

MODULES OVER MONADS, MONADIC SYNTAX AND THE CATEGORY OF UNTYPED LAMBDA-CALCULI

ANDRÉ HIRSCHOWITZ AND MARCO MAGGESI

ABSTRACT. We define a notion of module over a monad and use it to propose a new definition (or semantics) for abstract syntax (with binding constructions). Using our notion of module, we build a category of *exponential* monads, which can be understood as the category of lambda-calculi, and prove that it has an initial object (the pure untyped lambda-calculus). Our definitions and results are formalized in the proof assistant Coq.

1. INTRODUCTION

Our monads are monads over sets. The algebraic structure of monad reflects the behaviour of substitution (better than 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. In the present work, we introduce the notion of module over a monad, together with a construction of the so-called derivative of a module, and we propose a prominent role for these notions in the understanding of syntax and semantics.

It is a standard practice in computer science to define languages through a set of formulas, called the syntax, and a *reduction* relation on it, called the semantics. 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. This view is inadequate for those languages which allow *binding* constructions. We quote from the Poplmark Challenge [Pop]: “Representing binders has been recognized as crucial by the theorem proving community, and many different solutions to this problem have been proposed. In our

Date: December 10, 2005.

(still limited) experience, non emerge as clear winners.” For one such solution and a fair account of earlier ones, see e.g. [MM02]. For some more recent solutions, see ([Hof99, FPT99, GP99]). Based on modules over monads, we propose what we hope to become the “winning” abstract notion of syntax, which we call monadic syntax. It goes as follows.

When we view the formulas of a syntax as depending on a set of variables, we get a monad. We define the derivative M' of a monad M by the formula $M'(X) := M(X^*)$, where X^* is obtained from X by adding one element. It is a monad again, but this is too much structure. Our notion of module over a monad is tailored in order to reflect how formulas in M can be plugged into formulas in M' . This notion opens new room for binding constructions: for instance we now can see the lambda-calculus as a monad \mathbf{LC} equipped with two constructions $\mathbf{abs} : \mathbf{LC}' \rightarrow \mathbf{LC}$ and $\mathbf{app} : \mathbf{LC} \times \mathbf{LC} \rightarrow \mathbf{LC}$ (here \rightarrow and \times refer to natural notions of morphism and product of modules, and we use the fact that a monad can be viewed as a module over itself). We say that $(\mathbf{abs}, \mathbf{app})$ is a representation in \mathbf{LC} of the (binding) signature $(1; 0), (0, 0; 0)$. This leads to a very simple extension of the algebraic point of view to higher-order syntax: given a 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. We have already checked for instance that this principle allows a fair treatment of the first part of the Poplmark challenge (see [Pop])

In the present paper, we explore our new representation of binders on the emblematic example of the lambda-calculus, by providing a formal (and original) answer to the question of Dana Scott [Sco80]: “I am trying to find out where lambda-calculus *should* come from”.

The algebraic structure of (untyped) lambda-calculus reflects the behaviour of application and abstraction. The operation of abstraction deeply involves so-called variables. This is probably the reason why, although some abstract characteristic properties have certainly been given earlier for lambda-calculi, see e.g. [LS88], none has been accepted as a standard definition. We make a new try by defining an untyped lambda-calculus to be a monad equipped with an isomorphism of modules to its derivative. The reverse isomorphism is the abstraction, while the isomorphism itself is an avatar of the application. The familiar β and η rules just express that these two morphisms are inverse of each other. There is a natural category of untyped lambda-calculi, where our main result asserts the existence of an initial object: the pure (untyped) lambda-calculus.

We are among those mathematicians who believe that time is “almost” ripe for computer-checked proofs. Accordingly we propose (only)

a computer-checked proof of our theorem. Indeed, while the ideas of the proof are just the natural ones, nobody should be interested in the details, which fortunately the Coq Proof Assistant [Coq] was able to check. The way this formal proof will be eventually accounted for has not yet been arranged with our editor.

2. MODULES OVER MONADS

The monads we consider here are monads over sets (c.f. e.g., [ML98]).

Example 2.1 (Lambda-Calculus). We see the lambda calculus as a monad LC , where $\text{LC}(X)$ is the set of lambda-terms (modulo $\alpha\beta\eta$ -conversion) built on free variables taken in X .

Definition 2.2 (Derived monad). Being given a monad R , we define its derivative R' to be the functor $X \mapsto R(X^*)$, where X^* is obtained from X by adding one element, called ∞_X . It is easily checked how R' is again a monad.

Definition 2.3 (Module over a monad). We define a module over a monad R to be a functor M from sets to sets equipped with a natural transformation $M \circ R \rightarrow M$, called substitution, and satisfying the usual associativity condition identifying the two natural transformations from $M \circ R \circ R$ to M .

Example 2.4. We can see our monad R as a module over itself, which we call the tautological module.

Example 2.5. Other trivial R -modules are the initial and final functors \emptyset and $*$.

Definition 2.6 (Derived module). We define the derivation of a R -module M to be the functor $X \mapsto M(X^*)$. It is easily checked how this is a R -module again. We denote by M' this derivative module. We can iterate the construction and denote by $M^{(n)}$ the n -th derivative.

Definition 2.7. We say that a natural transformation $F : M \rightarrow N$ is a module morphism if it is compatible with substitution.

Example 2.8. We easily check that the natural inclusion of a module into its derivative is a module morphism.

Example 2.9. Note that there are two natural inclusions of the derivative M' into the second derivative M'' .

Definition 2.10 (Category of R -modules). We check easily that module morphisms among R -modules yield a subcategory of the functor category.

Definition 2.11 (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.

Example 2.12. The final module $*$ is the product of the empty family.

Example 2.13. Given a R -module M , we have a natural “evaluation” morphism

$$\mathbf{app} : M' \times R \longrightarrow M.$$

Proposition 2.14. *Derivation yields a cartesian functor from R -modules to R -modules.*

Definition 2.15 (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 get

$$\mathbf{comp}(X) : f^*M(A(X)) \longrightarrow f^*M(X)$$

by composing the structural morphism $f^* : M(A(X)) \rightarrow M(B(X))$ with the structural morphism $M(B(X)) \rightarrow M(X)$.

Lemma 2.16. *The pull-back of a module is a module.*

Hint. Add $M(B(B(X)))$ in the middle of the diagram, which then splits into three pieces, whose commutativity is given respectively by the fact that M is a B -module, the fact that the map from $M(B(-))$ to $M(-)$ is functorial, and the fact that f is a morphism. \square

Definition 2.17 (Pull-back (2)). 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 modules, then so is the natural transformation $f^*g : f^*M \rightarrow f^*N$.

Proposition 2.18. *Pull-back commutes with products and with derivation.*

Proposition 2.19. *Any morphism of monads $f : A \rightarrow B$ yields a morphism of A -modules, still denoted f , from A to f^*B .*

3. SYNTAX

Definition 3.1 (Arity). An *arity* is a list of integers [FPT99]. In the arity of a construction, the length of the list represents the number of arguments, while the integers say the number of variables on the corresponding argument, which are bound by the construction. We denote by \mathbb{N}^* the set of arities.

Example 3.2. The arity of the **app** operation of the lambda-calculus is $(0, 0)$, while the arity of the **abs** construction is (1) .

Definition 3.3 (Signatures). We define a (binding) signature $\Sigma = (O, a)$ to be a family of arities $a : I \rightarrow \mathbb{N}^*$.

Definition 3.4 (Representation of an arity in a monad). Given a monad R , we define a representation of the arity $a = (a_1, \dots, a_n)$ in R to be a module morphism

$$r : \prod R^{(a_i)} \longrightarrow R.$$

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

Example 3.5. 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.

Definition 3.6 (Representation of a signature). Given a signature $\Sigma = (I, a)$ and a monad R , we denote by $R^{(a)}$ the product of (possibly higher) derivative R -modules: $\prod_{i \in I} R^{a_i}$. A representation of Σ in R , consists of, for each i in I , a representation of a_i in R .

Definition 3.7 (The category of representations). Given a signature $\Sigma = (I, a)$ we build the category Mon^Σ of representations of Σ as follows. Its objects are monads equipped with a representation of Σ . A morphism from (R, r) to (S, s) is a morphism f from R to S compatible with the representations in the sense that, for each i in I , the following diagram commutes:

$$(1) \quad \begin{array}{ccc} R^{(a)} & \longrightarrow & R \\ \downarrow & & \downarrow \\ f^* S^{(a)} & \longrightarrow & f^* S \end{array}$$

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 3.8. *These morphisms, together with the obvious composition, turn Mon^Σ into a category.*

Theorem 3.9. *For any signature $\Sigma = (I, a)$, the category Mon^Σ has an initial object, which we call the syntactic monad generated by Σ , and denote by $\hat{\Sigma}$. Furthermore, this representation $\hat{\Sigma} = (S, \rho)$ satisfies the following recursion principle:*

given a family of sets $E(X, s)$ indexed by $\prod_{X \in \text{Set}} S(X)$, given for each $i \in I$, for every set X and for each element $x = (x_1, \text{dots}, x_k) \in S^{(a_i)}(X)$ (where k is the length of our arity $a_i = (\alpha_1, \text{dots}, \alpha_k)$), an application $r_{i, X, x}$ from $\prod_{1 \leq j \leq k} E(X^{(\alpha_j)}, x_j)$ to $E(X, \rho(X, x))$, there exists a unique section f of the family E with the property that for any set X , for any $i \in I$ and each element $x = (x_1, \text{dots}, x_k) \in S^{(a_i)}(X)$, the following recursion relation holds:

$$f(X, \rho(a_i)(x)) = r_{i, X, x}(f(X^{(\alpha_1)}, x_1), \dots, f(X^{(\alpha_k)}, x_k)).$$

4. MONADIC GROUPS

We can easily rephrase the definition of monads for first-order algebraic structures in terms of modules. We sketch here the case of groups, in order to open the track for lambda-calculi.

Definition 4.1. A monad R is said to be a monadic group when its tautological module is equipped with a structure of group object in the category of R -modules.

Example 4.2. We define the free monadic group F : $F(X)$ is the free group generated by X . The monadic and group structures on \hat{G} are the obvious ones.

Example 4.3. Given a group G , we define a monadic group \hat{G} by $\hat{G}(X) := G \amalg F(X)$. The monadic and group structures on \hat{G} are the obvious ones.

Example 4.4. Given a group G , we define another monadic group \tilde{G} by $\tilde{G}(X) := G^{G^X}$. The group structure on \tilde{G} is the obvious ones.

Definition 4.5. We define a morphism between two monadic groups to be a morphism between the underlying monads, which is compatible with the group structure. Monadic groups and these morphisms form the category MG of monadic groups.

Proposition 4.6. *The free monadic group is an initial object in MG .*

5. LAMBDA-CALCULI

Now we turn to our higher-order application.

Definition 5.1. We call (untyped) lambda-calculus a monad equipped with an isomorphism **app** (with inverse called **abs**) of its tautological module with its derivative.

Example 5.2. The **abs** construction turns the monad **LC** into a lambda-calculus in the above sense. The inverse of **abs** is the avatar **app**₁ of **app** which sends a term M in $\mathbf{LC}(X)$ to **app**($M, \mathbf{var}(\infty_X)$) in $\mathbf{LC}(X^*)$. The β and η rules express precisely that **abs** and **app**₁ are inverse of each other.

Example 5.3. [Sco80] Let us consider a cartesian closed category C with all products, equipped with a reflexive object D , that is to say D is equipped with an isomorphism **abs** : $(D \rightarrow D) \rightarrow D$. We build a monad M_D by setting:

$$M_D(X) := \text{Hom}(D^X \rightarrow D).$$

There is a tautological morphism from D^X to $D^{M_D(X)}$ which, by composition, yields the substitution morphism from $\text{Hom}(D^{M_D(X)}, D)$ to $\text{Hom}(D^X \rightarrow D)$. The **abs** isomorphism is obtained by composing **abs**_{*} : $\text{Hom}(D^X, D \rightarrow D) \rightarrow \text{Hom}(D^X, D)$ with curryfication: $\text{Hom}(D^{X^*}, D) = \text{Hom}(D^X \times D, D) \rightarrow \text{Hom}(D^X, D \rightarrow D)$.

We take for morphisms among lambda-calculi those morphisms among the underlying monads which are compatible with the given isomorphisms. This yields a category of lambda-calculi where lives our main

Theorem 5.4. *The lambda-calculus LC is an initial object in the category of lambda-calculi.*

6. PROOFS

For the moment, our computer proofs can be downloaded at ??? and run on the Coq CVS version.

REFERENCES

- [BCP03] Gilles Barthe, Venanzio Capretta, and Olivier Pons, *Setoids in type theory*, J. Funct. Programming **13** (2003), no. 2, 261–293, Special issue on “Logical frameworks and metalanguages”. MR 1 985 376
- [BHM00] Nick Benton, John Hughes, and Eugenio Moggi, *Monads and effects*, 2000.
- [Bor94] Francis Borceux, *Handbook of categorical algebra. 2*, Encyclopedia of Mathematics and its Applications, vol. 51, Cambridge University Press, Cambridge, 1994, Categories and structures. MR **96g**:18001b
- [Coq] *The Coq Proof Assistant*, <http://coq.inria.fr>.
- [DK80] W. G. Dwyer and D. M. Kan, *Simplicial localizations of categories*, J. Pure Appl. Algebra **17** (1980), no. 3, 267–284. MR **81h**:55018
- [FPT99] Marcelo Fiore, Gordon Plotkin, and Daniele Turi, *Abstract syntax and variable binding (extended abstract)*, 14th Symposium on Logic in Computer Science (Trento, 1999), IEEE Computer Soc., Los Alamitos, CA, 1999, pp. 193–202. MR 1 943 414
- [GP99] Murdoch Gabbay and Andrew Pitts, *A new approach to abstract syntax involving binders*, 14th Symposium on Logic in Computer Science (Trento, 1999), IEEE Computer Soc., Los Alamitos, CA, 1999, pp. 214–224. MR 1 943 416
- [GTWW77] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright, *Initial algebra semantics and continuous algebras*, J. Assoc. Comput. Mach. **24** (1977), no. 1, 68–95. MR 58 #25071
- [GZ67] P. Gabriel and M. Zisman, *Calculus of fractions and homotopy theory*, Ergebnisse der Mathematik und ihrer Grenzgebiete, Band 35, Springer-Verlag New York, Inc., New York, 1967. MR 35 #1019
- [Hof99] Martin Hofmann, *Semantical analysis of higher-order abstract syntax*, 14th Symposium on Logic in Computer Science (Trento, 1999), IEEE Computer Soc., Los Alamitos, CA, 1999, pp. 204–213. MR 1 943 415
- [Hov99] Mark Hovey, *Model categories*, Mathematical Surveys and Monographs, vol. 63, American Mathematical Society, Providence, RI, 1999. MR **99h**:55031
- [LS88] J. Lambek and P. J. Scott, *Introduction to higher order categorical logic*, Cambridge Studies in Advanced Mathematics, vol. 7, Cambridge University Press, Cambridge, 1988, Reprint of the 1986 original. MR **89a**:03024
- [McB] Conor McBride, *The derivative of a regular type is its type of one-hole contexts*, <http://www.dur.ac.uk/c.t.mcbride/diff.ps>.
- [ML98] Saunders Mac Lane, *Categories for the working mathematician*, second ed., Graduate Texts in Mathematics, vol. 5, Springer-Verlag, New York, 1998. MR **2001j**:18001
- [MM02] Raymond C. McDowell and Dale A. Miller, *Reasoning with higher-order abstract syntax in a logical framework*, ACM Trans. Comput. Log. **3** (2002), no. 1, 1–56 (electronic). MR **2004e**:68066

- [Mog89] Eugenio Moggi, *Computational lambda-calculus and monads*, Proceedings 4th Annual IEEE Symp. 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.
- [Mog91] ———, *Notions of computation and monads*, Information and Computation **93** (1991), no. 1, 55–92.
- [Pop]
- [Qui67] Daniel G. Quillen, *Homotopical algebra*, Lecture Notes in Mathematics, No. 43, Springer-Verlag, Berlin, 1967. MR 36 #6480
- [Sco71] Dana S. Scott, *The lattice of flow diagrams*, Symposium on Semantics of Algorithmic Languages, Lecture Notes in Mathematics, Vol. 188. Springer, Berlin, 1971, pp. 311–366. MR 43 #4577
- [Sco80] ———, *Relating theories of the λ -calculus*, To H.B. Curry: essays in Combinatory Logic, lambda calculus and Formalisms. (R. Hindley and J. Seldin, eds.), Academic Press, 1980, 1980.
- [Sim] Carlos T. Simpson, *Computer theorem proving in math*, arXiv math.HO/0311260.
- [Wei94] Charles A. Weibel, *An introduction to homological algebra*, Cambridge Studies in Advanced Mathematics, vol. 38, Cambridge University Press, Cambridge, 1994. MR **95f**:18001