The linearity of constructions and nested syntax

André Hirschowitz\textsuperscript{1} and Marco Maggesi\textsuperscript{2}

\textsuperscript{1} Laboratoire J.-A. Dieudonné, Université de Nice - Sophia Antipolis
ah@unice.fr
\textsuperscript{2} Dipartimento di Matematica U. Dini, Università degli studi di Firenze,
maggesi@math.unifi.it

Abstract. We use the notion of module over a monad to propose a new definition for abstract syntax and semantics (with binding constructions) encapsulating compositionality and various substitution lemmas.

1 Introduction

1.1 Monads

Our monads are monads over sets. The algebraic structure of monad reflects the behaviour of substitution (and so does the companion structure of operad). The operation of substitution involves so-called variables, and this is why the notion of monad is much more abstract than, for instance, the notion of monoid. However, the role of monads in the understanding of structures has been early recognized (see e.g. [ML98] [Bor94]) as well as their role in the understanding of computation (see e.g. [Mog89] [Mog91] [BHM00]). In these familiar roles, monads appear in general with their companion algebras, which are some kind of right-modules. In the present work, we propose a prominent role for the notion of left-module over a monad in the understanding of syntaxes (with binding constructions) and their semantics.

The monadic (or operadic) structure of syntax has already been explored, at least in the special case of λ-calculus, by several authors [AR99] [BP99a] [FPT99] [GJ06] [HM06]. In this paper we go further and we use the monadic framework (with the related notion of modules) to clarify the structure of the syntax category.

1.2 Syntax with binders

Basically, an (untyped, first-order, abstract) syntax is specified by a signature, which is a family of arities and yields an inductive set of formulas with corresponding induction and recursion principles. A more sophisticated approach is provided by the so-called algebraic point of view [GTWW77] [Sco71], where the abstract syntax is an initial object in a category of semantics algebras, and the initial morphism into each algebra is understood as its semantics. Our work is
concerned with the extension of this view to the case of languages which allow
binding constructions. We quote from the PoplMark Challenge [ABF+05]:

“Representing binders has been recognized as crucial by the theorem
proving community, and many different solutions to this problem have
been proposed. In our (still limited) experience, none emerge as clear
winners.”

For one such solution and a fair account of earlier ones, see e.g. [MM02]. For
other recent solutions, see [Hof99] [FPT99] [GP99]. Beside the above-mentioned
contributions there is a series of works devoted to nested data-types ([AR99]
[BP99a] [FPT99] [GJ06]) which, although not explicitly addressed to our prob-
lem, are extremely relevant to our purpose. Based on modules over monads, we
propose a completion of the solution implicitly proposed in [BP99a], hopefully
improving the solution explicitly introduced in [FPT99] (see section 8).

1.3 Nested syntax

We explain how nested syntax goes on the emblematic example of the $\lambda$-calculus.
We consider the syntactic $\lambda$-calculus as an assignment $V \mapsto \text{SLC}(V)$, where
$V$ is a (variable) set (of variables) while $\text{SLC}(V)$ is the set of $\lambda$-terms (modulo $\alpha$-
conversion) built on free variables taken in $V$. Similarly the semantic $\lambda$-calculus
is the assignment $V \mapsto \text{LC}(V)$, where now $\text{LC}(V)$ is the set of $\lambda$-terms (modulo $\alpha\beta\eta$-
conversion) built on free variables taken in $V$.

Our first (standard) observation is that both assignments are monads and
that the natural transformation
\[ \text{sem}: \text{SLC} \longrightarrow \text{LC} \]
is a morphism of monads. Next we turn to the $\text{app}$ construction, which is easily
understood as a natural transformation:
\[ \text{app}: \text{LC} \times \text{LC} \longrightarrow \text{LC} \]
and similarly
\[ \text{Sapp}: \text{SLC} \times \text{SLC} \longrightarrow \text{SLC}. \]

Our second observation is which structure is preserved by these transformations:
for instance we observe that $\text{LC}$ and $\text{LC} \times \text{LC}$ are left-modules over the monad
$\text{LC}$, and that $\text{app}$ is a morphism of left-modules. We say that $\text{app}$ is linear.

Our third observation extends the previous one to the $\text{abs}$ construction: we
see it as a collection of applications
\[ \text{abs}_V: \text{LC}(V^*) \longrightarrow \text{LC}(V) \]
where $V^*$ is obtained from $V$ by adding one element; then we recognize the
assignment $V \mapsto \text{LC}(V^*)$ as a $\text{LC}$-left-module which we denote $\text{LC}'$ (and call the
derivative of $\text{LC}$), so that $\text{abs}$ turns out to be also linear.
Then we just say that \texttt{app} is a representation in \texttt{LC} of arity \((0,0)\), because there are two arguments and no derivative, while \texttt{abs} is a representation in \texttt{LC} of arity \((1)\), because there is one derived argument. So that \((\texttt{app}, \texttt{abs})\) is a representation in \texttt{LC} of the signature \(\Sigma := ((0,0),(1))\). Finally we define in the natural way the category of representations of \(\Sigma\) and observe that \texttt{SLC} is initial there, while \texttt{sem} is the initial morphism in this category.

So we view the formulas of a syntax as depending on a set of variables, yielding a monad. Given an arbitrary ”signature”, we define the category of its representations, and prove it to possess an initial object, which we call the syntax generated by the given signature. As expected, this syntax is equipped with a recursion principle which seems perfectly suited for semantics reasoning.

1.4 Relation with other approaches to syntax and semantics

To compare our work with previous ”algebraic” definitions of syntax amounts to compare (via functors) the corresponding categories of semantics. Larger categories correspond to stronger recursion principles, while smaller categories select nicer semantics. The algebraic point of view restricts attention to semantics enjoying compositional.

Other standard properties usually found in the literature of abstract syntax concern compatibility with substitution (substitution lemmas). Our category seems to be the smallest one currently available thus encapsulating substitution lemmas as much as currently possible.

1.5 Experiments

Nested syntax has been experienced via the Proof Assistant Coq on several standard situations which we describe below in section 7. In all these cases, the relevant monad has been built using the whole power of Coq inductive data types definitions.

This ability of Coq is the counter-part of the higher-rank polymorphism allowed in recent versions of Haskell that permits to implement nested data-types as studied by Bird and Paterson [BP99b].

1.6 Organization of the paper

Our work is organized as follows. Standard material about functors and monads is reviewed in section 2. In section 3 we review modules over monads. In section 4, we give our treatment of syntaxes and their semantics. The relation between our syntax with the more general setting of nested data-types is treated in section 5. In section 6, we sketch how our treatment extends to simply-typed syntax. In section 7, we describe various experiments. In section 8, we compare our work to previous contributions on the same subject.
2 Families, functors and monads

We briefly recall here some standard material about families of sets, functors and monads. Experienced readers may want to skip this section or just use it as reference for our notations.

**Definition 1 (Families of sets).** We denote by \( \text{Fam} \) the category of families of sets. Objects of \( \text{Fam} \) are maps \( F : \text{Set} \to \text{Set} \) and a morphism between two objects \( \phi : F \to G \) is a collection of functions \( \{ \phi_X : F(X) \to G(X) \} \) indexed by \( X \) ranging over sets.

The category \( \text{End}(\text{Set}) \) of endofunctor of \( \text{Set} \) is a subcategory of \( \text{Fam} \). Both categories have products and coproducts that can be constructed “pointwise”, e.g., given \( F, G \) two families, the product family is \( F \times G : X \mapsto F(X) \times G(X) \).

We denote by \( F \cdot G \) the composition of families (or functors) and by \( F^2, F^3 \) the iterated compositions \( F \cdot F, F \cdot F \cdot F \).

A monad is a monoid in the category of endofunctor (c.f. e.g., [ML98]).

**Definition 2 (Monad).** A monad \( R = (R, \mu, \eta) \) is given by a functor \( R : \text{Set} \to \text{Set} \), and two natural transformations \( \mu : R^2 \to R \) and \( \eta : I \to R \) such that the following diagrams commute:

\[
\begin{array}{ccc}
R^3 & \xrightarrow{R \mu} & R^2 \\
\downarrow{\mu R} & & \downarrow{\mu} \\
R^2 & \xrightarrow{\mu} & R
\end{array}
\quad
\begin{array}{ccc}
I \cdot R & \xrightarrow{\eta R} & R^2 \\
\downarrow{1_R} & & \downarrow{\eta} \\
R & \xrightarrow{R \eta} & R \cdot I
\end{array}
\]

**Definition 3 (Maybe monad).** The functor \( X \mapsto X^* \) which takes a set and add one point has a natural structure of monad that we call, borrowing from the terminology of the library of the programming language Haskell, **Maybe monad**.

**Definition 4 (Hofunctors).** We call hofunctor (higher order functor) the endofunctors over \( \text{End}(\text{Set}) \), i.e., a functor of kind \( H : (\text{Set} \to \text{Set}) \to (\text{Set} \to \text{Set}) \).

As in the case of families and functors, (co)products of hofunctors can be constructed pointwise. Forgetting the action on maps, every hofunctor can also be considered an endofunctor over \( \text{Fam} \). One important example of hofunctor for the rest of this paper is given by derivation.

**Definition 5 (Derivation).** We define the derivation \( F' \) of a functor \( F : \text{Set} \to \text{Set} \) to be the functor \( F' = F \cdot \text{Maybe} \). We can iterate the construction and denote by \( F^{(n)} \) the \( n \)-th derivative. It is easily checked how the derivation \( R' \) of a monad \( R \) is again a monad.\(^3\)

---

\(^3\) Apart from section 6, the monads we consider here are monads over sets.

\(^4\) In the programming language Haskell, \( \mu \) and \( \eta \) correspond **join** and **return**.

\(^5\) This correspond to the **MaybeT** monad transformer in Haskell.
3 Modules over monads

Mathematicians of the last century have agreed upon the two companion notions of monad and operad to encode substitution. Being a monoid in a suitable monoidal category, a monad (or an operad) has associated right and left modules. Algebras over a monad, which are just a special kind of right modules, have been extensively considered. Other kind of modules are in the air.

**Definition 6 (Module over a monad).** Since we consider only left-modules here, we simply call them modules. We define a module over a monad \( R \) to be a functor \( M \) from sets to sets equipped with a natural transformation \( \rho : M \cdot R \to M \), called substitution, and satisfying the usual associativity condition identifying the two natural transformations from \( M \cdot R \cdot R \) to \( M \), more precisely, we require that the following diagrams commute

\[
\begin{array}{ccc}
M \cdot R^2 & \xrightarrow{M\mu} & M \cdot R \\
\downarrow{\rho R} & & \downarrow{\rho} \\
M \cdot R & \xrightarrow{\rho} & M
\end{array}
\quad
\begin{array}{ccc}
M \cdot R & \xrightarrow{M\eta} & M \cdot I \\
\downarrow{\rho} & & \downarrow{1_M} \\
M & & M
\end{array}
\]

**Example 1.** We can see our monad \( R \) as a module over itself, which we call the tautological module.

**Example 2.** For any functor \( F \) and \( R \)-module \( M \), the composition \( F \cdot M \) is a \( R \)-module. In particular, the initial and final functors \( \emptyset \) and \( * \) and any other constant functor are trivial \( R \)-module.

As for functors and monads, derivation is well-behaved also on modules:

**Lemma 1 (Derived module).** The derivation \( M' = M \cdot \text{Maybe} \) of a \( R \)-module \( M \) has a natural structure of \( R \)-module.

**Definition 7.** We say that a natural transformation of \( R \)-modules \( F : M \to N \) is linear if it is compatible with substitution. We take linear natural transformations as module morphisms.

**Definition 8 (Category of \( R \)-modules).** We check easily that module morphisms among \( R \)-modules yield a subcategory of the functor category.

**Example 3.** We easily check that the natural transformation of a module into its derivative is linear.

**Example 4.** Note that there are two natural inclusions of the derivative \( M' \) into the second derivative \( M'' \). Both are linear.

**Definition 9 (Product of modules).** We check easily that the cartesian product of two \( R \)-modules as functors is naturally a \( R \)-module again and is the cartesian product also in the category of \( R \)-modules. We also have finite products as usual. The final module \( * \) is the product of the empty family.
Example 5. Given a $R$-module $M$, we have a natural “evaluation” morphism $\text{eval}: M' \times R \rightarrow M$.

Proposition 1. Derivation yields a cartesian functor from $R$-modules to $R$-modules.

Definition 10 (Pull-back). Given a morphism $f: A \rightarrow B$ of monads and a $B$-module $M$, we define its pull-back $f^*M$ as follows: we set $f^*M(X) := M(X)$ and we define the action $\rho_{f^*M}: f^*M \cdot A \rightarrow f^*M$ as the composition of $Mf: M \cdot A \rightarrow M \cdot B$ with the action of $B$ on $M$:

$$\rho_{f^*M} := \rho_M \cdot Mf.$$ 

Lemma 2. The pull-back of a module is a module.

Proof. Our thesis is the commutativity of the diagram

which follows from the commutativity of the three pieces since $M$ is a $B$-module, the map from $M(B(\_)) \rightarrow M(\_)$ is functorial, and $f$ is a morphism.

Definition 11 (Pull-back (functoriality)). We upgrade the pull-back into a functor $f^*: \text{Mod}_B \rightarrow \text{Mod}_A$ by checking that if $g: M \rightarrow N$ is a morphism of $B$-modules, then so is the natural transformation $f^*g: f^*M \rightarrow f^*N$.

Proposition 2. Pull-back commutes with products and with derivation.

Proposition 3. Any morphism of monads $f: A \rightarrow B$ yields a morphism of $A$-modules, still denoted $f$, from $A$ to $f^*B$.

4 Syntax

Definition 12 (Arity). An arity is a list of integers [FPT99]. Informally, we attach to any syntactic construction an arity: the length of the list represents the number of arguments, while the integers say the number of variables in the corresponding argument which are bound by the construction. We denote by $\mathbb{N}^*$ the set of arities.

Example 6. The arity of the $\text{app}$ operation of the $\lambda$-calculus is $(0,0)$, while the arity of the $\text{abs}$ construction is $(1)$. 
Definition 13 (Signatures). We define a (binding) signature $\Sigma = (I, a)$ to be a family of arities $a: I \to \mathbb{N}^*$. Associated to any signature $\Sigma = (I, a)$ we consider the hofunctor

$(-)^\Sigma: (\text{Set} \to \text{Set}) \to (\text{Set} \to \text{Set})$

given by

$N^\Sigma := \sum_{i \in I} \prod_{k \in a_i} N^{(k)}$

We observe that if $R$ is a monad and $M$ is a $R$-module, then $M^\Sigma$ has a natural structure of $R$-module (c.f. examples in section 3).

Definition 14 (Representation of a signature). Given a monad $R$, we define a representation of the signature $\Sigma$ in $R$ to be a $R$-module morphism

$R^\Sigma \to R$.

In other words, representations of an arity $(a_1, \ldots, a_r)$ are module morphisms $\Pi R^{(a_i)} \to R$ and representations of a signature $\Sigma = (I, a)$ consist of, for each $i \in I$, a representation of $a(i)$ in $R$.

Example 7. As mentioned in the introduction, a representation of $(0, 0)$ in LC is given by the app construction, while a representation of $(1)$ in LC is given by the abs construction. Taken together, app and abs constitute a representation of the signature $\Sigma = ((0, 0), (1))$ in the monad LC.

Definition 15 (The category of representations). Given a signature $\Sigma = (I, a)$ we build the category $\text{Mon}^\Sigma$ of representations of $\Sigma$ as follows. Its objects are monads equipped with a representation of $\Sigma$. A morphism from $(R, r)$ to $(S, s)$ is a morphism $f$ from $R$ to $S$ compatible with the representations in the sense that the following diagram of $R$-modules commutes:

$R^\Sigma \to R$

$\downarrow$ \hspace{1cm} $\downarrow f$

$f^*(S^\Sigma) \to f^*S$

where the horizontal arrows come from the representations and the vertical arrows come from $f$ (it is used here that $f^*$ commutes with derivation and products).

Proposition 4. These morphisms, together with the obvious composition, turn $\text{Mon}^\Sigma$ into a category.

We are now in position to present our characterisation of (binding) syntaxes.

Theorem 1. For any signature $\Sigma$, the category $\text{Mon}^\Sigma$ has an initial object, which we call the syntactic monad generated by $\Sigma$, and denote by $(\hat{\Sigma}, \hat{\sigma})$. Given a representation $(S, s)$ of $\Sigma$ we denote the initial morphism by $rfold_s: \hat{\Sigma} \to S$.

The above theorem will be proved in section 5 together with other general recursion principles.
5 Nested syntax and nested data-types

In this section we relate our nested syntax with the broader discipline of nested data-types and use inductive data-types techniques to prove theorem 1 and other important recursion principles for the initial algebra \(\hat{\Sigma}\).

Following the work of Altenkirch and Reus, we implement nested data-type using families of sets. Other ideas used in this section are taken from the work of Bird and Paterson. We only sketch the necessary construction and refer to [AR99] [BP99a] for more details.

The first step of this construction is the definition of \(\Sigma\)-algebra, a cousin notion of \(\Sigma\)-representation of section 4.

**Definition 16 (\(\Sigma\)-algebra).** Let \(\Sigma\) be a signature. For every family \(F\) we pose
\[
F^{1+\Sigma} := I + F^{\Sigma}
\]
where \(I\) is the identity family. The construction \((\cdot)^{1+\Sigma}\) provides an endofunctor of families and a hofunctor. We call \(\Sigma\)-algebra a couple \((A, a)\) given by a family \(A\) and a morphism \(a: A^{1+\Sigma} \rightarrow A\).

It is obvious that any \(\Sigma\)-representation \((S, s)\) induces a \(\Sigma\)-algebra given by \(\eta + s: S^{1+\Sigma} \rightarrow S\). Furthermore every morphism between representations yields a morphism between the corresponding algebras. Altogether, we have a functor from the category of \(\Sigma\)-representations to the category of \(\Sigma\)-algebras.

Now we are in the position to state the following general, non-dependent, recursion principle which is also the basis for the proof of theorem 1.

**Theorem 2 (Non-dependent recursion principle).** For every signature \(\Sigma\) there exists an initial object \((\hat{\Sigma}, \hat{\sigma})\) in the category of \(\Sigma\)-algebras which possesses a natural structure of monad and \(\Sigma\)-representation. Given a \(\Sigma\)-algebra \((S, s)\) we denote by \(\text{fold}_s : \hat{\Sigma} \rightarrow S\) the induced initial morphism.

**Proof.** The existence of the initial object is a well-known result related to the fact that \((\cdot)^{1+\Sigma}\) is a strictly positive operator. The fact that \(\hat{\Sigma}\) is a monad can be proved with the same procedure followed in [AR99] for the special case of \(\lambda\)-calculus. The linearity of the induced morphism \(\hat{\Sigma}^\Sigma \rightarrow \hat{\Sigma}\) follows easily by structural induction.

The next theorem establish that the initial algebra \(\hat{\Sigma}\) of the above theorem actually is the representation stated in theorem 1 and initial morphism \(\text{rfold}\) is obtained from \(\text{fold}\) with the restriction to subcategory of \(\Sigma\)-representations induces the.

**Theorem 3.** Given a \(\Sigma\)-representation \((S, s)\) the induced initial \(\Sigma\)-algebra morphism \(\text{fold}_s : \hat{\Sigma} \rightarrow S\) is a morphism of representations.
Proof. Let \( \phi = \text{fold}_s \), we want to show that the following diagram is commutative

\[
\begin{array}{c}
\hat{\Sigma} \cdot \hat{\Sigma} \xrightarrow{\mu_{\Sigma}} \hat{\Sigma} \\
\phi \downarrow \quad \downarrow \phi \\
S \cdot \hat{\Sigma} \xrightarrow{\rho} S
\end{array}
\]

where \( \rho \) is the \( \hat{\Sigma} \)-action induced by the pull-back through \( \phi \), i.e., \( \rho = \mu_{\hat{\Sigma}} \cdot \hat{\Sigma} \phi \).

The proof is by structural induction on terms in \( R(R(X)) \). First we need to prove that given a set \( X \), the diagram commute on terms in \( \hat{\Sigma}(\hat{\Sigma}(X)) \) coming from \( \hat{\Sigma}(X) \), that is

\[
\phi \cdot \mu_{\hat{\Sigma}} \cdot \eta_{\hat{\Sigma}} \hat{\Sigma} = \rho \cdot \phi \hat{\Sigma} \cdot \eta_{\hat{\Sigma}} \hat{\Sigma}.
\]

This can be proved by showing that both side of the previous equations reduces to \( \phi \):

\[
\begin{align*}
\phi \cdot \mu_{\hat{\Sigma}} \cdot \eta_{\hat{\Sigma}} \hat{\Sigma} &= \phi \\
\mu_{\Sigma} \cdot S \phi \cdot (\phi \cdot \eta_{\hat{\Sigma}}) \hat{\Sigma} &= \mu_{\Sigma} \cdot \eta_{\hat{\Sigma}} S \cdot \hat{\Sigma} \phi = \phi.
\end{align*}
\]

Now we have to settle the induction step. We consider the cubic diagram

where the bigger square (the front side of the cube) is commutative by induction hypothesis and the commutativity of the inner square over terms coming from \( \hat{\Sigma} \cdot \hat{\Sigma} \) is our thesis. The floor and the ceiling of the cube are commutative because our \( \hat{\sigma} \) and \( s \) are representations. The fact that \( \phi \) is a \( \Sigma \)-algebra morphism implies that the remaining two vertical sides of the cube are also commutative. Then our thesis follows.

**Theorem 4 (Dependent recursion principle).** Let \( \Sigma \) be a signature. If a \( \Sigma \)-algebra \( (S, s) \) is fibered over \( \hat{\Sigma} \), in the sense that there is an assigned morphism of \( \Sigma \)-algebras \( n : S \rightarrow \hat{\Sigma} \), the induced initial algebra morphism \( \text{fold}_s : \hat{\Sigma} \rightarrow S \) respects the fibration, i.e., we have the following commutative diagram of families

\[
\begin{array}{c}
\hat{\Sigma} \cdot \hat{\Sigma} \xrightarrow{\mu_{\Sigma}} \hat{\Sigma} \\
\phi \downarrow \quad \downarrow \phi \\
S \cdot \hat{\Sigma} \xrightarrow{\rho} S
\end{array}
\]

where \( \rho \) is the \( \hat{\Sigma} \)-action induced by the pull-back through \( \phi \), i.e., \( \rho = \mu_{\hat{\Sigma}} \cdot \hat{\Sigma} \phi \).
fibered over $\hat{\Sigma}$

\[
\begin{array}{c}
\hat{\Sigma}^{1+\Sigma} \xrightarrow{(\text{fold}_s)^{1+\Sigma}} S^{1+\Sigma} \\
\downarrow \delta \quad \quad \downarrow s \\
\hat{\Sigma} \quad \quad \quad \quad \quad \quad S
\end{array}
\]

Observe that the family $G$ in the previous theorem can be equivalently given as a a family of sets $P(X, t) := n^{-1}_X(t)$ indexed over $\Pi_{X \in \text{Set}} T(X)$ (in other terms $G(X) = \Pi_{t \in T(X)} P(X, t)$). In this setting the previous principle can be rephrase in a way that it is cumbersome to describe but perhaps more common in practice. E.g., the Coq version of this principle, corresponding to the type $T$ described in section 7 is:

\[
\begin{array}{l}
\text{T_rec : } \forall P : \forall X : \text{Set}, \ T X \rightarrow \text{Set}, \\
\quad (\forall (X : \text{Set}) (x : X), \\
\quad \quad P X (\text{var} X x)) \rightarrow \\
\quad (\forall (X : \text{Set}) (t : T X), \\
\quad \quad P X t \rightarrow \forall t0 : T X, P X t0 \rightarrow P X (\text{app} X t t0)) \rightarrow \\
\quad (\forall (X : \text{Set}) (t : T (\text{option} X)), \\
\quad \quad P (\text{option} X) t \rightarrow P X (\text{abs} X t)) \rightarrow \\
\quad \forall (X : \text{Set}) (t : T X), P X t.
\end{array}
\]

6 Typed syntax

In this section, we sketch how our notion of nested syntax can be accommodated to the typed case, by visiting the emblematic example of the simply-typed $\lambda$-calculus.

We denote by $T$ the set of simple types $T := * \mid T \rightarrow T$ and by $T\text{-Set}$ the category of sets over $T$ (i.e., maps from $T$ to sets). Following [Zs06], we consider the syntactic typed $\lambda$-calculus as an assignment $V \mapsto \text{STLC}(V)$, where $V = \Pi_{t \in T} (V_t)$ is a (variable) set (of typed variables) while $\text{STLC}(V)$ is the set of typed $\lambda$-terms (modulo $\alpha$-conversion) built on free variables taken in $V$. Similarly the semantic $\lambda$-calculus is the assignment $V \mapsto \text{TLC}(V)$, where now $\text{LC}(V)$ is the set of $\lambda$-terms (modulo $\alpha\beta\eta$-conversion) built on free typed variables taken in $V$. We observe that both assignments are monads over the category of $T$-sets and that the natural transformation $\text{Tsem} : \text{STLC} \rightarrow \text{TLC}$ is a morphism of monads.

Next, for each type $A$, $\text{TLC}_A$ is a functor over $T\text{-sets}$, which is equipped with substitution turning it into a module over $\text{TLC}$. And given two types $A, B$, we have

\[
\text{app}_{A,B} : \text{TLC}_A \rightarrow B \times \text{LC}_A \rightarrow \text{TLC}_A
\]
which is linear.

For the abs construction, we need a notion of partial derivative for a module:

For a module $M$ over $T$-sets, and a type $A \in T$, we set

$$\delta_A M(V) := M(V_A^*)$$

where $V_A^*$ is obtained from $V$ by adding one element with type $A$. It is easily checked how $\delta_A M$ is again a module. Now given two types $A, B$, it turns out that

$$\text{abs}_{A,B} : \delta_A \text{TLC}_B \rightarrow \text{TLC}_{A \rightarrow B}$$

is linear.

Next we say that $\text{app}_{A,B}$ is a representation in TLC of typed arity $(A \rightarrow B) \times A \rightarrow B$, while $\text{abs}_{A,B}$ is a representation in TLC of typed arity $\delta_A B \rightarrow (A \rightarrow B)$. Accordingly, we say that $(\text{app}, \text{abs})$ is a representation in TLC of the type signature $\sigma := ((A \rightarrow B) \times A \rightarrow B)_{A,B \in T} \Pi ((\delta_A B \rightarrow (A \rightarrow B))_{A,B \in T}$. Finally we define in the natural way the category of representations of $\sigma$ and observe that STLC is initial there, while $Tsem$ is the initial morphism to TLC in this category.

7 Nested syntax for computer formalizations

In the present section we briefly describe the use of nested syntax that have been experienced in some computer formalisations using the proof assistant Coq.

7.1 The $\lambda$-calculus in Coq

The first three applications build upon the formalization of the syntactic $\lambda$-calculus, i.e., the monad SLC that has been already described in the introduction. The paradigm that permits to easily build the monad SLC in the Calculus of Inductive Constructions is the use of nested data-types [BP99a], in the same way it has been done initially in Haskell [BP99b]. Concretely, the crucial definition is that of the terms, that we archive as follows:

```coq
Inductive option (X : Set) : Set := Some : X -> option X
| None : option X.

Inductive T (X : Set) : Set := var : X -> T X
| app : T X -> T X -> T X
| abs : T (option X) -> T X.
```

Based on the formalization of SLC we enumerate three contributions with classical and new results on $\lambda$-calculus.

7.2 The Church-Rosser theorem

The Church–Rosser theorem has been formalized by Claudine Noblet-Faure [NF06]. Her code is available at www-math.unice.fr/~nobletc.

---

A version of Coq that allows for recursively non uniform parameters in inductive types is needed to execute this code, i.e., Coq V8.1beta or higher.
7.3 The category of untyped \(\lambda\)-calculi

This contribution concerns a new characterisation of the pure untyped \(\lambda\)-calculus modulo \(\alpha\beta\eta\)-equivalence (we refer to it as *semantic \(\lambda\)-calculus*) as initial object in a suitable category of monads. Here we give a short exposition of the statement of the theorem and we refer to our computer formalization available from http://math.unice.fr/~maggesi/lc.tar.gz for the details.

7.4 The category of simply-typed \(\lambda\)-calculi

The analogous theorem for the simply typed \(\lambda\)-calculus, in the spirit of section 6, has been formalized by Julianna Zsidó [Zsi06]. Her code is available at http://math.unice.fr/~zsido/st/.

7.5 A contribution to the PoplMark challenge

Our final computer formalization developed using the viewpoint of nested syntax is about \(F_\subset\). A lot of attention to system \(F_\subset\) have been recently catalyzed by the PoplMark Challenge\(^7\). The PoplMark Challenge is a concrete set of benchmarks for measuring the current ability to formalize the metatheory of programming languages. Among the various issues of formalizing metatheory, the treatment of syntax with binding is a prominent one, as stressed by the authors of the challenge itself. As part 1a of this challenge prescribes, we have proven transitivity for the \(\text{sub}\) relation on \(F_\subset\)-types. Our code is available at http://web.math.unifi.it/users/maggesi/fsub/.

8 Related and future works

As mentioned above, our work is a variation upon [FPT99] involving some ideas strongly related to [BP99a]. Here we discuss this relationship in more detail.

8.1 Comparing categories

To compare "algebraic" definitions of syntax amounts to compare (via functors) the corresponding categories of semantics. We can relate the contributions of [AR99], [BP99a], [FPT99] and ours in this way. Indeed, they consider the category \(F_\Sigma\) (resp. \(H_\Sigma\), resp. \(O_\Sigma\)) of representations of a signature \(\Sigma\) in families [AR99], (resp. endofunctors [BP99a], resp. operads [FPT99]) while we consider the category \(M_\Sigma\) of representations of \(\Sigma\) in monads and the category \(L_\Sigma\) of linear representations in monads.

\(^7\) For an introduction to the challenge, the vision behind the project and the description of the problems we refer to the site of the project http://www.cis.upenn.edu/group/proj/plclub/mmm/ and the expository paper [ABF+05].
There are natural functors from monads to operads, endofunctors and families, yielding natural functors:

\[ LM : L_\Sigma \to M_\Sigma,\ M\Sigma \to H_\Sigma,\ H\Sigma \to F_\Sigma \]

and \[ MO : M_\Sigma \to O_\Sigma.\] While \[ MO \] is not too far from being an equivalence, the other three functors define subcategories with strictly less morphisms (\( LM \)), or strictly less objects and morphisms (in particular \( MH \)).

8.2 Relationship to [FPT99]

We tackle the same problem as [FPT99] and pay attention for the same two main features of syntax: renaming (functoriality) and substitution.

As mentioned earlier, two mathematical structures have been universally recognized for substitution: monads and operads. Both are naturally equipped with companion structures of modules and thus notions of linearity. The main difference between monads and operads is that for monads, the set of variables is arbitrary, while for operads, the set of variables is of the form \([1, ..., n]\).

The work [FPT99] is on the side of operads. They do not strictly use operads and build instead their own variant. Accordingly, they do not consider modules, while we incorporate in the picture more properties of the substitution by observing the linearity of syntactic constructions.

Our second contribution is to switch to the monad side, in other words we consider presheaves over sets instead of presheaves over some small category of finite subsets of \( N \). Beyond conceptual simplicity, we can mention two advantages:

- As explained in [BP99b], deBruijn shifts, which are there anyway, appear much more natural in \( \text{Set} \).
- In order to extend the nested approach to other contexts, like the simply-typed context sketched above, we can take monads in a suitable category, and we just have to find the suitable class of modules, and the suitable derivations while the operadic approach will also need ad hoc variants of operads for each new context.

8.3 Relationship to [BP99a]

The work [BP99a] is not concerned with syntax but with (recursive) data-types. Although they observe that a nested data-type yields a monad, they feel the argument of the monad as a variable type, not as a set of variables. Accordingly, their categorical semantics for nested data-types do not pay much attention to renaming and substitution.

Summarizing, we marry the works of [FPT99] and [BP99a], bringing linearity as a wedding-gift.

8.4 Relationship to other works

We now try to compare our approach to three other ones: the classical HOAS approach, currently represented by the Twelf project, the nominal approach
initiated in [GP99], and the silent approach of those who consider that de Bruijn implementation is perfectly suited. We feel that using HOAS as implemented in Twelf as well as using nominal syntax as described for instance in [UT05] requires a serious overhead, while anyone can start programming with de Bruijn indices. As already understood in [BP99a], nested syntax provides a refined typing for terms which rules out all the low-level problems related to free versus bound variables. This typing is so simple and natural that you can try it right now. Beyond this refined typing, what we hope to bring to silent programmers is a more unified view of all aspects of substitution in the context of syntax.

8.5 Future works

Future works include extension to typed contexts as alluded above, as well as to equational contexts: the case of semantic λ-calculus, treated above, of a syntax with two equations, definitely calls for a wide generalization. Another future work should be to revisit the applications treated with other methods, e.g. [FT01] “semantics of name and value passing”, and check how our approach behaves for this applications.

References


