Semantical analysis of higher-order abstract syntax

Martin Hofmann*

Synopsis: A functor category semantics for higher-order abstract syntax is proposed with the following aims: relating higher-order and first order syntax, justifying induction principles, suggesting new logical principles to reason about higher-order syntax.

1 Introduction

It is the aim of this paper to advocate the use of functor categories as a semantic foundation of higher-order abstract syntax (HOAS). By way of example, we will show how functor categories can be used for at least the following applications:

- relating first-order and higher-order abstract syntax (proofs of adequacy) without using reduction rules,
- justifying induction principles and other axioms assumed in conjunction with HOAS,
- suggesting new logical principles to reason about HOAS,
- making precise the relationship between proofs involving HOAS and properties of first-order syntax (what exactly does this or that HOAS proof prove?)

The main aim of this paper is of pedagogical nature; it is our hope that practical workers in the field will take the time to familiarise themselves with the required category-theoretic background and that the methods described in this and related papers by other authors (see Section 2 below) will soon make their way into the standard toolbox of people studying syntax of programming languages and similar systems.

1.1 Higher-order abstract syntax

Higher-order abstract syntax (HOAS) aims at representing the syntax of logical systems, in particular programming languages in a variable-free way or rather in a way which avoids explicit treatment of object-level variables. For example, the syntax of the untyped lambda calculus can be specified follows.

$$\begin{array}{rcl} tm & ::= & app : tm, tm \to tm \\ & \mid & lam : (tm \to tm) \to tm \end{array} \tag{1}$$

This specification means that the type tm of lambda expressions has two constructors: app and lam, the first of which is an ordinary binary first-order constructor, whereas the second one is a higher-order constructor which takes a meta-level function as argument. Using brackets for meta-level abstraction and parenthesisation with commas for possibly iterated meta-level application we can write down closed terms of type tm like

$$\begin{array}{l} \mathbf{t} \stackrel{\text{def}}{=} lam([x{:}\, tm]lam([y{:}\, tm]x)) \\ \mathbf{f} \stackrel{\text{def}}{=} lam([x{:}\, tm]lam([y{:}\, tm]y)) \\ \mathbf{1} \stackrel{\text{def}}{=} lam([x{:}\, tm]lam([f{:}\, tm]app(f, x))) \end{array} \tag{2}$$

These correspond to the lambda terms $\lambda xy.x$, $\lambda xy.y$, and $\lambda xf.fx$ in standard notation. More generally, we can inductively define an encoding function $\ulcorner - \urcorner$ such that, when $t(\vec{x})$ is a λ-term with free variables $\vec{x} = x_1, \ldots, x_n$ then $\ulcorner t(\vec{x}) \urcorner$ is a term of type tm involving free meta-level variables $x_1, \ldots, x_n$ of type tm.

The advantage of this higher-order syntax as opposed to a first-order formal description of lambda terms like

$$\begin{array}{rcl} tm & ::= & var : var \to tm \\ & \mid & app : tm, tm \to tm \\ & \mid & lam : var, tm \to tm \end{array} \tag{3}$$

is that in higher-order syntax substitution and weakening can be inherited from the metalanguage. For instance, we can specify beta reduction by saying that $app(lam(f), t)$ should reduce to $f(t)$, whereas in first-order syntax we have to talk about free and bound variables and define capture-free substitution in order to state such a reduction rule. More detailed references on HOAS are [3, 9].

As is well-known, this simplicity comes for a price. In order to retain conservativity of the higher-order syntax over the first-order one we need to restrict the function space in arguments of higher-order constructors like lam to definable ones and therefore we lose the possibility of defining

*LFCS Edinburgh, Mayfield Rd, Edinburgh EH9 3JZ, UK. E-mail: mxh@dcs.ed.ac.uk

functions on lambda terms by recursion or case distinction. For example, if we would allow the definition of a function $is\_app : tm \to tm$ by the clauses

$$\begin{array}{lcl} is\_app(app(t,t')) & = & \mathbf{t} \\ is\_app(lam(f)) & = & \mathbf{f} \end{array} \tag{4}$$

Then $lam([x\colon tm](is\_app)(x)) : tm$ and this term, albeit closed, does not correspond to an ordinary lambda term.

1.2 Induction principle for HOAS

The situation is less clear with induction principles on the propositional level. Suppose that our metalanguage contains a type *Prop* of propositions closed under impredicative universal quantification An example of such a metalanguage is the COQ system [2], see [11, 4] for examples of HOAS developments in COQ.

In such a system we can formulate the following induction principle for untyped lambda terms.

$$\begin{array}{l} IND \stackrel{\text{def}}{=} \forall P\colon tm \to \mathit{Prop} \\ (\forall t, t'\colon tm.P(t) \Rightarrow P(t') \Rightarrow P(app(t,t'))) \Rightarrow \\ (\forall f\colon tm \to tm.(\forall t\colon tm.P(t) \Rightarrow P(f(t))) \Rightarrow \\ \quad P(lam(f))) \Rightarrow \quad \forall t\colon tm.P(t) \end{array} \tag{5}$$

Is this principle "sound" and if so, what does it really mean?

Concerning the question of soundness we see immediately that it depends on the nature of the propositions, or rather predicates, the outermost universal quantification of IND ranges over.

If these are so strong that functions can be defined by their (*Prop*-valued) graphs then it becomes unsound as we can in this case recover the definition of is_app. More formally, this happens if we assume the principle of unique choice $AC!$ familiar from topos theory:

$$\begin{array}{ll} AC! \stackrel{\text{def}}{=} & \forall A, B\colon Set.\forall R\colon A \to B \to \mathit{Prop}. \\ & (\forall a\colon A.\exists! b\colon B.R(a,b)) \to \\ & \quad \exists f\colon A \to B.\forall a\colon A.R(a, f(a)) \end{array} \tag{6}$$

Here $\exists!$ means unique existence. It can be defined from universal quantification using the familiar encoding of existential quantification and Leibniz equality. We can now use the defining clauses of is_app to define its graph as a relation $IS\_APP(t,t')$ which we can prove to be functional using IND and hence obtain the function is_app using $AC!$. The logic then becomes unsound in the sense that it ascertains the existence of elements of type tm which do not correspond to object-level terms. It is possible to formulate $AC!$ in an even stronger form which actually gives a term of type $A \to B$ rather than merely asserting its existence. Formally, this can be done by replacing the latter $\exists$-quantifier by a Σ-type. With such a formulation even the basic adequacy property that closed terms of type tm correspond to object-level lambda terms fails.

On the other hand, the principle IND trivially does not harm adequacy if we decree that *Prop* contains only one, namely a true proposition. Then, of course, propositions have no actual meaning.

We will show that there exists a sensible compromise: an interpretation of propositions which achieves soundness of IND yet does have the property that propositions are meaningful, in particular existential statements really mean existence of object-level terms. Of course $AC!$ is not valid under this interpretation.

1.3 Modal operators

Another approach to marrying HOAS with induction has recently been put forward by Pfenning et al. [5]. They argue that a function like is_app does make sense when applied to closed terms only. In order to "reify" this restriction they introduce a modal operator $\Box$ operating on types in the metalanguage. The idea is that in the running example $\Box tm$ should denote the set of closed terms and, more generally, $\Box(tm^n \to tm)$ denotes the set of terms in at most n-free variables. Here and in the sequel we write $A^n \to B$ for the iterated function space $A \to A \to \ldots \to A \to B$.

The operator $\Box$ obeys a type-theoretic version of intuitionistic S4 modal logic. In particular, if $e : A$ is a term all of whose free variables have a type of the form $\Box C$ then we obtain a term of type $\Box A$ corresponding to the necessitation rule. Intuitively, this means that when we substitute a closed term for each free variable of a term then we end up with a closed term.

The proposed iteration principle then allows one to define a function $f = It_A(H\_app, H\_lam)$ of type $\Box tm \to A$ from functions $H\_app : A \to A \to A$ and $H\_lam : (A \to A) \to A$. The idea is that when we apply f to a closed term e of type tm then in order to obtain $f(e) : A$ we normalise e and then literally replace each instance of app with H_app and every instance of H_lam with lam thus giving a term of type A.

The iterator is then brought to use in fascinating examples like a function of type $\Box tm \to \mathit{nat}$ counting the number of bound variables.

In *loc. cit.* the iteration principle is justified using a syntactic method which makes the above informal reasoning precise. Below, we will see how functor categories can be used to give a very intuitive interpretation of the modal operator and the iteration principle which provides a "reduction-free" proof of adequacy and also suggests generalisations of the iteration principle to dependent types and to propositions.

1.4 HOAS for alpha conversion

Fed up with the difficulties with HOAS and induction Despeyroux *et al.* [4] have proposed a different encoding of lambda terms which gives up the built-in treatment of substitution yet retains the built-in treatment of α-conversion and the handling of free and bound variables. They assume a type var of "variables" or "names" and define lambda terms as an inductive type with the following constructors:

$$\begin{aligned} &var : var \to tm \\ &app : tm \to tm \to tm \\ &lam : (var \to tm) \to tm \end{aligned} \tag{7}$$

So, a λ-term is viewed as either a variable, or an application of two lambda terms, or a function from variables to lambda terms. Notice that this is an ordinary positive inductive definition; there are no negative occurrences of the inductively defined type tm in arguments to the constructors as would be the case if we would understand the usual HOAS definition of lambda terms as an inductive one.

Again, there is an obvious mapping of closed λ-terms to closed terms of type tm, e.g.,

$$\begin{aligned} \mathbf{t} &\stackrel{\text{def}}{=} lam([x\colon var]lam([y\colon var]var(x))) \\ \mathbf{f} &\stackrel{\text{def}}{=} lam([x\colon var]lam([y\colon var]var(y))) \end{aligned} \tag{8}$$

Whether or not we have adequacy, i.e., whether every closed term of type tm denotes (i.e., can be reduced to) the encoding of an object-level term, depends on what exactly is the type var of variables. If—as has been done in [4]—we take $var = nat$, i.e., the type of natural numbers, then adequacy is lost since we can then define functions of type $var \to tm$ by case distinction, recursion, etc.

If, on the other hand, as suggested in [12], we leave the type of variables unspecified, e.g. take var as a variable of type Set then adequacy does hold as can be seen by a syntactic argument.

The problem with this approach is that substitution is not only no longer for free, but indeed, it is not even definable; at least not as a function $subst : (var \to tm) \to tm \to tm$ which takes a term with a designated hole and a term to substitute for that hole.

The solution of [4] to this problem consists of defining for each $n \in \mathbb{N}$ a predicate WF on $var^n \to tm$ singling out certain "well-formed" terms and then defining substitution as a functional relation on "well-formed" terms.

However, the functor-category approach provides us with another solution, namely it shows that we can soundly assume that not only tm, but also the types $var^n \to tm$ validate an induction principle which in particular allows us to view the following equations which a putative substitution function should satisfy as an inductive definition:

$$\begin{aligned} &subst(var, u) = u \\ &subst([x\colon var]var(y), u) = var(y) \\ &subst([x\colon var]app(t_1x, t_2x), u) = \\ &\quad app(subst(t_1, u), subst(t_2, u)) \\ &subst([x\colon var]lam(tx), u) = \\ &\quad lam([y\colon var]subst([y\colon var]t(x, y), u)) \end{aligned} \tag{9}$$

Notice, how renaming of bound variables is here delegated to the metalanguage.

1.5 Equality of variables

Honsell *et al.* [11] have recently proposed an encoding of the π-calculus in COQ which is similar in spirit to the approach from [4].

Here π-calculus processes are presented as an inductive datatype $proc$ with one constructor corresponding to each defining clause in the concrete first-order syntax. The constructors involving higher-order are the ones corresponding to input and "new":

$$\begin{aligned} &nil : proc \\ &out : name \to name \to proc \to proc \\ &in : name \to (name \to proc) \to proc \\ &new : (name \to proc) \to proc \\ &\ldots \end{aligned} \tag{10}$$

Consider, for example, the process $P \stackrel{\text{def}}{=} x(y).\nu z.\bar{y}z.0$ which reads a channel name y along a channel x creates a new name (z) and outputs it along the just received channel y. Its encoding is

$$P \stackrel{\text{def}}{=} in(x, [y\colon name]new([z\colon name]out(y, z, nil))) \tag{11}$$

In order to formulate the operational semantics of the π-calculus as a relation on processes one needs equality of names and also an extensionality principle for processes involving free names. They add these in the form of unproved axioms and argue intuitively for their soundness without, however, being able to provide a rigorous proof.

Using functor categories it is possible to justify all of their axioms and to derive adequacy of their encoding.

To keep this abstract within the page limits we reformulate the salient features of a selection of their axioms in the context of the encoding of the untyped lambda calculus (7). In the full paper the π-calculus encoding from [11] will be studied in more detail.

Suppose that we postulate that Leibniz equality of variables is decidable:

$$DEC \stackrel{\text{def}}{=} \forall x, y\colon var. x{=}y \vee \neg x{=}y \tag{12}$$

together with

$$EXT \stackrel{\text{def}}{=} \forall p, q\colon var \to tm.\ \forall x\colon var. x \notin p \wedge x \notin q \to p(x) = q(x)) \Rightarrow p{=}q \tag{13}$$

Here, $x \notin p$ means that x does not occur freely in p. This predicate admits an inductive definition using equality of variables for the cases $var(x)$ and $lam(f)$, see [11] for details. The principle EXT expresses that a bound variable can be chosen distinct from all names occurring freely in the body. (In the context of π-calculus the corresponding principles ascertains freshness of new names provided by the ν-binder.)

Again, it is easy to see that these principles are in conflict with unique choice $AC!$ because that principle would allow us to define the characteristic function of equality on names

$$eq : var \to var \to nat \tag{14}$$

with $\forall x, y\colon var. x{=}y \iff eq(x, y){=}1$ and that in turn would give rise to "exotic terms" like

$$Q \stackrel{\text{def}}{=} [x\colon var]\mathsf{if}\ eq(x, y)\ \mathsf{then}\ u\ \mathsf{else}\ v \tag{15}$$

where $y\colon var$ and $u, v\colon tm$ are variables.

This in itself would not be so bad, but from EXT we could conclude that, in fact,

$$Q = [x\colon var]v \tag{16}$$

Therefore, $u = v$ by inserting y for x, so all terms would be Leibniz-equal.

As we shall see, without *AC!* these axioms can be soundly and sensibly interpreted.

2 Related work

The idea of using functor categories to describe binding of variables and freshness seems to be "in the air". In a semantical context it has been around for a while, notably in the theory of idealised Algol [16, 15] and in semantic models of the π-calculus [18, 6]. The only place in the literature where functor categories have been used explicitly to justify higher-order syntax is [10]. It is, however, fair to say that the possibility of using functor categories for HOAS is part of the folklore. Indeed, I believe that most if not all of the results to be elaborated in this paper are known to one or the other person who has used functor categories in a semantic context. The point of this paper is mostly to make these techniques available to syntactically minded people who want to use HOAS to describe and reason about programming languages.

During the work on this paper I have become aware of two independent projects of a similar nature which appear in this volume. One is Pitts and Gabbay's proposal to use ZF set theory with atoms as a meta-language for syntax with variable binding [8]. The other one is [7] which proposes and studies presheaves over the category of finite sets as a model for a higher-order syntax in the style of [4]. There is an overlap between this paper and the first part of Section 7 below.

3 Functor categories

Let $\mathbb{C}$ be a small category with cartesian products and terminal object. A presheaf over $\mathbb{C}$ consists of a family of sets $(A_X)_{X\in\mathbb{C}}$ indexed by objects of $\mathbb{C}$ and for each morphism $f \in \mathbb{C}(X, Y)$ a function $A_f : A_Y \to A_X$ such that $A_{id}(a) = a$ and $A_{g\circ f}(a) = A_f(A_g(a))$.

A morphism or natural transformation from presheaf A to presheaf B is a family of functions $(m_X)_{X\in\mathbb{C}}$ such that $m_X : A_X \to B_X$ and for each $f : X \to Y$ and $a \in A_Y$ we have $m_Y(A_f(a)) = B_f(m_X(a))$ (naturality). In this way, the presheaves form a category: the functor category $\widehat{\mathbb{C}} = \mathcal{S}et^{\mathbb{C}^{op}}$.

The category $\widehat{\mathbb{C}}$ has cartesian products and a terminal object given pointwise by $(A \times B)_X = A_X \times B_X$ and $\top_X = \{\star\}$.

For each object $Y \in \mathbb{C}$ we have the so-called *representable presheaf* $\mathcal{Y}(Y) \in \widehat{\mathbb{C}}$ given by $\mathcal{Y}(Y)_X = \mathbb{C}(X, Y)$ and $\mathcal{Y}(Y)_f(u) = u \circ f$. This assignment extends to a full and faithful functor $\mathcal{Y} : \mathbb{C} \longrightarrow \widehat{\mathbb{C}}$ called *Yoneda embedding*, the morphism part is given by post-composition. If $m : \mathcal{Y}(X) \longrightarrow \mathcal{Y}(Y)$ then $m = \mathcal{Y}(m_X(id_X))$ by naturality thus establishing fullness. The Yoneda embedding preserves cartesian products as $\mathcal{Y}(U \times V)_Z \cong \mathbb{C}(Z, U) \times \mathbb{C}(Z, V) = \mathcal{Y}(U)_Z \times \mathcal{Y}(V)_Z$.

For each presheaf A we have a natural isomorphism $A_X \cong \widehat{\mathbb{C}}(\mathcal{Y}(X), A)$ If $a \in A_X$ then $\mathcal{Y}(X)_Z \ni u \mapsto A_u(a) \in A_Z$. Conversely, if $m \in \widehat{\mathbb{C}}(\mathcal{Y}(X), A)$ then $m_X(id_X) \in A_X$.

If $X \in \mathbb{C}$ and $A \in \widehat{\mathbb{C}}$ a presheaf A^X is defined by $A^X_Y = A_{X\times Y}$. The category $\widehat{\mathbb{C}}$ is cartesian closed; the function space $A{\Rightarrow}B$ is given by $(A{\Rightarrow}B)_X = \widehat{\mathbb{C}}(A, B^X)$. The application map $(A{\Rightarrow}B) \times A \longrightarrow B$ sends $m \in \widehat{\mathbb{C}}(A, B^X)$ and $a \in A_X$ to $B_\delta(m_X(a))$ where $\delta \in \mathbb{C}(X, X \times X)$ is the diagonal. Conversely, if $m : C \times A \to B$ then $\mathrm{curry}(m) : C \longrightarrow A{\Rightarrow}B$ sends $c \in C_X$ to $A_Y \ni a \mapsto m_{X\times Y}(C_\pi(c), A_{\pi'}(a))$ where π, π' are the first and second projections.

Function spaces with representable presheaves admit a simpler characterisation, namely we have

$$\mathcal{Y}(X){\Rightarrow}A \cong A^X \tag{17}$$

To see this, we calculate as follows: $(\mathcal{Y}(X){\Rightarrow}A)_Y \cong \widehat{\mathbb{C}}(\mathcal{Y}(Y), \mathcal{Y}(X){\Rightarrow}A) \cong$

$\widehat{\mathbb{C}}(\mathcal{Y}(Y) \times \mathcal{Y}(X), A) \cong \widehat{\mathbb{C}}(\mathcal{Y}(X \times Y), A) \cong A_{X \times Y} \cong A_Y^X$. This characterisation is the most important reason for the applicability of presheaves to higher-order syntax.

In a nutshell the idea is as follows. Interpret the metalanguage in an appropriate functor category $\widehat{\mathbb{C}}$ where $\mathbb{C}$ is chosen such that all metalanguage types appearing in negative positions (such as tm in the typing of lam in Eqn. 1) are representable. Then use Equation 17 to analyse the types of constants. If our metalanguage is merely simply-typed lambda calculus then the structure of functor categories exhibited so far suffices to interpret it. Dependent types in the metalanguage can also be accommodated in any presheaf category, see [10] for details. Also universes can be easily modelled. If impredicativity is desired like for COQ's Set one needs to consider presheaves relative to some constructive set theory which supports such universes in the first place. See [1] for details.

Being a topos every presheaf category also supports a notion of predicates, propositions, and higher-order logic: a predicate on some presheaf F is a subobject of F, i.e., a family of subsets $U_X \subseteq F_X$ such that $u \in U_X \supset F_f(u) \in U_Y$ for each $f \in F_X, f : Y \to X$. Such predicates are in 1-1 correspondence with morphisms from F to *Prop* $= \Omega$ where Ω_X is the set of predicates on $\mathcal{Y}(X)$. This "canonical" interpretation of propositions, however, always validates *AC!* and will therefore sometimes have to be replaced by a different one.

4 Adequacy

Consider the HOAS encoding (1) of the untyped lambda calculus. A functor category model of the relevant part of the metalanguage can be obtained as follows: Let $\mathbb{S}$ be the category which has as objects finite sets of variables $X = \{x_1, \dots, x_n\}$ and λ-*substitutions* as morphisms.

A λ-substitution from $X = \{x_1, \dots, x_m\}$ to $Y = \{y_1, \dots, y_n\}$ is given by a function σ which to each variable $y_i \in Y$ assigns an (object level) λ-term $\sigma(y_i)$ whose free variables are among the variables in X. For example, $\sigma(y_1) = \lambda z.z$, $\sigma(y_2) = \lambda z.zx_1$ defines a morphism from $\{x_1\}$ to $\{y_1, y_2\}$.

Let us write Tm_Y for the set of object-level λ-terms modulo α-conversion whose free variables are contained in Y. If $t \in Tm_Y$ and $\sigma : X \to Y$ is a λ-substitution then we obtain $t[\sigma] \in Tm_X$ as the simultaneous capture-free substitution of each y_i by $\sigma(y_i)$. This allows us to define composition in $\mathbb{S}$ by $(\sigma \circ \tau)(z) = \tau(z)[\sigma]$. It can be verified that this makes $\mathbb{S}$ into a category and Tm into a presheaf over $\mathbb{S}$ with morphism part given by $Tm_\sigma(t) = t[\sigma]$. In fact, Tm is the representable presheaf $\mathbb{S}(-, \{y\})$ for some arbitrarily chosen variable y.

Application of λ-terms takes two terms $t_1, t_2 \in Tm_X$ and produces $t_1 t_2 \in Tm_X$. Since application is compatible with substitution we obtain a natural transformation $App : Tm \times Tm \longrightarrow Tm$ given by $App_X(t_1, t_2) = t_1 t_2$.

The category $\mathbb{S}$ has cartesian products which are given on objects by disjoint union of variable sets. In particular, we have $X \times \{y\} = X \cup \{y'\}$ where $y' \notin X$. This means that for any presheaf $F \in \widehat{\mathbb{S}}$ we have

$$(Tm \Rightarrow F)_X \cong F_{X \cup \{x\}} \tag{18}$$

when $x \notin X$. Therefore, in particular $(Tm \Rightarrow Tm)_X \cong Tm_{X \cup \{x\}}$. Note that $Tm \Rightarrow Tm$ is not representable. For this to be the case, we would need an object $Z \in \mathbb{S}$ such that $(Tm \Rightarrow Tm)_X \cong \mathbb{S}(X \cup \{x\}, \{y\}) \cong \mathbb{S}(X, Z)$. But there is no such set Z. With $Z = \{y\}$ we have back-and-forth morphisms given by abstraction and application to the variable x. However, these do not form a bijection.

The characterisation of $Tm \Rightarrow Tm$ provides us with a family of maps

$$Lam_X : (Tm \Rightarrow Tm)_X \longrightarrow Tm_X \tag{19}$$

by $Lam_X(t) = \lambda x.t$. Since λ-abstraction also commutes with substitution we obtain in fact a natural transformation. This suggests to interpret the metalanguage used for the HOAS encoding (1) in the functor category $\widehat{\mathbb{S}}$ and in particular to interpret tm, app, lam by Tm, App, Lam, respectively.

This interpretation—when appropriately formalised—then associates to each meta-level term $M : tm$ with free variables $x_1 \colon tm, \dots, x_n \colon tm$ a natural transformation

$$[\![M]\!] : Tm^n \longrightarrow Tm$$

in such a way that $\beta\eta$-equal terms receive equal denotation.

But by the Yoneda Lemma such natural transformations are in 1-1 correspondence with $\mathbb{S}$-morphisms from $X = \{x_1, \dots, x_n\}$ to $\{x\}$, i.e., with elements of Tm_X. Concretely, if $f : Tm^n \longrightarrow Tm$ is a $\widehat{\mathbb{S}}$-map then we obtain $dec(f) \in Tm_X$ by

$$dec(f) \stackrel{\text{def}}{=} f_X(x_1, \dots, x_n) \tag{20}$$

Moreover, by inducion on the definition of $\ulcorner - \urcorner$ it follows that

$$[\![\ulcorner t \urcorner]\!]_X(s_1, \dots, s_n) = t[\sigma] \tag{21}$$

when $t = t(y_1, \dots, y_n)$ is an object-level term with free variables among $Y = \{y_1, \dots, y_n\}$ and $s_i = \sigma(y_i)$ are the components of a λ-substitution $\sigma : X \longrightarrow Y$. Putting these together shows

Proposition 4.1 *If $M(x_1, \dots, x_n)$ is a meta-language term of type tm with free variables $x_i : tm$ then there exists an object level term $t(x_1, \dots, x_n)$, namely $t = dec([\![M]\!])$ such that*

$$[\![M]\!] = [\![\ulcorner t \urcorner]\!]$$

Using a logical relation it is possible to show the stronger result that

$$M =_{\beta\eta} \ulcorner \mathit{dec}([\![M]\!]) \urcorner$$

which is the usual statement of adequacy.

Notice that our proof of existence of object-level terms corresponding to meta-language terms did not rely on any notion of term rewriting in the meta-language.

5 Induction principle

Recall the induction principle (5). Our aim is to interpret it in the model $\widehat{\mathbb{S}}$ from the previous section.

As argued in the introduction, the soundness of IND depends on our notion of proposition; in particular it is trivially true under an interpretation in which every proposition is true. It is false under an interpretation in which functional proposition-valued relations determine actual functions. In particular, this rules out an interpretation of propositions as sets in the style of Martin-Löf type theory.

The usual topos-theoretic interpretation of predicates as sub-objects also cannot be used as it validates $AC!$.

Intuitively, a principle like IND can only be sound if the quantification in the conclusion $\forall x\colon tm.\phi(x)$ ranges over closed terms only. We can achieve this effect by decreeing that a predicate over some presheaf $F \in \widehat{\mathbb{S}}$ is a subset of $F_\emptyset$, in particular a predicate over Tm is a set of closed terms. More formally, we put $\mathbf{Pred}(F) \stackrel{\text{def}}{=} \mathcal{P}(F_\emptyset)$. Of course, we cannot make arbitrary such declarations; we have to convince ourselves that propositional connectives and quantifiers as well as inference rules and axioms admit an adequate and convincing interpretation. The arising proof-obligations can be packaged in the form of a *tripos* [13], i.e., we have to show that $\mathbf{Pred}(-)$ extends to one such. Before defining this concept in detail we will describe its salient features for the particular case at hand.

Firstly, $\mathbf{Pred}$ extends to a functor $\mathbf{Pred} : \widehat{\mathbb{S}}^{op} \longrightarrow Set$ with morphism part given by inverse image: if $m : G \longrightarrow F$ and $U \subseteq F_\emptyset$ then $\mathbf{Pred}(m)(U) = \{x \in G_\emptyset \mid m_\emptyset(x) \in U\}$.

This functor is itself representable; we have a presheaf *Prop* $\in \widehat{\mathbb{S}}$ such that

$$\mathbf{Pred}(F) \cong \widehat{\mathbb{S}}(F, \mathit{Prop}) \qquad (22)$$

Setting $F = \mathcal{Y}(X)$ suggests to take $\mathit{Prop}_X = \mathbf{Pred}(\mathcal{Y}(X)) = \mathcal{P}(Tm^X_\emptyset)$ In other words a proposition "at stage X" is a subset of $\mathbb{S}(\emptyset, X)$.

For two predicates $U, V \in \mathbf{Pred}(F)$ the implication is defined as usual in classical logic by $U \Rightarrow V = \bar{U} \cup V$.

Finally, if $m : F \longrightarrow G$ is a morphism and $U \in \mathbf{Pred}(G)$ then we can define $\forall_m(U) \in \mathbf{Pred}(F)$ by

$$\forall_m(U) = \{f \in F_\emptyset \mid \forall g \in G_\emptyset . m_\emptyset(f) = g \text{ implies } g \in U\} \qquad (23)$$

In case where m is a projection this interprets the usual universal quantification.

A predicate $U \in \mathbf{Pred}(F)$ is true if it equals the whole of $U_\emptyset$. In particular, a closed proposition $U \in \mathbf{Pred}(\top)$ is true if it is the singleton set.

Definition 5.1 *Let $\mathcal{E}$ be a category. A contravariant functor $\mathbf{Pred} : \mathcal{E}^{op} \longrightarrow Set$ is called a* tripos *if*

- *each set $\mathbf{Pred}(X)$ forms a Heyting algebra (model for intuitionistic propositional logic), and this structure is preserved by the morphism part of $\mathbf{Pred}$, i.e., $\mathbf{Pred}(u) : \mathbf{Pred}(Y) \longrightarrow \mathbf{Pred}(X)$ is a Heyting algebra morphism for each $u \in \mathcal{E}(X, Y)$,*
- *The functor $\mathbf{Pred}$ is representable, i.e., there is an object Prop $\in \mathcal{E}$ so that $\mathbf{Pred}(X) \cong \mathcal{E}(X, \mathit{Prop})$,*
- *For each morphism $\pi : X \longrightarrow Y$ (typically a product projection, i.e., $X = Y \times D$, in which case $\forall_\pi$ quantifies over D.) there is a function $\forall_\pi : \mathbf{Pred}(X) \longrightarrow \mathbf{Pred}(Y)$ so that $\phi \Rightarrow \forall_\pi(\psi) = \mathbf{Pred}(\pi)(\phi) \Rightarrow \psi$ where $\Rightarrow$ is implication in $\mathbf{Pred}(Y)$ and $\mathbf{Pred}(X)$.*
- *Quantification commutes with substitution in the sense that whenever*

$$\begin{tikzcd} X' \arrow[r, "\pi'"] \arrow[d, "u'"'] & Y' \arrow[d, "u"] \\ X \arrow[r, "\pi"'] & Y \end{tikzcd}$$

is a pullback in $\mathcal{E}$ then $\mathbf{Pred}(u)(\forall_\pi(\phi)) = \forall_{\pi'}(\mathbf{Pred}(u')(\phi)$ for each $\phi \in \mathbf{Pred}(X)$. Again, a typical example arises when $X = Y \times D, X' = Y' \times D, u' = u \times D$.

If $\mathcal{E}$ models some metalanguage, e.g., simply typed lambda calculus then a tripos over $\mathcal{E}$ models higher order logic over that language in the sense that there is a type of propositions, term formers for implication and universal quantification and an additional judgment $\Gamma \vdash \phi$ which states that ϕ is a true proposition involving variables from Γ. Of course this presupposes that $\Gamma \vdash \phi : \mathit{Prop}$. The usual intuitionistic rules plus the axiom $\forall p.q\colon \mathit{Prop}.p \leftrightarrow q q \rightarrow p{=}q$ are validated.

A tripos does not model propositions-as-types, i.e., for each proposition $\phi : \mathit{Prop}$ a type $Prf(\phi)$ of proofs of ϕ.

If that is desired, one must interpret types as pairs (X, ϕ) where $X \in \mathcal{E}$ and $\phi \in \mathbf{Pred}(X)$.

The following result (due to Jaap van Oosten) allows us to construct many triposes easily.

Theorem 5.2 *1. For any topos $\mathcal{E}$ the setting $\mathbf{Pred}(X) = \mathcal{E}(X, \Omega) \cong Sub(X)$ determines a tripos structure.*

2. If $\mathcal{E}, \mathcal{F}$ are categories with cartesian products and terminal object, $\mathbf{Pred}_{\mathcal{F}}$ a tripos structure on $\mathcal{F}$, and $f : \mathcal{E} \longrightarrow \mathcal{F}$ a finite limit preserving functor with a right adjoint $f^* : \mathcal{F} \longrightarrow \mathcal{E}$, then $\mathbf{Pred}_{\mathcal{E}}(X) \stackrel{def}{=} \mathbf{Pred}_{\mathcal{F}}(f(X))$ determines a tripos structure on $\mathcal{E}$.

Our tripos given by $\mathbf{Pred}(X) = \mathcal{P}(\mathbb{S}(\emptyset, X))$ arises by taking $\mathcal{E} = \widehat{\mathbb{S}}, \mathcal{F} = Set, f(A) = A_{\emptyset}$. It is clear that f preserves finite limits as these are computed pointwise; the required right adjoint is obtained by $f^*(X)_Y = X^{\mathbb{S}(\emptyset, Y)}$.

Let us now see how this interpretation validates our induction principle. After a somewhat laborious unfolding of the definitions we find that IND states the following: for each $P \in \mathbf{Pred}(Tm) = \mathcal{P}(Tm_{\emptyset})$, i.e., for each set of closed terms, it holds that if

(A) for any two $t_1, t_2 \in P$ we have $t_1 t_2 \in P$,

(L) for each $t \in Tm_{\{x\}}$ which has the property that whenever $s \in P$ then $t[x{:=}s] \in P$, it holds that $\lambda x.t \in P$

then $P = Tm_{\emptyset}$.

To see that this is valid let P be a set of closed terms such that (A) and (L) above are valid. By induction on object-level lambda terms we can now show the following

Lemma: For each set of variables $X = \{x_1, \ldots, x_n\}$ and $t \in Tm_X$ the following holds. If $\sigma \in \mathbb{S}(\emptyset, X)$ is such that $\sigma(x) \in P$ for each $x \in X$ then $t[\sigma] \in P$.

The desired conclusion that $P = Tm_{\emptyset}$ then follows by specialising to $X = \emptyset$.

5.1 Applications of IND

A first trivial application is that the following proposition is provable using IND:

$$\begin{array}{l} \forall x{:}\, tm.(\exists y, z{:}\, tm.x = app(y, z)) \vee \\ (\exists f{:}\, tm{\to}tm.x{=}lam(f)) \end{array} \tag{24}$$

Here $y = z$ denotes Leibniz equality definable in higher-order logic. Its denotation in the model is exactly syntactic equality of closed terms. So, the meaning of this proposition is that every closed term is syntactically equal either to an application or to an abstraction.

A more interesting application builds on typed terms, i.e., we introduce a type $ty : Set$ together with constants $o : ty$ and $arr : ty \to ty \to ty$. We also replace our type of terms ty by a family of types $tm : ty \to Set$ and use appropriate constants app and lam, e.g., $lam : \Pi a, b{:}\, ty.(tm(a){\to}tm(b)){\to}tm(arr(a, b))$. We furthermore assume that ty is an inductive type, i.e., that we can define functions and families of types or propositions by induction over ty. This, of course, requires a dependently typed metalanguage such as Martin-Löf type theory.

This metalanguage can be interpreted as presheaves over the category which has sets of typed variables as objects and simply-typed substitutions as morphisms. The interpretation of ty is the constant presheaf of type expressions; the interpretation of $tm(a)$ is the again the appropriate representable presheaf.

A typed version of principle *IND* can then be validated which can be used to establish the following proposition

$$\forall a{:}\, ty.\forall x{:}\, tm(a).\exists y{:}\, tm(a).whnf(y) \wedge whred(x, y)$$

where $whred(x, y)$ and $whnf(x)$ denote weak-head reduction and weak-head normality, both of which are readily definable in higher-order logic. The proof goes by defining a family of Tait-style reducibility predicates $RED : \Pi a{:}\, ty.tm(a) \to Prop$ by

$$\begin{array}{l} RED(o, t) = \forall P{:}\, Prop.P \\ RED(arr(a, b), t) = \exists f{:}\, tm(a){\to}tm(b). \\ \ whred(t, lam(a, b, f)) \wedge \\ \ \forall x{:}\, tm(a).RED(a, x) \Rightarrow RED(b, f(x)) \end{array}$$

Now (the typed version of) IND gives us $\forall a{:}\, ty.\forall t{:}\, tm(a).RED(a, t)$ from which the desired conclusion is direct. We remark that it is not possible to define typing as a relation between type expressions and untyped terms because our notion of predicate would mean that typing relates closed terms to types only, but the restriction of typing to closed terms doesn't seem to admit a reasonable inductive definition.

6 Modal operator

Let us move back to the encoding of untyped lambda calculus and its justification using the model $\mathbb{S}$. We have a functor $\Box : \widehat{\mathbb{S}} \longrightarrow \widehat{\mathbb{S}}$ given by $(\Box F)_X = F_{\emptyset}$.

We have $\Box \circ \Box = \Box$ and there is a natural transformation $\varepsilon : \Box \to \mathrm{Id}$ given by $(\varepsilon_F)_X(f) = F_{\langle\rangle}(f)$ where $f \in F_X$ and $\langle\rangle \in \mathbb{S}(X, \emptyset)$ is the empty substitution.

These data make $\Box$ a comonad on $\widehat{\mathbb{S}}$ and in particular, if $m : \Box F \longrightarrow G$ then we can "raise" m to a morphism from $\Box F$ to $\Box G$, namely $\Box m$ in view of $\Box^2 = \Box$.

This allows us to model Pfenning's modal lambda calculi [17, 5].

The interpretation of the type $\Box tm$ then becomes the presheaf $\Box Tm$ which is the constant presheaf of closed terms. Similarly, the interpretation of $\Box(tm \to tm)$ becomes the constant presheaf consisting of the terms in exactly one free variable and more generally, $\Box(tm^n \to tm)$ is the constant presheaf of terms in n distinguished free variables. The interpretation of $(\Box tm) \to tm$ on the other hand becomes the constant presheaf consisting of all set-theoretic functions between closed terms. We remark that this is very well in line with the intuitions provided in [5] which, however, do not receive semantical underpinning there.

The types $\Box(tm^n \to tm)$ now carry an inductive structure in $\widehat{\mathbb{S}}$ with respect to the usual topos-theoretic interpretation of predicates, i.e., where $\mathbf{Pred}(F)$ consists of all subobjects of F and accordingly $Prop_X$ is the set of subobjects of $\mathcal{Y}(X)$.

Under this interpretation the axiom of unique choice is validated and allows us to define recursion principles from induction principles in the usual set-theoretic way. We will formulate a general induction principle which requires a type *nat* of natural numbers interpreted as the constant presheaf $\mathbf{N}_X = \mathbb{N}$ which is the natural numbers object in $\widehat{\mathbb{S}}$. We use the abbreviation $tm^{(n)}$ for $tm^n \to tm$. We also write $app^{(n)} : tm^{(n)} \to tm^{(n)} \to tm^{(n)}$ and $lam^{(n)} : tm^{(n+1)} \to tm^{(n)}$ for the obvious liftings of app and lam obtained from the equation $tm^{(n+1)} = tm \to tm^{(n)}$. If we aim for a less ambitious metalanguage then—without looking at the semantics again—we can derive more specialised induction and recursion principles from that one in the language of topos logic.

Our induction principle takes the following form:

$$\begin{aligned}
&IND_{\Box} \equiv \forall P : \Pi n{:}\,nat.\Box(tm^{(n)}) \to Prop \\
&(\forall n{:}\,nat.\forall t.t'{:}\,\Box tm^{(n)}.P(n,t) \Rightarrow P(n,t') \Rightarrow \\
&\qquad P(n, \Box app^{(n)}(t,t'))) \Rightarrow \\
&(\forall n{:}\,nat.\forall t{:}\,\Box tm^{(n+1)}.P(n+1,t) \Rightarrow \\
&\qquad P(n, \Box lam^{(n)}(t))) \Rightarrow \\
&\forall n{:}\,nat.\forall t{:}\,\Box(tm^{(n)}).P(n,t)
\end{aligned} \tag{25}$$

Here $\Box$ in front of app and lam is a syntactic reflection of the "raising" operation which is allowed here since all free variables of the term under question are of boxed type. The modal lambda calculi in *op. cit.* make this more precise.

The proof that this principle is valid in $\widehat{\mathbb{S}}$ is straightforward from the explicitation of the denotations of the types $\Box tm^{(n)}$ sketched above.

Since we have not changed our semantic universe the adequacy proof from Section 4 continues to hold.

From $IND_{\Box}$ we can derive other principles using merely higher-order logic with unique choice. In particular, all the recursion principles from [5] including the one mentioned in Section 1.3 can be validated in the same way as AC! allows us to recover primitive recursion from induction. Of course, when other base types and constants are present then we may have to change our base category $\mathbb{C}$, but obviously $\Box$ can be defined in those cases as well.

7 Variables and names

In order to validate the HOAS from [4] we need a presheaf of variables that is representable so that we can unravel the function spaces of the form $var \to A$. In order to achieve this, we use the subcategory $\mathbb{V} \subseteq \mathbb{S}$ which consists of the variable substitutions only, i.e., those $\sigma \in \mathbb{S}(X,Y)$ for which $\sigma(y)$ is a variable (from X) for each $y \in Y$. We have the representable presheaf $Var \cong \mathcal{Y}(\{x\})$ given by $Var_X = X$ and the presheaf of terms as before (i.e. $Tm_X = \mathbb{S}(X, \{x\})$) which is now no longer representable though. Equation 17 gives us $(Var \Rightarrow Tm)_X \cong Tm_{X \cup \{x\}}$ thus allowing us to interpret the signature from [4] in $\widehat{\mathbb{V}}$. It is now possible to validate Martin-Löf-style induction-recursion principles for tm and also for the functional types $var^n \to tm$ which allow one to define substitution as sketched above in Section 1.4. Rather than showing directly how these principles are validated we will exhibit Tm and more generally $Var^n \Rightarrow Tm$ as initial algebras for appropriate signature functors. The interpretation of induction / recursion principles are then a routine generalisation of the derivation of the induction scheme for a natural numbers object as explained e.g. in Section II.4 of [14].

Theorem 7.1 *Let $\mathcal{E}, \mathcal{F}$ be categories and $f : \mathcal{E} \longrightarrow \mathcal{F}$ be a functor with right adjoint f^*.*

Furthermore, let $T : \mathcal{E} \longrightarrow \mathcal{E}$ and $T' : \mathcal{F} \longrightarrow \mathcal{F}$ be functors such that $T' \circ f = f \circ T$ by some natural isomorphism ϕ.

If $a : T(A) \longrightarrow A$ is an initial T-algebra then $\phi \circ f(a) : T'(f(A)) \longrightarrow f(A)$ is an initial T'-algebra.

Proof. By showing that f and f^* lift to a pair of adjoint functors between the categories of T-, resp. T'-algebras. The initial T-algebra being an initial object of the former category will therefore be preserved. $\Box$

The following properties of $Var \Rightarrow -$ are established by direct verfication.

Proposition 7.2 *1. The functor $Var \Rightarrow - : \widehat{\mathbb{V}} \longrightarrow \widehat{\mathbb{V}}$ has a right adjoint R given by $R(F)_X = \widehat{\mathbb{V}}(Var \Rightarrow \mathcal{Y}(X), F)$,*

2. $Var \xrightarrow{const} Var \Rightarrow Var \xleftarrow{id} \top$ is a coproduct diagram, i.e., an element of $Var \Rightarrow Var$ is either a constant or the identity and we can define functions on $Var \Rightarrow Var$ by case distinction on which of the two holds.

3. *The presheaf Tm is an initial algebra for the functor*

$$T(X) = Var + X \times X + Var \Rightarrow X$$

with structure map $T(Tm) \longrightarrow Tm$ obtained by case distinction from var, app, and lam.

Iterating part ii together with the fact that $Var \Rightarrow -$ preserves coproducts shows that $Var^n \Rightarrow Var \cong 1 + \cdots + 1 + Var$, i.e., an n-ary function on variables is either one of the n projections or a constant. It also follows from the preservation of coproducts that

$$T(Var^n \Rightarrow X) \cong \\ Var \Rightarrow ((Var^n \Rightarrow Var) + X \times X + Var \Rightarrow X)$$

thus providing an initial algebra characterisation of $Var^n \Rightarrow Tm$ by Theorem 7.1. In particular, we have that $Var \Rightarrow Tm$ is initial algebra for the functor $T(X) = 1 + Var + X \times X + Var \Rightarrow X$ thus providing us with the recursion principle required to define substitution according to Eqn. 9. Again, we can syntactically derive from the initial algebra property more specialised induction and recursion prinicples without having to justify these anew.

In order to justify axioms EXT, DEC and the other axioms from [11] which are not reproduced here we need again a special interpretation of the type *Prop* of propositions which does not validate *AC!*.

Let us first see that *DEC* is not valid in the full topos logic of $\widehat{\mathbb{V}}$. We have that $Eq_X \subseteq Var_X \times Var_X$ defined by $E_X = \{(x,x) \mid x \in X\}$ forms a subobject of $Var \times Var$, in fact it is the denotation of Leibniz equality on Var in the topos logic. However the complement of Eq given by $InEq_X = \{(x,y) \mid x \in X, y \in X, x \neq y\}$ fails to be a subobject because it is not preserved by non-injective variable substitutions. Of course, the proposition $\neg x{=}y$ has a denotation in the topos logic, but its meaning is absurdity and consequently the topos logic of $\widehat{\mathbb{V}}$ validates the proposition $\forall x, y{:}\, var. \neg\neg x{=}y$! So, DEC fails.

In order to enforce DEC we simply arrange matters so that $InEq$ becomes an allowable predicate. More formally, we use the subcategory $\mathbb{I} \subseteq \mathbb{V}$ consisting of the *injective* variable substitutions, i.e., $\mathbb{I}(X,Y) = \{\sigma : Y \to X \mid \sigma \text{ injective}\}$.

The restriction of presheaves in $\widehat{\mathbb{V}}$ to injective substitutions gives us a functor $f : \widehat{\mathbb{V}} \longrightarrow \widehat{\mathbb{I}}$ and (as every such restriction functor it preserves finite limits and has a right adjoint given by $f^*(A)_Y = \widehat{\mathbb{I}}(\mathbb{S}(-,Y), A)$. So, Theorem 5.2 shows that we obtain a tripos on $\widehat{\mathbb{V}}$ by defining $\mathbf{Pred}(A)$ as the set of subobjects of F in $\widehat{\mathbb{I}}$. More concretely, a predicate on F is given by a family of subsets $U_X \subseteq F_X$ such that for each injective σ, i.e., $\sigma \in \mathbb{I}(Y,X)$ and $f \in U_X$ we have $F_\sigma(f) \in U_Y$. We have argued informally above that DEC holds in this tripos; the validity of EXT amounts to the fact that whenever $p, q \in Tm(X)$ and $py = qy \in Tm_{X \cup \{y\}}$ for some $y \notin X$ then $p = q$. A truism.

Finally, we need to verify that this tripos also validates the induction principle for tm, i.e., the proposition

$$\begin{array}{l} \forall P{:}\, tm \to \textit{Prop}. \\ \quad (\forall x{:}\, var. P(var(x)) \to \\ \quad (\forall x, y{:}\, tm. P(x) \to P(y) \to P(app(x,y))) \to \\ \quad (\forall f{:}\, var \to tm. (\forall x{:}\, var. P(f(x)) \to P(lam(f)))) \\ \qquad \forall t{:}\, tm. P(t) \end{array} \tag{26}$$

and, more generally, analogous principles for $var^n \to tm$. Notice that in the absence of propositions-as-types this is not an immediate consequence of the initial algebra property.

However, in the case at hand, we can deduce it from Thm. 7.1 with $T' : \widehat{\mathbb{I}} \longrightarrow \widehat{\mathbb{I}}$ given by

$$T'(X) = Var + X \times X + Var \multimap X$$

where $(Var \multimap F)_X \stackrel{\text{def}}{=} F_{X \cup \{y\}}$. Unlike in $\widehat{\mathbb{V}}$ this is *not* the function space $Var \Rightarrow F$, but only a right adjoint to a certain tensor product on $\widehat{\mathbb{I}}$. The reason is that the diagonal map δ used in the verification that $F^A \cong \mathcal{Y}(A) \Rightarrow F$ is not available in $\mathbb{I}$. Astonishingly, one has in $\widehat{\mathbb{I}}$ a map $\alpha : Var \Rightarrow F \longrightarrow Var \multimap F$, but it fails to be injective. It would thus be possible to interpret the metalanguage in $\widehat{\mathbb{I}}$ directly, hence use the full topos logic, but the interpretation of lam obtained as the composition of the third inductive constructor corresponding to T' with the map α would not be injective.

Axiom EXT would then be valid with conclusion $lam(p) = lam(q)$ rather than $p = q$.

This is basically the approach of [18, 6] except that domain-valued functors are used there.

In fact, in these papers the subcategory $Sh_{\neg\neg}(\mathbb{I})$ of $\widehat{\mathbb{I}}$ consisting of pullback preserving functors only is used. This category forms again a topos and it validates full classical logic, i.e., $\forall p{:}\, \textit{Prop}. p \vee \neg p$. One has again a finite limit preserving functor $f : \widehat{\mathbb{I}} \longrightarrow Sh_{\neg\neg}(\mathbb{I})$ admitting a right adjoint f^* (namely the inclusion $Sh_{\neg\neg}(\mathbb{I}) \subseteq \widehat{\mathbb{I}}$), so one can again use Thm. 5.2 to obtain another tripos structure on $\widehat{\mathbb{V}}$ which would validate classical logic. Fortunately, this new tripos is nothing but the one given by restricting the old one to double negation closed predicates, i.e.,

$$\mathbf{Pred}_{\neg\neg}(F) = \{P \in \mathbf{Pred}(F) \mid \neg\neg P = P\}$$

Since the axioms we were interested in were purely implicational formulas with decidable conclusion their validity carries over to $\mathbf{Pred}_{\neg\neg}$. This shows that by working with $\mathbf{Pred}_{\neg\neg}$ one can justify full classical logic and not just DEC while still having EXT plus induction principles.

As indicated, this can be seen directly by syntactic considerations without even mentioning $Sh_{\neg\neg}(\mathbb{I})$. We close by remarking that $Sh_{\neg\neg}(\mathbb{I})$ is equivalent to the categorical structure underlying [8], so that in some sense our tripos $\mathbf{Pred}_{\neg\neg}$ can be seen as an amalgamation of $\widehat{\mathbb{V}}$ (the approach also taken by Fiore *et al.*) with Gabbay and Pitts' setting thus providing a formal link between the three works on higher-order syntax present in this volume.

References

[1] T. Altenkirch, M. Hofmann, and T. Streicher. Reduction-free normalisation for system F. Unpublished. Available from www.dcs.ed.ac.uk/home/mxh/papers, May 1996.

[2] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Yann Coscoy, David Delahaye, Daniel de Rauglaudre, Jean-Christophe Fillâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, Henri Laulhère, César Muñoz, Chetan Murthy, Catherine Parent-Vigouroux, Patrick Loiseleur, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. *The Coq Proof Assistant Reference Manual - Version 6.2*. INRIA, Rocquencourt, France, May 1998. Available at `ftp://ftp.inria.fr/INRIA/coq/V6.2/doc`.

[3] D. Basin and S. Matthews. Logical frameworks. Draft of book chapter. Full reference to appear in final version, June 1998.

[4] J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-order abstract syntax in Coq. In M. Dezani and G. Plotkin, editors, *Typed Lambda Calculi and Applications*, pages 124–138. Springer LNCS vol. 902, 1995.

[5] J. Despeyroux, F. Pfenning, and C. Schürmann. Primitive Recursion for Higher-Order Abstract Syntax. Technical Report CMU-CS-96-172, Carnegie Mellon University, 1996.

[6] M. Fiore, E. Moggi, and D. Sangiorgi. A fully abstract model for the π-calculus. In *Proc. 11th Symp. Logic in Comp. Sci. (LICS), New Brunswick, N. J.*, pages 43–55. IEEE, 1996.

[7] M. Fiore, G. Plotkin, and D. Turi. Abstract Syntax and Variable Binding. 1999.

[8] Gabbay and Pitts. A new semantical account of higher-order syntax. In *Proc. 14th Symp. Logic in Comp. Sci. (LICS), Trento*, 1999.

[9] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[10] Martin Hofmann. Syntax and semantics of dependent types. In A. M. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute, pages 79–130. Cambridge University Press, 1997.

[11] F. Honsell, M. Miculan, and I. Scagnetto. π-calculus in (co)inductive type theory. Accepted for publication in TCS. Full reference to appear in full paper.

[12] Furio Honsell and Marino Miculan. A Natural Deduction Approach to Dynamic Logics. In *Proc. BRA TYPES workshop, Torino, June 1995, Springer LNCS 1158*, 1996.

[13] J. M. E. Hyland, P. T. Johnstone, and A. M. Pitts. Tripos theory. *Mathematical Proceedings of the Cambridge Philosophical Society*, 88:205–232, 1980.

[14] Joachim Lambek and Philip Scott. *Introduction to Higher-Order Categorical Logic*. Cambridge University Press, 1986.

[15] P. W. O'Hearn and R. D. Tennent. Semantics of local variables. In *Applications of Categories in Computer Science 1991*, number 177 in London Mathematical Society Lecture Note Series, pages 217–238. Cambridge University Press, 1992.

[16] F. J. Oles. Type categories, functor categories and block structure. In M. Nivat and J. C. Reynolds, editors, *Algebraic Semantics*, pages 543–574. Cambridge University Press, 1985.

[17] Frank Pfenning and Hao-Chi Wong. On a modal lambda calculus for S4. In *Proceedings of the 11th Conference on Mathematical Foundations of Programming Semantics (MFPS), New Orleans, Louisiana*. Electronic Notes in Theoretical Computer Science, Volume 1, Elsevier, 1995.

[18] Ian Stark. A fully abstract domain model for the π-calculus. In *Proc. 11th Symp. Logic in Comp. Sci. (LICS), New Brunswick, N. J.*, pages 36–42. IEEE, 1996.