A Framework for Typed HOAS and Semantics*

Marino Miculan
miculan@dimi.uniud.it

Ivan Scagnetto
scagnett@dimi.uniud.it

Department of Mathematics and Computer Science, University of Udine
Via delle Scienze 206, I-33100 Udine, Italy

ABSTRACT

We investigate a framework for representing and reasoning about syntactic and semantic aspects of typed languages with variable binders.

First, we introduce *typed binding signatures* and develop a theory of typed abstract syntax with binders. Each signature is associated to a category of "presentation" models, where the language of the typed signature is the initial model.

At the semantic level, types can be given also a computational meaning in a (possibly different) *semantic* category. We observe that in general, semantic aspects of terms and variables can be reflected in the presentation category by means of an adjunction. Therefore, the category of presentation models is expressive enough to represent both the syntactic and the semantic aspects of languages.

We introduce then a *meta*logical system, inspired by the internal languages of the presentation category, which can be used for reasoning on both the syntax and the semantics of languages. This system is composed by a core equational logic tailored for reasoning on the syntactic aspects; when a specific semantics is chosen, the system can be modularly extended with further "semantic" notions, as needed.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal definitions and Theory—*syntax, semantics*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of programming languages—*algebraic approaches to semantics, denotational semantics*; F.4.3 [**Formal Languages**]: Mathematical Logic and Formal Languages—*Algebraic language theory*

General Terms

Languages, Theory.

*Research supported by Italian MIUR project COFIN 2001013518 CoMeta.

PPDP'03, August 27–29, 2003, Uppsala, Sweden.

Keywords

Typed abstract syntax with variable binding, categorical metamodels, metalanguages for syntax and semantics of languages, initial algebra semantics, presheaf categories.

Introduction

It is well known that, in order to reason about programs and programming languages, one has to face both the problem of representing the syntax with the related machinery (e.g., substitution, α-conversion, etc.) and of modeling the underlying semantics. Even if at a first sight these abstraction layers may seem independent from each other, they are closely related: for instance, once the interpretation of terms has been fixed, the syntactic substitution usually uniquely determines a corresponding notion of semantic substitution, via the so-called "substitution lemma". Another contact point is the pervasive notion of *(intrinsic) type*. Syntactically, types denote *languages*, usually defined inductively by grammars, possibly with binders and thus quotiented by α-equivalences. The inductive structure of typed syntax is fundamental for recursive definitions and inductive reasoning. Semantically, types denote *domains*, ranged over by *variables* and where syntactic expressions are given a meaning. Both syntactic and semantic manipulations (e.g. substitution) have to respect the types, otherwise non-wellformed terms or meaningless structures would arise.

Therefore, a framework (a *meta*model) aiming to provide mathematical notions and tools for reasoning about programming languages should address both the syntactic and the semantic aspects on a par. The metamodel should enlighten the interactions and dependencies between these two abstraction levels.

The aim of this paper is indeed to define and investigate such a framework. Our setting is inspired by previous work by Fiore, Hofmann, Plotkin, Turi, among others [2–4, 7, 8]. Like in these works, we advocate the use of *presheaf categories* for a satisfactory representation of the *abstract syntax* of (typed) languages with variables and binding operators. But moreover, it turns out that the same presheaf categories can be used to address the semantic aspects as well.

We proceed as follows. First, we give an account of typed higher-order abstract syntax and semantics, and their interconnection. We introduce the notion of *typed binding signatures* and develop a theory of algebraic typed abstract syntax with variable binders. To the types T of a signature we associate a category of *presentation models*

$$\mathrm{Mod}_T(\mathcal{S}) \triangleq (\mathcal{S}et^{\mathcal{U}})^T \cong \mathcal{S}et^{\mathcal{U}\times T}$$

where $\mathcal{U}$ is a suitable category of typing contexts. The language of any typed signature can be defined as the initial algebra of the corresponding signature functor on $\mathrm{Mod}_T(\mathcal{S})$.

On the other hand, types can be also given a *semantics*, i.e. a meaning in any suitable category $\mathcal{C}$, depending on the particular computational notion we are focus on. Therefore, another category $\mathrm{Mod}_T(\mathcal{C})$, of *semantic* models arises. Under a mild condition, this turns out to be connected to the presentation category via an adjunction, as in [2,3]. Along this adjunction, semantic aspects of terms and variables can be reflected at the syntactic level, and syntactic (i.e., structural) aspects can be recovered at the semantic level. Therefore any framework for abstract syntax, complete with semantics for the syntax in it, can be reflected in the same, well-established presheaf category $\mathrm{Mod}_T(\mathcal{S})$.

We illustrate this schema with some examples. When the semantic category is the presentation category itself (§3.3), we get a monoidal closed category whose monoids are fully typed (i.e., heterogeneous) substitutions over higher-order abstract syntax. In another application (§3.4), the distinction between "names" and "variables over names" (a common situation in process algebras, and languages for syntax manipulation) is formally clarified.

We introduce then a *meta*logical system, based on the internal languages of the presentation category, which can be used for reasoning on both the syntax and the semantics of languages on a par. The metalogical system is composed by a core equational logic $\mathcal{H}_\Sigma$ tailored for reasoning on the syntactic aspects. This system makes no assumption on the "intended meaning" of variables (e.g., they are not seen as "names" from the beginning, as in [8,19]). Still $\mathcal{H}_\Sigma$ allows to represent faithfully the typed languages with binders and defining programs by structural recursion over typed abstract syntax with binders. The main purpose of $\mathcal{H}_\Sigma$ is to act as a basis for further extensions: when types are given also a semantic interpretation, the core metalanguage can be enriched with new type and term constructors and new propositions, in a modular way. We will exemplify this approach in the case that variables are denoting *names:* this semantic interpretation will induce both rules of the Theory of Contexts and rules close to those of Nominal Logic [8,19].

Synopsis. In §1 we introduce typed binding signatures. In §2 we develop an algebraic theory of typed abstract syntax, by means of presheaf categories. In §3 we address the problem of connecting the semantic and the syntactic aspects of a language, along an adjunction. Three case studies are presented, in order to illustrate the generality and applicability of the framework. In §4 we introduce the metalanguage $\mathcal{H}_\Sigma$ (parametric on the signature of the object language) which allows one to reason about typed languages in higher-order abstract syntax. Some conclusions are finally drawn in §5.

1. TYPED BINDING SIGNATURES

In this section, we introduce the category of typed binding signatures, which is a direct extension of both typed signatures [11, §1.6] and binding signatures [4].

DEFINITION 1. *A* typed binding signature *is a triple* $\Sigma = (T, U, \mathcal{O})$ *where* T *is a (countable) set of* (basic) types*,* $U \subseteq T$ *is the set of* types with variables *and* $\mathcal{O} : (U^* \times T)^* \times T \longrightarrow \mathcal{S}et$ *assigns to each tuple* $((\vec{\sigma}_1, \tau_1), \ldots, (\vec{\sigma}_n, \tau_n), \tau)$ *a set of* function symbols*.*

In the following, we will denote the set of types of a signature Σ also by $|\Sigma|$.

The set $(U^* \times T)^* \times T$ will be denoted by $Ar(\Sigma)$, and, following [17, Chapter 3], its elements will be called *arities* (of the signature Σ), ranged over by $\alpha, \beta, \ldots$. Arities and function symbols will be written in the usual "arrow" notation: we will write $f : (\vec{\sigma}_1 \to \tau_1) \times \cdots \times (\vec{\sigma}_n \to \tau_n) \to \tau$ for $f \in \mathcal{O}((\vec{\sigma}_1, \tau_1), \ldots, (\vec{\sigma}_n, \tau_n), \tau)$. Notice that only types in U may appear in "negative" positions of these arities. When $U = \emptyset$ we get back the traditional notion of typed signature [11]; when $T = U = 1$, we get the notion of binding signature [4].

Finally, notice that a symbol may be overloaded, that is it may have several arities.

Example 1. The signature of untyped λ-calculus is $\Sigma_\lambda = (\{\star\}, \{\star\}, \mathcal{O}_\lambda)$, where $\mathcal{O}_\lambda((\star, \star), \star) = \{\lambda\}$, $\mathcal{O}_\lambda((\star), (\star), \star) = \{app\}$.

Let $T_{\lambda\supset}$ be the smallest set containing a given set of atomic type symbols and closed under the "$\supset$" constructor. Then, the signature of simply typed λ-calculus is $\Sigma_{\lambda\supset} = (T_{\lambda\supset}, T_{\lambda\supset}, \mathcal{O}_{\lambda\supset})$ where, for $\sigma, \tau \in T_{\lambda\supset}$: $\mathcal{O}_{\lambda\supset}((\tau, \sigma), \tau\supset\sigma) = \{\lambda\}$, $\mathcal{O}_{\lambda\supset}((\tau\supset\sigma), (\tau), \sigma) = \{app\}$.

The signature of π-calculus is $\Sigma_\pi = (\{\eta, \iota\}, \{\eta\}, \mathcal{O}_\pi)$, where $\mathcal{O}_\pi(\iota) = \{0\}$, $\mathcal{O}_\pi((\iota), \iota) = \{\tau, !\}$, $\mathcal{O}_\pi((\iota), (\iota), \iota) = \{|, +\}$, $\mathcal{O}_\pi((\eta), (\eta), (\iota), \iota) = \{match, output\}$, $\mathcal{O}_\pi((\eta, \iota), \iota) = \{\nu\}$, $\mathcal{O}_\pi((\eta), (\eta, \iota), \iota) = \{input\}$.

A morphism $u : \Sigma \to \Sigma'$ of signatures consists of a function $u : |\Sigma| \to |\Sigma'|$ between type sets (which is extended to arities in the obvious way) such that $u(U) \subseteq U'$ and for each arity α of Σ, a function $u_\alpha : \mathcal{O}(\alpha) \to \mathcal{O}'(u(\alpha))$. Thus, if $f : (\vec{\sigma}_1 \to \tau_1) \times \cdots \times (\vec{\sigma}_n \to \tau_n) \to \tau$, then $u_\alpha(f) : (u(\vec{\sigma}_1) \to u(\tau_1)) \times \cdots \times (u(\vec{\sigma}_n) \to u(\tau_n)) \to u(\tau)$. Typed binding signatures and their morphisms form a category $\mathcal{TS}ign$.

Terms. Let $\Sigma = (T, U, \mathcal{O})$ be a signature, and let $Var = (Var_\tau)_{\tau \in U}$ a U-indexed family of infinite sets of variables (ranged over by x^τ). The set S_τ of terms of type τ is defined by the following grammar:

$$\begin{array}{rcll} S_\tau \ni t ::= & x^\tau & & \text{if } \tau \in U \\ & \mid & op((\vec{x}_1)t_1, \ldots, (\vec{x}_n)t_n) & \end{array}$$

where $op : (\vec{\sigma}_1 \to \tau_1) \times \cdots \times (\vec{\sigma}_n \to \tau_n) \to \tau$ and for $i = 1, \ldots, n$: $t_i \in S_{\tau_i}$. The usual conventions about free/bound variables and α-conversion apply.

A more precise definition of the terms generated by a binding signature Σ is given in a type theoretic form, as e.g. in [11,18]. Let $Var = \{v_1, v_2, \ldots\}$ be an enumeration of distinct elements, called *variable symbols*; a *typing context* (denoted by Γ) is a list $v_1 : \tau_1, \ldots, v_n : \tau_n$, where $\tau_i \in U$. The *term calculus* for Σ is the system for deriving typing judgments of the usual form

$$\Gamma \vdash_\Sigma t : \tau$$

where $\tau \in T$, and whose rules are given in Figure 1. The index $_\Sigma$ will be dropped when clear from the context. The system consists of the projection rule (1), where necessarely $\tau \in U$, and a specific rule (5) for each constructor *op* of the signature. In these latter rules, the contexts of the premises are extended with new variable symbols, that is, $\Gamma, \vec{v} : \vec{\sigma}$ is a shorthand for $\Gamma, v_{n+1} : \sigma_1, \ldots, v_{n+l} : \sigma_l$ where $|\Gamma| = n$ and $|\vec{\sigma}| = l$. As usual, terms are taken up-to α-conversion.

$$\frac{(v_i:\tau_i)\in\Gamma}{\Gamma\vdash v_i:\tau_i}\quad(1)\qquad \frac{\Gamma,v_{n+1}:\tau_{n+1},v_{n+2}:\tau_{n+2}\vdash t:\tau}{\Gamma,v_{n+1}:\tau_{n+2},v_{n+2}:\tau_{n+1}\vdash t\{v_{n+1}/v_{n+2},v_{n+2}/v_{n+1}\}:\tau}\quad(3)$$

$$\frac{\Gamma\vdash t:\tau}{\Gamma,v_{n+1}:\tau_{n+1}\vdash t:\tau}\quad(2)\qquad \frac{\Gamma,v_{n+1}:\tau_{n+1},v_{n+2}:\tau_{n+1}\vdash t:\tau}{\Gamma,v_{n+1}:\tau_{n+1}\vdash t\{v_{n+2}/v_{n+1}\}:\tau}\quad(4)$$

$$\frac{\Gamma,\vec{v}_1:\vec{\sigma}_1\vdash t_1\{\vec{v}_1/\vec{x}_1\}:\tau_1\quad\ldots\quad\Gamma,\vec{v}_k:\vec{\sigma}_k\vdash t_k\{\vec{v}_1/\vec{x}_1\}:\tau_k}{\Gamma\vdash op((\vec{x}_1)t_1,\ldots,(\vec{x}_k)t_k):\tau}\ op\in\mathcal{O}((\vec{\sigma}_1\to\tau_1)\times\cdots\times(\vec{\sigma}_n\to\tau_k)\to\tau)\quad(5)$$

Figure 1: Term calculus for binding signature $\Sigma=(T,U,\mathcal{O})$ (in all rules, $|\Gamma|=n$)

The remaining three rules are the usual ones for structural manipulation of contexts. Strictly speaking, these rules are admissible and thus could be omitted.

Given a signature Σ, a context Γ and a type τ, we denote the *set of (α-equivalence classes of) terms of type τ at* Γ by $S_\tau(\Gamma)\triangleq\{t\mid\Gamma\vdash t:\tau\}$.

2. ALGEBRAIC TYPED HOAS

In this section we give an abstract algebraic presentation of typed abstract syntax with bindings.

The term calculus of Figure 1 stratify sets of terms according to the typing contexts. As pointed out in [4, 7], an adequate universe for reflecting this situation is some functor category of the form $\mathcal{S}et^{\mathcal{C}}$, where the index category $\mathcal{C}$ is a suitable category of contexts. In the case of typed syntax [3], the category of contexts turns out to be the free cocartesian category over the category of variable types, whose features we recall in §2.1. Then, in §2.2 we analyze the category $\mathcal{S}$ where each syntactic sort will be interpreted. Using the properties and type constructors of $\mathcal{S}$, in §2.3 the language generated by a typed binding signature is described as the initial Σ-algebra, for a suitable Σ signature endofunctor on the category of "presentation models" $\mathrm{Mod}_T(\mathcal{S})$.

2.1 The structure of typed contexts

Let $\Sigma=(T,U,\mathcal{O})$ be a signature. Following [3], a context $\Gamma=\{x_1:\tau_1,\ldots,x_n:\tau_n\}$ can be defined more abstractly as a map $\Gamma:n\to U$ in $\mathcal{S}et$, where n is the cardinal $\{1,\ldots,n\}$; in the following, it will be denoted by $|\Gamma|$. These maps are the objects of the comma category $\mathcal{U}\triangleq(in\downarrow U)$ where $in:\mathbb{F}\hookrightarrow\mathcal{S}et$ is the inclusion functor of $\mathbb{F}$, the full subcategory of $\mathcal{S}et$ of finite cardinals. Morphisms of $\mathcal{U}$ are type-preserving renamings, that is $\rho:\Gamma_1\to\Gamma_2$ is a morphism $\rho:|\Gamma_1|\to|\Gamma_2|$ in $\mathbb{F}$ such that $\Gamma_1=\Gamma_2\circ\rho$.

Context concatenation corresponds to coproduct in $\mathcal{U}$: if $\Gamma_1:n_1\to U$ and $\Gamma_2:n_2\to U$, then $\Gamma_1+\Gamma_2=[\Gamma_1,\Gamma_2]:n_1+n_2\to U$. In fact, the category $\mathcal{U}$ of typed cartesian contexts over U is the free cocartesian category over U.

Let $\tau\in T$; by abuse of notation, let us denote by τ also the context $\tau:1\to U$, object of $\mathcal{U}$. We have then the following coproduct in $\mathcal{U}$, for two chosen inclusion maps:

$$\Gamma\xrightarrow{old_{\tau,\Gamma}}\Gamma+\tau\xleftarrow{fresh_{\tau,\Gamma}}\tau$$

Notice that all morphisms of $\mathcal{U}$ can be generated by closing under composition three families of maps, which correspond to the structural rules in Figure 1:

- exchange: $s:\Gamma+\tau_1+\tau_2\to\Gamma+\tau_2+\tau_1$ such that $\Gamma+\tau_1+\tau_2=(\Gamma+\tau_2+\tau_1)\circ s$;
- weakening: $w:\Gamma\to\Gamma+\tau$ such that $\Gamma=(\Gamma+\tau)\circ w$;
- contraction: $c:\Gamma+\tau+\tau\to\Gamma+\tau$ such that $\Gamma+\tau+\tau=(\Gamma+\tau)\circ c$;

2.2 The ambient category $\mathcal{S}$

The category where each syntactic sort will be interpreted is $\mathcal{S}\triangleq\mathcal{S}et^{\mathcal{U}}$, whose objects are sets stratified by typed contexts. As usual for topoi, they are called *types* of the category, and are ranged over by A,B,C. Notice that they *do not* correspond to syntactic types $\tau\in T$. Syntactic types are interpreted in $\mathcal{S}$ by a functor $S:T\longrightarrow\mathcal{S}$; $S_\tau(\Gamma)$ is the set of terms of type τ in the context Γ. Given a variable substitution $\rho:\Gamma\to\Gamma'$, then $S_\tau(\rho):S_\tau(\Gamma)\to S_\tau(\Gamma')$ is the extension by structural recursion of ρ to terms of type τ; for this reason, we will denote $S_\tau(\rho)(t)$ as $t\{\rho\}$.

A morphism $f:A\longrightarrow B$ in $\mathcal{S}$ is a natural transformations, that is a family of functions $\{f_\Gamma:A(\Gamma)\to B(\Gamma)\mid\Gamma\in\mathcal{U}\}$, satisfying the naturality condition: for all $\rho:\Gamma\to\Gamma'$ in $\mathcal{U}$, $f_\Gamma\circ A(\rho)=B(\rho)\circ f_{\Gamma'}$. Thus, a morphism between languages (of a given type τ) has to be stratified accordingly to the context structure and has to respect the structural operations on contexts.

Notice that the naturality condition rules out some morphisms which one may expect to define on syntactic terms. For instance, the family of functions $fvcount_\tau(\Gamma):S_\tau(\Gamma)\to Nat$ in $\mathcal{S}et$, counting the free variables of terms (of any given language), is not natural (consider, e.g., the contraction map $c:\Gamma+\tau+\tau\to\Gamma+\tau$). Therefore, there is no morphism $fvcount_\tau:S_\tau\longrightarrow Nat$ in $\mathcal{S}$ counting the free variables of terms. Still, the morphisms of $\mathcal{S}$ are rich enough to contain interesting maps, such as fully typed simultaneous substitution (see §3.3) or normalization functions [3].

As mentioned before, sets of terms are indexed also by types in T. Thus a complete description of the language generated by a signature is a T-indexed family of functors in $\mathcal{S}$. These families form functors from T to $\mathcal{S}$, that is objects of the category of *presentation models* of T:

$$\mathrm{Mod}_T(\mathcal{S})\triangleq\mathcal{S}^T=(\mathcal{S}et^{\mathcal{U}})^T\cong\mathcal{S}et^{\mathcal{U}\times T}$$

where T is viewed as a discrete category. In this paper, we are interested in simple type theories; more complex theories, e.g. with subtyping or polymorphism, can be modelled by adding suitable structure to the category T. When T is discrete, the structure of $\mathrm{Mod}_T(\mathcal{S})$ is essentially the same of $\mathcal{S}$, which we recall next, following the pattern of [3, 4].

Limits and colimits. Like the degenerate case $\mathcal{S}et^{\mathbb{F}}$, the category $\mathcal{S}$ has a rich and interesting structure: it is a complete and cocomplete cartesian category (it is a topos, actually), where limits and colimits can be computed pointwise [14]. Thus we have automatically sums, products and exponentials. We recall that the exponential is defined on

objects as follows:

$$(A \Rightarrow B)_\Gamma \triangleq \mathcal{S}(\mathbf{y}(\Gamma) \times A, B) \tag{6}$$

where $\mathbf{y} : \mathcal{U}^{op} \longrightarrow \mathcal{S}$ is the Yoneda embedding: $\mathbf{y}(\Gamma) = \mathcal{U}(\Gamma, \_)$, and for $\rho : \Gamma \to \Gamma'$, $\mathbf{y}(\rho) = \_ \circ \rho$.

Typed variables. For each $\tau \in T$, we define the presheaf of *(abstract) variables of type* τ as

$$Var_\tau \triangleq \mathbf{y}(\tau) \triangleq \mathcal{U}(\tau, \_)$$

where $\tau : 1 \to T$ is the abstract context with one variable of type τ. More explicitly, $Var_\tau(\Gamma)$ is the restriction of the cardinal $|\Gamma|$ to the elements of type τ:

$$Var_\tau(\Gamma) = \mathcal{U}(\tau, \Gamma) \cong \Gamma^{-1}(\tau) \qquad Var_\tau(\rho) = \rho \restriction \Gamma^{-1}(\tau)$$

There is also the presheaf $AllVar \in \mathcal{S}$ of all declared variables, defined by $AllVar(\Gamma) = |\Gamma|$, and $AllVar(\rho) = \rho$. Notice that $AllVar$ is not representable.

Typed operations. A non-syntactic construction of types in $\mathcal{S}$ is the following. Let $\mathcal{C}$ be a cartesian category, and $A : U \longrightarrow \mathcal{C}$ a functor. Let $A^* : \mathcal{U}^{op} \longrightarrow \mathcal{C}$ the *slice extension* of A, defined as follows:

$$A^*(\Gamma) \triangleq \prod_{1 \le i \le |\Gamma|} A(\Gamma(i))$$

$$A^*(\rho) \triangleq (\pi_{\rho 1}, \ldots, \pi_{\rho|\Gamma|}) \qquad (\rho : \Gamma \to \Gamma')$$

Then, for B an object of $\mathcal{C}$, the presheaf of *(typed) operations* from A to B is the functor $\langle A, B\rangle : \mathcal{U} \longrightarrow \mathcal{S}et$ defined as follows

$$\langle A, B\rangle(\Gamma) \triangleq \mathcal{C}\,(A^*(\Gamma), B)$$

$$\langle A, B\rangle(\rho)(f) \triangleq f \circ \langle \pi_{\rho 1}, \ldots, \pi_{\rho|\Gamma|}\rangle \qquad (\rho : \Gamma \to \Gamma')$$

Context extension. For each $\tau \in T$, we can define a *typed context extension* constructor $\delta_\tau : \mathcal{S} \longrightarrow \mathcal{S}$, by precomposition with the operation of context extension at the level of contexts: $\delta_\tau A \triangleq A \circ (\_ + \tau)$. More explicitly:

$$(\delta_\tau A)(\Gamma) = A(\Gamma + \tau) \quad (\delta_\tau A)(\rho) = A(\rho + id_\tau)$$

Therefore, elements of $(\delta_\tau A)(\Gamma)$ can be seen as elements of A with possibly a "hole" of type τ. Indeed:

PROPOSITION 1. *For all $\tau \in T, A \in \mathcal{S}$: $\delta_\tau A \cong Var_\tau \Rightarrow A$*

PROOF. $(Var_\tau \Rightarrow A)_\Gamma = \mathcal{S}(\mathbf{y}(\Gamma) \times Var_\tau, A) \cong \mathcal{S}(\mathbf{y}(\Gamma + \tau), A) \cong A_{\Gamma+\tau} = (\delta_\tau A)_\Gamma$. □

This equivalence is at the core of higher-order abstract syntax representations in type theory-based λ-calculi; see e.g. [9], among others.

The same property can be shown also by observing that δ_τ is right adjoint of $\_ \times Var_\tau$, that is

$$\mathcal{S}(A, \delta_\tau B) \cong \mathcal{S}(A \times Var_\tau, B) \tag{7}$$

Indeed, given $f : A \longrightarrow \delta_\tau B$, we can define $g : A \times Var_\tau \longrightarrow B$ as $g_\Gamma(a, i) = f_\Gamma(a)\{\rho\}$, where $\rho = [id_\Gamma, i] : \Gamma + \tau \to \Gamma$. On the other hand, given $g : A \times Var_\tau \longrightarrow B$, we define $f_\Gamma(a) = g_{\Gamma+\tau}(a\{old_{\tau,\Gamma}\}, fresh_{\tau,\Gamma})$.

Moreover, δ_τ is left adjoint of the functor $\langle Q_\tau, \_\rangle : \mathcal{S} \longrightarrow \mathcal{S}$, where $Q_\tau : \mathcal{U}^{op} \longrightarrow \mathcal{S}$ is $Q_\tau(\Gamma)(\Gamma') \triangleq \mathcal{U}(\Gamma, \Gamma' + \tau)$. In other words, we have

$$\mathcal{S}(\delta_\tau A, B) \cong \mathcal{S}(A, \langle Q_\tau, B\rangle) \tag{8}$$

Indeed, given $f : \delta_\tau A \longrightarrow B$, we can define $g : A \longrightarrow \langle Q_\tau, B\rangle$ as

$$g_\Gamma(a) : Q_\tau(\Gamma) \longrightarrow B$$

$$(g_\Gamma(a))_{\Gamma'} : \mathcal{U}(\Gamma, \Gamma' + \tau) \longrightarrow B_{\Gamma'}$$

$$(g_\Gamma(a))_{\Gamma'}(\rho) \triangleq f_{\Gamma'}(a\{\rho\})$$

On the other hand, if $g : A \longrightarrow \langle Q_\tau, B\rangle$ then we can define $f : \delta_\tau A \longrightarrow B$ as $f_\Gamma(a) \triangleq (g_{\Gamma+\tau}(a))_\Gamma(id_{\Gamma+\tau})$.

Thus we have proved the following

PROPOSITION 2. $- \times Var_\tau \dashv \delta_\tau \dashv \langle Q_\tau, -\rangle$

COROLLARY 1. *δ_τ preserves both limits and colimits.*

Moreover, each δ_τ is a monad over $\mathcal{S}$. The unit $up_\tau : \mathrm{Id} \longrightarrow \delta_\tau$ and the multiplication $contr_\tau : \delta_\tau^2 \longrightarrow \delta_\tau$ arise from the structural rules of contexts, i.e., from weakening and contraction of contexts in $\mathcal{U}$:

$$up_{\tau,A,\Gamma} \triangleq A(w) : A(\Gamma) \to A(\Gamma + 1)$$

$$contr_{\tau,A,\Gamma} \triangleq A(c) : A(\Gamma + \tau + \tau) \to A(\Gamma + \tau)$$

In the same way, the exchange map $s : \Gamma + \tau_1 + \tau_2 \to \Gamma + \tau_2 + \tau_1$ lifts to a natural transformation

$$swap_{\tau_1,\tau_2} : \delta_{\tau_1}\delta_{\tau_2} \longrightarrow \delta_{\tau_2}\delta_{\tau_1}\ , \tag{9}$$

which exchanges the order of "holes". Notice that $swap_{\tau_1,\tau_2} \circ swap_{\tau_2,\tau_1} = Id_{\mathcal{S}}$.

In the following, for $\vec{\tau} = \tau_1, \ldots, \tau_n$ a list of types, we will denote by $\delta_{\vec{\tau}}$ the composition $\delta_{\tau_1} \circ \cdots \circ \delta_{\tau_n}$.

Finally, we observe that the following is a coproduct diagram:

$$Var_\tau \xrightarrow{old_\tau} \delta_\tau\, Var_\tau \xleftarrow{fresh_\tau} 1 \tag{10}$$

where $old_\tau = \{old_{\tau,\Gamma} \restriction Var_{\tau,\Gamma} : Var_{\tau,\Gamma} \longrightarrow Var_{\tau,\Gamma} + 1\}$ and $fresh_\tau = \{fresh_{\tau,\Gamma} : 1 \longrightarrow Var_{\tau,\Gamma} + 1\}$. Notice that

$$\delta_\tau\, Var_\sigma = Var_\sigma \quad \text{ if } \sigma \neq \tau\ .$$

Thus, δ_τ can be seen as a *partial differentiation operator*.

2.3 Abstract typed syntax with binders as initial algebras

Using the type constructors and properties of $\mathcal{S}$ described in the previous section, we can now give an algebraic characterization of the language generated by any typed binding signature $\Sigma = (T, U, \mathcal{O})$.

Recall that the set of terms (up-to α-equivalence) of type τ at Γ is defined as $S_\tau(\Gamma) \triangleq \{t \mid \Gamma \vdash t : \tau\}$. This definition can be extended to a presheaf $S_\tau : \mathcal{S}$ by adding the obvious action on morphisms: for $\rho : \Gamma \to \Gamma'$ and $t \in S_\tau(\Gamma)$,

$$S_\tau(\rho)(t) \triangleq t\{\rho\} = t\{x_{\rho 1}/x_1, \ldots, x_{\rho n}/x_n\} \in S_\tau(\Gamma')\ .$$

Each S_τ is the object part of a functor $S : T \longrightarrow \mathcal{S}$, the *abstract syntax*; since in this paper we consider only T discrete, the action on morphisms is trivial. Therefore, the language of terms generated by Σ is an object S of $\mathrm{Mod}_T(\mathcal{S})$. We will see now that S can be characterized as the initial algebra of a functor associated to the signature, similarly to the specific case of simply typed λ-calculus in [3].

Given the signature Σ above, we associate the functor $\Sigma : \mathrm{Mod}_T(\mathcal{S}) \longrightarrow \mathrm{Mod}_T(\mathcal{S})$ defined as follows:

$$(\Sigma(R))_\tau \triangleq (\tau \in U?\, Var_\tau) + \coprod_{op \in \mathcal{O}((\vec{\sigma}_1 \to \tau_1) \times \cdots \times (\vec{\sigma}_n \to \tau_n) \to \tau)} \prod_{1 \le i \le n} \delta_{\vec{\sigma}_i} R_{\tau_i}$$

where "$(\tau \in U?A)$" is a syntactic shorthand for "A" if $\tau \in U$, "0" otherwise.

We can consider thus the category Σ-Alg, of Σ-algebras, that is pairs (R, r) where $R \in \mathrm{Mod}_T(\mathcal{S})$ and $r : FR \longrightarrow R$ (the *structure*). Each object of this category can be used to interpret the syntactic constructors of the signature: given a Σ-algebra (R, r), an operator $op : (\vec{\sigma}_1 \to \tau_1) \times \cdots \times (\vec{\sigma}_n \to \tau_n) \to \tau$ can be mapped to the corresponding component of the structure r, mapping the component $\prod_{1 \le i \le n} \delta_{\vec{\sigma}_i} R_{\tau_i}$ back into R_τ.

The functor S can be endowed with a structure s of Σ-algebra, given by the syntactic constructor themselves. For each type τ, and context Γ, we define $s_{\tau,\Gamma} : (\Sigma(S))_\tau(\Gamma) \longrightarrow S_\tau(\Gamma)$, as follows:

- if $\tau \in U$, then $s_{\tau,\Gamma}(x) = x$ for $x \in Var_\tau(\Gamma)$
- for $(t_1, \ldots, t_n) \in (\prod_{1 \le i \le n} \delta_{\vec{\sigma}_i} S_{\tau_i})_\Gamma = \prod_{1 \le i \le n} S_{\tau_i, \Gamma + \vec{\sigma}_i}$, let

  $$s(t_1, \ldots, t_n) = op((\vec{x}_1{:}\vec{\sigma}_1)t_1, \ldots, (\vec{x}_n{:}\vec{\sigma}_n)t_n)$$

  where $\vec{x}_i$ is the list of variables in the domain of $\Gamma + \vec{\sigma}_i$ added to $|\Gamma|$.

THEOREM 1. *The syntactic model S, associated to a signature Σ, endowed with the structure s above, is the Σ-initial algebra, and thus the initial object of Σ-Alg.*

This result allows to define morphisms out of $\mathcal{S}$ by initiality, i.e., by structural recursion on the syntax of typed terms with binders, as we will see in §3.

Terms with "holes". Let $\sigma \in T$ be a type, and let us consider the functor $\delta_\sigma \circ S : T \longrightarrow \mathcal{S}$. Since

$$(\delta_\sigma S)_\tau(\Gamma) = \delta_\sigma(S_\tau(\Gamma)) = S_\tau(\Gamma + \sigma) \ ,$$

the functor $\delta_\sigma S$ can be seen as the language generated by the signature Σ, but where we admit a "hole" (a metalevel variable) of type σ. More generally, given a list $\vec{\sigma}$ of types, we can consider the functor $\delta_{\vec{\sigma}} S$ of *term contexts with $\vec{\sigma}$ holes*. We need often to reason by induction on the structure of term contexts, much like on the structure of plain terms. For instance, substitution can be defined by structural recursion over term contexts (see [4, 7] for the untyped case). Thus it is important to give a characterization of these languages as initial algebras.

Let us consider the functor $\delta_{\vec{\sigma}}\Sigma$ over $\mathrm{Mod}_T(\mathcal{S})$. Using the *swap* isomorphism (9) several times, we have that for any $\vec{\sigma}'$: $\delta_{\vec{\sigma}}\delta_{\vec{\sigma}'} \cong \delta_{\vec{\sigma}'}\delta_{\vec{\sigma}}$. By Proposition 2, $\delta_{\vec{\sigma}}$ preserves products and coproducts, thus it distributes on the "op" part of Σ as follows:

$$((\delta_{\vec{\sigma}}\Sigma)(R))_\tau = (\tau \in U?\, \delta_{\vec{\sigma}}\, Var_\tau) + \coprod_{op \in \mathcal{O}((\vec{\sigma}_1 \to \tau_1) \times \cdots \times (\vec{\sigma}_n \to \tau_n) \to \tau)} \prod_{1 \le i \le n} \delta_{\vec{\sigma}_i}\delta_{\vec{\sigma}} R_{\tau_i}$$

Now, by the coproduct (10), we have that

$$\delta_{\vec{\sigma}}\, Var_\tau \cong Var_\tau + n$$

where n is the number of occurrences of τ in $\vec{\sigma}$. Writing n as $\mathcal{U}(\tau, \vec{\sigma})$, we have:

$$\begin{aligned}((\delta_{\vec{\sigma}}\Sigma)(R))_\tau &\cong (\Sigma\delta_{\vec{\sigma}}(R))_\tau + (\tau \in U?\mathcal{U}(\tau, \vec{\sigma})) \\ &= ((\Sigma + (\_ \in U?\mathcal{U}(\_, \vec{\sigma})))\delta_{\vec{\sigma}}R)_\tau\end{aligned}$$

If we define the endofunctor $\Sigma_{\vec{\sigma}}$ on $\mathrm{Mod}_T(\mathcal{S})$ as follows

$$(\Sigma_{\vec{\sigma}}(R))_\tau \triangleq \Sigma(R)_\tau + (\tau \in U?\mathcal{U}(\tau, \vec{\sigma})) \ ,$$

we have proved that $\delta_{\vec{\sigma}}\Sigma \cong \Sigma_{\vec{\sigma}}\delta_{\vec{\sigma}}$. Therefore, by applying [7, Theorem 7.1], we can "lift" the (initial) Σ-algebras to the $\Sigma_{\vec{\sigma}}$-algebras. More formally, we have the following characterization of languages with holes:

THEOREM 2. *For all $\vec{\sigma}$, the $\Sigma_{\vec{\sigma}}$-initial algebra is $\delta_{\vec{\sigma}}S$.*

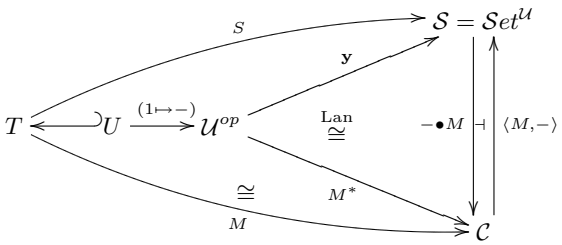
Notice that by definition of $\Sigma_{\vec{\sigma}}$, the structure of $\Sigma_{\vec{\sigma}}$-algebras are essentially the same of Σ-algebras, possibly plus constant injections for the "holes" (for the types with variables). This means that definitions by structural recursion over term contexts have exactly the same pattern of recursions over terms, but possibly extended with as many base cases as needed for the "holes" of the correct type.

3. REFLECTING THE SEMANTICS

As we have seen in the previous section, the category $\mathrm{Mod}_T(\mathcal{S})$ is well suited for the characterization of the syntax generated by typed signatures over types T. One may wonder whether this category is rich enough for containing also any given *semantics* of typed languages. In this section, we address this question.

In general, one can give types and terms an interpretation in a category $\mathcal{C}$, different from $\mathcal{S}$. Let us denote by $M : T \to \mathcal{C}$ (the intended "meaning") this interpretation. For each τ, we can see variables of type Var_τ as "ranging over" the abstract generals in M_τ. The choice of the category $\mathcal{C}$ depends on the particular computational notion we are focusing on; see e.g. [16]. For instance, in order to interpret typed λ-calculus, we can choose any cartesian closed category, but not necessarily $\mathcal{S}$. Thus, in general, the preferred semantic model M of types is in the category $\mathrm{Mod}_T(\mathcal{C})$, while the syntactic model S is in $\mathrm{Mod}_T(\mathcal{S})$. Nevertheless, we often need to reason on *semantic* objects (i.e., in $\mathcal{C}$) using *structural* properties of terms, such as structural recursion which corresponds to initiality of $\mathcal{S}$ on Σ-algebras over $\mathcal{S}$.

In fact, if $\mathcal{C}$ is cocomplete, there is an adjunction which connects the given "semantic" category $\mathcal{C}$ to the "preferred syntactic" category $\mathcal{S}$, as in [2, 3]. For our typed binding signatures, the situation is as follows:



(11)

This setting gives insights on the interplay between syntactic and semantic notions. The adjunction allows one to "transfer" constructions from the first to the second and vice versa. Therefore, any framework for abstract syntax, complete with semantics for the syntax in it, can be reflected in the same, well-established presheaf category $\mathcal{S}$.

In this section, we first describe the construction (11), which is a simple generalization of that in [2]. Then, we will exemplify its use in some interesting situations: initial semantics for the simply typed λ-calculus, typed substitution in a monoidal category whose monoids generalize Lawvere algebraic theories and a suitable semantics of names for a nominal calculus such as the π-calculus.

3.1 From abstract syntax to semantics, and back

Let $\mathcal{C}$ be a cocomplete category, and $M \in \mathrm{Mod}_T(\mathcal{C})$ a functor (the "meaning" of types). The definition of presheaves of typed operations (§2.2) can be extended to a functor $\langle M, \_\rangle : \mathcal{C} \longrightarrow \mathcal{S}$, by adding the postcomposition action on morphisms: for $f : A \to B$ in $\mathcal{C}$, $\langle M, f\rangle_\Gamma = f \circ \_ : \langle M, A\rangle_\Gamma \longrightarrow \langle M, B\rangle_\Gamma$.

By applying [14, Theorem I.5.2], we have that there is a left adjoint $\_ \bullet M : \mathcal{S} \longrightarrow \mathcal{C}$ given by

$$A \bullet M \triangleq Colim \quad \int A \xrightarrow{\pi_A} \mathcal{U}^{op} \xrightarrow{M^*} \mathcal{C}$$

where $\int A$ is the *category of elements* of A. Then, applying [13, Theorem X.7.1], we obtain $A \bullet M = \int^{\Gamma \in \mathcal{U}} A_\Gamma \cdot M^*(\Gamma)$, i.e., the left Kan extension of M^* along the Yoneda embedding ($\mathrm{Lan}_{\mathbf{y}}(M^*)$), where $\cdot : \mathcal{S}et \times \mathcal{C} \longrightarrow \mathcal{C}$ is the usual copower. Hence, we have the following:

Proposition 3. *If $\mathcal{C}$ is a cocomplete category, there is an adjunction $\_ \bullet M \dashv \langle M, \_\rangle$, where $\_ \bullet M : \mathcal{S} \longrightarrow \mathcal{C}$ and $\langle M, \_\rangle : \mathcal{C} \longrightarrow \mathcal{S}$ are given as follows*

$$A \bullet M \triangleq \int^{\Gamma \in \mathcal{U}} A_\Gamma \cdot M^*(\Gamma) \qquad \langle M, B\rangle_\Gamma \triangleq \mathcal{C}(M^*(\Gamma), B)$$

Thus, we have a bijection $\phi_{A,B} : \mathcal{C}(A \bullet M, B) \cong \mathcal{S}(A, \langle M, B\rangle)$ natural in $A \in \mathcal{S}$ and $B \in \mathcal{C}$. Moreover, we can "lift" the adjunction to the model categories $\mathrm{Mod}_T(\mathcal{S})$ and $\mathrm{Mod}_T(\mathcal{C})$:

Theorem 3. *If $\mathcal{C}$ is a cocomplete category, there is an adjunction*

$$Mod_T(\mathcal{S}) \underset{G}{\overset{F}{\underset{\longleftarrow}{\overset{\longrightarrow}{\bot}}}} Mod_T(\mathcal{C}) \tag{12}$$

where F is given by $F_A(\tau) \triangleq A_\tau \bullet M$ and G is defined by $G_B(\tau) \triangleq \langle M, B_\tau\rangle$.

Proof. (Sketch) In order to prove this result, we have to show that there is a bijection $\phi'_{A,B} : \mathcal{C}^T(F_A, B) \cong \mathcal{S}^T(A, G_B)$. Hence, given $A \in \mathcal{S}^T$, $B \in \mathcal{C}^T$ and $f : F_A \longrightarrow B$, $\phi'_{A,B}(f)$ must be a natural transformation from A to G_B. It follows that, given $\tau \in T$, $\phi'_{A,B}(f)_\tau$ must be a morphism from A_τ to $\langle M, B_\tau\rangle$. To conclude it is sufficient to define $\phi'_{A,B}(f)_\tau$ as $\phi_{A_\tau, B_\tau}(f_\tau)$, where $\langle \_ \bullet M, \langle M, \_\rangle, \phi\rangle$ is the adjunction of Proposition 3. So doing, the required properties of ϕ' (bijectivity and naturality) follow from those of ϕ. □

In the following, by abuse of notation, we will write $\_ \bullet M$ and $\langle M, \_\rangle$ instead of F and G, respectively.

Notice that $Var \bullet M \cong M$. Indeed, $Var_\tau \bullet M = \int^\Gamma \Gamma^{-1}(\tau) \cdot M^*(\Gamma) \cong M_\tau$.

3.2 Typed Initial Semantics

Let us consider the signature $\Sigma_{\lambda\to}$ of simply typed λ-calculus, given in Example 1. This case has been presented first and studied in depth by Fiore in the semantic analysis of normalisation by evaluation [3]. Here we consider this case to show how the application of the general setting (11) may bring into light new connections between semantic and syntactic aspects, even for a so well-known language.

The signature endofunctor $\Sigma : \mathrm{Mod}_T(\mathcal{S}) \longrightarrow \mathrm{Mod}_T(\mathcal{S})$ is the following:

$$(\Sigma(R))_\tau = Var_\tau + \coprod_{\sigma \in T} R_{\sigma \supset \tau} \times R_\sigma + \coprod_{\substack{\tau_1, \tau_2 \in T \\ \tau = (\tau_1 \supset \tau_2)}} \delta_{\tau_1} R_{\tau_2}$$

whose initial algebra Λ is, by Theorem 1, the language of simply typed λ-terms.

For each type $\tau \in T$, let us denote by M_τ its usual interpretation in a given Cartesian closed category $\mathcal{C}$ (see e.g. [11]). The image of M in $\mathrm{Mod}_T(\mathcal{S})$ along the adjunction (12) is $\langle M, M\rangle$, the *clone of typed operations* of M, being the generalization of the similar structure used in universal algebra.

Using the properties of $\mathcal{C}$, we can give $\langle M, M\rangle$ a structure $\gamma : \Sigma\langle M, M\rangle \longrightarrow \langle M, M\rangle$ of Σ-algebra in $\mathcal{S}$:

$$\gamma_{\tau,\Gamma}(i : Var_\tau(\Gamma)) = \pi_i$$
$$\gamma_{\tau,\Gamma}(f : M^*(\Gamma) \to M_{\sigma\supset\tau}, a : M^*(\Gamma) \to M_\sigma) = \\ = ev \circ \langle unfold \circ f, a\rangle$$
$$\gamma_{\tau,\Gamma}(f : M^*(\Gamma + \tau_1) \to M_{\tau_2}) = fold \circ curry(f)$$

By initality of Λ there exists a unique Σ-homomorphism $tr : \Lambda \longrightarrow \langle M, M\rangle$. At stage Γ, and for type τ, this map translates, *by structural recursion*, a λ-term t such that $\Gamma \vdash t : \tau$, in (the representation of) a map $tr_\tau(t) : M^*(\Gamma) \to M_\tau$. Hence, tr acts as a (compositional) *compiler*. It is easy to see that this corresponds to the usual interpretation of typed λ-calculus.

On the other hand, we can consider also $\Lambda \bullet M$ in $\mathrm{Mod}_T(\mathcal{C})$. Elements of $(\Lambda \bullet M)_\tau$ are (equivalence classes of) tuples of the form $(t, \vec{d}) : \Lambda_\tau(\Gamma') \cdot M^*(\Gamma')$. These tuples can be seen as a program t, equipped with enough data (of the right types) to be associated to the free variables of t, i.e., the inputs. Moreover, $\Lambda \bullet M$ is a Σ'-algebra, where $\Sigma' : \mathrm{Mod}_T(\mathcal{C}) \longrightarrow \mathrm{Mod}_T(\mathcal{C})$ is defined by the same polynomial of Σ: for $M : T \longrightarrow \mathcal{C}$:

$$(\Sigma'(M))_\tau \triangleq M_\tau + \coprod_{\sigma \in T} M_{\sigma \supset \tau} \times M_\sigma + \coprod_{\substack{\tau_1, \tau_2 \in T \\ \tau = (\tau_1 \supset \tau_2)}} \delta'_{\tau_1} M_{\tau_2}$$

where $\delta'_{\tau_1} : \mathcal{C} \longrightarrow \mathcal{C}$ is the lifting of δ_{τ_1} on $\mathcal{S}$ along $\_ \bullet M$. Recalling that $M_\tau \cong (Var \bullet M)_\tau$, we can prove that $\Sigma'(\_ \bullet M) \cong (\Sigma\_) \bullet M$, and thus, by [7, Theorem 7.1], the Σ'-initial algebra is exactly $\Lambda \bullet M$, because Λ is the initial Σ-algebra. But also M can be given a Σ'-structure, as above; therefore there exists a unique Σ'-homomorphism $eval : \Lambda \bullet M \longrightarrow M$. Given a program equipped with data $(t, \vec{d}) \in \Lambda_\tau \bullet M$, $eval(t, \vec{d})$ *evaluates* to a value in M_τ, by *structural recursion over t*. Therefore, *eval* acts as an *interpreter*.

Notice that *tr* and *eval* are images of each other along the adjunction $\_ \bullet M \dashv \langle M, \_\rangle$.

3.3 Monoids and typed substitution

In this subsection we give an algebraic theory of typed substitution, along the lines of [4]. This theory naturally

arises when the abstract generals denoted by the variables are terms themselves. Therefore, let $\mathcal{C} = \mathcal{S}$; we have that

$$(A \bullet M)_\tau \triangleq A_\tau \bullet M = Colim \quad \int A_\tau \xrightarrow{\pi_{A_\tau}} \mathcal{U}^{op} \xrightarrow{M^*} \mathcal{S} \quad .$$

The latter can be expressed as the following coequalizer, where Δ is an object of $\mathcal{U}$:

$$\coprod_{\substack{\Gamma,\gamma\in(A_\tau)(\Gamma)\\ \rho:\Gamma\longrightarrow\Gamma'}} (M^*(\Gamma'))(\Delta) \underset{\theta'}{\overset{\theta}{\rightrightarrows}} \coprod_{\Gamma,\gamma\in(A_\tau)(\Gamma)} (M^*(\Gamma))(\Delta) \xrightarrow{\psi} (A_\tau \bullet M)(\Delta)$$

where θ maps each $(M^*(\Gamma'))(\Delta)$, indexed by (Γ', γ, ρ), to the corresponding summand indexed by $(\Gamma', \gamma' = \gamma\rho)$ by means of $\mathbf{1} : (M^*(\Gamma'))(\Delta) \longrightarrow (M^*(\Gamma'))(\Delta)$, while θ' maps the same $(M^*(\Gamma'))(\Delta)$ to the summand indexed by (Γ, γ) by means of $(M^*(\rho))(\Delta) : (M^*(\Gamma'))(\Delta) \longrightarrow (M^*(\Gamma))(\Delta)$. Then, observing that the second coproduct can be written as $\coprod_\Gamma ((A_\tau)(\Gamma) \times (M^*(\Gamma))(\Delta))$, we obtain the following:

$$((A \bullet M)_\tau)(\Delta) = \left(\coprod_{\Gamma\in\mathcal{U}} \left((A_\tau)(\Gamma) \times \prod_{i\in|\Gamma|} (M_{\Gamma(i)})(\Delta) \right) \right)_{/\approx}$$

where $\approx$ is the equivalence relation generated by the coequalizer's property, i.e.,

$$(a; b_{\rho(1)}, \ldots, b_{\rho(n)}) \approx (a\{\rho\}; b_1, \ldots, b_m) \ , \qquad (13)$$

where $\rho \in \mathcal{U}(\Gamma, \Gamma')$, i.e., $\rho : |\Gamma| \longrightarrow |\Gamma'|$ and $\Gamma = \Gamma' \circ \rho$.

If $T = U$, the $\bullet$ turns $\mathrm{Mod}_T(\mathcal{S})$ into a closed monoidal category $(\mathrm{Mod}_T(\mathcal{S}), \bullet, Var)$ whose monoids correspond to *typed* (i.e., heterogeneous) substitutions; indeed, the intuitive meaning of an object $(a; b_1, \ldots, b_n) \in ((A \bullet M)_\tau)(\Gamma')$ is that of a "suspended" substitution "respecting types": for some Γ such that $|\Gamma| = n$, it is $a \in A_\tau(\Gamma)$ and $b_i \in M_{\Gamma(i)}(\Gamma')$.

Similarly to [4], we can now define in the internal language of $\mathrm{Mod}_T(\mathcal{S})$ the simultaneous typed substitution

$$\sigma : \Lambda \bullet \Lambda \longrightarrow \Lambda$$

for the typed λ-calculus of §3.2: for $(t; \vec{u}) \in (\Lambda \bullet \Lambda)_\tau(\Gamma)$:

$$\begin{aligned}
\sigma_{\tau,\Gamma}(t; \vec{u}) = \ & \mathsf{case}\ t\ \mathsf{of} \\
& var_\tau(i) \Rightarrow u_i \\
& app_{\tau',\tau}(t_1, t_2) \Rightarrow app_{\tau',\tau}(\sigma_{\tau'\supset\tau,\Gamma}(t_1; \vec{u}), \sigma_{\tau',\Gamma}(t_2; \vec{u})) \\
& lam_{\tau_1,\tau_2}(t') \Rightarrow lam_{\tau_1,\tau_2}(\sigma_{\tau_1,\Gamma+\tau_2}(t', [\vec{u}, var_{\tau_2}(fresh_{\tau_2})])) \\
& \mathsf{end}
\end{aligned}$$

where $var : Var \longrightarrow \Lambda$, $app : \Lambda \times \Lambda \longrightarrow \Lambda$ and $lam : \delta\Lambda \longrightarrow \Lambda$ form the algebraic structure of Λ (§3.2).

It is worth noticing that, if we have only one type (i.e., $T = \mathbf{1}$), then $\mathcal{S} \cong \mathcal{S}et^{\mathbb{F}}$ and, as pointed out in [4], the monoids in $(\mathcal{S}, \bullet, Var)$ correspond to Lawvere algebraic theories. Thus we can say that the typed monoids are a generalization of the latter, and it is interesting future work to determine what generalisation it is.

3.4 Variables and Names

We apply the framework above to address a common issue of HOAS techniques, namely: what is a variable, what is a name, and what is a variable ranging over names? This is important e.g. for languages for HOAS manipulation like FreshML [5], and for nominal calculi like the π-calculus, which we use here for definiteness.

While variables are syntactic entities, names are semantic entities; thus we choose $\mathcal{C} \triangleq \mathcal{S}et^{\mathbb{I}}$, where $\mathbb{I}$ is the subcategory of $\mathbb{F}$ with injective morphisms only. Following the pattern of §3.3, we obtain

$$((A \bullet M)_\tau)(n) = \left(\coprod_{\Gamma\in\mathcal{U}} \left((A_\tau)(\Gamma) \times \prod_{i\in|\Gamma|} (M_{\Gamma(i)})(n) \right) \right)_{/\approx}$$

where $n \in \mathbb{I}$ and $\approx$ is the equivalence relation defined as before (13).

Since the maps of $\mathbb{I}$ are injective, it is possible to define an *apartness* tensor $\#$ on $\mathcal{S}et^{\mathbb{I}}$ as follows:

$$(A\#B)_n \triangleq \{\langle a, b\rangle \in A_n \times B_n \mid supp(a) \cap supp(b) = \emptyset\}$$

where A, $B \in \mathcal{S}et^{\mathbb{I}}$ and $supp(a)$ stands for the support of a, whose precise definition can be found in [6]. For our purposes one can think of $supp(a)$ as the set of free names of a ($fn(a)$). It follows that, if $N \triangleq M_\eta$ (recall from the examples in §1 that η is the type of names of π-calculus) is the presheaf representing names in $\mathcal{S}et^{\mathbb{I}}$, we have that $(N\#B)_m$ stands for the set of pairs (n, b) where $n \in N_m = m$, $b \in B_m$ and $n \notin fn(b)$. Then, by means of the adjunction $\_ \bullet M \dashv \langle M, \_\rangle$, one can transfer this semantical notion to the "presentation" category $\mathcal{S}$ by defining

$$A\#_{\mathcal{S}}B \triangleq \langle M, (A \bullet M)\#(B \bullet M)\rangle$$

If we choose $A \triangleq Var_\eta$ and $B \triangleq S_\iota$, where the latter is the presheaf representing processes (remember from the examples in §1 that ι is the type of π-calculus processes), then we have

$$\begin{aligned}
Var_\eta\#_{\mathcal{S}}S_\iota \triangleq & \langle M, (Var_\eta \bullet M)\#(S_\iota \bullet M)\rangle \\
= & \langle M, N\#(S_\iota \bullet M)\rangle
\end{aligned}$$

For Γ an object of $\mathcal{U}$, a term $t \in (Var_\eta\#_{\mathcal{S}}S_\iota)(\Gamma)$ is a natural transformation $t : M^*(\Gamma) \longrightarrow N\#(S_\iota \bullet M)$. Thus, $(Var_\eta\#_{\mathcal{S}}S_\iota)(\Gamma)$ corresponds to the datatype of pairs $(x, P) \in Var_\eta(\Gamma) \times S_\iota(\Gamma)$ such that the *variable* x can be interpreted only on *names* which do not occur free in the interpretation of P. In other words, it represents the *freshness* predicate "$\notin$", used e.g. in [8]: $x \notin P$ holds iff (the interpretation of) x is not a name occurring free in (the interpretation of) P.

4. A METALANGUAGE FOR TYPED HOAS

As we have seen in the previous sections, the category $\mathcal{S}$ is rich enough to provide suitable presentations for both the abstract syntax and (almost) any semantics of an object typed language with binders. Therefore, a formal system for reasoning on objects and morphisms of $\mathcal{S}$ would be useful because it would allow for reasoning on both the syntactic and semantic aspects of object systems in higher-order abstract syntax, at once. In this section we describe briefly such a metalanguage.

An important point is that the system should have a modular structure. This is required by the fact that the same syntax may be given many semantics: while the properties and rules for the syntactic part are the same, different semantics induce different properties and rules. Therefore, we design the system to be composed by a semantic-free core system, called $\mathcal{H}_\Sigma$ and presented in §4.1 and §4.2, dealing with only the purely syntactic aspects of the abstract syntax of the signature one focus on.

Since variables in $\mathcal{H}_\Sigma$ are free from any intended meaning, $\mathcal{H}_\Sigma$ is strictly weaker than more specific logics, such as e.g. the Theory of Contexts [8] or the Nominal Logic [19] which rely on the "variables as names" interpretation. However, when a specific interpretation is chosen, the core system $\mathcal{H}_\Sigma$ can be extended by reflecting into the metalanguage the given semantic aspects, as needed. We give an example of this process in §4.3, where we show that the rules one could introduce when variable are intended to denote names are similar to those of a typed Nominal Logic.

4.1 The system $\mathcal{H}_\Sigma$

Rather than giving a complete formal system, we describe the term and type constructors (with their rules) that one can safely add to ones favorite formal system, in a modular fashion. We require only that the formal system includes many sorted equational logic (and thus, linear calculi are ruled out).

Types and terms of the metalanguage are ranged over by A, B, C and by a, b, f, t, respectively. Contexts of the metalanguage will be denoted with Δ, possibly with indexes, and subject to the usual rules of formation [11, 18]. We assume that the formal system has the following judgments:

$A\ type\ [\Delta]$	A is a well-formed type in context Δ
$a : A\ [\Delta]$	a is a well-formed term of type A in context Δ
$\Phi \vdash \phi\ [\Delta]$	ϕ is true when all propositions in Φ are true, in context Δ

and that there is a distinct type Prop (i.e., $\mathsf{Prop}\ type\ [\Delta]$ is derivable) whose terms represent propositions: if $\phi : \mathsf{Prop}\ [\Delta]$ then ϕ is a well-formed proposition in context Δ. In this section, we are interested in an equational logic, so we assume that we can derive

$$\frac{a_1 : A\ [\Delta] \quad a_2 : A\ [\Delta]}{a_1 =_A a_2 : \mathsf{Prop}\ [\Delta]}$$

as well as the usual rules of congruence.

For improving readability of rules, we will drop the common part of the contexts and hypotheses from assumptions and conclusions, giving the rules a "natural deduction style" flavour. For instance, the usual elimination rule of sum types

$$\frac{e : A + B\ [\Delta] \quad c_1(x) : C\ [\Delta, x : A] \quad c_2(x) : C\ [\Delta, x : B]}{case(e, c_1, c_2) : C\ [\Delta]}$$

would be written as

$$\frac{e : A + B \quad c_1(x) : C\ [x : A] \quad c_2(x) : C\ [x : B]}{case(e, c_1, c_2) : C}$$

Strictly speaking, we do not require the usual type constructors $\mathbf{1}, \times, +, \Rightarrow$ to be present; if needed, these may be provided by the underlying formal system (and interpreted as usual; see e.g., [11,18]). We focus on the rules specific of $\mathcal{H}_\Sigma$, which are presented in Figure 2.

For each $\tau \in T$, there are distinguished types Var_τ and S_τ, and a type constructor $A \mapsto \delta_\tau A$. Following the "weak HOAS" paradigm [8], Var_τ has no constructor, so the only terms (in normal form) inhabiting Var_τ are variables of the metalanguage themselves.

The adjunction (7) means that $\delta_\tau A$ is essentially a "variable abstraction" type, much like $Var_\tau \Rightarrow A$. Thus there are two term constructors corresponding to *variable abstraction* and *variable instantiation* (rules 17 and 18, respec-

Formation rules for types

$$\frac{}{Var_\tau\ type}\tau \in U \tag{14}$$

$$\frac{}{S_\tau\ type}\tau \in T \tag{15}$$

$$\frac{A\ type}{\delta_\tau A\ type}\tau \in U \tag{16}$$

Formation rules for terms

$$\frac{a : A\ [x : Var_\tau]}{(x{:}Var_\tau)a : \delta_\tau A} \tag{17}$$

$$\frac{t : \delta_\tau A \quad x : Var_\tau}{t@x : A} \tag{18}$$

$$\frac{x : Var_\tau}{var_\tau(x) : S_\tau}\tau \in U \tag{19}$$

$$\frac{t_1 : \delta_{\vec\sigma_1} S_{\tau_1} \quad \dots \quad t_n : \delta_{\vec\sigma_n} S_{\tau_n}}{op(t_1, \dots, t_n) : S_\tau} \quad op \in \mathcal{O}((\vec\sigma_1 \to \tau_1) \times \cdots \times (\vec\sigma_n \to \tau_n) \to \tau) \tag{20}$$

$$\frac{v : \delta_\sigma Var_\tau \quad a_1 : A \quad a_2 : \delta_\tau A}{case(v, a_1, a_2) : A} \tag{21}$$

$$\frac{F \in ES(\vec\rho, J, A) \quad \tau \in J \quad t : \delta_{\vec\rho} S_\tau}{rec(F, t) : A_\tau} \tag{22}$$

where $ES(\vec\rho, J, \mathcal{A})$ is defined in Definition 2

Conversion rules

$$\frac{a : A\ [x : Var_\tau] \quad v : Var_\tau}{((x : Var_\tau)a)@v =_A a\{v/x\}} \tag{23}$$

$$\frac{a_1 : A \quad a_2 : \delta_\tau A}{case((x{:}Var_\tau)x, a_1, a_2) =_A a_1} \tag{24}$$

$$\frac{x : Var_\tau \quad a_1 : A \quad a_2 : \delta_\tau A}{case((y{:}Var_\sigma)x, a_1, a_2) =_A a_2@x} \tag{25}$$

$$\frac{F \in ES(\vec\rho, J, A) \quad F_{op} \in F \quad t_1 : \delta_{\vec\rho} B_1 \quad \dots \quad t_n : \delta_{\vec\rho} B_n}{rec(F, (\vec x{:}Var_{\vec\rho})op(t_1@\vec x, \dots, t_n@\vec x)) =_{A_\tau} F_{op}(t'_1, \dots, t'_n)}$$

where $t'_i \triangleq \begin{cases} (\vec y{:}Var_{\vec\sigma_i})rec(F, (\vec x{:}Var_{\vec\rho})(t_i@\vec x)@\vec y) & \text{if } B_i \equiv \delta_{\vec\sigma_i} S_{\tau_i} \text{ for some } \tau_i \in J \\ t_i & \text{otherwise} \end{cases}$

and $\vec x{:}Var_{\vec\rho} \triangleq x_1{:}Var_{\rho_1}, \dots, x_k{:}Var_{\rho_k}$, and

$\vec y{:}Var_{\vec\sigma_i} \triangleq y_1{:}Var_{\sigma_{i,1}}, \dots, y_{m_i}{:}Var_{\sigma_{i,m_i}}$ where $m_i = |\vec\sigma_i|$ (26)

Figure 2: The system $\mathcal{H}_\Sigma$ for the signature $\Sigma = (T, U, \mathcal{O})$

tively), which behave as expected (rule 23). If the underlying logic contains typed λ-calculus (like, e.g., the Calculus of Constructions [10]), these term constructors can be conveniently simulated by abstraction and application, respectively. However, $\delta_\tau\, Var_\tau$ has the peculiar property to be a coproduct (which is not true for general abstraction types), and this is reflected by rules 21, 24 and 25.

The encoding of the object language is obtained by adding a specific constructor of types S_τ for each constructor in the signature (rule 20). For types with variables, a suitable "coercion" is added (rule 19).

Finally, an important feature of the system $\mathcal{H}_\Sigma$ is the possibility of defining terms by *mutual, higher-order recursion.* In fact, for Theorem 1 each type S_τ is inductive; but moreover, for Theorem 2, also each $\delta_{\rho_1} \dots \delta_{\rho_k} S_\tau$ is inductive, for $\vec{\rho} = \rho_1 \dots \rho_k$ types of the signature. Terms of type $\delta_{\vec{\rho}} S_\tau$ represent syntactic terms of type τ possibly with holes of type $\rho_1 \dots \rho_k$. Therefore, the system $\mathcal{H}_\Sigma$ allows for *elimination schemata* also for these datatypes of terms with holes:

DEFINITION 2. *Let $J \subseteq T$ and $\vec{\rho} = \rho_1, \dots, \rho_k$ ($k \geq 0$) be a set and a list of signature types, respectively. Let $A = \{A_\tau \mid \tau \in J\}$ be a J-indexed family of well formed types (that is, "A_τ type" is derivable for all $\tau \in J$).*

An elimination schema for $\vec{\rho}, J$ over A *is a set of terms*

$$F = \{F_{op} \mid op \in \mathcal{O}((\vec{\sigma}_1 \to \tau_1) \times \dots \times (\vec{\sigma}_n \to \tau_n) \to \tau), \tau \in J\} \cup \\ \cup \{F_{var_\tau} \mid \tau \in J \cap U\}$$

such that for all $\tau \in J$:

- *for all $op \in \mathcal{O}((\vec{\sigma}_1 \to \tau_1) \times \dots \times (\vec{\sigma}_n \to \tau_n) \to \tau)$:*

$$F_{op}(x_1, \dots, x_n) : A_\tau \ [x_1 : B_1, \dots, x_n : B_n]$$

where $B_i \triangleq \begin{cases} \delta_{\vec{\sigma}_i} A_{\tau_i} & \text{if } \tau_i \in J \\ \delta_{\vec{\rho}\vec{\sigma}_i} S_{\tau_i} & \text{otherwise;} \end{cases}$

- *and if $\tau \in U$, then* $\quad F_{var_\tau}(x) : A_\tau \ [x : \delta_{\vec{\rho}} Var_\tau]$.

We denote by $ES(\vec{\rho}, J, \mathcal{A})$ the class of all elimination schemata for $\vec{\rho}, J$ over A.

Given an elimination schema F, the rule (22) defines a term in A_τ by structural recursion on a given term (possibly with holes) $t \in \delta_{\vec{\rho}} S_\tau$. In the conversion rule (26), notice that recursion can cross binders, by creating local variables for "filling" the holes. If $k = 0$, and thus $\vec{\rho} = \emptyset$, we get the case of elimination schemata over plain terms (with binders, though). In this simple case, the rules (22) and (26) become respectively as follows:

$$\frac{F \in ES(\emptyset, J, A) \quad \tau \in J \quad t : S_\tau}{rec(F, t) : A_\tau}$$

$$\frac{F \in ES(\emptyset, J, A) \quad F_{op} \in F \quad t_1 : B_1 \quad \dots \quad t_n : B_n}{rec(F, op(t_1, \dots, t_n)) =_{A_\tau} F_{op}(t'_1, \dots, t'_n)}$$

$$\text{where } t'_i \triangleq \begin{cases} (\vec{y}\colon Var_{\vec{\sigma}_i}) rec(F, t_i @ \vec{y}) \\ \qquad\qquad \text{if } B_i \equiv \delta_{\vec{\sigma}_i} S_{\tau_i} \text{ for some } \tau_i \in J \\ t_i \qquad\qquad \text{otherwise} \end{cases}$$

Example 2. As an example program in $\mathcal{H}_\Sigma$, we provide the definition by mutual higher-order recursion of the infinite family of typed capture-avoiding substitution functions $\mathsf{subst}^\tau_u : \delta_\rho \Lambda_\tau \to \Lambda_\tau$, parametric in a given term $u : \Lambda_\rho$. For $t : \delta_\rho \Lambda_\tau$, the intended meaning of $\mathsf{subst}^\tau_u(t)$ is $t(z)\{u/z\}$, where z denotes the "hole" in t to be filled with u.

Let $J = T$ and

$$F \triangleq \{F_{var_\tau} : \delta_\rho\, Var_\tau \to \Lambda_\tau \mid \tau \in T\} \cup \\ \{F_{app_{\tau_1,\tau_2}} : \Lambda_{\tau_1 \supset \tau_2} \to \Lambda_{\tau_1} \to \Lambda_{\tau_2} \mid \tau_1, \tau_2 \in T\} \cup \\ \{F_{lam_{\tau_1,\tau_2}} : \delta_{\tau_1} \Lambda_{\tau_2} \to \Lambda_{\tau_1 \supset \tau_2} \mid \tau_1, \tau_2 \in T\}$$

where

$$F_{var_\tau}(v) \triangleq case(v, u, (x\colon Var_\tau) var_\tau(x))$$
$$F_{app_{\tau_1,\tau_2}}(t_1, t_2) \triangleq app_{\tau_1,\tau_2}(t_1, t_2)$$
$$F_{lam_{\tau_1,\tau_2}}(t) \triangleq lam_{\tau_1,\tau_2}(t)$$

It is easy to see that

$$rec(F, t) : \Lambda_\tau \ [u : \Lambda_\rho, t : \delta_\rho \Lambda_\tau],$$

and thus we can define the required substitution functions $\mathsf{subst}^\tau_u(t) \triangleq rec(F, t)$.

4.2 Properties of $\mathcal{H}_\Sigma$

$\mathcal{H}_\Sigma$ satisfies all the usual properties of simply typed λ-calculi, if the underlying calculus does. The kind of "normal" form of terms depends on the kind of reduction of the underlying calculus; usually, this is some variant of the β-head normal form, where all redexes are applied. Let us denote by $N(\Delta, A)$ the set of terms in head normal form of type A in a context Δ.

THEOREM 4. *Let Δ be a context, a, b be terms and A, B be types.*

- *If $a : A \ [\Delta]$ and $a =_A b \ [\Delta]$, then $b : A \ [\Delta]$*
- *If $a : A \ [\Delta]$, then there exists $b \in N(\Delta, A)$ such that $a =_A b \ [\Delta]$*

Normal terms of type S_τ in the metalanguage correspond faithfully to the terms of type τ of the language generated by the signature.

PROPOSITION 4. *Let $\Gamma = x_1 : \tau_1, \dots, x_n : \tau_n$ be a context of the object language, and $\tau \in T$. There is a bijection between terms of the object language such that $\Gamma \vdash t : \tau$ and terms a in normal form of the metalanguage such that $a : S_\tau \ [x_1 : Var_{\tau_1}, \dots, x_n : Var_{\tau_n}]$.*

In the system $\mathcal{H}_\Sigma$, we can prove that for all τ, $\delta_\tau\, Var_\tau$ is a coproduct, and for $\sigma \neq \tau$, $\delta_\tau\, Var_\sigma$ is equivalent to Var_σ. More formally, let us define the following syntactic shorthands:

$$fresh_\tau \triangleq (x : Var_\tau) x$$
$$old_\tau(a) \triangleq (x : Var_\tau) a \qquad x \notin fv(a)$$

PROPOSITION 5. *For all $\tau \neq \sigma \in T$, the following rules are derivable:*

$$\frac{a : Var_\sigma}{old_\tau(a) : \delta_\tau\, Var_\sigma} \qquad \frac{Var_\tau \ type}{fresh_\tau : \delta_\tau\, Var_\tau}$$

$$\frac{a_1 : A \quad a_2 : \delta_\tau A}{case(fresh_\tau, a_1, a_2) =_A a_1}$$

$$\frac{v : Var_\sigma \quad a_1 : A \quad a_2 : \delta_\tau A}{case(old_\tau(v), a_1, a_2) =_A a_2 @ v}$$

$$\frac{v : Var_\tau \quad a : \delta_\tau A}{unfold(old_\sigma(v), a) =_A a@v}$$

$$\frac{f : A\ [x_1 : A_1, \ldots, x_n : A_n]}{\delta_\tau f : \delta_\tau A\ [x_1 : \delta_\tau A_1, \ldots, x_n : \delta_\tau A_n]}$$

where $unfold(old_\sigma(v), a) \triangleq case(old_\sigma(v), a@v, a)$ *and* $\delta_\tau f \triangleq (x : Var_\tau) f\{x_1@x/x_1, \ldots, x_n@x/x_n\}$.

Moreover, let us denote

$$contr_\tau(a) \triangleq (x : Var_\tau)(a@x)@x$$

$$swap_{\tau,\sigma}(a) \triangleq (x : Var_\sigma)(y : Var_\tau)(a@y)@x$$

PROPOSITION 6. *The following rules are derivable.*

$$\frac{a : \delta_\tau \delta_\tau A}{contr(a) : \delta_\tau A} \qquad \frac{a : \delta_{\sigma\tau} A}{swap(a) : \delta_{\tau\sigma} A}$$

$$\frac{a : A\ [x : Var_\tau, y : Var_\tau]}{contr((x : Var_\tau)(y : Var_\tau)a) =_{\delta_\tau A} (x : Var_\tau)a\{x/y\}}$$

$$\frac{a : A\ [x : Var_\sigma, y : Var_\tau]}{contr((x : Var_\sigma)(y : Var_\tau)a) =_{\delta_{\tau\sigma} A} (y : Var_\tau)(x : Var_\sigma)a}$$

Also, we can prove the monadic properties of δ_τ:

PROPOSITION 7. *The following rules are derivable:*

$$\frac{e : \delta_\tau A \quad f : \delta_\tau B\ [x : A]}{let(x{\Leftarrow}e, f) : \delta_\tau B}$$

$$\frac{e : A \quad f : \delta_\tau B\ [x : A]}{let(x{\Leftarrow}old_\tau(e), f) =_{\delta_\tau B} f\{e/x\}}$$

$$\frac{e : \delta_\tau A}{let(x{\Leftarrow}e, old_\tau(x)) =_{\delta_\tau B} e}$$

$$\frac{e : \delta_\tau A \quad f : \delta_\tau B\ [x : A] \quad g : \delta_\tau C\ [y : B]}{let(y{\Leftarrow}let(x{\Leftarrow}e, f), g) =_{\delta_\tau C} let(x{\Leftarrow}e, let(y{\Leftarrow}f, g))}$$

where $let(x{\Leftarrow}e, f) \triangleq contr_\tau((\delta_\tau f)\{e/x\})$.

Soundness. $\mathcal{H}_\Sigma$ is an internal language of $\mathcal{S}$: each type of $\mathcal{H}_\Sigma$ is interpreted as an object of $\mathcal{S}$; in particular, the type Prop is interpreted as the usual subobject classifier Ω of $\mathcal{S}$ (which is a topos). The semantics of $a : A\ [x_1 : A_1, \ldots, x_n : A_n]$ is a natural transformation $[\![a]\!] : A_1 \times \cdots \times A_n \to A$, and $\Phi \vdash a_1 =_A a_2\ [\Delta]$ means that $[\![a_1]\!], [\![a_2]\!] : [\![\Delta]\!] \to A$ are the same morphism, if all propositions in Φ are satisfied.

Therefore, in order to prove the soundness of the system, we have to check that each rule can be interpreted in $\mathcal{S}$. This is easily verified by inspection, once we have fixed the interpretation of types and term constructors. All these constructors are interpreted as expected; e.g.,

$$[\![\delta_\tau A]\!] = \delta_\tau [\![A]\!]$$

$$[\![\Gamma \vdash (x : Var_\tau)t : \delta_\tau A]\!] = \phi([\![\Gamma, x : Var_\tau \vdash t : A]\!])$$

where ϕ is the isomorphism of adjunction (7). In particular, recursion operators are interpreted as the unique maps given by initiality of S in Σ-Alg. Thus we have:

PROPOSITION 8. *If* $\vdash t_1 =_A t_2\ [\Delta]$, *then* $[\![t_1 : A\ [\Delta]]\!] = [\![t_2 : A\ [\Delta]]\!]$ *in* $\mathcal{S}$.

Finally, notice that $\mathcal{H}_\Sigma$ admits all properties and principles which hold in the standard interpretation of an intuitionistic higher-order logic in the topos $\mathcal{S}$. In particular, the principle of Unique Choice is not ruled out *a priori:* it is consistent with $\mathcal{H}_\Sigma$. Things can change, however, if we extend $\mathcal{H}_\Sigma$ with rules derived from semantic interpretation, as we see next.

4.3 Extending $\mathcal{H}_\Sigma$ with semantic properties

We have seen in §3 that there is an adjunction connecting any given category $\mathcal{C}$ cocomplete to $\mathcal{S}$. Along this adjunction, we can build new types in $\mathcal{S}$, from those in $\mathcal{C}$. This allows us for introducing new type constructors and propositions in the metalanguage, in a modular way. For instance, in the case of $\mathcal{C} = \mathcal{S}et^{\mathbb{I}}$ (§3.4), we can add new rules for the tensor type "#" and the "freshness" proposition, such as the following:

$$\frac{A\ type}{Var_\eta \# A\ type}$$

$$\frac{x : Var_\eta \quad a : A}{x \notin a : \mathsf{Prop}}$$

$$\frac{t : Var_\eta \# A}{\Phi \vdash \pi_1(t) \notin \pi_2(t)}$$

and the following structural rules for $\notin$

$$\frac{\{\Phi, \nabla_i \vdash y \notin t_i@\vec{x}_i\ [\Gamma, \vec{x}_i : \vec{\sigma}_i]\}_{i \in 1\ldots n}}{\Phi \vdash x \notin op(t_1, \ldots, t_n)\ [\Gamma]}$$

$$op : (\vec{\sigma}_1 \to \tau_1) \times \cdots \times (\vec{\sigma}_n \to \tau_n) \to \tau$$

where $\nabla_i \triangleq \{x_{ij} \notin y \mid j \leq |\vec{x}_i|, y \in dom(\Gamma)\} \cup \{x_{ij} \notin x_{ik} \mid j \neq k \leq |\vec{x}_i|\}$. For instance, for the λ-calculus we have the rule (among others):

$$\frac{\Phi, \{x \notin z \mid z \in dom(\Gamma)\} \vdash y \notin t@x\ [\Gamma, x : Var_\tau]}{\Phi \vdash y \notin lam(t)\ [\Gamma]}$$

which is similar to the rules used in the Theory of Contexts [8]. The $\notin$ predicate can be used also to introduce the (multi-sorted) $\mathbin{\#}$ quantifier of Nominal Logic [19], together with the corresponding formation, introduction and elimination rules:

$$\frac{\phi : \delta_\tau \mathsf{Prop}}{\mathbin{\#}_\tau \phi : \mathsf{Prop}}$$

$$\frac{\Phi, \nabla_x \vdash \phi@x\ [\Gamma, x : Var_\tau]}{\Phi \vdash \mathbin{\#}_\tau \phi\ [\Gamma]}$$

$$\frac{\psi : \mathsf{Prop}\ [\Gamma] \quad \Phi \vdash \mathbin{\#}_\tau \phi\ [\Gamma] \quad \Phi, \nabla_x, \phi@x \vdash \psi\ [\Gamma, x : Var_\tau]}{\Phi \vdash \psi\ [\Gamma]}$$

where, again, $\nabla_x \triangleq \{x \notin y \mid y \in dom(\Gamma)\}$.

The interpretation (and the soundness) of these "semantic"-based rules is given by using the properties of types reflected from $\mathcal{C}$ into $\mathcal{S}$. In general, this may require to define peculiar notions of truth and validity, by interpreting Prop on a non-standard subobject classifier, as in [1,7]. In such a case, not all properties valid in the standard interpretation may be valid anymore. In particular, the principles of Choice and Unique Choice may become invalid, as it is known from [1,7]. Also for this reason we have preferred to keep the core system $\mathcal{H}_\Sigma$ as simple as possible, open also to these possibilities.

5. CONCLUSIONS

In this paper we have presented a framework for representing and reasoning on the syntax and the semantics of languages with types and binders. Following the approach of [2, 4], we have developed an algebraic theory of typed abstract syntax in a suitable "presentation category" $\mathrm{Mod}_T(\mathcal{S})$, which is ultimately a presheaf category. We have pointed out that, by means of a general construction, the

same category can be used to interpret not only the syntax but also any given semantics of the type theory, under a mild condition. For instance, we have shown in §3.4, how the semantic notion of "name" is related to its syntactic counterpart of "variable ranging over names".

The internal structure of $\mathrm{Mod}_T(\mathcal{S})$ becomes therefore very interesting and useful. Hence, we have presented a metalogical system $\mathcal{H}_\Sigma$, a particular internal logic of $\mathrm{Mod}_T(\mathcal{S})$, which can be used to reason on both the syntactic and the semantic aspects of the languages. The system $\mathcal{H}_\Sigma$ is composed by a core equational logic for reasoning on general aspects of abstract syntax; when a specific semantic interpretation is chosen, the system can be further extended with type and term constructors deriving from the semantics, thus streamlining the approach followed in [1, 7].

This general framework can be useful from a foundational point of view since it allows to illuminate differencies and relationships between the syntactic and the semantic layers of programming languages.

Related work. The algebraic treatment of typed binding signatures of §2 is a direct consequence of the work in [2, 4] on untyped binding signatures and in [3] for the specific case of simply typed λ-calculus. Metalogical systems for reasoning on datatypes with variables and binders, following many different approaches, have been presented in [8, 15, 19, 20], among others. In our opinion, the development we carried out in this paper tries to close the gap between [2, 4] (which are oriented towards an algebraic point of view) and [1, 7] (which, on the other hand, are inspired by a logical perspective). Indeed, as we have seen in §3.4 we can also reflect at the presentation level a notion of predicate (the freshness predicate $\notin$ used at the metalanguage level in [1, 7]), still retaining all the properties derived from the initial algebra semantics in [2, 4].

Future work. The development presented in this paper can be generalized further, in order to cope with more complex languages and semantics. For instance, in order to represent faithfully e.g. the typed π-calculus or object-based calculi, we need to take into account the notion of *subtyping*. This can be easily accomodated by taking T as a partial order or a preorder $(T, \leq)$, viewed as a category. Other important typing notions are *polymorphism*, *equational theories* of types, and *dependent types*. For instance, using polymorphic types *a la* ML, together with $Cpo^{\mathbb{I}}$ as semantic category, we can give a formal setting for languages like FreshML [5].

A still open problem is the formal modularization of the "reflections" of the semantic notions in the presentation category, and in the metalogical system. This task should require first a modularization of the semantics itself, such as in Moggi's approach based on computational monads [16]. Thus, one could reflect the "monad logic" onto $\mathcal{S}$.

On a more practical side, the notions of Σ-initial algebra and the initial algebra semantics should be useful for the implementation of Logical Frameworks featuring recursion, possibly higher-order, over typed languages with binders, along the lines of the metalogical system $\mathcal{H}_\Sigma$.

6. REFERENCES

[1] A. Bucalo, M. Hofmann, F. Honsell, M. Miculan, and I. Scagnetto. Consistency of the theory of contexts. Submitted, 2001.

[2] M. Fiore and D. Turi. Semantics of name and value passing. In H. Mairson, editor, *Proc. 16th LICS*, pages 93–104, Boston, USA, 2001. IEEE Computer Society Press.

[3] M. P. Fiore. Semantic analysis of normalisation by evaluation for typed lambda calculus. In *PPDP'02 - Principles and Practice of Declarative Programming*. ACM Press, 2002.

[4] M. P. Fiore, G. D. Plotkin, and D. Turi. Abstract syntax and variable binding. In Longo [12], pages 193–202.

[5] `FreshML`: A fresh approach to name binding in metaprogramming languages. `http://www.cl.cam.ac.uk/~amp12/research/freshml/`, 2002. Research project.

[6] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax involving binders. In Longo [12], pages 214–224.

[7] M. Hofmann. Semantical analysis of higher-order abstract syntax. In Longo [12], pages 204–213.

[8] F. Honsell, M. Miculan, and I. Scagnetto. An axiomatic approach to metareasoning on systems in higher-order abstract syntax. In *Proc. ICALP'01*, volume 2076 of *Lecture Notes in Computer Science*, pages 963–978. Springer-Verlag, 2001.

[9] F. Honsell, M. Miculan, and I. Scagnetto. π-calculus in (co)inductive type theory. *Theoretical Computer Science*, 253(2):239–285, 2001.

[10] INRIA. *The Coq Proof Assistant*, 2002. `http://coq.inria.fr/doc/main.html` .

[11] B. Jacobs. *Categorical Logic and Type Theory*, volume 141 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 1999.

[12] G. Longo, editor. *Proceedings, Fourteenth Annual IEEE Symposium on Logic in Computer Science*, 1999. IEEE Computer Society Press.

[13] S. Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, Berlin, 1971.

[14] S. Mac Lane and I. Moerdijk. *Sheaves in Geometry and Logic: a First Introduction to Topos Theory*. Universitext. Springer-Verlag, 1994.

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

[16] E. Moggi. Notions of computation and monads. *Information and Computation*, 1, 1993.

[17] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*, volume 7 of *International Series of Monograph on Computer Science*. Oxford University Press, 1990.

[18] A. 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.

[19] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 2003. Special issue on TACS2001, to appear.

[20] C. Schürmann. Recursion for higher-order encodings. In *Proc. CSL 2001*, volume 2142 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.