Category theory on the computer

Benedikt Ahrens
Laboratoire J. A. Dieudonné,
Université de Nice
ahrens@unice.fr

Abstract

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 goal is to build a library of category theory [2] in the proof assistent Coq *[1]. It makes extensive use of modern features of* Coq*, such as type classes and generalized rewriting.*

1. Introduction

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. Two features which were recently added to our favourite prover Coq however, help significantly to reduce the problems that were encountered on earlier attempts. We present these features and explain their use and benefits for our library of category theory.

2. Generalized Equality

The main difficulty in formalizing category theory is the notion of *equality*. The objects of interest in category theory are commutative diagrams of morphisms, i. e. equality of (compositions of) morphisms. The built-in Coq equality – defined for any type – is defined as an inductive type (see listing 1) and is logically equivalent to Leibniz equality.

```
----- Listing 1 : Coq's equality ------------------
  Inductive eq (A : Type) (x : A) : A -> Prop :=
    eq_refl : x = x.
```

It turns out that for a mathematician's purposes this equality is too restrictive, when it comes to function types (where we'd like to have an extensional equality) or dependent pairs, where an object is accompanied by additional data and properties. It is hence more convenient to equip our arrow types with a *custom equality*[5]. Any equivalence relation may be chosen as such. A type equipped with an equivalence relation is called a *setoid*. We'll write `x==y` for `x` and `y` being related.
A Coq function from morphisms to morphisms should then respect the equalities on the domain and codomain. The Coq keyword for this property is `Proper`. As an example, in the definition of category (see listing 2) the composition or morphisms is a setoid morphism.
After having proven that a function is a morphism of setoids we can rewrite a setoid equality in the argument of this function, just as with the normal Coq equality.
In listing 3 the `rewrite` tactic will automatically check that composition is indeed declared to be a morphism of setoids, and will fail if we omit the field `comp_oid` in the declaration of the category type class.

```
----- Listing 2 : The typeclass of categories -----
 Class Cat (obj:Type)(mor: obj -> obj -> Type) := {
  mor_oid:> forall a b, Setoid (mor a b);
  id: forall a, mor a a;
  comp: forall {a b c},
    mor a b -> mor b c -> mor a c;
  comp_oid:> forall a b c,
    Proper (equiv ==> equiv ==> equiv) (@comp a b c);
  id_r: forall a b (f: mor a b),
    comp f (id b) == f;
  id_l: forall a b (f: mor a b),
    comp (id a) f == f;
  Assoc: forall a b c d (f: mor a b)
            (g:mor b c) (h: mor c d),
    comp (comp f g) h == comp f (comp g h)
}.
Notation "x ';;' y" := (comp x y).
```

```
----- Listing 3 : generalized rewriting -----------
  ...
  H : f == g
  ============================
   f;; h == g;; h

Coq < rewrite H.
  ...
  H : f == g
  ============================
   g;; h == g;; h
```

3. Type classes

A *type class* [4] is a set of types and propositions specified for one or more parametric types. For given type parameters we can then give an *instance* of the type class by giving for each type a term of that type, and for each proposition a proof of that proposition when instantiated with the functions and types in question. Type classes allow for consistent naming conventions, since functions on different types but with same semantics can be given the same name – a technique called *overloading*.

```
----- Listing 4 : the type class of groups --------
Class group (A:Type) := {
  mult : A -> A -> A;
  I : A;
  neutral_r : forall a, mult a I = a ;
  neutral_l : forall a, mult I a = a }.

Notation "a 'o' b" := (mult a b) (at level 60).

Program Instance Z_group : group Z := {
  mult a b := a + b;
  I := 0  }.
```

In listing 4 we define *groups*, an algebraic structure, as a type class. We then declare one instance of this class by giving $\mathbb{Z}$ the structure of a group, where the group action is given by addition and the neutral element by zero. The instance declaration is not finished at that point, however. When entering the above command into the system, Coq will generate two subgoals, one for each of the neutrality conditions. It will instantiate the general neutral element I with the zero of $\mathbb{Z}$, and the goals to solve will look like in listing 5.

```
----- Listing 5 : goals for instance Z ------------
  a : Z
  ============================
   a + 0 = a
```

They can then be worked on using the interactive proving mode. After finishing the last obligation with `Qed`, the instance is declared and stored in a data base. For a type of lists over a type, list concatenation can also be made an instance of the group type class. We can then refer to both operations – addition on $\mathbb{Z}$ and concatenation on lists by `mult x y` and the declared notation `x o y` for integers resp. lists `x` and `y`.
In our formalization type classes play a central role in that every object is first defined as a type class. We have already given the example of categories, allowing for overloading of the composition. Another nice example is that of initial and terminal object as given in listing 6.

```
----- Listing 6 : type class of initial object ----
Class Initial (C : Category) := {
  I : C ;
  Imor: forall a : C, I ---> a;
  Imor_unique: forall b (f : I ---> b), f == Imor b
}.
```

Finally setoids as presented in section 2 are implemented as type classes as well. Here the relation symbol is overloaded with the infixed `==`.

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

[5] Matthieu Sozeau. A New Look at Generalized Rewriting in Type Theory. *Journal of Formalized Reasoning*, pages 41–62, 2009.