Information and Computation 208 (2010)545–564

Contents lists available at ScienceDirect

Information and Computation

journal homepage: www.elsevier.com/locate/ic

Modules over monads and initial semantics

André Hirschowitz [a,*], Marco Maggesi [b]

[a] UMR 6621, CNRS, Université de Nice Sophia—Antipolis, Parc Valrose, 06108 Nice Cedex2, France
[b] Università degli Studi di Firenze, Viale Morgagni, 67/a - 50134 Firenze, Italy

ARTICLE INFO

Article history:
Received 23 November 2007
Revised 21 July 2009
Available online 22 January 2010

ABSTRACT

Inspired by the classical theory of modules over a monoid, we introduce the natural notion of module over a monad. The associated notion of morphism of left modules ("linear" natural transformations) captures an important property of compatibility with substitution, not only in the so-called homogeneous case but also in the heterogeneous case where "terms" and variables therein could be of different types. In this paper, we present basic constructions of modules and we show how modules allow a new point of view concerning higher-order syntax and semantics.

1. Introduction

Substitution is a major operation. Mathematicians of the last century have coined two strongly related notions which capture the formal properties of this operation. The first one is the notion of monad, while the second one is the notion of operad. We focus on the notion of monad. The relevance of monads to computer science has been stressed constantly (see, e.g. [11]). Moggi was the first to describe how monads could contribute to the modeling of computational effects [4,23,24]. This breakthrough opened the way for the programming language Haskell [17], which offers strong support for programming through monads.

A monad in the category C is a monoid in the category of endofunctors of C (see Section 2) and as such, has right and left modules. A priori these are endofunctors (in the same category) equipped with an action of the monad. In fact, we introduce a slightly more general notion of modules over a monad, based on the elementary observation that we can readily extend the notion of a right action of a monad in C to the case of a functor from any category B to C, and symmetrically the notion of a left action of a monad in C to the case of a functor from C to any category D. We are mostly interested in left modules. As usual, the interest of this new notion is that it generates a companion notion of morphism. We take as morphisms those natural transformations among (left) modules which are compatible with the structure, namely which commute with the action (we also call these morphisms *linear* natural transformations).

We propose here a first reference for basic properties and constructions concerning left modules, together with basic examples taken in the area of (possibly higher-order) syntax and semantics. Indeed, there is currently a search for a convincing discipline suited to the programming of theorem-provers in this area [2], and we hope to show, through examples in Haskell and in Coq, the adequacy of the language of left modules for the encoding of higher-order syntax and semantics.

In Section 2, we briefly review the theory of monads. In Section 3, we present left modules and, in Section 4, we introduce linearity. In Sections 5 and 6, we present the treatment of first-order typed syntax and of higher-order untyped syntax based on modules. In Section 7, we show, on the example of the λ-calculus, how semantics can be approached via modules. Finally

* Corresponding author.
E-mail address: ah@math.unice.fr (A. Hirschowitz).
URL: http://math.unice.fr/∼ah (A. Hirschowitz), http://www.math.unifi.it/∼maggesi (M. Maggesi).

0890-5401/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.ic.2009.07.003

in Section 8, we discuss related and future work. Appendix A describes the formal proof in the Coq proof assistant of one of our examples.

We are pleased to thank the referees for very helpful criticism and suggestions.

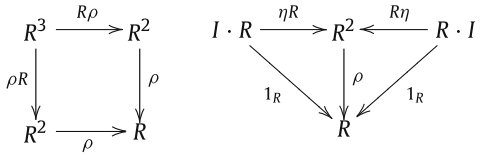
2. Monads

We briefly recall some standard material about monads. Experienced readers may want to skip this section or just use it as a reference for our notations. Mac Lane's book [19] can be used as a reference on this material. We fix a category C which admits sums and a final object $*$. The main examples of such a category are the category of sets, the category of endofunctors $\mathsf{Set} \to \mathsf{Set}$ and the relative categories Set/D considered in Section 5.

2.1. The category of monads

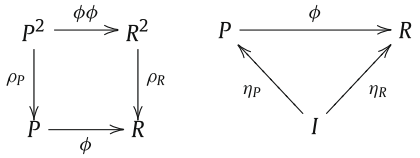
A monad over C is a monoid in the category $\mathsf{C} \to \mathsf{C}$ of endofunctors of C. In more concrete terms:

Definition 1 (*Monad*)**.** A monad $R = (R, \rho, \eta)$ over C is given by a functor $R\colon \mathsf{C} \to \mathsf{C}$, and two natural transformations $\rho\colon R^2 \to R$ and $\eta\colon I \to R$ such that the following diagrams commute:



The natural transformations ρ and η are often referred to as *product* (or *composition*) and *unit* of the monad R. In the programming language Haskell, their names are `join` and `return`, respectively.

Definition 2 (*Morphisms of monads*)**.** A morphism from the monad P to the monad R (both over C) is a natural transformation $\phi\colon P \to R$ which respects composition and unit, i.e., such that the following diagrams commute:



It can be easily checked that these morphisms yield a category, the category of monads over C.

2.2. Examples of monads

Now we focus on some examples and constructions of monads which are especially relevant for the rest of the paper.

Example 1. The identity functor $I\colon \mathsf{C} \longrightarrow \mathsf{C}$ is perhaps the most trivial example of monad (with ρ and η given by the identity natural transformation).

Definition 3. For any object X in a category D, we define the *constant functor* $\underline{X}\colon \mathsf{C} \to \mathsf{D}$ to be the functor which assigns X to every object of C and 1_X to every arrow of C. For instance if D has a final object $*$, then $\underline{*}$ is the final functor (in the functor category $\mathsf{C} \to \mathsf{D}$).

Example 2. In particular, if $\mathsf{C} = \mathsf{D}$ the (just defined) endofunctor $\underline{*}$ has a natural structure of monad (with ρ and η given by the universal property of $\underline{*}$) and it is the final monad over C.

Example 3 (`Maybe` *monad*)**.** The endofunctor $X \mapsto X + *$, which takes an object in C and "adds one point" has a natural structure of monad over C. Borrowing from the Haskell terminology, we call it the `Maybe` monad. The monomorphisms $X \to X + *$ and $* \to X + *$, for each object X in C, give the natural transformations $I \to$ `Maybe` and $\underline{*} \to$ `Maybe`. Both are morphisms of monads.

Example 4 (*Peano naturals*)**.** Here we build the monad $\mathsf{nat}\colon \mathsf{Set} \to \mathsf{Set}$ of naturals, where $\mathsf{nat}(X)$ features the set of formulas built out from the familiar signature $(0, \mathtt{succ})$ and the variables in X in the usual way. Since a formula is determined by

its number of occurrences of succ and its unique leaf (which is either 0 or an element of X), this is formalized by taking $\mathsf{nat}(X) := \mathbb{N} \times (X \amalg \{0\})$.

For η_X we take $x \mapsto (0, x)$ and for ρ_X we take $\rho_X(n, (m, z)) := (n + m, z)$.

Note that this monad nat has an endotransformation $\mathsf{succ}\colon \mathsf{nat} \to \mathsf{nat}$ defined by $\mathsf{succ}_X(n, x) := (n + 1, x)$, which is not a morphism of monad. Indeed it does not even respect the units.

Example 5 (*Lists*)**.** To construct the monad of lists L (over Set), first take the functor $L\colon \mathsf{Set} \to \mathsf{Set}$ satisfying, for each set X

$$L(X) = \Sigma_{n\in\mathbb{N}} X^n = * + X + X \times X + X \times X \times X + \cdots$$

So $L(X)$ is the set of all finite lists with elements in X. Then consider as composition the natural transformation $\rho\colon L \cdot L \to L$ given by the *join* (or *flattening*) of lists of lists:

$$\rho[[a_1, \ldots], [b_1, \ldots], \ldots, [z_1, \ldots]] = [a_1, \ldots, b_1, \ldots, z_1, \ldots].$$

The unit $\eta\colon I \to L$ is the collection of singleton maps $\eta_X\colon x \in X \mapsto [x] \in L(X)$.

Example 6 (*Lambda calculus*)**.** This example will be worked out with the necessary details in Section 6.3, but let us give early some basic ideas (already present, e.g. in [1]). We denote by $FV(M)$ the set of free variables of a λ-term M. For a fixed set X, consider the collection of λ-terms (modulo α-conversion) with free variables in X:

$$\mathsf{SLC}(X) := \{M \mid FV(M) \subset X\}$$

Given a set X we take as unit morphism $\eta_X\colon X \to \mathsf{SLC}(X)$ the application assigning to an element $x \in X$ the corresponding variable in $\mathsf{SLC}(X)$. Every map $f\colon X \to Y$ induces a morphism $\mathsf{SLC}(f)\colon \mathsf{SLC}(X) \to \mathsf{SLC}(Y)$ ("renaming") which makes SLC an endofunctor. Let us now explain what can be called the tautological substitution, which is a natural transformation

$$\rho_X\colon \mathsf{SLC}(\mathsf{SLC}(X)) \to \mathsf{SLC}(X).$$

Since indexing variables by lambda-terms may appear counter-intuitive, we briefly explain why this makes perfect sense. This is about terms equipped with an intention of substitution: think for instance of a term with three free variables which you intend to replace, respectively, by I, K and S. This intention is reflected by seeing this term not in $\mathsf{SLC}(\{x, y, z\})$ but rather in $\mathsf{SLC}(\{I, K, S\})$. More generally, terms equipped with an intention of substitution (by closed terms) are reflected in $\mathsf{SLC}(\mathsf{SLC}(\emptyset))$. Finally terms equipped with an intention of substitution by terms with variables in X are perfectly reflected in $\mathsf{SLC}(\mathsf{SLC}(X))$, and the tautological substitution replaces each variable, which is now a term, by this term.

Equipped with this η and this ρ, SLC is a monad, which we call *monad of the "syntactic"* λ-calculus.

Moreover, the monad composition is compatible with the β and η conversions. Then, taking the quotient by the β, or the η, or the $\beta\eta$ equivalence relation we get three new monads, and the four associated projections are monad morphisms. In particular, we denote by LC the quotient by $\beta\eta$-equivalence and we call it *monad of the "semantic"* λ*-calculus* (cfr. Section 7).

The formal development illustrated in the appendix contains the proofs of these facts (see Appendices A.3 and A.4).

Definition 4 (*Derivative*)**.** We define the *derivative* F' of an endofunctor $F\colon \mathsf{C} \to \mathsf{C}$ to be the functor $F' = F \cdot \mathtt{Maybe}$. We can iterate the construction, and denote by $F^{(n)}$ the nth derivative.

Example 7 (*The derivative of a monad*)**.** The derivative of a monad R has an induced structure of monad defined as follows.[1] We define the natural transformation $\gamma\colon \mathtt{Maybe} \cdot R \to R \cdot \mathtt{Maybe}$ such that, for each X, $\gamma_X\colon \mathtt{Maybe}(R(X)) \to R(\mathtt{Maybe}(X))$ acts on $R(X)$ by functoriality of R and sends $*$ to $\eta_R(*)$. It is easily checked that γ is what is called a distributive law (see [3]) of R over $\mathtt{Maybe}$. This is the standard piece of data for turning $R \cdot \mathtt{Maybe}$ into a monad. We just mention the corresponding notations: if η and ρ are the unit and the composition of the monad R, we denote by η' the unit of R', which is the composition $I \xrightarrow{\eta} R \longrightarrow R'$ and by ρ' the composition of R' which results from the following composition:

$$R \cdot \mathtt{Maybe} \cdot R \cdot \mathtt{Maybe} \xrightarrow{R\cdot\gamma\cdot\mathtt{Maybe}} R \cdot R \cdot \mathtt{Maybe} \cdot \mathtt{Maybe} \xrightarrow{\rho_R\cdot\rho_{\mathtt{Maybe}}} R \cdot \mathtt{Maybe}.$$

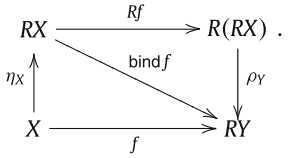
Thanks to this notion of derivative, the abstraction on λ-terms gives a natural transformation $\mathsf{abs}\colon \mathsf{SLC}' \to \mathsf{SLC}$ which takes a λ-term $M \in \mathsf{SLC}(X + *)$ and binds the "variable" $*$. This fails to be a morphism of monads. Indeed, it does not even respect units.

[1] This corresponds to the `MaybeT` monad transformer in Haskell.

2.3. The Kleisli category and the bind construction

The Kleisli category is an important canonical construction associated to every monad. We recall here its definition and its relation with the bind construction. For more details on this topic we refer to book of Mac Lane [19]. These constructions will be generalized to modules (with more detailed proofs) in Section 3.3. We consider a monad (R, ρ, η) over C.

Definition 5 (*The* bind *operator*)**.** Given an arrow $f : X \to RY$, we define the morphism $\mathsf{bind}\, f : RX \to RY$ by $\mathsf{bind}\, f := \rho_Y \cdot Rf$, which yields the following commutative diagram:



We have the following associativity and unity equations for bind:

$$\mathsf{bind}\, g \cdot \mathsf{bind}\, f = \mathsf{bind}(\mathsf{bind}\, g \cdot f), \quad \mathsf{bind}\, \eta_X = 1_{RX}, \quad \mathsf{bind}\, f \cdot \eta_X = f \tag{1}$$

for any pair of arrows $f : X \to RY$ and $g : Y \to RZ$.

Definition 6 (*Kleisli category*)**.** The *Kleisli category* associated to R, denoted $\mathsf{Kl}(R)$, is defined as follows:

- The objects of $\mathsf{Kl}(R)$ are those of C. We will write X_R when we intend to consider an object X of C as an object of $\mathsf{Kl}(R)$.
- Given two objects X and Y, the arrows from X_R to Y_R in $\mathsf{Kl}(R)$ are those from X to RY in C. We will write $f_R : X_R \longrightarrow Y_R$ when we intend to consider an arrow $f : X \longrightarrow RY$ in C as an arrow of $\mathsf{Kl}(R)$.
- The identity arrow of X_R is given by $(\eta_X)_R$.
- Given $f : X \to RY$ and $g : Y \to RZ$ two arrows in C, we define the composition in $\mathsf{Kl}(R)$ of f_R and g_R with the rule

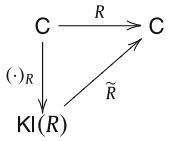
$$g_R \cdot f_R = (\mathsf{bind}\, g \cdot f)_R = (\rho_Z \cdot R(g) \cdot f)_R$$

The category properties of $\mathsf{Kl}(R)$ are easy consequences of equations (1).

Definition 7 (*Kleisli extension*)**.** We define the *Kleisli extension* for R, $\widetilde{R} : \mathsf{Kl}(R) \longrightarrow \mathsf{C}$ as follows:

- $\widetilde{R} X_R = RX$ for each object X;
- $\widetilde{R} f_R = \mathsf{bind}\, f$ for each arrow $f : X \to RY$ in C.

The functor properties of $\widetilde{R}$ reduce to the first two equations in (1). The functor $\widetilde{R}$ extends R in the sense that the following diagram of functors is commutative:



As is well-known, $\widetilde{R}$ is right adjoint to $(.)_R$.

It is well-known that a monad can be alternatively defined through the bind and η operators. Indeed, the equations

$$Rf = \mathsf{bind}(\eta \cdot f), \qquad \rho_X = \mathsf{bind}\, 1_{RX} \tag{2}$$

express the functor and monad structure of R in terms of the bind and η operators and we have the following proposition.

Proposition 1. *Let* $R : \mathrm{obj}(\mathsf{C}) \to \mathrm{obj}(\mathsf{C})$ *be a mapping* (*where* $\mathrm{obj}(\mathsf{C})$ *denotes the set of objects of* C), *and* bind, η *two operators*

$$\mathsf{bind} : \forall X, Y : \mathrm{obj}(\mathsf{C}), \mathrm{Hom}(X, RY) \longrightarrow \mathrm{Hom}(RX, RY), \qquad \eta : \forall X : \mathrm{obj}(\mathsf{C}), X \to RX$$

satisfying (1). *Consider on* R *the structure defined by equations* (2). *Then* (R, ρ, η) *is a monad and* $\widetilde{R} = (R, \mathsf{bind})$ *is the associated Kleisli extension*.

It is also easy to characterize morphisms of monads in terms of the bind construction.

Proposition 2. *Let R and S be two monads over the category C and consider a family ϕ of arrows $\phi_X : RX \to SX$ for X varying on $\mathrm{obj}(\mathsf{C})$. Assume that ϕ commutes with the unit and* bind *of the two monads, that is, for each $X, Y \in \mathrm{obj}(\mathsf{C})$ and $f : X \to RY$ we have*

$$\phi_X \cdot \eta_{R,X} = \eta_{S,X}$$

and

$$\phi_Y \cdot \mathsf{bind}_R f = \mathsf{bind}_S (\phi_Y \cdot f) \cdot \phi_X.$$

Then $\phi : R \to S$ is a natural transformation and a morphism of monads.

3. Modules over monads

In this section, we fix a monad $R := (R, \rho, \eta)$ over a category C as above, and we introduce and study left R-modules.

3.1. Left modules

Definition 8 (*Left modules*)**.** A left R-module is given by a functor $M : \mathsf{C} \to \mathsf{D}$ equipped with a natural transformation $\mu : M \cdot R \to M$, called the *action*, which is compatible with the monad composition, more precisely, we require that the following diagrams commute:

$$\begin{array}{ccc} M \cdot R^2 & \xrightarrow{M\rho} & M \cdot R \\ {\scriptstyle \mu R}\downarrow & & \downarrow{\scriptstyle \mu} \\ M \cdot R & \xrightarrow{\mu} & M \end{array} \qquad \begin{array}{ccc} M \cdot R & \xleftarrow{M\eta} & M \cdot I \\ {\scriptstyle \mu}\downarrow & \swarrow_{1_M} & \\ M & & \end{array} \tag{3}$$

We will refer to the category D as the *range* of M.

Remark 1. The companion definition of modules over an operad (cf. e.g. [21,13]) follows easily from the observation [25] that operads are monoids in a suitable monoidal category. By the way, this monoidal structure is central in [10].

3.2. Examples of left modules

Example 8. We can see our monad R as a left module over itself (with range C), which we call the *tautological* module. Note that this module structure does not help our Proposition 3 to encompass our Proposition 1.

Example 9. Any constant functor $\underline{W} : \mathsf{C} \to \mathsf{D}$, for W an object in D, is a left R-module (in an obvious way).

Example 10. For any functor $F : \mathsf{D} \to \mathsf{E}$ and any left R-module $M : \mathsf{C} \to \mathsf{D}$, the composition $F \cdot M$ is a left R-module (in the evident way).

Example 11 (*Derived module*)**.** As for functors and monads, derivation is well-behaved on left modules: for any left R-module M, the derivative $M' = M \cdot \mathtt{Maybe}$ has a natural structure of left R-module where the action $M' \cdot R \to M'$ is the composition

$$M \cdot \mathtt{Maybe} \cdot R \xrightarrow{M\gamma} M \cdot R \cdot \mathtt{Maybe} \xrightarrow{\mu\mathtt{Maybe}} M \cdot \mathtt{Maybe}$$

and γ is the distributive law introduced for Example 7.

3.3. The Kleisli extension of a left module and the mbind construction

We fix a left module (M, μ) with range in D over the monad (R, ρ, η).

Definition 9 (*The* mbind *operator*)**.** Given an arrow $f : X \to RY$, we define the morphism $\mathsf{mbind}\, f : MX \to MY$ by $\mathsf{mbind}\, f := \mu_Y \cdot Mf$.

The axioms of left module imply the following equations over mbind:

$$\mathsf{mbind}\, g \cdot \mathsf{mbind}\, f = \mathsf{mbind}(\mathsf{bind}\, g \cdot f), \qquad \mathsf{mbind}\, \eta_X = 1_{MX} \tag{4}$$

for any pair of arrows $f\colon X \to RY$ and $g\colon Y \to RZ$. These equations express exactly that the following defines a functor.

Definition 10 (*Kleisli extension*)**.** We define the *Kleisli extension* of M, $\widetilde{M}\colon \mathsf{Kl}(R) \longrightarrow \mathsf{D}$ as follows:

- $\widetilde{M}X_R = MX$ for each object X;
- $\widetilde{M}f_R = \mathsf{mbind}\, f$ for each arrow $f\colon X \to RY$ in C.

Just as in the case of a monad, a left module can be alternatively defined through the mbind and η operators. Indeed, we have the following relations:

$$Mg = \mathsf{mbind}(\eta_Y \cdot g), \qquad \mu_X = \mathsf{mbind}\, 1_{RX} \tag{5}$$

which express the functoriality and the module action in terms of the mbind and η operators. And we have a result analogous to Proposition 1.

Proposition 3.

1. *Given a functor* $\phi\colon \mathsf{Kl}(R) \longrightarrow \mathsf{D}$, *there exists a unique left module* M *over* R *satisfying* $\widetilde{M} = \phi$.
2. *In other words, let* $M\colon \mathsf{obj}(\mathsf{C}) \to \mathsf{obj}(\mathsf{D})$ *be a mapping* ($\mathsf{obj}(\cdot)$ *denotes the set of objects*) *and* mbind *be an operator*

$$\mathsf{mbind}\colon \forall X\, Y : \mathsf{obj}(\mathsf{C}),\ \mathrm{Hom}(X, RY) \longrightarrow \mathrm{Hom}(MX, MY)$$

satisfying (4). *Then the formulas given by equations* (5) *turn* M *into an* R*-module*.

Proof. The first assertion follows directly from the second one by the very definition of functor. For the second assertion, let us give the explicit calculation showing that the operators defined by equations (5) satisfy the axioms of left-modules. First, we check that M satisfies the functor axioms: let X, Y, Z be objects of C and $f\colon X \to Y$, $g\colon Y \to Z$ two arrows. Then

$$\begin{aligned} M\, 1_X &= \mathsf{mbind}(\eta_X \cdot 1_X) = \mathsf{mbind}\, \eta_X = 1_X, \\ Mg \cdot Mf &= \mathsf{mbind}(\eta_Z \cdot g) \cdot \mathsf{mbind}(\eta_Y \cdot f) \\ &= \mathsf{mbind}(\mathsf{bind}(\eta_Z \cdot g) \cdot \eta_Y \cdot f) = \mathsf{mbind}(\eta_Z \cdot g \cdot f) = M(g \cdot f). \end{aligned}$$

Now let us show that the action μ is compatible with the monad structure as expressed by (3). For each object X, we have

$$\begin{aligned} &\mu_X \cdot M\, \eta_X \mathsf{mbind}\, 1_{RX} \cdot \mathsf{mbind}(\eta_{RX} \cdot \eta_X) = \\ &\qquad = \mathsf{mbind}(\mathsf{bind}\, 1_{RX} \cdot \eta_{RX} \cdot \eta_X) = \mathsf{mbind}(1_{RX} \cdot \eta_X) = 1_{MX} \end{aligned}$$

which proves the commutativity of the right diagram in (3). Finally, the commutativity of the left part of the same diagram is shown by the following calculations:

$$\begin{aligned} \mu_X \cdot \mu_{RX} &= \mathsf{mbind}\, 1_{RX} \cdot \mathsf{mbind}\, 1_{R(RX)} = \mathsf{mbind}(\mathsf{bind}\, 1_{RX} \cdot 1_{R(RX)}) = \mathsf{mbind}(\mathsf{bind}\, 1_{RX}), \\ \mu_X \cdot M\, \rho_X &= \mathsf{mbind}\, 1_{RX} \cdot \mathsf{mbind}(\eta_{RX} \cdot \rho_X) = \mathsf{mbind}(\mathsf{bind}\, 1_{RX} \cdot \eta_{RX} \cdot \mathsf{bind}\, 1_{RX}) \\ &= \mathsf{mbind}(1_{RX} \cdot \mathsf{bind}\, 1_{RX}) = \mathsf{mbind}(\mathsf{bind}\, 1_{RX}). \qquad \square \end{aligned}$$

4. Linearity

The notion of module is coined in order to select the "right" notion of morphism. A morphism will be a *linear* natural transformation among modules. Here we fix a monad $R := (R, \rho, \eta)$ on a category C as above.

4.1. Categories of left modules

Definition 11 (*Linear natural transformations*)**.** Given two left R-modules M, N with the same range, we say that a natural transformation $\tau\colon M \to N$ is *linear* if it is compatible with the actions, in the sense that the following diagram commutes:

$$\begin{array}{ccc} M \cdot R & \xrightarrow{\tau R} & N \cdot R \\ {\scriptstyle \mu_M}\downarrow & & \downarrow{\scriptstyle \mu_N} \\ M & \xrightarrow{\tau} & N \end{array}$$

We take linear natural transformations as left module morphisms.

Remark 2. Here the term *linear* refers to linear algebra: linear applications between modules over a ring are group morphisms compatible with the action of the ring. This is compatible with the usual flavor of the word linear (no duplication, no junk, see the first example below).

Definition 12 (*Category of left modules*)**.** We check easily that linear morphisms between left R-modules with the same range yield a category (equipped with a forgetful functor to the functor category). We denote by $\mathrm{Mod}^{\mathsf{D}}(R)$ the category of left R-modules with range D.

Definition 13 (*Product of left modules*)**.** We check easily that the cartesian product of two left R-modules as functors (having as range a cartesian category D) is naturally a left R-module again and is the cartesian product also in the category $\mathrm{Mod}^{\mathsf{D}}(R)$. We also have finite products as usual. The final left module $*$ is the product of the empty family.

We just mention (for completeness) that our category of left modules admits limits and colimits as soon as the target category does. We also mention that right modules can be introduced in a similar way. However, it is less clear to us how they could interact with the Kleisli construction.

4.2. Examples of non linear and linear natural transformations

Example 12. Here we choose for R the monad on Set generated by a binary constructions $+$ and we build (by recursion) a natural transformation $\alpha\colon R \to R$ as follows: for a variable x, we take $\alpha(x) := x + x$, while for the other case we take $\alpha(a + b) := \alpha(a) + \alpha(b)$. It is easily verified that α is not linear with respect to the tautological module structure: when checking the diagram against $\eta((\eta x) + (\eta y))$, we obtain $(\eta x + \eta x) + (\eta y + \eta y) \neq (\eta x + \eta y) + (\eta x + \eta y)$.

Example 13. We easily check that the natural transformation of a left module into its derivative is linear. Note that there are two natural inclusions of the derivative M' into the second derivative M''. Both are linear.

Example 14. Consider again the monad of lists L. The concatenation of two lists is an L-linear morphism $L \times L \to L$.

Definition 14 (*Evaluation*)**.** Here our domain category C is Set. Given an R-module M with range in Set, we have a natural "evaluation" morphism $\mathsf{eval}\colon M' \times R \to M$, where M' is the derivative of M, defined as follows. Given a set X and two elements $x \in RX$ and $m \in M(X + *)$ we define $\mathsf{eval}_X(m, x) := \mathsf{mbind}\, w_x\, m$, where $w_x\colon (X + *) \to RX$ is the sum of η_X with the constant map with value x.

More generally, for each non-negative number n, by composition, we have a natural "evaluation" morphism $\mathsf{eval}_n\colon M^{(n)} \times R^n \to M$, where $M^{(n)}$ is the nth derivative of M.

4.3. Linearity and the Kleisli extensions

Our approach via Kleisli constructions to monads and modules extends to linearity as follows.

We have the following characterization of linearity (compare with Proposition 2 for the analogous characterization of monad morphisms):

Proposition 4. *Let R be a monad over C and consider two left R-modules M, N with range in D. Then the family of arrows $\tau_X\colon MX \to NX$ (for X varying in $\mathrm{obj}(\mathsf{C})$) is a linear natural transformation if and only if, for each arrow $f\colon X \longrightarrow RY$ the diagram*

$$\begin{array}{ccc} MX & \xrightarrow{\mathsf{mbind}_M f} & MY \\ {\scriptstyle \tau_X}\downarrow & & \downarrow{\scriptstyle \tau_Y} \\ NX & \xrightarrow[\mathsf{mbind}_N f]{} & NY \end{array} \tag{6}$$

is commutative.

Proof. Let us assume first that τ is linear and consider an arrow $f\colon X \longrightarrow RY$. Then we have

$$\tau_Y \cdot \mathsf{mbind}_M f = \tau_Y \cdot (\mu_M)_Y \cdot Mf = (\mu_N)_Y \cdot \tau_{RY} \cdot Mf = (\mu_N)_Y \cdot Nf \cdot \tau_X = \mathsf{mbind}_N f \cdot \tau_X.$$

Hence diagram 6 is commutative.

Conversely, assume that for each arrow $f : X \to RY$ the diagram 6 is commutative. Given an arrow $g : X \to Y$ we consider $f = (\eta_R)_Y \cdot g$ and we get

$$\tau_Y \cdot Mg = \tau_Y \cdot \mathsf{mbind}_M f = \mathsf{mbind}_N f \cdot \tau_X = Ng \cdot \tau_X$$

where the first and the last equalities follows by the first equation of (5). This shows that τ is a natural transformation. Finally, for any arrow $f : X \longrightarrow RY$ we have

$$\tau_X \cdot (\mu_M)_X = \tau_X \cdot \mathsf{mbind}_M\, 1_{RX} = \mathsf{mbind}_N\, 1_{RX} \cdot \tau_{RX} = (\mu_N)_X \cdot \tau_{RX}$$

hence τ is linear. □

The commutativity in the previous statement can be reinterpreted as follows:

Proposition 5. *Let R and D as above. We define the Kleisli extension as a functor*:

$$\widetilde{(\cdot)} : \ \mathrm{Mod}^D(R) \to \mathrm{Hom}(\mathsf{Kl}(R), D)$$

by sending M to $\widetilde{M}$ as previously defined, and sending $\tau : M \to N$ to $\tilde{\tau}$ defined, for any $X \in \mathrm{obj}(C)$, by $\tilde{\tau}_X : \widetilde{M}X \to \widetilde{N}X := \tau_X$. Then $\widetilde{(\cdot)}$ is a functor and an isomorphism of categories.

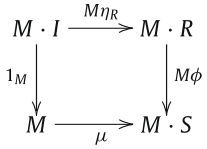
4.4. Base change

Proposition 6 (*Base change*). *Given a morphism $\phi : R \to S$ of monads and a left S-module M, we have an R-action on M given by*

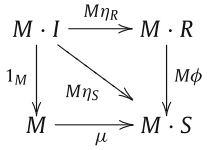
$$M \cdot R \xrightarrow{M\phi} M \cdot S \xrightarrow{\mu_M} M.$$

We denote by ϕ^*M the resulting R-module and we refer to ϕ^* as the base change operator.

Proof. We first have to prove the commutativity of the diagram

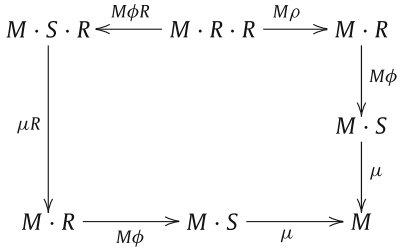


We split this diagram as follows:

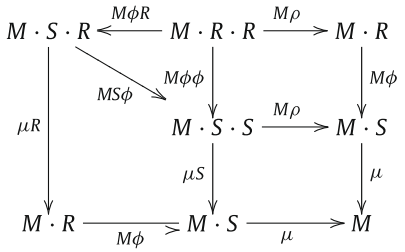


The upper triangle is commutative because ϕ is a morphism ($\phi \cdot \eta_R = \eta_S$). The lower triangle is commutative because μ is an action.

Next we have to prove the commutativity of the following diagram.



which we break into four pieces as follows:



The upper square is commutative because ϕ is a morphism. The lower square is commutative because M is an S-module. The triangle is commutative by definition of the composition $\phi\phi$. The final piece is commutative because μ is natural. □

Definition 15 (*Base change functor*)**.** Given a morphism $\phi\colon R \to S$ of monads, we define the base change functor $\mathrm{Mod}^{\mathsf{D}}(S)$ to $\mathrm{Mod}^{\mathsf{D}}(R)$. We denote it again by ϕ^* since its object part is the previous defined base change operator.

For the morphism part, given a morphism $\alpha\colon M \to N$ of left S-modules, we take $\phi^*(\alpha) := \alpha$. Indeed, this natural transformation being S-linear is easily checked to be also R-linear. This obviously defines a functor as announced.

The following results are easy consequences of the definitions.

Proposition 7. *The base change functor commutes with products and with derivation*.

Proposition 8. *Given a morphism of monads $\phi\colon R \to S$, the underlying natural transformation also underlies a morphism of left R-modules, still denoted ϕ, from the tautological R-module to ϕ^*S*.

5. First-order typed syntax

In this section, we check the approach to syntax through modules against the first-order typed case. This approach starts by defining arities and signatures, then it defines representations of arities or signatures in monads, and the category of such representations, and finally looks for an initial object in this category. We fix a set D of *types* (or *degrees*). From now on, we will say *modules* instead of *left modules*.

Definition 16 (*D-sets*)**.** We call D-set any family of sets indexed by D and write $D \to \mathsf{Set}$ for the (large) set of D-sets. For morphisms among D-sets, we take families of applications indexed by D. This turns $D \to \mathsf{Set}$ into a category. For each D-set X and $d \in D$, we denote by X_d rather than $X(d)$ the component of X in degree d. We denote by τ_d the corresponding functor from Set/D to Set. There is an essentially equivalent view, where a D-set is a set over D, namely a set A equipped with a map from A to D. This map assigns to each element of A its *type* (or *degree*). The corresponding category is the slice category Set/D, where morphisms are type-preserving applications. The categories $D \to \mathsf{Set}$ and Set/D are equivalent. For an object X in Set/D we write again X_d for the inverse image of d in X, and we write again τ_d for the functorial assignment $X \mapsto X_d$.

Definition 17 (*D-arities and D-signatures*)**.** A D-arity is a finite non-empty list of elements in D. We use a colon to single out the first element of the list, which stands for the type of the result, while the other elements of the list stand for the types of the arguments. A signature is a family of arities, i.e., a pair $\Sigma = (O, \alpha)$ where $\alpha(o) = (b(o) : \underline{a}(o))$ is an arity for every $o \in O$.

Definition 18 (*Elementary D-modules*)**.** We say that a monad over Set/D is a D-monad. Given a D-monad R, and a type $d \in D$ we denote by R_d the R-module with range in Set obtained by applying τ_d to R, in other words: $R_d := \tau_d \cdot R$. More generally, for a list $\underline{a} := (a_1, \ldots, a_n)$ of elements in D, we write $R_{\underline{a}}$ for the product $R_{a_1} \times \cdots \times R_{a_n}$. These modules are our *elementary* modules over R.

Definition 19 (*Representation of a D-arity*)**.** Given a D-monad R, we define a *representation* of the D-arity $\alpha = (b : a_1, \ldots, a_n) = (b : \underline{a})$ in R to be a morphism of modules:

$$r\colon \prod_{i=1}^{n} R_{a_i} \longrightarrow R_b$$

or, in other words:

$$r\colon R_{\underline{a}} \longrightarrow R_b.$$

We also say that r is a *construction* of arity α in R.

Definition 20 (*Representation of a D-signature*)**.** A representation of the D-signature $\Sigma = (O, \alpha)$ in the D-monad R, consists of, for each o in O, a representation of the D-arity $\alpha(o)$ in R.

Definition 21 (*The category of representations*)**.** Given a signature $\Sigma = (O, \alpha)$, we build the category Rep_D^Σ of representations of Σ in D-sets as follows. Its objects are D-monads equipped with a representation of Σ. A morphism from (R, r) to (S, s) is a morphism ϕ from R to S compatible with the representations in the sense that, for each o in O, with $\alpha(o) = (b : a_1, \ldots, a_n)$, the following diagram of R-modules commutes:

$$\begin{array}{ccc} \prod_{i=1}^n R_{a_i} & \longrightarrow & R_b \\ \downarrow & & \downarrow \\ \phi^* \prod_{i=1}^n S_{a_i} & \longrightarrow & \phi^* S_b \end{array} \tag{7}$$

where the horizontal arrows come from the representations and the vertical arrows come from ϕ.

Proposition 9. *These morphisms, together with the obvious composition, turn* Rep_D^Σ *into a category*.

Theorem 1. *The category* Rep_D^Σ *has an initial object which we call the inductive D-monad generated by* Σ*, and denote by* $\widehat{\Sigma}$.

The proof of this theorem is straightforward. We just mention that the initial object may be constructed as a set of abstract syntax trees. As usual, our "syntactic" object $\widehat{\Sigma}$ comes with corresponding recursion and induction principles which we omit to state explicitly.

Example 15. Here we consider the syntax for typed lists. For the set of types, we take the inductive set generated by the grammar $\tau = * \mid \mathsf{list}\,\tau$ (we could of course take more base types). Next we describe our signature *TL*. It consists of the following two families, nil and cons, of arities, both indexed by τ: for each $t \in \tau$, $\mathsf{nil}_t = (\mathsf{list}\,t :)$ and $\mathsf{cons}_t = (\mathsf{list}\,t :\ t,\ \mathsf{list}\,t)$. According to our theorem, this signature defines a syntactic monad. This monad assigns to each set of typed variables the set of typed lists built out from these variables.

6. Higher-order untyped syntax

In the previous section, we have considered typed constructions without bindings. Here we consider untyped constructions with bindings. The approach is the same as in the previous section. We define (untyped higher-order) arities and signatures. Then we define the representations of such signatures in a monad, and morphisms between representations. Finally we look for an initial object in the category of representations of a signature, which will be "the" syntax associated with the given signature. We give a general existence theorem. Then we show, on the example of the λ-calculus, how easily such a higher-order syntax can be implemented in Haskell.

6.1. The general construction

All monads in this section and the next are assumed to be over the category Set of (small) sets.

Definition 22 (*HO-arities*)**.** We define an *arity* (or HO-arity) to be a non-empty list of integers. We say the arity is *flat* when the first element of the list is zero. Given an arity $a = (b : a_1, \ldots, a_n)$, we denote by $\mathit{flat}(a)$ the (flat) arity $(0 : a_1, \ldots, a_n, 0^b)$, where 0^b stands for a list of b zeros.

The intended meaning is that a (possibly binding) construction will have an arity. The number n will be the number of arguments, while a_i will be the number of bound variables in the ith argument. Finally, the number b is the number of "fresh" variables in the output of the construction.

Thus, the difference with the definition in [10] is the extra integer (b) opening new room for non flat arities. The purpose of non flat arities is to offer some extra flexibility in the presentation of theories. Here is the emblematic example.

Example 16. The semantic λ-calculus is a monad *LC* where the abstraction is a linear isomorphism $\mathsf{abs} : LC' \to LC$. The arity of this construction is $(0 : 1)$ because it has one argument where one variable is bound. The inverse isomorphism is what we call the app_1 construction, $\mathsf{app}_1 : LC \to LC'$, namely the curryfication of the usual app, which accepts only one argument (hence its output waits for an extra argument). To this construction app_1, we assign the non flat arity $(1 : 0)$, meaning that it expects a single argument where no extra variable is bound, and that its output expects one extra argument. More importantly, presenting *LC* with abs and app_1 (instead of the usual app) allows for a strikingly simple formulation of the β and η rules. For more details see the next section.

Definition 23 (*Binding signatures*)**.** We define a (binding) signature $\Sigma = (O, \alpha)$ to be a family of arities $\alpha\colon O \to \mathbb{N} \times \mathbb{N}^*$. Here $\mathbb{N}^*$ stands for the set of lists of natural numbers. A signature is said to be flat if it consists of flat arities. For a signature $\Sigma = (O, \alpha)$, we denote by $\mathit{flat}(\Sigma)$ the flat signature $(O, o \mapsto \mathit{flat}(\alpha(o)))$.

Next, we describe what are the elementary modules in the present context.

Definition 24 (*Elementary modules*)**.** Given a monad R over Set, and a possibly empty list $\underline{a} := (a_1, \dots, a_n)$ of integers, we denote by $R^{(\underline{a})}$ the R-module with range in Set obtained as product of derivatives: $R^{(\underline{a})} := R^{(a_1)} \times \cdots \times R^{(a_n)}$.

Definition 25 (*Representation of an arity*)**.** Given a monad R over Set, we define a representation of the arity $\alpha = (b : \underline{a})$ in R to be a module morphism

$$r\colon R^{(\underline{a})} \longrightarrow R^{(b)}.$$

We also say that r is a *construction* of arity α in R. In case $b = 0$ we say that r is a flat construction in R.

Definition 26 (*The category of representations*)**.** Given a signature $\Sigma = (O, \alpha)$, we build the category Mon^{Σ} of representations of Σ as follows. Its objects are monads equipped with a representation of Σ. A morphism ϕ from (R, r) to (S, s) is a morphism from R to S compatible with the representations in the sense that, for each o in O with $\alpha(o) = (b : \underline{a})$, the following diagram commutes:

$$\begin{array}{ccc} R^{(\underline{a})} & \longrightarrow & R^{(b)} \\ \downarrow & & \downarrow \\ \phi^* S^{(\underline{a})} & \longrightarrow & \phi^* S^{(b)} \end{array} \tag{8}$$

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

Proposition 10. *These morphisms, together with the obvious composition, turn* Mon^{Σ} *into a category which comes equipped with a forgetful functor to the category of monads.*

Theorem 2. *Let* Σ *be a* (*binding*) *signature. Then the category* Mon^{Σ} *of representations of* Σ *has an initial object* $\widehat{\Sigma}$.

Proof. The problem can be reduced to the case of flat signatures. This reduction is discussed in the next section (Proposition 11). Thus, let us assume that $\Sigma = (O, \alpha)$ is a flat signature.

Our task is fourfold. We have to build a monad $\widehat{\Sigma}$, to equip it with a representation ρ, to build a morphism ι from ρ to any other representation Θ and to prove the uniqueness of such a morphism.

The monad.

We give a brief presentation of the monad, which is fairly standard. The elements in $\widehat{\Sigma}(X)$ are syntactic trees with free variables in X (and for bound variables our "de Bruijn indices": $\mathsf{None}(X)$, $\mathsf{None}(X^*)$, ...) as expected. The unit of $\widehat{\Sigma}$ takes an element x in the given set X and maps it to the syntactic tree with a single node labeled with x. The composition of the monad takes a syntactic tree with free variables in $\widehat{\Sigma}(X)$ and maps it to the syntactic tree obtained by replacing each leaf labeled by a free variable f by the syntactic tree f.

In such a syntactic tree $t \in \widehat{\Sigma}(X)$, to each node γ is associated its *shift*. This is an integer n such that the subtree at γ may be seen as a syntactic tree in $\widehat{\Sigma}(X^{(n)})$ where $X^{(k)}$ denotes $X + * \cdots *$ (k times). The shift function is defined in order to satisfy the arity condition: if the arity of the constructor at γ is $(0 : a_1, \dots, a_p)$, the shifts at the sons of γ are $n + a_1, \dots, n + a_p$.

The initial representation.

For each $o \in O$ we have a corresponding arity $\alpha(o) = (0 : \underline{a}) = (0 : a_1, \dots, a_n)$ in Σ and a natural transformation $r_o\colon \widehat{\Sigma}^{(\underline{a})} \to \widehat{\Sigma}$. For each set X, $r_{o,X}$ maps the tuple $(t_1, \dots, t_n) \in \widehat{\Sigma}^{(\underline{a})}(X)$ to the tree with the root labeled with o and $t_1, \dots, t_n$ as subtrees. This is a natural transformation since it obviously commutes with renaming. Due to our original approach, we have to prove slightly more, namely that this natural transformation is linear. This just means that the given construction commutes not only with renaming, but with any substitution, which is again obvious.

Uniqueness of the initial morphism of representations.

We start by checking the uniqueness of the initial morphism, which is simpler than the existence. So we consider another representation r of Σ into a monad R and take two such morphisms p and q from ρ to r. We must prove that for any t in any $\widehat{\Sigma}(X)$, $p(t)$ and $q(t)$ are equal.

This is proven by induction on the structure of the syntactic tree of t. The case where t is a variable is obvious since p and q are morphisms of monads, hence commute with the units.

Otherwise, t reads $\rho_{o,X}(s_1,\dots,s_n)$ where $s_1,\dots,s_n$ are subtrees. We have

$$p(t) = r_{o,X}(p(s_1),\dots,p(s_n)) = r_{o,X}(q(s_1),\dots,q(s_n)) = q(t).$$

The first and third equalities follow from the fact that p and q are morphisms of representations, while the second equality follows by induction.

Existence of the initial morphism of representations.

The construction of the initial morphism is parallel to the previous proof. So we consider another representation r of Σ into a monad R and define our morphism from ρ to r by recursion on the structure of trees. In the case where the given syntactic tree t is a variable $\eta_{\widehat{\Sigma},X}(x)$, we take $\iota_X(x) := \eta_{R,X}(x)$. Otherwise t reads $\rho_{o,X}(s_1,\dots,s_n)$, and we take

$$\iota_X(t) := r_{o,X}(\iota_{X^{(a_1)}}(s_1),\dots,\iota_{X^{(a_n)}}(s_n)).$$

Let us note that, since a functor on a large category is not a set, it is not clear how this construction could be formalized without assuming the existence of a universe, and working inside it. Note also that this issue does not show up when formalizing in Coq, thanks to the implicit hierarchy of universes.

Now we have to check that this transformation is a natural transformation and a monad morphism. Thanks to Proposition 2 both properties can be verified at once by showing that ι commutes with the η and bind operators. The preservation of the unit follows directly by the definition of ι.

For the preservation of bind we show that the natural transformation $\iota^{(k)}\colon \widehat{\Sigma}^{(k)} \to R^{(k)}$ (or more precisely $\iota^* R^{(k)}$) is $\widehat{\Sigma}$-linear for all k, that is, for every arrow $f\colon X \to \widehat{\Sigma}Y$ and $t \in \widehat{\Sigma}^{(k)}X$ we have

$$\iota_Y^{(k)}\left(\mathsf{mbind}_{\widehat{\Sigma}^{(k)}} f\, t\right) = \mathsf{mbind}_{R^{(k)}}\left(\iota_Y \cdot f\right)\left(\iota_X^{(k)} t\right). \tag{9}$$

We remark that the two maps $\iota_X^{(k)}$ and $\iota_{X^{(k)}}$ are the same. For $k = 0$ the previous formula reduces to compatibility of ι and bind which is our thesis. We proceed by induction on the structure of t. If t is a variable $\eta_{\widehat{\Sigma},X^{(k)}}(x)$ then both sides of the equation are $\iota_Y^{(k)}(f^{(k)}(x))$, where $f^{(k)}\colon X^{(k)} \to \widehat{\Sigma}Y^{(k)}$ denotes f shifted by k. Otherwise, t reads $\rho_{o,X^{(k)}}(s_1,\dots,s_n)$ and we can assume that equation (9) holds for the subtrees $s_j \in \widehat{\Sigma}^{(a_j+k)}X$. Then we have

$$\begin{aligned}
&\iota_Y^{(k)}\left(\mathsf{mbind}_{\widehat{\Sigma}^{(k)}} f\,(\rho_{o,X^{(k)}}(s_1,\dots,s_n))\right)\\
&= \iota_Y^{(k)}\left(\rho_{o,Y^{(k)}}\left(\mathsf{mbind}_{\widehat{\Sigma}^{(a_1+k)}} f\, s_1,\dots,\mathsf{mbind}_{\widehat{\Sigma}^{(a_n+k)}} f\, s_n\right)\right) && (10)\\
&= r_{o,Y^{(k)}}\left(\iota_Y^{(a_1+k)}\left(\mathsf{mbind}_{\widehat{\Sigma}^{(a_1+k)}} f\, s_1\right),\dots,\iota_Y^{(a_n+k)}\left(\mathsf{mbind}_{\widehat{\Sigma}^{(a_n+k)}} f\, s_n\right)\right) && (11)\\
&= r_{o,Y^{(k)}}\left(\mathsf{mbind}_{R^{(a_1+k)}}\left(\iota_Y \cdot f\right)\left(\iota_X^{(a_1+k)} s_1\right),\dots,\mathsf{mbind}_{R^{(a_n+k)}}\left(\iota_Y \cdot f\right)\left(\iota_X^{(a_n+k)} s_n\right)\right) && (12)\\
&= \mathsf{mbind}_{R^{(k)}}\left(\iota_Y \cdot f\right)\left(r_{o,Y}\left(\iota_X^{(a_1+k)} s_1,\dots,\iota_X^{(a_n+k)} s_n\right)\right) && (13)\\
&= \mathsf{mbind}_{R^{(k)}}\left(\iota_Y \cdot f\right)\left(\iota_X^{(k)} t\right) && (14)
\end{aligned}$$

where equalities (10) and (13) hold by the linearity of the constructors (see Proposition 4), equalities (11) and (14) hold by the definition of ι and equality (12) follows by induction. □

6.2. Flattening

Definition 27 (*Flattening a representation of an arity*)**.** Given a monad R and a representation r of the arity $\alpha = (b : a_1,\dots,a_n) = (b : \underline{a})$ in R, we define the natural transformation

$$\mathit{flat}(r)\colon R^{(\underline{a})} \times R^b \longrightarrow R$$

as the composition

$$R^{(\underline{a})} \times R^b \xrightarrow{r \times R^b} R^{(b)} \times R^b \xrightarrow{\mathsf{eval}_b} R$$

where eval_b is the operator defined in Definition 14. It is easily checked that $\mathit{flat}(r)$ is a representation of $\mathit{flat}(\alpha)$.

Example 17. A representation of $(1 : 0)$ in SLC is given by the $\mathsf{app}_1\colon \mathsf{SLC} \longrightarrow \mathsf{SLC}'$ construction. The associated representation of $\mathit{flat}(1 : 0) = (0 : 0, 0)$ is the usual app.

Proposition 11 (*Flattening is bijective for arities*)**.** *Given a monad R and an arity $\alpha = (b : a_1, \dots, a_n)$ the flat map defined above defines a bijection from the set of representations of α in R to the set of representations of $\mathit{flat}(\alpha)$ in R.*

Definition 28 (*Flattening a representation of a signature*)**.** Given a monad R and a representation r of the signature $\Sigma = (O, \alpha)$ we define $\mathit{flat}(r)$ to be the family $o \mapsto \mathit{flat}(r(o))$ which is a representation of $\mathit{flat}(\Sigma)$.

Proposition 12 (*Flattening is bijective for signatures*)**.** *Given a monad R and a signature $\Sigma = (O, \alpha)$ the flat map defined above defines a bijection from the set of representations of Σ in R to the set of representations of $\mathit{flat}(\Sigma)$ in R.*

Proposition 13. *Given a signature Σ, the flat construction extends as a functor*

$$\mathrm{Mon}^{\Sigma} \longrightarrow \mathrm{Mon}^{\mathit{flat}\,\Sigma}$$

from the category of representations of Σ to the category of representations of $\mathit{flat}(\Sigma)$. Furthermore, this functor is an isomorphism of categories.

The previous proposition means that, as announced, our Theorem 2 can be proved by reduction to the special case of a flat signature.

6.3. Implementation in Haskell

The programming language Haskell [17] offers strong support for programming through monads. In this section, we show, on the special case of the usual signature $\Sigma = ((0 : 0, 0), (0 : 1))$ of the λ-calculus, how higher-order syntax can be nicely implemented in Haskell. The initial monad $\widehat{\Sigma}$ associated with Σ by our Theorem 2 will be denoted by SLC (SLC standing for "syntactic" λ-calculus, cfr. the example in Section 2). We consider $\mathsf{SLC}(X)$ as the set of λ-terms with free variables taken from X where the Σ-representation morphism $\mathsf{SLC} \times \mathsf{SLC} + \mathsf{SLC}' \longrightarrow \mathsf{SLC}$ gives the familiar constructors of the λ-calculus in the *nameless* encoding [5], namely, the SLC-linear natural transformations

$$\mathsf{app} \colon \mathsf{SLC} \times \mathsf{SLC} \to \mathsf{SLC}, \quad \mathsf{abs} \colon \mathsf{SLC}' \to \mathsf{SLC}.$$

We denote by var the unit of the monad which is the constructor of variables. The bind operator of SLC gives us the substitution (or instantiation) of free variables and the monad axioms together with the SLC-linearity of the app and abs constructors give us an elegant and concise description of the basic properties of substitution, namely the recursive rules for bind :

$$\begin{aligned} \mathsf{bind}\, f\ (\mathsf{var}\ v) &= f\ v \\ \mathsf{bind}\, f\ (\mathsf{app}\ x\ y) &= \mathsf{app}\ (\mathsf{bind}\, f\ x)\ (\mathsf{bind}\, f\ y) \\ \mathsf{bind}\, f\ (\mathsf{abs}\ x) &= \mathsf{abs}\ (\mathsf{mbind}\, f\ x) \end{aligned}$$

and its mandatory properties:

$$\mathsf{bind}\ \mathsf{var}\ x = x\ , \quad \mathsf{bind}\, g\ (\mathsf{bind}\, f\ x) = \mathsf{bind}\ (\mathsf{bind}\, g \cdot f)\ x.$$

Notice the occurrence of mbind in the substitution rule for abs which is where the SLC-module structure of the derivative SLC' comes into play.

Most of this can be condensed in the following ten lines of Haskell code, which appear already in [1], implementing altogether the (nested) datatype for λ-terms and the substitution algorithm. Following the Haskell practice we use the (infix) operator >>= to denote the bind operator with arguments switched. We also switched the arguments in the definition of the mbind operator for consistency. Also recall that in the Haskell terminology `return` denotes the unit of the monad.

```
module SLC where
import Monad (liftM)

data SLC a = Var a | App (SLC a) (SLC a) | Abs (SLC (Maybe a)) -- the nested data type

instance Monad SLC where  -- the monad structure on SLC
  return = Var
  Var x >>= f = f x
  App x y >>= f = App (x >>= f) (y >>= f)
  Abs x >>= f = Abs (mbind x f)           -- notice the ‘‘forward" call to mbind

mbind :: SLC (Maybe a) -> (a -> SLC b) -> SLC (Maybe b)
mbind x f = x >>= maybe (Var Nothing) (liftM Just . f).
```

The last line of this code implements the structure of SLC-module over the derivative SLC'. Observe that the definitions of bind (>>=) and mbind are mutually recursive. We recall from [17] that `Maybe` is the inductive datatype defined by

```
data Maybe a = Nothing | Just a
```

and `maybe` is the associated catamorphism

```
maybe :: b -> (a -> b) -> Maybe a -> b.
```

The operator

```
liftM :: (Monad m) => (a -> b) -> m a -> m b
```

is the functor action on maps (defined in terms of the >>= operator), so the de Bruijn "bump" of free variables is given by the term

```
liftM Just :: (Monad m) => m a -> m (Maybe a).
```

As usual with programming, some properties remain to be checked, namely:

- the mandatory properties mentioned above, which ensure that we have a monad
- the fact that mbind defines a module structure.

This is done in the appendix, where we present the same substitution algorithm with the necessary verifications implemented with the Coq proof assistant.

Notice that the fact that abs is linear can be read out from the code, thanks to Proposition 4.

7. Algebraic semantics: an example

In this section, we illustrate, through the example of the λ-calculus, how modules and their morphisms interplay with semantics. We also show how this point of view may be implemented in Coq [9]. Our approach concerns algebraic semantics. For our case of the signature Σ considered in the previous section, algebraic semantics amounts to "choose" some set of equations, to clarify when a representation of Σ in a monad satisfies these equations, and to show that the category of representations of Σ satisfying our equations has an initial object. This leads us to an algebraic characterization of the semantic monad of the λ-calculus, LC, which we construct below.

7.1. The semantic λ-calculus

For any set X, consider the equivalence relation $\equiv_{\beta\eta}$ on $\mathsf{SLC}(X)$ given by the reflexive symmetric transitive closure of β and η conversions and define $\mathsf{LC}(X) := \mathsf{SLC}(X)/\equiv_{\beta\eta}$. It can be shown that $\equiv_{\beta\eta}$ is compatible with the structure of SLC so LC has a structure of monad, and that the projection $\mathsf{SLC} \to \mathsf{LC}$ is a morphism of monads. Application and abstraction of SLC induce two LC-linear morphisms which we still denote by app and abs.

Now the key fact is that the linear morphism $\mathsf{abs}\colon \mathsf{LC}' \to \mathsf{LC}$ is an isomorphism. In fact, it is easy to construct its inverse $\mathsf{app}_1\colon \mathsf{LC} \to \mathsf{LC}'$:

$$\mathsf{app}_1\, x = \mathsf{app}\,(\hat{x}, *)$$

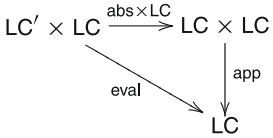
where $x \mapsto \hat{x}$ denotes the natural inclusion $\mathsf{LC} \to \mathsf{LC}'$. The equation

$$\mathsf{abs} \cdot \mathsf{app}_1 = 1_{\mathsf{LC}}$$

clearly corresponds to the η-rule while the other equation

$$\mathsf{app}_1 \cdot \mathsf{abs} = 1_{\mathsf{LC}'}$$

can be considered the ultimate formulation of the β-rule. We do not consider the more classical formulation of the β-rule which can be stated as the commutativity of the diagram



7.2. Exponential monads

So we choose for equations the two rules stated above. The price to pay for this aesthetic choice is the shift from our original flat signature Σ, to the non flat $\Sigma' := ((1:0),(0:1))$. This makes no serious difference, thanks to our results in 6.2, since we have $\mathit{flat}\,\Sigma' = \Sigma$.

Next we make explicit that a representation $(\sigma_{\mathsf{app}}, \sigma_{\mathsf{abs}})$ of Σ' in a monad R satisfies the η-rule when the following holds:

$$\sigma_{\mathsf{abs}} \cdot \sigma_{\mathsf{app}} = 1_R$$

and satisfies the β-rule when the following holds:

$$\sigma_{\mathsf{app}} \cdot \sigma_{\mathsf{abs}} = 1_{R'}.$$

Now we call exponential monads those representations of Σ' which satisfy these two equations. The implementation of such equations for instance in Coq is straightforward, see the code for `ExpMonad` in Appendix A. Exponential monads form a full subcategory of the category of representations of Σ', and LC is an object there.

In this context, we have the expected result:

Theorem 3. *The monad* LC *is initial in the category of exponential monads*.

We have developed a formal proof of the above theorem in the Coq proof assistant [9], which is discussed in the appendix.

This theorem allows to define the (pure untyped semantic) λ-calculus as the initial exponential monad.

It is highly expected that our proof extends readily to arbitrary signatures and equations.

8. Related and future work

We have introduced the notion of module over a monad, and more importantly the notion of linearity for transformations among such modules and we have tried to show that this notion is ubiquitous as soon as syntax and semantics are concerned. Our thesis is that the point of view of modules opens some new room for initial semantics, as we sketched for the λ-calculus.

Despite the natural ideas involved, the notion of module over a monad has been essentially ignored till now: modules over monads are mentioned on a blog by Urs Schreiber[2] and some kind of right modules have been introduced by Bohm and Menini in [6]. Let us also mention that an experienced reader could recognize the notion of module in an old paper of Street [26]. On the other hand, modules over operads have been introduced more than ten years ago by Markl [20,21] and are commonly used by topologists (see, e.g. [13,18,8]). In [12], such modules over operads have been considered, under the name of actions, in the context of semantics.

The idea that the notion of monad is suited for modelling substitution concerning syntax (and semantics) is deeply rooted in the folklore and appears at least in [1,5]. It has been retained by many other recent contributions concerned with syntax, although some other settings have been considered. Notably in [10] the authors work within a setting roughly based on operads (although they do not write this word down; operads and monads are not too far from each other and the connection between them is fairly well understood). As they mention, their approach is, to some extent, equivalent to an approach through monads. It has been applied, e.g. in [28] and generalized, e.g. in [27]. This line of research is further explored in particular by Marcelo Fiore.

Another approach to syntax with bindings, initiated by Gabbay and Pitts [14], relies on a systematic consideration of freshness, an issue which is definitely ignored in the monadic or operadic approach.

Let us also mention the contributions [15,22] where a much more general notion of signature than ours is treated in a (higher-order) algebra-oriented approach. Future work should integrate such general signatures in a module-oriented approach.

As far as applications to semantics are concerned, the typed case is of major interest. In this respect, the approach of [10] has been successfully applied to the semantics of process calculi, see, e.g. [12]. Future work should definitely check our module-oriented approach against this and other related applications. For a first step in this direction, see [30], where the semantic simply typed lambda-calculus is treated in the spirit of our Section 7. The next challenge is to treat the dependently-typed case in a similar way.

Finally let us mention one more track which we envision: the syntactic λ-calculus functor equipped with the β^* relation may be seen as a monad on the category of preordered sets. It seems clear that this construction is pertinent for the study of the operational semantics of the λ-calculus.

Appendix A. Formal proof of Theorem 3

In this appendix, we present our formal proof of Theorem 3 in the Coq proof assistant [9]. We recall the statement of the theorem.

[2] http://golem.ph.utexas.edu/string/archives/000715.html

The monad LC of the semantic untyped λ-calculus is an initial object in the category of exponential monads.

The logical foundation implemented by Coq is the Calculus of (Co)Inductive Constructions (CIC). This is a rich and expressive formalism where the theory of monads and modules can be encoded in a direct manner. In our development we implement monads over sets, i.e., with carriers of type `Set`→`Set` (cf. the definition at the beginning of Appendix A.2). The same approach cannot be adopted in those systems which lack second-order type polymorphism, e.g. the theorem provers of the HOL family. (However, let us mention that an interesting proposal for extending Higher Order Logic with second-order polymorphism and a prototype implementation based on the HOL Light system [16] have been recently made by Völker in [29].)

Our formalization is developed in the spirit of maximal code reusability. All axiomatic definitions are introduced with records. In particular, we avoid the use of the Coq module system (although tempting) since Coq modules are not first class objects thus limiting the expressiveness of the concepts formalized on them.[3]

We include here only a small fraction of the code without proofs. The full sources can be retrieved from the archive of user contributions maintained by the Coq team.[4]

A.1. Structure of the formalization

The structure of our proof can be outlined in the following four major parts:

1. axioms and support library;
2. formalization of monads, modules and exponential monads;
3. formalization of syntactic and semantic λ-calculus;
4. the main theorem.

The second and third parts are independent of each other. As for what this paper is concerned, the first part can be considered as an extension of the Coq system for practical purposes. This part contains some meta-logical material (tactics and notations) and gives an axiomatic definition of quotient types (with extensionality of functions and existence of a section). These axioms were found to be inconsistent in old versions of Coq (prior to 8.0) by Chicli, Pottier and Simpson [7]. The source of the inconsistency was the impredicativity of the type `Set`. In the new version of Coq the type `Set` is predicative by default. As they observe, this version has a classical Boolean model, where our axioms for quotients are true (thanks to the Choice Axiom).

A.2. Formalization of monads and modules

After the preliminary material, our formalization opens the theory of monads and (left) modules (files Monad.v, Mod.v, Derived_Mod.v). This is constructed starting from a rather straightforward translation of the Haskell monad library. As an example, we report here our definitions of monads and modules in the Coq syntax.

```
Record Monad : Type := {
  monad_carrier :> Set -> Set;
  bind : forall X Y, (X -> monad_carrier Y) -> monad_carrier X -> monad_carrier Y;
  unit : forall X, X -> monad_carrier X;
  bind_bind : forall X Y Z
    (f : X -> monad_carrier Y) (g : Y -> monad_carrier Z) (x : monad_carrier X),
    bind Y Z g (bind X Y f x) = bind X Z (fun u => bind Y Z g (f u)) x;
  bind_unit : forall X Y (f : X -> monad_carrier Y) (x : X), bind X Y f (unit X x) = f x;
  unit_bind : forall X (x : monad_carrier X), bind X X (unit X) x = x
}.
Notation ``x >>= f" := (@bind _ _ _ f x).

Record Mod (P : Monad) : Type := {
  mod_carrier :> Set -> Set;
  mbind : forall X Y (f : X -> P Y) (x : mod_carrier X), mod_carrier Y;
  mbind_mbind : forall X Y Z (f : X -> P Y) (g : Y -> P Z) (x : mod_carrier X),
    mbind Y Z g (mbind X Y f x) = mbind X Z (fun u => f u >>= g) x;
  unit_mbind : forall X (x : mod_carrier X), mbind X X (@unit P X) x = x
}.
Notation ``x >>>= f" := (@mbind _ _ _ _ f x).
```

[3] During the preparation of the present paper, Coq has been extended with a new mechanism of type classes which allows to highly improve the readability of our contribution and narrow the gap between the Coq code and the Haskell code shown in this paper. We plan to port our code to the new version of Coq and take advantage of the new mechanism in the near future.

[4] http://coq.inria.fr/contribs-eng.html

The library also includes the definition of morphism of monads and modules and other related categorical material. Other definitions which are specific to our objective are those of derived module and exponential monad. The latter reads as follows:

```
Record ExpMonad : Type := {
  exp_monad :> Monad;
  exp_abs : Mod_Hom (Derived_Mod exp_monad) exp_monad;
  exp_app : Mod_Hom exp_monad (Derived_Mod exp_monad);
  exp_eta : forall X (x : exp_monad X), exp_abs _ (exp_app _ x) = x;
  exp_beta : forall X (x : Derived_Mod exp_monad X), exp_app _ (exp_abs _ x) = x
}.
```

The terms `exp_eta` and `exp_beta` assert that `exp_abs` and `exp_app` are inverses of each other, i.e., that an exponential monad M is isomorphic with its derivative M' (as an M-module). Morphisms of exponential monads $M \longrightarrow N$ are given by monad morphisms which commute with the structural isomorphisms $M \simeq M'$ and $N \simeq N'$, i.e., such that both `exp_abs` and `exp_app` commute with the monad morphism:

```
Record ExpMonad_Hom (M N : ExpMonad) : Type := {
  expmonad_hom :> Monad_Hom M N;
  expmonad_hom_app : forall X (x : M X),
    expmonad_hom _ (exp_app M _ x) = exp_app N _ (expmonad_hom _ x);
  expmonad_hom_abs : forall X (x : Derived_Mod M X),
    expmonad_hom _ (exp_abs M _ x) = exp_abs N _ (expmonad_hom _ x)
}.
```

A.3. Formalization of the λ-calculus

This part contains the definition of the syntactic and the semantic untyped λ-calculus (files Slc.v and Lc.v, respectively). We use nested datatypes to encode λ-terms in the Calculus of (Co)Inductive Constructions. Notice that this encoding can be considered a typeful variant of the well-known de Bruijn encoding [5]. As does the de Bruijn encoding, it represents λ-terms modulo α-conversion.

The correspondence of the following code with the Haskell fragment shown in Section 6.3 is partially obfuscated by the fact that we cannot directly encode the mutual recursive definitions of bind and its derivation without hitting the check of structural recursion performed by Coq. We solve the problem by giving an independent recursive definition of `fct` (corresponding to the term `liftM` in Haskell) before the actual implementation of the substitution algorithm (term `slc_bind`).

Nevertheless, the following code performs the same steps as the corresponding Haskell code. In addition, the Lemma `slc_bind_abs` in the next section should help the reader to convince himself that the two implementations in question are actually equivalent.

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

Fixpoint fct X Y (f : X -> Y) (x : term X) { struct x } : term Y :=
  match x with var a => var (f a)
             | app x y => app (x //- f) (y //- f)
             | abs x => abs (x //- (optmap f)) end
where ``x //- f" := (@fct _ _ f x).

Definition shift X (x : term X) : term (option X) := x //- @Some X.

Definition comm X Y (f : X -> term Y) (x : option X) : term (option Y) :=
  match x with Some a => shift (f a) | None => var None end.

Fixpoint slc_bind X Y (f : X -> term Y) (x : term X) { struct x } : term Y :=
  match x with var x => f x
             | app x y => app (x //= f) (y //= f)
             | abs x => abs (x //= comm f) end
where ``x //= f" := (@slc_bind _ _ f x).
```

where `optmap` is the functoriality of `option`:

```
Definition optmap (X Y : Set) (f : X -> Y) (x : option X) : option Y :=
  match x with Some a => Some (f a) | _ => None end.
```

We also define the app_1 operator

```
Definition app1 X (x : term X) : term (option X) := app (shift x)
  (var None).
```

Once the basic definitions are settled, we prove a series of basic lemmas which includes the associativity of substitution, which is the most important ingredient in proving that the λ-calculus is a monad.

```
Lemma slc_bind_bind : forall X Y Z (f : X -> term Y) (g : Y -> term Z) (x : term X),
  x //= f //= g = x //= fun u => f u //= g.
```

Finally, we introduce the $\beta\eta$-relation on λ-terms in two steps. First we define the one-step *beta*-relation

```
Inductive Beta X : term X -> term X -> Prop :=
  | beta_intro : forall (x1 : term (option X)) (x2 : term X),
      Beta (app (abs x1) x2) (x1 //= (default (fun a => var a) x2)).
```

where `default` is defined by

```
Definition default (X Y : Set) (f : X -> Y) (def : Y) (x : option X) : Y :=
  match x with Some a => f a | new => def end.
```

Then we define the $\beta\eta$-relation as follows:

```
Inductive lcr (X : Set) : term X -> term X -> Prop :=
  | lcr_var : forall a : X, var a == var a
  | lcr_app : forall x1 x2 y1 y2 : term X, x1 == x2 -> y1 == y2 -> app x1 y1 == app x2 y2
  | lcr_abs : forall x y : term (option X), x == y -> abs x == abs y
  | lcr_beta : forall x y : term X, Beta x y -> x == y
  | lcr_eta : forall x : term X, abs (app1 x) == x
  | lcr_sym : forall x y : term X, y == x -> x == y
  | lcr_trs : forall x y z : term X, lcr x y -> lcr y z -> lcr x z
where ``x == y" := (@lcr _ x y).
```

which constitutes an equivalence relation that we call `r`:

```
Let r X : Eqv (term X) := Build_Eqv (@lcr X) (@lcr_rfl X) (@lcr_sym X) (@lcr_trs X).
```

Then we prove some compatibility lemmas of r with constructors and other operations. The compatibility with substitution is stated as follows:

```
Lemma lcr_bind : forall X Y (f g : X -> term Y) (x y : term X),
  (forall u, f u == g u) -> x == y -> x //= f == y //= g.
```

A.4. Proof of the main theorem

The fourth and last part summarizes the results proved in the other parts and proves the main theorem. It starts by proving that our definitions of syntactic and semantic lambda calculus provide indeed two monads, denoted `SLC` and `LC`, respectively. The notation *termX*//*r* in the definition of `lc` below denotes the quotient by the equivalence relation r.

```
Definition SLC : Monad :=
  Build_Monad term slc_bind var slc_bind_bind slc_bind_var slc_var_bind.

Definition lc : Set -> Set := fun X => term X // r.

Definition LC : Monad :=
  Build_Monad lc lc_bind lc_var lc_bind_assoc lc_bind_var lc_var_bind.
```

It is now possible to check that the present substitution algorithm is consistent with the one written in Haskell (see 6.3):

```
Lemma slc_bind_abs : forall X Y (f : X -> SLC Y) (x : Derived_Mod SLC X),
  abs x >>= f = abs (x >>>= f).
```

Moreover, we can show the crucial fact that the two transformations abs and app_1 are indeed morphisms of modules:

```
Let abs_hom : Mod_Hom (Derived_Mod LC) LC :=
  Build_Mod_Hom (Derived_Mod LC) LC lc_abs lc_abs_hom.

Let app1_hom : Mod_Hom LC (Derived_Mod LC) :=
  Build_Mod_Hom LC (Derived_Mod LC) lc_app1 lc_app1_hom.
```

Now comes the proof that `LC` is an exponential monad. This is stated in Coq through the following:

```
Definition ELC : ExpMonad := Build_ExpMonad abs_hom app1_hom lc_eta lc_beta.
```

Next comes the construction of the initial morphism which is initially defined as a fixpoint on terms.

```
Variable M : ExpMonad.
Fixpoint iota_fix X (x : term X) { struct x } : M X :=
  match x with var a => unit M a
             | app x y => exp_app M _ (iota_fix x) >>= default (@unit M X) (iota_fix y)
             | abs x => exp_abs M _ (iota_fix x) end.
```

Then we prove that `iota_fix` is compatible with the $\beta\eta$ equivalence relation and thus it yields an operator `iota : forall X, lc X -> M X`. The construction of the initial morphism ends with the verification that it is actually a morphism of monads, and of exponential monads.

```
Let iota_monad : Monad_Hom LC M := Build_Monad_Hom LC M iota iota_bind iota_var.

Let exp_iota : ExpMonad_Hom ELC M :=
  Build_ExpMonad_Hom ELC M iota_monad iota_app1 iota_abs.
```

Finally, we prove that `iota_monad` is unique.

```
Theorem iota_unique : forall (j : ExpMonad_Hom ELC M) X (x : lc X), j X x = exp_iota X x.
```

The Coq terms `ELC`, `iota_monad` and `iota_unique` altogether form the formal proof of the initiality of the monad `LC` in the category of exponential monads.

References

[1] Thorsten Altenkirch, Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types, in: CSL'99: Proceedings of the 13th International Workshop and Eighth Annual Conference of the EACSL on Computer Science Logic, London, UK, Springer, Berlin, 1999, pp. 453–468.

[2] B. Aydemir, A. Bohannon, M. Fairbairn, J. Foster, B. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, S. Zdancewic. Mechanized metatheory for the masses: the POPLmark challenge, in: Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005), 2005.

[3] Jon Beck, Distributive laws, in: Seminar on Triples and Categorical Homology Theory, Mathematics and Statistics, vol. 80, Springer, Berlin, 1969, pp. 119–140.

[4] Nick Benton, John Hughes, Eugenio Moggi. Monads and effects, in: Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9–15, 2000, Advanced Lectures, London, UK, Springer, Berlin, 2002, pp. 42–122.

[5] Richard S. Bird, Ross Paterson, De Bruijn notation as a nested datatype, J. Funct. Program., 9 (1) (1999) 77–91.

[6] Gabriella Böhm, Claudia Menini, Pre-torsors and Galois comodules over mixed distributive laws. ArXiv e-prints, 806, June 2008.

[7] Laurent Chicli, Loic Pottier, Carlos Simpson, Mathematical quotients and quotient types in Coq, in: Herman Geuvers, Freek Wiedijk (Eds.), TYPES, Lecture Notes in Computer Science, vol. 2646, Springer, Berlin, 2002, pp. 95–107.

[8] Michael Ching, Bar constructions for topological operads and the Goodwillie derivatives of the identity, Geom. Topol. 9 (2005) 833–933. (electronic)

[9] The Coq Proof Assistant. Available from: http://coq.inria.fr.

[10] Marcelo Fiore, Gordon Plotkin, Daniele Turi, Abstract syntax and variable binding (extended abstract), in: Proceedings of the 14th Symposium on Logic in Computer Science (Trento, 1999), IEEE Computer Society, Los Alamitos, CA, 1999, pp. 193–202.

[11] Marcelo P. Fiore, On the structure of substitution, in: Invited address for the 22nd Mathematical Foundations of Programming Semantics Conference (MFPS XXII), 2006, DISI, University of Geneva, Italy.

[12] Marcelo P. Fiore, Daniele Turi, Semantics of name and value passing, in: Logic in Computer Science, 2001, pp. 93–104.

[13] Benoit Fresse, Koszul duality of operads and homology of partition posets, in: Homotopy Theory: Relations with Algebraic Geometry, Group Cohomology, and Algebraic K-Theory, Contemporary Mathematics, vol. 346, American Mathematical Society, Providence, RI, 2004, pp. 115–215.

[14] Murdoch Gabbay, Andrew Pitts, A new approach to abstract syntax involving binders, in: Proceedings of the 14th Symposium on Logic in Computer Science (Trento, 1999), IEEE Computer Society, Los Alamitos, CA, 1999, pp. 214–224.

[15] Neil Ghani, Tarmo Uustalu, Explicit substitutions and higher-order syntax, in: MERLIN'03: Proceedings of the 2003 ACM SIGPLAN Workshop on Mechanized Reasoning About Languages with Variable Binding, New York, NY, USA, 2003, ACM Press, New York, pp. 1–7.

[16] John Harrison, The HOL Light theorem prover. Available from: http://www.cl.cam.ac.uk/∼jrh13/hol-light/.

[17] S. Peyton Jones (Ed.), Haskell 98 Language and Libraries, The Revised Report, Cambridge University Press, Cambridge, 2003.
[18] Muriel Livernet, From left modules to algebras over an operad: application to combinatorial Hopf algebras, 2006, math/0607427v1.
[19] Saunders Mac Lane, Categories for the working mathematician, Graduate Texts in Mathematics, second ed., vol. 5, Springer, New York, 1998.
[20] Martin Markl, Models for operads, 1994. hep-th/9411208v1.
[21] Martin Markl, A compactification of the real configuration space as an operadic completion, J. Algebra 215 (1) (1999) 185–204.
[22] Ralph Matthes, Tarmo Uustalu, Substitution in non-wellfounded syntax with variable binding, Theor. Comput. Sci. 327 (1–2) (2004) 155–174.
[23] Eugenio Moggi, Computational lambda-calculus and monads, in: Proceedings Fourth Annual IEEE Symposium on Logic in Computer Science, LICS'89, Pacific Grove, CA, USA, 5–8 June 1989, IEEE Computer Society Press, Washington, DC, 1989, pp. 14–23.
[24] Eugenio Moggi, Notions of computation and monads, Inform. Comput. 93 (1) (1991) 55–92.
[25] V.A. Smirnov, Homotopy theory of coalgebras, Math. USSR Izv. 27 (1986) 575–592.
[26] Ross Street, The formal theory of monads, J. Pure Appl. Algebra 2 (2) (1972) 149–168.
[27] Miki Tanaka, John Power, A unified category-theoretic formulation of typed binding signatures, in: MERLIN'05: Proceedings of the Third ACM SIGPLAN Workshop on Mechanized Reasoning About Languages with Variable Binding, New York, NY, USA, ACM Press, New York, 2005, pp. 13–24.
[28] Miki Tanaka, John Power, Pseudo-distributive laws and axiomatics for variable binding, Higher Order Symbol. Comput. 19 (2–3) (2006) 305–337.
[29] Norbert Völker, Hol2p – a system of classical higher order logic with second order polymorphism, in: Klaus Schneider, Jens Brandt (Eds.), TPHOLs, Lecture Notes in Computer Science, vol. 4732, Springer, Berlin, 2007, pp. 334–351.
[30] Julianna Zsidó, Le lambda calcul vu comme monade initiale, Master's thesis, Université de Nice – Laboratoire J. A. Dieudonné, 2005/06, Mémoire de Recherche – master 2.