Categorical semantics of programming languages (in Coq)

Benedikt Ahrens

May 2010

Formalized mathematics have met an increasing interest in the last years. They give a way to build digital searchable libraries of mathematics, and computer-verified proofs allow for a level of trust in the theory which the pen and paper versions cannot supply.

Our first goal is to build a library of category theory [2] in the proof assistent Coq [1]. A comprehensive library surely demands more than one person's work, so our library can only provide a starting point for further developments.

Secondly, we want to apply our library to formalize a specific theory of programming languages. In [4] Hirschowitz and Maggesi show that the untyped lambda calculus is the initial object in a suitable *category of representations* of the *signature* given by abstraction and application. We want to extend this theory by adding *semantical* conditions to the representations, working in categories "with structure".

1 Categories formalized

The formalization of category theory has been said to be "notoriously hard to formalize " [3], so the absence of a comprehensive library of category theory in any theorem prover is not surprising. The *ConCaT* contribution [5], up to this point the most evolved development of category theory in Coq, comes with a serious drawback: the setoid record contains the carrier as a field. As a consequence some of the types defined after the setoid type cannot be used as a carrier of a setoid, since their type is of a higher level in the type hierarchy as is allowed for a setoid carrier. As an example, in order to define the functor category, one is obliged to define a second type of setoids, and after that a second type of categories using the new setoids.

The setoid typeclass recently added to Coq [7] however avoids universe clashes by being parametrized by the carrier, instead of the carrier being a field of the setoid record. We take advantage of this technology and define a category as a type of objects and, for each couple of objects, a type of morphisms for which we'll define our own equality - a setoid.

Typeclasses play a central role in our formalization in general. Their use allows overloading of common operators and equality. Giving two examples, equality between morphisms for *any* category will be denoted by $==$, and the terminal

object of any category - which has such - is called Term. When using these overloaded identifiers, the typeclass mechanism will find the right instance of the corresponding class itself, without any intervention by the user.

The appendix contains a list of objects of category theory and their corresponding name in our library. In the following each definition we give is accompanied by the corresponding name in the COQ library, given in `typewriter font`.

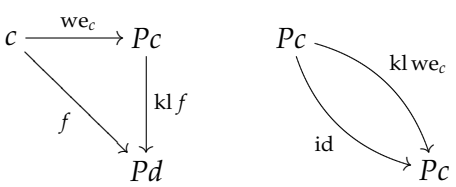
2 Monads & Modules

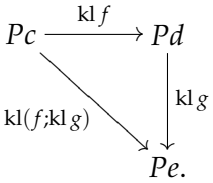
Our definition of monads is the one known in the Haskell community, which we'll briefly recall.

2.1 Definition `[Monad]`**:** A *monad* P over a category $\mathcal{C}$ is given by

- a map $P\colon \mathcal{C} \to \mathcal{C}$ on the objects of $\mathcal{C}$ (observe the abuse of notation) ,
- for each object c of $\mathcal{C}$, a morphism $\mathrm{we}_c \in \mathrm{Hom}(c, Pc)$ and
- for all objects c, d of $\mathcal{C}$, a map $\mathrm{kl}_{c,d}\colon \mathrm{Hom}(c, Pd) \to \mathrm{Hom}(Pc, Pd)$,

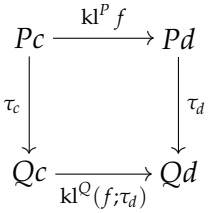
such that the following diagrams commute for all suitable morphisms f and g:

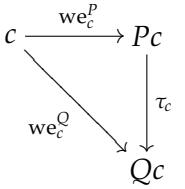




For two monads P and Q over the same category $\mathcal{C}$, we can define a *morphism of monads*:

2.2 Definition `[Monad_Hom]`**:** A *morphism of monads* P and Q is given by a collection of morphisms $\tau_c \in \mathrm{Hom}(Pc, Qc)$ such that the following diagrams commute for all suitable morphisms f:



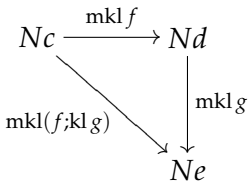


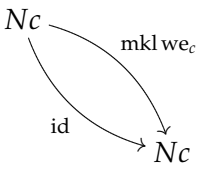
Given a monad P over $\mathcal{C}$, we can define *modules over* P as follows:

2.3 Definition `[Module]`**:** Let $\mathcal{D}$ be a category. A *module* N *over* P *with codomain* $\mathcal{D}$ is given by

- a map $N\colon \mathcal{C} \to \mathcal{D}$ on the objects of the categories involved and
- for all objects c, d of $\mathcal{C}$ a map $\mathrm{mkl}_{c,d}\colon \mathrm{Hom}(c, Pd) \to \mathrm{Hom}(Nc, Nd)$

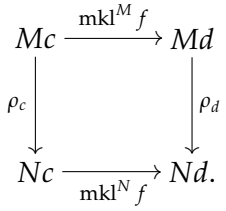
such that the following diagrams commute for all suitable morphisms f and g:





2.4 Definition `[Mod_Hom]`**:** Let M and N be two modules over P with codomain $\mathcal{D}$. A *morphism of modules* is given by a collection of morphisms $\rho_c \in \mathrm{Hom}(Mc, Nc)$ such that for all morphisms $f \in \mathrm{Hom}(c, Pd)$ the following diagram

commutes:



The modules on P with codomain $\mathcal{D}$ and morphisms between them form a category which will be denoted by $\mathrm{Mod}_{\mathcal{D}}^{P}$ `[MOD]`.

2.1 Examples

The following examples play a central role in our theory.

Tautological Module [`Taut_Mod`] Every monad P over $\mathcal{C}$ can be viewed as a module (also denoted by P) over itself, i. e. in $\mathrm{Mod}_{\mathcal{C}}^{P}$.

Constant module, Terminal module [`Const_Mod, MOD_Terminal`] For any object $d \in \mathcal{D}$ the constant map $T_d \colon \mathcal{C} \to \mathcal{D}$, $c \mapsto d$ can be given the structure of an P-module for any monad P. In particular, if $\mathcal{D}$ has a terminal object $1_{\mathcal{D}}$, then the constant module $c \mapsto 1_{\mathcal{D}}$ is terminal in $\mathrm{Mod}_{\mathcal{D}}^{P}$.

Pullback module [`PbMod, PBMOD`] Given $h \colon P \to Q$ a morphism of monads over $\mathcal{C}$ and $M \in \mathrm{Mod}_{\mathcal{D}}^{Q}$ a Q-module, we can define a P-module $\mathrm{Pb}\, M$ by setting

$$\mathrm{mkl}_{c,d}^{\mathrm{Pb}} f := \mathrm{mkl}^{M}(f; h_d).$$

This module is called the *pullback (module) h^*M of M along h*.

Ind. mod. morphism [`PbMod_ind_Hom`] With the same notation as in the preceding example, the monad morphism h induces a morphism of modules $h \colon P \to h^*Q$ in $\mathrm{Mod}_{\mathcal{C}}^{P}$.

Products [`Prod_Mod, MOD_PROD`] Suppose the category $\mathcal{D}$ is equipped with a product. Let $M, N \in \mathrm{Mod}_{\mathcal{D}}^{P}$ be modules with codomain $\mathcal{D}$. Then the map $c \mapsto Mc \times Nc$ can be extended to a module called the *product of M and N*. This construction can again be extended to a product on $\mathrm{Mod}_{\mathcal{D}}^{P}$.

2.2 Derivation

We denote by *derivation* a monad which intuitively adds a distinct variable to a set of variables. This construction varies from category to category, and to our knowledge no universal characterization has been given yet. We'll define this functor for the few categories we're interested in.

For the category **Type** the derivation is given by taking the coproduct with the terminal element: $V \mapsto V^* := V \amalg \{*\}$.This is known as the *Maybe* monad in Haskell or the *option* type constructor in Ocaml. The construction extends to categories of types "with structure", for example the category of preorders **PO**.

Let T be a discrete category (a type) and $\mathcal{C}$ one of **Type** or **PO**. For each object u of T an object $A \in [T, \mathcal{C}]$ of the functor category $[T, \mathcal{C}]$ can be derived with respect to u by setting

$$\partial_u At := \begin{cases} At^*, & \text{if } t = u \\ At & \text{otherwise.} \end{cases}$$

This yields a monad $\partial_u A$ (see `opt_T_monad` for indexed types and

`opt_TP_monad` for indexed preorders).

Derived module Given a monad P over $[T, \mathcal{C}]$ and $M \in \mathrm{Mod}^P_{[T,\mathcal{C}]}$, we can define the *derived module* w. r. t. $u \in T$ by setting

$$\partial_u MA := M(\partial_u A)$$

and

$$\mathrm{mkl}^{\partial_u M} f := \mathrm{mkl}^M(\mathrm{def}_u((f; \mathrm{lift}\,\mathrm{we}), *_u)),$$

where

$$\mathrm{def}_u(g, b)t := \begin{cases} [gt, (* \mapsto b)], & \text{if } t = u \\ gt & \text{otherwise.} \end{cases}$$

It turns out that derivation is an endofunctor over $\mathrm{Mod}^P_{[T,\mathcal{C}]}$ (see `DER_MOD` for **PO**, `ITDER_MOD` for **Type**).

2.3 Fibres

For a module $M \in \mathrm{Mod}^P_{[T,\mathcal{C}]}$ and $u \in T$ we can define the *fibre module* M_u by setting

$$M_u A := MA(u) \quad \in \mathcal{C}$$

and

$$\mathrm{mkl}^{M_u} f := (\mathrm{mkl}^M f)u.$$

This yields a module $M_u \in \mathrm{Mod}^P_{\mathcal{C}}$. The construction extends to a functor (see `FIB_MOD` and `ITFIB_MOD`).

3 Signatures & Representations

We extend the notion of *signature* given in [8, Ch. 6] by adding propositions to the possible arities. This extension captures the fact that we'd like to impose conditions on the terms of the syntax to construct. The propositional arities hence add *semantics* to the syntax.

The same syntax can be represented in different categories, which allow for different propositional arities. More precisely assuming that we work on a category $[T, \mathcal{C}]$ the propositional arities make use of the structure with which the objects of $\mathcal{C}$ are equipped.

In the following we will discuss the initiality of the programming language **PCF** (see e. g. [6] for an explicit description). In addition to the purely syntactic description we want embed the beta reduction preorder into our categorical characterization. We choose hence $\mathcal{C}$ to be the category **PO** of preorders. By $\mathcal{T}$ we will denote the types of **PCF**:

$$\mathcal{T} ::= \mathrm{Bool} \mid \mathrm{Nat} \mid \mathcal{T} \Rightarrow \mathcal{T}.$$

3.1 A failed attempt

We then gave the following definition, which turned out to be not sufficiently restrictive in order to prove initiality of **PCF**:

3.1 Definition: A *representation* of **PCF** *with beta reduction relation* is given by a monad P over $[\mathcal{T}, \mathbf{PO}]$ and a lot of (families of) morphisms of modules:

- $\mathrm{App}_{s,t}$: $P_{s\Rightarrow t} \times P_s \to P_t$
- $\mathrm{Abs}_{s,t}$: $(\partial_s P)_t \to P_{s\Rightarrow t}$
- Rec_t: $P_{t\Rightarrow t} \to P_t$
- for every constant c of type t a module morphism

  $$c : * \to P_t$$

  - the module $*$ being the terminal module in $\mathrm{Mod}^P_{\mathbf{PO}}$ -

where s and t range over T. They are subject to the following conditions for any $V \in [\mathcal{T}, \mathbf{PO}]$ and suitable x, y, z:

- $\mathrm{App}(\mathrm{Abs}\, x, y) < x[*_s := y]$
- $x < y \Rightarrow \mathrm{App}(x, z) < \mathrm{App}(y, z)$
- $y < z \Rightarrow \mathrm{App}(x, y) < \mathrm{App}(x, z)$
- $x < y \Rightarrow \mathrm{Abs}\, x < \mathrm{Abs}\, y$
- $x < y \Rightarrow \mathrm{Rec}\, x < \mathrm{Rec}\, y$.

3.2 Definition: A *morphism of representations* from P to R is given by a monad morphism $\rho\colon P \to R$ between the underlying monads such that for each module morphism of the representations we get a commutative diagram in the category $\mathrm{Mod}^{P}_{\mathbf{PO}}$, e. g. for App:

$$\begin{array}{ccc} P_{s\Rightarrow t} \times P_s & \xrightarrow{\mathrm{App}^P} & P_t \\ \downarrow{\scriptstyle \rho_{s\Rightarrow t}\times\rho_s} & & \downarrow{\scriptstyle \rho_t} \\ \rho^* R_{s\Rightarrow t} \times \rho^* R_s & \xrightarrow{\rho^*\,\mathrm{App}^R} & \rho^* R_t. \end{array}$$

The formal definition can be found in [8].

We can upgrade representations and their morphisms to a category, which we'll call $\mathrm{Rep}_{\mathbf{PO}}(\mathbf{PCF})$.

In our formalisation we show that **PCF** is an object in this category of representations. We could not, however, give a morphism to another arbitrary representation, say R. When trying to prove monotonicity of the function from **PCF** to R, we ran into a circular dependency of lemmata, which we were unable to escape from.

3.2 A new try

We hence chose to consider only those monads over $[\mathcal{T}, \mathbf{PO}]$ which "come from" a monad over $[\mathcal{T}, \mathbf{Type}]$. In practice we consider couples of *parallel monads* (R, S), where $R \in [\mathcal{T}, \mathbf{PO}]$ and $S \in [\mathcal{T}, \mathbf{Type}]$, verifying compatibility conditions involving the forgetful functor from (families of) preorders to (families of) types. This work is still in its beginnings, however, and we do not yet know if it will lead to the result we'd like to have.

References

[1] The Coq Proof Assistant. http://coq.inria.fr.

[2] Benedikt Ahrens. Categorical Semantics in Coq. http://math.unice.fr/~ahrens.

[3] John Harrison. Formalized Mathematics. Technical report, 1996.

[4] A. Hirschowitz and M. Maggesi. Modules over Monads and Linearity. *Theorem Proving in Higher Order Logics*, pages 278–293, 2008.

[5] Gérard Huet and Amokrane Saïbi. Constructive category theory. In *In Proceedings of the Joint CLICS-TYPES Workshop on Categories and Type Theory, Goteborg*. MIT Press, 1998.

[6] J. M. E. Hyland and C.-H. Ong. On full abstraction for pcf. *Information and Computation*, 163:285–408, 2000.

[7] M. Sozeau and N. Oury. First-class type classes. *Theorem Proving in Higher Order Logics*, pages 278–293, 2008.

[8] Julianna Zsidó. *Typed Abstract Syntax*. PhD thesis, University of Nice, France, 2010.

A Library of formalized category theory

MATHEMATICAL OBJECT	IDENTIFIER IN THE LIBRARY[2]
categories, product (product functor)	`PROD (Prod_functor)`
category	`Category`
category Delta	`DELTA`
category of categories	`Cat_CAT`
category of indexed types	`ITYPE`
category of haskell modules	`MOD`
category of haskell monads	`MONAD`
category of sets	`SET`
category, small	`SMALLCAT`
category of types	`TYPE`
coproduct	`Cat_Coprod`
discrete category	`DISCRETE`
functor	`Functor`
functor category	`FunctCat`
functor from haskell monad	`MFunc`
initial object	`Initial`
module (haskell style)	`Module`
module, morphisms	`Module_Hom`
module, pullback (functorial)	`PbMod, PBMOD`
module, tautological	`Taut_Mod`
monad, w. eta & mu	`MonaD`
monad, haskell style	`Monad`
monad (haskell style), morphism of	`Monad_Hom`
monads, equivalences	`Monad_h_from_Monad_struct, Monad_from_Monad_h_struct`
monad, morphisms	`Monad_Hom`
monoidal category	`mon_cat`
natural transformation	`NT`
natural transformation, hor. comp.	`hcompNT`
strict monoidal category	`smon_cat`
subcategory	`SubCat`
terminal object	`Terminal`

The library currently holds around 350 definitions, 250 lemmata and ca. 150 instances (i. e. examples). See [2] for a complete index.