{:}\upsilon \to \iota.(\forall x{:}\upsilon.(R\ (P\ x))) \Rightarrow (R\ \nu P)), \tag{C.7}$$

We prove that $G^*(\top) \bullet G^*(!_{TU}) = p \bullet G^*(\alpha) \bullet T /\!\!/ G(h)$. We first translate the latter equation in terms of composition in the category $\check{\mathcal{V}}$ and we obtain the following:

$$G^*(\top) \circ \langle \pi, G^*(!_{TU}) \rangle = p \circ \langle \pi, G^*(\alpha) \rangle \circ \langle \pi, (T /\!\!/ G)_h \rangle.$$

Then, unfolding the definitions of $G^*$ and $T /\!\!/ G$, we get:

$$\top \circ \pi' \circ \langle \pi, !_{TU} \circ \pi' \rangle = p \circ \langle \pi, \alpha \circ \pi' \rangle \circ \langle \pi, Th \circ st_{G,U} \rangle,$$

i.e., we have to prove that $\top \circ !_{TU} \circ \pi' = p \circ \langle \pi, \alpha \circ Th \circ st_{G,U} \rangle$. So, taken any $Z \in \mathcal{V}$, $g \in G_Z$ and $u \in (TU)_Z$, we have that $\top_Z((!_{TU})_Z(\pi'_Z(\langle g, u \rangle))) = \top_Z((!_{TU})_Z(u)) = \top_Z(*) = \mathcal{I}(Z, \_)$, while for the second member of the equation we have the following:

**($u = in_1(*)$)**

$$\begin{aligned} & p_Z(\langle \pi_Z(\langle g, in_1(*) \rangle), \alpha_Z((Th)_Z((st_{G,U})_Z(\langle g, in_1(*) \rangle)))\rangle) = \\ =& p_Z(\langle g, \alpha_Z((Th)_Z(in_1(*)))\rangle) = p_Z(\langle g, \alpha_Z(in_1(*))\rangle) = p_Z(\langle g, 0 \rangle) \end{aligned}$$

Hence, $p_Z(\langle g, 0 \rangle) = (ev_{Prop,Proc})_Z(\langle g, 0 \rangle) \wedge \mathcal{I}(Z, \_) = g_Z(\langle 0, \mathrm{id}_Z \rangle) \wedge \mathcal{I}(Z, \_)$. Since we know that for all $Y \in \mathcal{V}$, and $\eta_R \in (Proc \Rightarrow Prop)_Y$, $Y \Vdash_{R:\iota \to o, \eta_R} (R\ 0)$ holds, we can deduce, by point 3 of Theorem 5.1, that $[\![R : \iota \to o, P : \iota \vdash_\Sigma (R\ P) : o]\!]_Z(\langle g, 0 \rangle) = ([\![R : \iota \to o, P : \iota \vdash_\Sigma R : \iota \to o]\!]_Z(\langle g, 0 \rangle))_Z(\langle [\![R : \iota \to o, P : \iota \vdash_\Sigma 0 : \iota]\!]_Z(\langle g, 0 \rangle), \mathrm{id}_Z \rangle) = g_Z(\langle 0, \mathrm{id}_Z \rangle) \geq \mathcal{I}(Z, \_)$, whence the thesis.

**($u = in_2(q)$)**

$$\begin{aligned} & p_Z(\langle \pi_Z(\langle g, in_2(q) \rangle), \alpha_Z((Th)_Z((st_{G,U})_Z(\langle g, in_2(q) \rangle)))\rangle) = \\ =& p_Z(\langle g, \alpha_Z((Th)_Z(in_2(\langle g, q \rangle)))\rangle) = p_Z(\langle g, \alpha_Z(in_2(h_Z(\langle g, q \rangle)))\rangle) = \\ =& p_Z(\langle g, \tau . h_Z(\langle g, q \rangle)\rangle) \end{aligned}$$

At this point we know, by equation C.4, that for all $Y \in \mathcal{V}$, and $\eta_R \in (Proc \Rightarrow Prop)_Y$, $Y \Vdash_{R:\iota \to o, \eta_R} \forall P{:}\iota.(R\ P) \Rightarrow (R\ \tau P)$ holds. By points 1 and 2 of Theorem 5.1, this amounts to say that, for all $V \in \mathcal{V}$, $l \in \mathcal{I}(Y, V)$ and $\eta_P \in Proc_V$,

$$V \Vdash_{(R:\iota \to o, P:\iota), \langle (Proc \Rightarrow Prop)_l(\eta_R), \eta_P \rangle} (R\ P)$$

implies

$$V \Vdash_{(R:\iota \to o, P:\iota), \langle (Proc \Rightarrow Prop)_l(\eta_R), \eta_P \rangle} (R\ \tau P).$$

Then we notice the following facts:

1. $p_Z(\langle g, h_Z(\langle g, q \rangle)\rangle) = \mathcal{I}(Z, \_)$;
2. $p_Z(\langle g, h_Z(\langle g, q \rangle)\rangle) = (ev_{Prop,Proc})_Z(\langle g, h_Z(\langle g, q \rangle)\rangle) \wedge \mathcal{I}(Z, \_) = g_Z(\langle h_Z(\langle g, q \rangle), \mathrm{id}_Z \rangle)$ $\wedge \mathcal{I}(Z, \_)$;
3. $[\![R : \iota \to o, P : \iota \vdash_\Sigma (R\ P) : o]\!]_Z(\langle g, h_Z(\langle g, q \rangle)\rangle) = ([\![R : \iota \to o, P : \iota \vdash_\Sigma R : \iota \to o]\!]_Z(\langle g, h_Z(\langle g, q \rangle)\rangle))_Z(\langle [\![R : \iota \to o, P : \iota \vdash_\Sigma P : \iota]\!]_Z(\langle g, h_Z(\langle g, q \rangle)\rangle), \mathrm{id}_Z \rangle) = g_Z(\langle h_Z(\langle g, q \rangle), \mathrm{id}_Z \rangle)$ (by point 3 of Theorem 5.1); it follows from the previous two facts that $g_Z(\langle h_Z(\langle g, q \rangle), \mathrm{id}_Z \rangle) \geq \mathcal{I}(Z, \_)$; hence $Z \Vdash_{(R:\iota \to o, P:\iota), \langle g, h_Z(\langle g, q \rangle)\rangle} (R\ P)$ holds;

4. from the previous fact and the inductive hypothesis we can deduce that

$$Z \Vdash_{(R:\iota\to o,P:\iota),\langle g,h_Z(\langle g,q\rangle)\rangle} (R\ \tau P)$$

holds, i.e.,

$$[\![R : \iota \to o, P : \iota \vdash_\Sigma (R\ \tau P) : o]\!]_Z(\langle g, h_Z(\langle g,q\rangle)\rangle) \geq \mathcal{I}(Z, \_);$$

5. by point 3 of Theorem 5.1, we have $[\![R : \iota \to o, P : \iota \vdash_\Sigma (R\ \tau P) : o]\!]_Z(\langle g, h_Z(\langle g, q\rangle)\rangle) = ([\![R : \iota \to o, P : \iota \vdash_\Sigma R : \iota \to o]\!]_Z(\langle g, h_Z(\langle g,q\rangle)\rangle))_Z(\langle[\![R : \iota \to o, P : \iota \vdash_\Sigma \tau P : \iota]\!]_Z(\langle g, h_Z(\langle g,q\rangle)\rangle), \mathrm{id}_Z\rangle) = g_Z(\langle tau([\![R : \iota \to o, P : \iota \vdash_\Sigma P : \iota]\!]_Z(\langle g, h_Z(\langle g,q\rangle)\rangle), \mathrm{id}_Z\rangle) = g_Z(\langle tau(h_Z(\langle g,q\rangle)), \mathrm{id}_Z\rangle) = g_Z(\tau.h_Z(\langle g,q\rangle), \mathrm{id}_Z) = p_Z(\langle g, \tau.h_Z(\langle g,q\rangle)\rangle) \wedge \mathcal{I}(Z,\_)$, whence the thesis.

**($u = in_3(q,r)$)**

$$\begin{aligned}&p_Z(\langle \pi_Z(\langle g, in_3(q,r)\rangle), \alpha_Z((Th)_Z((st_{G,U})_Z(\langle g, in_3(q,r)\rangle)))\rangle) = \\ =&p_Z(\langle g, \alpha_Z((Th)_Z(in_3(\langle g,q,g,r\rangle)))\rangle) = \\ =&p_Z(\langle g, \alpha_Z(in_3(h_Z(\langle g,q\rangle), in_3(h_Z(\langle g,r\rangle))))\rangle) = \\ =&p_Z(\langle g, h_Z(\langle g,q\rangle)|h_Z(\langle g,r\rangle)\rangle)\end{aligned}$$

Equation C.5, states that for all $Y \in \mathcal{V}$, and $\eta_R \in (\mathit{Proc} \Rightarrow \mathit{Prop})_Y$, $Y \Vdash_{R:\iota\to o,\eta_R} \forall P{:}\iota.(R\ P) \Rightarrow \forall Q{:}\iota.(R\ Q) \Rightarrow (R\ P|Q)$ holds. By points 1 and 2 of Theorem 5.1, this amounts to say that, for all $V \in \mathcal{V}$, $l \in \mathcal{I}(Y,V)$ and $\eta_P \in \mathit{Proc}_V$,

$$V \Vdash_{(R:\iota\to o,P:\iota),\langle(\mathit{Proc}\Rightarrow\mathit{Prop})_l(\eta_R),\eta_P\rangle} (R\ P)$$

implies

$$V \Vdash_{(R:\iota\to o,P:\iota),\langle(\mathit{Proc}\Rightarrow\mathit{Prop})_l(\eta_R),\eta_P\rangle} \forall Q{:}\iota.(R\ Q) \Rightarrow (R\ P|Q).$$

Applying again the same theorem, the latter judgment is in turn equivalent to say that, for all $W \in \mathcal{V}$, $m \in \mathcal{I}(V,W)$ and $\eta_Q \in \mathit{Proc}_W$,

$$W \Vdash_{(R:\iota\to o,P:\iota,Q:\iota),\langle(\mathit{Proc}\Rightarrow\mathit{Prop})_{m\circ l}(\eta_R),\mathit{Proc}_m(\eta_P),\eta_Q\rangle} (R\ Q)$$

implies

$$W \Vdash_{(R:\iota\to o,P:\iota,Q:\iota),\langle(\mathit{Proc}\Rightarrow\mathit{Prop})_{m\circ l}(\eta_R),\mathit{Proc}_m(\eta_P),\eta_Q\rangle} (R\ P|Q).$$

Then we notice the following facts:

1. $p_Z(\langle g, h_Z(\langle g,q\rangle)\rangle) = \mathcal{I}(Z,\_)$ and $p_Z(\langle g, h_Z(\langle g,r\rangle)\rangle) = \mathcal{I}(Z,\_)$;
2. $p_Z(\langle g, h_Z(\langle g,q\rangle)\rangle) = (ev_{\mathit{Prop},\mathit{Proc}})_Z(\langle g, h_Z(\langle g,q\rangle)\rangle) \wedge \mathcal{I}(Z,\_) = g_Z(\langle h_Z(\langle g,q\rangle), \mathrm{id}_Z\rangle) \wedge \mathcal{I}(Z,\_)$ and analogously $p_Z(\langle g, h_Z(\langle g,r\rangle)\rangle) = g_Z(\langle h_Z(\langle g,r\rangle), \mathrm{id}_Z\rangle) \wedge \mathcal{I}(Z,\_)$;
3. $[\![R : \iota \to o, P : \iota \vdash_\Sigma (R\ P) : o]\!]_Z(\langle g, h_Z(\langle g,q\rangle)\rangle) = ([\![R : \iota \to o, P : \iota \vdash_\Sigma R : \iota \to o]\!]_Z(\langle g, h_Z(\langle g,q\rangle)\rangle))_Z(\langle[\![R : \iota \to o, P : \iota \vdash_\Sigma P : \iota]\!]_Z(\langle g, h_Z(\langle g,q\rangle)\rangle), \mathrm{id}_Z\rangle) = g_Z(\langle h_Z(\langle g,q\rangle), \mathrm{id}_Z\rangle)$ (by point 3 of Theorem 5.1); it follows from the previous two facts that $g_Z(\langle h_Z(\langle g,q\rangle), \mathrm{id}_Z\rangle) \geq \mathcal{I}(Z,\_)$; hence $Z \Vdash_{(R:\iota\to o,P:\iota),\langle g,h_Z(\langle g,q\rangle)\rangle\rangle} (R\ P)$ holds;
4. similarly we have that $[\![R : \iota \to o, P : \iota, Q : \iota \vdash_\Sigma (R\ Q) : o]\!]_Z(\langle g, h_Z(\langle g,q\rangle), h_Z(\langle g,r\rangle)\rangle) = g_Z(\langle h_Z(\langle g,r\rangle), \mathrm{id}_Z\rangle) \geq \mathcal{I}(Z,\_)$; hence $Z \Vdash_{(R:\iota\to o,P:\iota),\langle g,h_Z(\langle g,r\rangle)\rangle\rangle} (R\ Q)$ holds;

5. from the previous facts and the inductive hypothesis we can deduce that

$$Z \Vdash_{(R:\iota\to o,P:\iota,Q:\iota),\langle g,h_Z(\langle g,q\rangle),h_Z(\langle g,r\rangle)\rangle} (R\ P|Q)$$

holds, i.e.,

$$[\![R:\iota\to o,P:\iota,Q:\iota\vdash_\Sigma (R\ P|Q):o]\!]_Z(\langle g,h_Z(\langle g,q\rangle),h_Z(\langle g,r\rangle)\rangle)\geq\mathcal{I}(Z,\_);$$

6. by point 3 of Theorem 5.1, we have $[\![R:\iota\to o,P:\iota,Q:\iota\vdash_\Sigma (R\ P|Q):o]\!]_Z(\langle g, h_Z(\langle g,q\rangle),h_Z(\langle g,r\rangle)\rangle) = ([\![R:\iota\to o,P:\iota,Q:\iota\vdash_\Sigma R:\iota\to o]\!]_Z(\langle g,h_Z(\langle g,q\rangle), h_Z(\langle g,r\rangle)\rangle))_Z(\langle[\![R:\iota\to o,P:\iota,Q:\iota\vdash_\Sigma P|Q:\iota]\!]_Z(\langle g,h_Z(\langle g,q\rangle),h_Z(\langle g,r\rangle)\rangle), \mathrm{id}_Z\rangle) = g_Z(\langle par(\langle[\![R:\iota\to o,P:\iota,Q:\iota\vdash_\Sigma P:\iota]\!]_Z(\langle g,h_Z(\langle g,q\rangle),h_Z(\langle g,r\rangle)\rangle), [\![R:\iota\to o,P:\iota,Q:\iota\vdash_\Sigma Q:\iota]\!]_Z(\langle g,h_Z(\langle g,q\rangle),h_Z(\langle g,r\rangle)\rangle)\rangle),\mathrm{id}_Z\rangle) = g_Z(\langle par(\langle h_Z(\langle g, q\rangle),h_Z(\langle g,r\rangle)\rangle),\mathrm{id}_Z\rangle) = g_Z(h_Z(\langle g,q\rangle)|h_Z(\langle g,r\rangle),\mathrm{id}_Z) = p_Z(\langle g,\ h_Z(\langle g,q\rangle)|h_Z(\langle g, r\rangle)\rangle)\wedge\mathcal{I}(Z,\_)$, whence the thesis.

**($u = in_4(v,w,q)$)**

$$\begin{aligned}&p_Z(\langle\pi_Z(\langle g,in_4(v,w,q)\rangle),\alpha_Z((Th)_Z((st_{G,U})_Z(\langle g,in_4(v,w,q)\rangle)))\rangle) =\\ =&p_Z(\langle g,\alpha_Z((Th)_Z(in_4(\langle v,w,g,q\rangle)))\rangle) =\\ =&p_Z(\langle g,\alpha_Z(in_4(\langle v,w,h_Z(\langle g,q\rangle)\rangle))\rangle) =\\ =&p_Z(\langle g,[v\neq w]h_Z(\langle g,q\rangle)\rangle)\end{aligned}$$

At this point we know, by equation C.6, that for all $Y\in\mathcal{V}$, and $\eta_R\in(Proc\Rightarrow Prop)_Y$, $Y\Vdash_{R:\iota\to o,\eta_R}\forall x{:}\upsilon.\forall y{:}\upsilon.\forall P{:}\iota.(R\ P)\Rightarrow(R\ [x\neq y]P)$ holds. By point 2 of Theorem 5.1 and point 5 of Corollary 5.1, this amounts to say that, for all $V\in\mathcal{V}$, $l\in\mathcal{I}(Y,V)$ $\eta_x,\eta_y\in V$ and $\eta_P\in Proc_V$,

$$V\Vdash_{(R:\iota\to o,x:\upsilon,y:\upsilon,P:\iota),\langle(Proc\Rightarrow Prop)_l(\eta_R),\eta_x,\eta_y,\eta_P\rangle}(R\ P)$$

implies

$$V\Vdash_{(R:\iota\to o,x:\upsilon,y:\upsilon,P:\iota),\langle(Proc\Rightarrow Prop)_l(\eta_R),\eta_x,\eta_y,\eta_P\rangle}(R\ [x\neq y]P).$$

Then we notice the following facts:

1. $p_Z(\langle g,h_Z(\langle g,q\rangle)\rangle)=\mathcal{I}(Z,\_)$;
2. $p_Z(\langle g,h_Z(\langle g,q\rangle)\rangle)=(ev_{Prop,Proc})_Z(\langle g,h_Z(\langle g,q\rangle)\rangle)\wedge\mathcal{I}(Z,\_) = g_Z(\langle\ h_Z(\langle g,q\rangle),\mathrm{id}_Z\rangle)$ $\wedge\mathcal{I}(Z,\_)$;
3. $[\![R:\iota\to o,P:\iota\vdash_\Sigma (R\ P):o]\!]_Z(\langle g,h_Z(\langle g,q\rangle)\rangle) = ([\![R:\iota\to o,P:\iota\vdash_\Sigma R:\iota\to o]\!]_Z(\langle g,h_Z(\langle g,q\rangle)\rangle))_Z(\langle[\![R:\iota\to o,P:\iota\vdash_\Sigma P:\iota]\!]_Z(\langle g,h_Z(\langle g,q\rangle)\rangle),\mathrm{id}_Z\rangle) = g_Z(\langle h_Z(\langle g,q\rangle),\mathrm{id}_Z\rangle)$ (by point 3 of Theorem 5.1); it follows from the previous two facts that $g_Z(\langle h_Z(\langle g,q\rangle),\ \mathrm{id}_Z\rangle)\geq\mathcal{I}(Z,\_)$; hence $Z\Vdash_{(R:\iota\to o,P:\iota),\langle g,h_Z(\langle g,q\rangle)\rangle\rangle}(R\ P)$ holds;
4. from the previous fact and the inductive hypothesis we can deduce that

$$Z\Vdash_{(R:\iota\to o,x:\upsilon,y:\upsilon,P:\iota),\langle g,v,w,h_Z(\langle g,q\rangle)\rangle}(R\ [x\neq y]P)$$

holds, i.e.,

$$[\![R:\iota\to o,x:\upsilon,y:\upsilon,P:\iota\vdash_\Sigma (R\ [x\neq y]P):o]\!]_Z(\langle g,v,w,h_Z(\langle g,q\rangle)\rangle)\geq\mathcal{I}(Z,\_);$$

5. by point 3 of Theorem 5.1, we have $[\![R{:}\iota \to o, x{:}\upsilon, y{:}\upsilon, P{:}\iota \vdash_\Sigma (R\ [x \neq y]P){:}o]\!]_Z$ $(\langle g, v, w, h_Z(\langle g, q\rangle)\rangle) = ([\![R{:}\iota \to o, x{:}\upsilon, y{:}\upsilon, P{:}\iota \vdash_\Sigma R{:}\iota \to o]\!]_Z(\langle g, v, w, h_Z(\langle g, q\rangle)\rangle))_Z(\langle [\![R : \iota \to o, x : \upsilon, y : \upsilon, P : \iota \vdash_\Sigma [x \neq y]P : \iota]\!]_Z(\langle g, v, w, h_Z(\langle g, q\rangle)\rangle), \mathrm{id}_Z\rangle) = g_Z(\langle \mathit{mismatch}(\langle [\![R{:}\iota \to o, x{:}\upsilon, y{:}\upsilon, P{:}\iota \vdash_\Sigma x{:}\upsilon]\!]_Z(\langle g, v, w, h_Z(\langle g, q\rangle)\rangle), [\![R : \iota \to o, x{:}\upsilon, y{:}\upsilon, P{:}\iota \vdash_\Sigma y{:}\upsilon]\!]_Z(\langle g, v, w, h_Z(\langle g, q\rangle)\rangle), [\![R{:}\iota \to o, x{:}\upsilon, y{:}\upsilon, P{:}\iota \vdash_\Sigma P{:}\iota]\!]_Z$ $(\langle g, v, w, h_Z(\langle g, q\rangle)\rangle)\rangle), \mathrm{id}_Z\rangle) = g_Z(\langle \mathit{mismatch}(\langle v, w, h_Z(\langle g, q\rangle)\rangle), \mathrm{id}_Z\rangle) = g_Z([v \neq w]h_Z\ (\langle g, q\rangle), \mathrm{id}_Z) = p_Z(\langle g, [v \neq w]h_Z(\langle g, q\rangle)\rangle) \wedge \mathcal{I}(Z, \_)$, whence the thesis.

$(u = in_5(q))$

$$\begin{aligned}
&p_Z(\langle \pi_Z(\langle g, in_5(q)\rangle), \alpha_Z((Th)_Z((st_{G,U})_Z(\langle g, in_5(q)\rangle)))\rangle) = \\
=&p_Z(\langle g, \alpha_Z((Th)_Z(in_5(\overline{q}_g)))\rangle) = \\
=&p_Z(\langle g, \alpha_Z(in_5(h_{Z\uplus z}((\overline{q}_g)_{Z\uplus z}(z, in_Z))))\rangle) = \\
=&p_Z(\langle g, (\nu z)h_{Z\uplus z}((\overline{q}_g)_{Z\uplus z}(z, in_Z))\rangle),
\end{aligned}$$

where $\overline{q}_g : \mathit{Var} \times \mathcal{V}(Z, \_) \longrightarrow G \times U$ is the natural transformation such that, for all $Y \in \mathcal{V}$, $y \in Y$ and $f \in \mathcal{V}(Z, Y)$, $(\overline{q}_g)_Y(y, f) = \langle G_f(g), q_Y(\langle y, f\rangle)\rangle$.

At this point we know, by equation C.7, that for all $Y \in \mathcal{V}$, and $\eta_R \in (\mathit{Proc} \Rightarrow \mathit{Prop})_Y$, $Y \Vdash_{R:\iota\to o, \eta_R} \forall P{:}\upsilon \to \iota.(\forall x{:}\upsilon.(R\ (P\ x))) \Rightarrow (R\ \nu P)$ holds. By points 1 and 2 of Theorem 5.1, this amounts to say that, for all $V \in \mathcal{V}$, $l \in \mathcal{I}(Y, V)$ and $\eta_P \in (\mathit{Var} \Rightarrow \mathit{Proc})_V$,

$$V \Vdash_{(R:\iota\to o, P:\upsilon\to\iota), \langle (\mathit{Proc}\Rightarrow\mathit{Prop})_l(\eta_R), \eta_P\rangle} \forall x{:}\upsilon.(R\ (P\ x))$$

implies

$$V \Vdash_{(R:\iota\to o, P:\upsilon\to\iota), \langle (\mathit{Proc}\Rightarrow\mathit{Prop})_l(\eta_R), \eta_P\rangle} (R\ \nu P).$$

Then we notice the following facts:

1. $V \Vdash_{(R:\iota\to o, P:\upsilon\to\iota), \langle (\mathit{Proc}\Rightarrow\mathit{Prop})_l(\eta_R), \eta_P\rangle} \forall x{:}\upsilon.(R\ (P\ x))$ iff, for all $W \in \mathcal{V}$, $m \in \mathcal{I}(V, W)$ and $\eta_x \in W$, the following holds:

$$W \Vdash_{(R:\iota\to o, P:\upsilon\to\iota, x:\upsilon), \langle (\mathit{Proc}\Rightarrow\mathit{Prop})_{m\circ l}(\eta_R), \eta_P, \eta_x\rangle} (R\ (P\ x)),$$

   i.e., iff

$$[\![\Delta \vdash_\Sigma (R\ (P\ x)) : o]\!]_W(\eta) \geq \mathcal{I}(W, \_),$$

   where $\Delta \triangleq R : \iota \to o, P : \upsilon \to \iota, x : \upsilon$ and $\eta \triangleq \langle (\mathit{Proc} \Rightarrow \mathit{Prop})_{m\circ l}(\eta_R), (\mathit{Var} \Rightarrow \mathit{Proc})_m(\eta_P), \eta_x\rangle$. The first member of the preceding inequality can be simplified as follows according to Theorem 5.1:

$$\begin{aligned}
&[\![\Delta \vdash_\Sigma (R\ (P\ x)) : o]\!]_W(\eta) \geq \mathcal{I}(W, _) \\
=&([\![\Delta \vdash_\Sigma R : \iota \to o]\!]_W(\eta))_W(\langle [\![\Delta \vdash_\Sigma (P\ x) : \iota]\!]_W(\eta), \mathrm{id}_W\rangle) \\
=&((\mathit{Proc} \Rightarrow \mathit{Prop})_{m\circ l}(\eta_R))_W(\langle (([\![\Delta \vdash_\Sigma P : \upsilon \to \iota]\!]_W(\eta))_W \\
&\quad (\langle [\![\Delta \vdash_\Sigma x : \upsilon]\!]_W(\eta), \mathrm{id}_W\rangle), \mathrm{id}_W\rangle) \\
=&((\mathit{Proc} \Rightarrow \mathit{Prop})_{m\circ l}(\eta_R))_W(\langle (((\mathit{Var} \Rightarrow \mathit{Proc})_m(\eta_P))_W(\langle \eta_x, \mathrm{id}_W\rangle), \mathrm{id}_W\rangle)
\end{aligned}$$

2. in particular, when $V \triangleq Z$, $l \triangleq \mathrm{id}_Z$, $\eta_R \triangleq g$ and $\eta_P \triangleq h \circ \overline{q}_g$, we have that the following holds:

$$\begin{aligned}
&((\mathit{Proc} \Rightarrow \mathit{Prop})_m(g))_W(\langle (((\mathit{Var} \Rightarrow \mathit{Proc})_m(h \circ \overline{q}_g))_W(\langle \eta_x, \mathrm{id}_W\rangle), \mathrm{id}_W\rangle) \\
=&((\mathit{Proc} \Rightarrow \mathit{Prop})_m(g))_W(\langle (h \circ \overline{q}_g)_W(\langle \eta_x, m\rangle), \mathrm{id}_W\rangle) \\
=&((\mathit{Proc} \Rightarrow \mathit{Prop})_m(g))_W(\langle h_W(\langle (\mathit{Proc} \Rightarrow \mathit{Prop})_m(g), q_W(\langle \eta_X, m\rangle)\rangle), \mathrm{id}_W\rangle)
\end{aligned}$$

3. $p_W(\langle (Proc \Rightarrow Prop)_m(g), h_W(\langle (Proc \Rightarrow Prop)_m(g), q_W(\langle \eta_X, m\rangle)\rangle), \mathrm{id}_W\rangle\rangle) = \mathcal{I}(W, \_)$;
4. $p_W(\langle (Proc \Rightarrow Prop)_m(g), h_W(\langle (Proc \Rightarrow Prop)_m(g), q_W(\langle \eta_X, m\rangle)\rangle), \mathrm{id}_W\rangle\rangle) = ((Proc \Rightarrow Prop)_m(g))_W(\langle h_W(\langle (Proc \Rightarrow Prop)_m(g), q_W(\langle \eta_X, m\rangle)\rangle), \mathrm{id}_W\rangle) \wedge \mathcal{I}(W, \_)$; hence, for all $W$, $m \in \mathcal{I}(Z, W)$ and $\eta_x \in W$ we have
$$W \Vdash_{(R:\iota\to o, P:\upsilon\to\iota, x:\upsilon), \langle (Proc\Rightarrow Prop)_m(g), h\circ\overline{q}_g, \eta_x\rangle} (R\ (P\ x));$$
5. it follows that $Z \Vdash_{(R:\iota\to o, P:\upsilon\to\iota), \langle g, h\circ\overline{q}_g\rangle} (R\ \nu P)$ holds by the previous point and the inductive hypothesis, i.e, $[\![R:\iota \to o, P:\upsilon \to \iota \vdash_\Sigma (R\ \nu P) : \iota]\!]_Z(\langle g, h\circ\overline{q}_g\rangle) = ([\![R:\iota \to o, P:\upsilon \to \iota \vdash_\Sigma R:\iota \to o]\!]_Z(\langle g, h \circ \overline{q}_g\rangle))_Z(\langle [\![R:\iota \to o, P:\upsilon \to \iota \vdash_\Sigma \nu P : \iota]\!]_Z(\langle g, h \circ \overline{q}_g\rangle), \mathrm{id}_Z\rangle) = g_Z(\langle new_Z(\langle h \circ \overline{q}_g\rangle), \mathrm{id}_Z\rangle) = g_Z(\langle (\nu z)((h \circ \overline{q}_g)_{Z\uplus z}(\langle z, in_Z\rangle)), \mathrm{id}_Z\rangle) = g_Z(\langle (\nu z)(h_{Z\uplus z}((\overline{q}_g)_{Z\uplus z}(\langle z, in_Z\rangle))), \mathrm{id}_Z\rangle) \geq \mathcal{I}(Z, \_)$ holds. The thesis follows since
$$\begin{aligned} & p_Z(\langle g, (\nu z) h_{Z\uplus z}((\overline{q}_g)_{Z\uplus z}(z, in_Z))\rangle) \\ = \; & g_Z(\langle (\nu z)(h_{Z\uplus z}((\overline{q}_g)_{Z\uplus z}(\langle z, in_Z\rangle))), \mathrm{id}_Z\rangle) \wedge \mathcal{I}(Z, \_). \end{aligned}$$

# Bibliography


[AG99] Martín Abadi and Andrew D. Gordon. A Calculus for Cryptographic Protocols: the Spi Calculus. *Information and Computation*, 148:1–70, 1999.

[AHMP92] 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.

[Bar81] Henk P. Barendregt. *The Lambda Calculus: its Syntax and Semantics.* North Holland, 1981.

[Bar92] Henk P. Barendregt. Lambda Calculi with Types. In Samson Abramsky, Dov Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

[BB92] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, 1992.

[Bel88] J. L. Bell. *Toposes and Local Set Theories. An Introduction.* Clarendon Press, Oxford, 1988.

[BGG+92] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. Van Tassel. Experience with embedding hardware description languages in HOL. In *Proceedings of the IFIP TC10/WG 10.2 international conference on theorem provers in circuit design: Theory, practice and experience*, volume A-10 of *IFIP Transactions*, pages 129–156. Elsevier, 1992.

[BHH+01] Anna Bucalo, Martin Hofmann, Furio Honsell, Marino Miculan, and Ivan Scagnetto. Consistency of the Theory of Contexts. Submitted, 2001.

[BW90] Michael Barr and Charles F. Wells. *Category theory for computing science.* Prentice-Hall, 1990.

[CC01] L. Caires and L. Cardelli. A Spatial Logic for Concurrency (Part I). In N. Kobayashi and B. C. Pierce, editors, *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001, Sendai, Japan, October 29-31, 2001, Proceedings*, volume 2215 of *Lecture Notes in Computer Science*, pages 1–38. Springer-Verlag, Berlin, 2001.

[CG98] Luca Cardelli and Andrew D. Gordon. Mobile Ambients. In *Foundations of Software Science and Computational Structures*, volume 1378, pages 140–155. Springer, 1998.

[CG00] Luca Cardelli and Andrew D. Gordon. Anytime, Anywhere. Modal Logics for Mobile Ambients. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages*, pages 365–377, 2000.

[CG01] Luca Cardelli and Andrew D. Gordon. Logical Properties of Name Restriction. In S. Abramsky, editor, *Typed Lambda Calculi and Applications*, volume 2044 of *Lecture Notes in Computer Science*, page 46. Springer, May 2001.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Control*, 76:95–120, 1988.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.

[Chu33] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 2(33-34):346–366 and 839–864, 1932/33.

[Coq93] Thierry Coquand. Infinite Objects in Type Theory. In *Proceedings of TYPES'93*, pages 62–78, 1993.

[CP90] Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and G. Mints, editors, *Proceedings of COLOG 88*, volume 417 of *LNCS*, pages 50–66. Springer-Verlag, 1990.

[CSW97] Gian Luca Cattani, Ian Stark, and Glynn Winskel. Presheaf models for the $\pi$-calculus. In *Proceedings of CTCS*, 1997.

[Cur34] H. B. Curry. Functionality in combinatory logic. In *Proceedings of National Academy of Sciences*, volume 20, pages 584–590, USA, 1934.

[Dal00] Silvano Dal Zilio. Spatial congruence for Ambients is Decidable. Technical Report MSR-TR-2000-41, Microsoft Research, May 2000.

[dB70] N. G. de Bruijn. The mathematical language AUTOMATH, its usage and some of its extensions. In *Symposium on automatic demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 29–61, Versailles 1968, 1970. Springer.

[DFH95] Joëlle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order syntax in Coq. In *Proceedings of TLCA'95*, volume Lecture Notes in Computer Science, vol. 905, Edinburgh, 1995. Springer-Verlag. Also appears as INRIA research report RR-2556, April 1995.

[DH94] Joëlle Despeyroux and André Hirschowitz. Higher-order syntax and induction in Coq. In *Proceedings of LPAR'94*, Kiev, Ukraine, July 1994. Also appears as INRIA research report RR-2292, June 1994.

[DPS96] Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive Recursion for Higher-Order Abstract Syntax. Technical Report CMU-CS-96-172, Carnegie Mellon University, 1996.

[FMS96] Marcelo Fiore, Eugenio Moggi, and Davide Sangiorgi. A fully-abstract model for the $\pi$-calculus. In *Proceedings of 11th LICS*. IEEE, 1996.

[FPT99] Marcelo P. Fiore, Gordon D. Plotkin, and Daniele Turi. Abstract syntax and variable binding. In *Proceedings of LICS99*, 1999.

[FT01] Marcelo Fiore and Daniele Turi. Semantics of name and value passing. In Harry Mairson, editor, *Proceedings of 16th LICS*, pages 93–104, Boston, USA, 2001. IEEE Computer Society Press, for The Institute of Electrical and Electronics Engineers, Inc.

[Gab00] Murdoch J. Gabbay. *A Theory of Inductive Definitions With $\alpha$-equivalence*. PhD thesis, Trinity College, Cambridge University, 2000.

[Gen69] G. Gentzen. Investigations into logical deduction. In M. Szabo, editor, *The collected papers of Gerhard Gentzen*, pages 68–131. North Holland, 1969.

[Geu93] Jan Herman Geuvers. *Logics and Type Systems*. PhD thesis, Katholieke Universiteit, Nijmegen, the Netherlands, 1993.

[Gim94] Eduardo Giménez. Codifying guarded recursion definitions with recursive schemes. In *Proceedings of TYPES'94*, 1994.

[Gim96] Eduardo Giménez. *Un Calcul de Constructions Infinies et son Application à la Vérification de Systèmes Communicants*. PhD thesis, École Normale Supérieure de Lyon, December 1996.

[Gir72] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[GM96] A. Gordon and T. Melham. Five axioms of $\alpha$-conversion. In *Proceedings of TPHOL'96*, volume 1125 of *Lecture Notes in Computer Science*, pages 173–190. Springer Verlag, 1996.

[GP99] M. J. Gabbay and A. M. Pitts. A New Approach to Abstract Syntax Involving Binders. In *14th Annual Symposium on Logic in Computer Science*, pages 214–224. IEEE Computer Society Press, Washington, 1999.

[GP01] M. J. Gabbay and A. M. Pitts. A New Approach to Abstract Syntax with Variable Binding. *Formal Aspects of Computing*, ?:?–?, 2001. Special issue in honour of Rod Burstall. To appear.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, January 1993.

[Hir97] D. Hirschkoff. Bisimulation proofs for the $\pi$-calculus in the Calculus of Constructions. In *Proc. TPHOL'97*, number 1275 in Lecture Notes in Computer Science. Springer-Verlag, 1997.

[HJP80] J. M. E. Hyland, P. T. Johnstone, and Andrew M. Pitts. Tripos theory. In *Mathematical Proceedings of the Cambridge Philosophical society*, volume 88, pages 205–232, 1980.

[HMS01a] Furio Honsell, Marino Miculan, and Ivan Scagnetto. An axiomatic approach to metareasoning on systems in higher-order abstract syntax. In *LNCS, Proceedings of ICALP'01*, volume 2076, pages 963–978. Springer-Verlag, 2001. Also available at `http://www.dimi.uniud.it/~miculan/Papers/`.

[HMS01b] Furio Honsell, Marino Miculan, and Ivan Scagnetto. $\pi$-calculus in (co)inductive type theory. *Theoretical computer science*, 239-285(2):239–285, 2001. First appeared as a talk at TYPES'98 annual workshop.

[Hof99] Martin Hofmann. Semantical analysis of higher-order abstract syntax. In Giuseppe Longo, editor, *Proceedings of fourteenth annual ieee symposium on logic in computer science*, Trento, Italy, 1999. IEEE Computer Society Press, for The Institute of Electrical and Electronics Engineers, Inc.

[How80] W. A. Howard. The formulæ-as-types notion of construction. In J. R. Hindley and J. P. Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic. Lambda Calculus and Formalism.* Academic Press, 1980.

[Hue92] Gérard Huet. Constructive computation theory - part I. Lecture notes, October 1992.

[Hue94] Gérard Huet. Residual theory in $\lambda$-calculus: a formal development. *Journal of Functional Programming*, 4(3):371–394, 1994.

[Jac95] Bart Jacobs. Parameters and parametrization in specification using distributive categories. *Fundamenta informaticae*, 24(3), 1995.

[Jac99] Bart Jacobs. *Categorical logic and type theory*, volume 14 of *Studies in Logic and the Foundations of Mathematics.* Elsevier, 1999.

[Joh77] Peter Johnstone. Topos theory. In *London Mathematical Society Monographs*, volume 10. Academic Press, London, 1977.

[KR35] S. C. Kleene and J. B. Rosser. The inconsistency of certain formal logics. *Annals of Mathematics*, 2(36):630–636, 1935.

[Mac71] Saunders MacLane. *Categories for the working mathematician.* Springer-Verlag, Berlin, 1971.

[Mel95] T. Melham. A mechanized theory of the $\pi$-calculus in HOL. *Nordic journal of computing*, 1(1):50–76, 1995.

[Mic97] Marino Miculan. *Encoding logical theories of programs.* PhD thesis, Dipartimento di Informatica, Università di Pisa, Pisa, Italy, March 1997.

[Mic01a] Marino Miculan. Developing (meta)theory of $\lambda$-calculus in the theory of contexts. Technical Report 2001/26, Department of Mathematics and Computer Science, University of Leicester, Siena, 2001.

[Mic01b] Marino Miculan. On the formalization of the modal $\mu$-calculus in the calculus of inductive constructions. *Information and Computation*, 164(1):199–231, January 2001.

[Mil93] Robin Milner. The polyadic $\pi$-Calculus: a Tutorial. In *Logic and Algebra of Specification*, volume 94 of *NATO ASI Series F*. Springer, Berlin, 1993.

[ML85] Per Martin-Löf. On the meaning of the logical constants and the justifications of the logic laws. Technical Report 2, Scuola di Specializzazione in Logica Matematica, Dipartimento di Matematica, Università di Siena, 1985.

[MM92] Saunders MacLane and Ieke Moerdijk. *Sheaves in geometry and logic: a first introduction to topos theory*. Universitext. Springer-Verlag, 1992.

[MM01] Raymond McDowell and Dale Miller. Reasoning with higher-order abstract syntax in a logical framework. *Acm transactions on computational logic*, 2001. To appear.

[Mog89] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, LFCS, University of Edinburgh, 1989.

[MP99] James McKinna and Robert Pollack. Some Lambda Calculus and Type Theory Formalized. *Journal of Automated Reasoning*, 23(3-4), November 1999. Special Issue on Formal Proof edited by Frank Pfenning.

[MPW89] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes. Technical Report ECS-LFCS-89-85, Dept. of Computer Science, University of Edinburgh, June 1989.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes. *Information and Computation*, 100(1):1–77, 1992.

[Ole85] Frank J. Oles. Type categories, functor categories and block structure. In M. Nivat and J. C. Reynolds, editors, *Algebraic semantics*. Cambridge University Press, 1985.

[Pit81] Andrew M. Pitts. *The theory of triposes*. PhD thesis, Cambridge University, 1981.

[Pit99] Andrew M. Pitts. Tripos theory in retrospect. In L. Birkedal and G. Rosolini, editors, *Tutorial workshop on realizability semantics, FLoC'99*, volume 23 of *Electronic Notes in Theoretical Computer Science*, Trento, Italy, 1999. Elsevier.

[Pit00] Andrew M. Pitts. Categorical logic. In S Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of logic in computer science*, volume 5. Oxford University Press, 2000.

[Pit01a] A. M. Pitts. Nominal logic: A first order theory of names and binding. In N. Kobayashi and B. C. Pierce, editors, *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001, Sendai, Japan, October 29-31, 2001, Proceedings*, volume 2215 of *Lecture Notes in Computer Science*, pages 219–242. Springer-Verlag, Berlin, 2001.

[Pit01b] Andrew M. Pitts. A First Order Theory of Names and Binding. Invited talk at IJCAR 2001 Workshop on Mechanized Reasoning about Languages with variable binding (MERLIN 2001), Siena, June 2001.

[PM93] Christine Paulin-Mohring. Inductive definitions in the system Coq; rules and properties. In Mark Bezem and Jan Friso Groote, editors, *Proceedings of International Conference on Typed Lambda Calculi and Applications*, volume 664 of *LNCS*, pages 328–345. Springer-Verlag, 1993. Also appears as LIP research report 92-49.

[Rey81] John C. Reynolds. The essence of Algol. In *Algorithmic languages. Proceedings of ACM Annual Conference*, pages 345–372. North-Holland, 1981.

[RHB01] Christine Röckl, Daniel Hirschkoff, and Stefan Berghofer. Higher-order abstract syntax with induction in Isabelle/HOL: Formalising the $\pi$-calculus and mechanizing the theory of contexts. In Furio Honsell and Marino Miculan, editors, *Proceedings of FoSSaCS 2001*, volume 2030 of *Lecture Notes in Computer Science*, pages 359–373, Genova, 2001. Springer-Verlag.

[Sch01] Carsten Schürmann. Recursion for Higher-Order Encodings. In *Proceedings of Computer Science Logic (CSL 2001)*, volume 2142 of *Lecture Notes In Computer Science*, pages 585–599, Paris, France, 2001.

[Sta96] Ian Stark. A fully abstract domain model for the $\pi$-calculus. In *Proceedings of LICS'96*, pages 36–42. IEEE, 1996.

[Tay88] Paul Taylor. Using Constructions as a metalanguage. Technical Report ECS-LFCS-88-70, Department of Computer Science, University of Edinburgh, December 1988.

[TCDT01] The Coq Development Team. *The Coq Proof Assistant Reference Manual - Version 7.1.* INRIA, Rocquencourt, France, October 2001. Available at `ftp://ftp.inria.fr/INRIA/coq/V7.1/doc`.

[vO91] Jaap van Oosten. *Exercises in realizability.* PhD thesis, Department of Mathematics and Computer Science, University of Amsterdam, 1991.

[Wer94] Benjamin Werner. *Une théorie des constructions inductives.* PhD thesis, Université Paris 7, 1994.