# Constructive Category Theory

#### Gérard Huet and Amokrane Saïbi

INRIA Rocquencourt<sup>1</sup>

B.P. 105 - 78153 Le Chesnay CEDEX

# 1 Introduction

These notes are part of a preliminary attempt at developing abstract algebra in constructive type theory. The developments that follow are inspired from a previous axiomatization by P. Aczel in LEGO in Jan. 1993, as an initial step to a program of formal development of Galois Theory[1]. Our version of type theory is the Calculus of Inductive Constructions, as implemented in Coq V5.10. This paper give the full transcript of the Coq axiomatisation.

In this note we develop one possible axiomatisation of the notion of category by modeling objects as types and Hom-sets as Hom-setoids of arrows parameterized by their domain and codomain types. Thus we may quotient arrows, but not objects. We develop in this setting functors, as functions on objects, and extentional maps on arrows. We show that CAT is a category, and we do not need to distinguish to this effect "small" and "big" categories. We rather have implicitly categories as relatively small structures indexed by a universe. Thus we just need two instances of the same notion of category in order to define CAT.

We then construct the Functor Category, with the natural definition of natural transformations. We then show the Interchange Law, which exhibits the 2-categorical structure of the Functor Category. We end this paper by giving a corollary to Yoneda's lemma.

This incursion in Constructive Category Theory shows that Type Theory is adequate to represent faithfully categorical reasoning. Three ingredients are essential:  $\Sigma$ -types, to represents structures, dependent types, so that arrows are indexed with their domains and codomains, and a hierarchy of universes, in order to escape the foundational difficulties. Some amount of type reconstruction is necessary, in order to write equations between arrows without having to indicate their type other than at their binder, and notational abbreviations, allowing e.g. infix notation, are necessary to offer the formal mathematician a language close to the ordinary informal categorical notation.

# 2 Relations

We assume a number of basic constructions, which define quantifiers and equality at the level of sort Type. These definitions are included in the prelude module Logic\_Type of the Coq system.

We start with a few standard definitions pertaining to binary relations.

Require Logic\_Type.

Section Orderings.

<sup>&</sup>lt;sup>1</sup>This research was partially supported by ESPRIT Basic Research Action "TYPES."

```
Variable U: Type.

Definition Relation := U -> U -> Prop.

Variable R: Relation.

Definition Reflexive := (x: U) (R x x).

Definition Transitive := (x,y,z: U) (R x y) -> (R y z) -> (R x z).

Definition Symmetric := (x,y: U) (R x y) -> (R y x).

Structure equivalence : Prop := {Prf_e_refl : Reflexive; Prf_e_trans : Transitive; Prf_e_sym : Symmetric}.

Structure partial_equivalence : Prop := {Prf_pe_sym : Symmetric; Prf_pe_trans : Transitive}.

End Orderings.

Syntactic Definition Equivalence := equivalence | 1.

Syntactic Definition Partial_equivalence := partial_equivalence | 1.
```

The notations used should be familiar to the reader, once he gets used to the fact that universal quantification is denoted by parenthesised typing judgments, like (x:T)(P x), standing for  $\forall x$ :  $T \cdot P(x)$ . Similarly, functional abstraction is denoted by square brackets, like [x:T](f x), standing for  $\lambda x : T \cdot f(x)$ .

The "Section" mechanism of Coq allows to parameterize the notions defined in the body of the section by the variables and axioms on which they depend. In our case, all the notions defined inside section Orderings are parameterized by parameters U and R. Thus, for instance, the definition of Reflexive becomes, upon closing of this section:

```
Definition Reflexive := [U:Type][R:(Relation\ U)](x:\ U) (R x x).
```

The combinator Equivalence is defined by the Syntactic Definition above. This is just a macro definition facility, which will in the rest of the session replace every occurrence of Equivalence by (equivalence?). These 'question marks arguments' will be automatically synthesized from the context. We use systematically this facility in the following development, with the convention that all the generic notions defined with macros have a name starting with an upper-case letter, and generate internally the same name with the corresponding lower-case letter, applied to question marks. The number of these question marks is indicated in the macro after the symbol |. In each context of use of such a macro, we shall expect the corresponding parameters to be derivable mechanically during type-checking.

## 3 Setoid

We now move to the development of "Setoids". Setoids are triples composed of a Type S, a relation R over S, and a proof that R is an equivalence relation over S. Thus a Setoid is a set considered

as the quotient of a Type by a congruence. Setoids were first investigated by M. Hofmann in the framework of Martin-Löf's type theory[7]. This terminology is due to R. Burstall.

## 3.1 The Setoid structure

Let us understand what happens by type synthesis. The subformula (Equivalence Equal) gets replaced by (equivalence? Equal). Now the type checker instantiates the question mark as Carrier, since Equal is declared to be a relation over Carrier. This way we use type synthesis to elide information which is implicit from the context.

Remark that in Coq  $\Sigma$ -types (records) are not primitive, but are built as inductive types with one constructor. The macro Structure constructs the corresponding inductive type and defines the projection functions for destructuring a Setoid into its constituent fields. It defines also a constructor Build\_Setoid to build a Setoid from its constituents. One can choose a different name for the constructor by putting the desired name before the opening brace of the Structure defining expression. Such specialized macros are user-definable in the same way as tactics.

In order to have a more pleasant notation, we first introduce a concrete syntax declaration allowing to parse (Carrier A) as |A|:

```
Grammar command command1 := ["|" command0(\$s)"|"] \rightarrow [\$0 = <<(Carrier \$s)>>].
```

Such a Grammar declaration may be read as follows. A grammar production for the command language consists of two parts. The first part represents a production which is added to the corresponding non-terminal entry; here command1, receives the new production "|" command0(\$s) "|". The second part is the semantic action which builds the corresponding abstract syntax tree when firing this production; here we indicate that we build an application of the constant Carrier to the result of parsing with entry command0 what is enclosed in the vertical bars. The various entries commandn stratify the commands according to priority levels.

Given a Setoid A, (Equal A) is its associated relation. We give it the infix notation =%S, the parameter A being synthesised by type-checking. Since the symbol =%S is not predefined, we have to declare it as a new token (for the "extensible" lexer of Coq).

```
Token "=%S". Grammar command command1 := [ command0($a) "=%S" command0($b) ] -> [$0 = <<(Equal ? $a $b)>>].
```

Note that =%S is a generic Setoid equality, since the type of its elements may in general be inferred from the context, as we shall see immediately.

The last extracted field is the proof that the equality of a Setoid is an equivalence relation. Right after this equivalence proof, we give as corollaries reflexivity, symmetry and transitivity of equality. We get these proofs easily with the help of Coq's proof engine, driven by tactics. Here as in the rest of the document, we do not give the proof scripts, just the statements of lemmas.

## 3.2 An example

As example of the preceding notions, let us define the Setoid of natural numbers. The type of its elements cannot be directly the inductively defined nat:Set, but it is easy to define an isomorphic Nat:Type.

```
Inductive Nat : Type := Z : Nat | Suc : Nat -> Nat.
Definition Eq_Nat := [N1,N2:Nat] N1==N2.
```

The == symbol which appears in the body of the definition of Eq\_Nat, is the standard polymorphic Leibniz equality defined in the Logic\_Type module. In our case, it is instantiated over the Type Nat, inferred from the type of N1.

Right after this, we give the equivalence proof of Eq\_Nat and build the setoid of natural numbers:

```
\label{lemma} \begin{tabular}{ll} Lemma & Eq_Nat_equiv : (Equivalence & Eq_Nat). \\ \\ Definition & Set_of_nat := (Build_Setoid & Nat & Eq_Nat_equiv). \\ \\ \end{tabular}
```

#### 3.3 Alternative: Partial Setoids

Alternatively, we could build Partial Setoids, where the equality equivalence is replaced by a weaker partial equivalence relation of coherence; total elements are defined as being coherent with themselves:

#### 3.4 The Setoid of Maps between two Setoids

We now define a Map between Setoid A and Setoid B as a function from |A| to |B| which respects equality. Remark the use of generic equality in map\_law.

Section maps.

A Map m over A and B is thus similar to a pair, packing a function (ap A B m) (of type |A|->|B|) with the proof (Pres A B m) that this function respects equality.

Two Maps f and g are defined to be equal iff they are extensionally equal, i.e.  $\forall x. f(x) = g(x)$ :

This last command allows writing A=>B, with appropriate precedence level, for the Setoid of Maps between Setoids A and B.

We end this section by defining a generic Ap, denoting the application function associated with a Map, and a (curried) binary application ap2, useful for what follows.

```
Syntactic Definition Ap := ap | 2.  
Definition ap2 := [A,B,C:Setoid][f:|(A=>(B=>C))|][a:|A|] (Ap (Ap f a)).  
Syntactic Definition Ap2 := ap2 | 3.
```

# 4 Categories

## 4.1 The category structure

We now axiomatise a category as consisting of a Type of Objects and a family of Hom Setoids indexed by their domain and codomain types.

Section cat.

```
Variable Ob : Type.
Variable Hom : Ob->Ob->Setoid.
```

The next component of a category is a composition operator, which for any Objects a,b,c, belongs to (Hom a b) => ((Hom b c) => (Hom a c)). We write this operator (parameters a,b,c, being implicit by type synthesis) as infix o.

```
Variable Op_comp : (a,b,c:0b) \mid ((Hom \ a \ b) \Rightarrow ((Hom \ b \ c) \Rightarrow (Hom \ a \ c))) \mid.

Definition Cat_comp := [a,b,c:0b](Ap2 \ (Op\_comp \ a \ b \ c)).
```

```
Grammar command command2 := [ command1($f) " o " command2($g) ] -> [$0 = <<(Cat_comp ? ? ? $f $g)>>].
```

Composition is assumed to be associative:

The final component of a category is, for every object a, an arrow in (H a a) which is an identity for composition:

```
Variable Id : (a:0b) \mid (Hom\ a\ a) \mid. 
 Definition idl_law := (a,b:0b)(f:|(Hom\ a\ b)|)((Id\ a)\ o\ f) =%S f. 
 Definition idr_law := (a,b:0b)(f:|(Hom\ b\ a)|)f =%S (f o (Id ?)). 
 End cat.
```

We give generic notations for the various laws:

```
Syntactic Definition Assoc_law := assoc_law | 2.

Syntactic Definition Idl_law := idl_law | 2.

Syntactic Definition Idr_law := idr_law | 2.
```

We are now able to define synthetically a Category:

We successively define the projections which extract the various components of a category.

Remark that we now use the infix notation o in the context of a local Category parameter C. It must be noticed that Grammar definitions inside Sections disappear when their section is closed. Thus the new rule giving syntax for Comp does not conflict with the previous one giving syntax for Cat\_comp.

Actually, a composition operator is nothing else than a binary function verifying the congruence laws for both arguments. Thus we provide a general method allowing the construction of a composition operator from such a function. We shall use systematically this tool in the following, for every category definition.

Section composition\_to\_operator.

```
: Type.
Variable Ob
                   : Ob->Ob->Setoid.
Variable Hom
Variable Comp_fun : (a,b,c:0b) | (Hom a b) | -> | (Hom b c) | -> | (Hom a c) |.
Definition Congl_law := (a,b,c:Ob)(f,g:|(Hom b c)|)(h:|(Hom a b)|)
           f = %S g \rightarrow (Comp_fun a b c h f) = %S (Comp_fun a b c h g).
Definition Congr_law := (a,b,c:0b)(f,g:|(Hom a b)|)(h:|(Hom b c)|)
           f = %S g \rightarrow (Comp_fun a b c f h) = %S (Comp_fun a b c g h).
Definition Cong_law := (a,b,c:0b)(f,f':|(Hom a b)|)(g,g':|(Hom b c)|)
           f = \%S f' \rightarrow g = \%S g' \rightarrow (Comp_fun a b c f g) = \%S (Comp_fun a b c f' g').
Hypothesis pcgl : Congl_law.
Hypothesis pcgr : Congr_law.
Variable a, b, c : Ob.
Lemma Comp1_map_law : (f:|(Hom a b)|)(Map_law (Comp_fun a b c f)).
Definition Comp1_map := [f:|(Hom a b)|](Build_Map (Comp_fun a b c f) (Comp1_map_law f)).
Lemma Comp_map_law : (map_law (Hom a b) (Hom b c)=>(Hom a c) Comp1_map).
Definition Build_Comp := (build_Map (Hom a b) (Hom b c)=>(Hom a c)
                                      Comp1_map Comp_map_law).
```

We now check that composition preserves the morphisms equalities, to the left and to the right, and prove as a corollary the congruence law for composition:

```
Lemma Prf_congl : (C:Category)(Congl_law ? ? (Comp C)).
Lemma Prf_congr : (C:Category)(Congr_law ? ? (Comp C)).
Lemma Prf_cong : (C:Category)(Cong_law ? ? (Comp C)).
```

#### 4.2 Hom equality

End composition\_to\_operator.

We need for the following a technical definition: two arrows in  $(Hom\ a\ b)$  of category C are equal iff the corresponding elements of the Setoid  $(Hom\ a\ b)$  are equal. This is a typical example where Type Theory obliges us to make explicit an information which does not even come up in

the standard mathematical discourse based on set theory. Of course we would like the standard "abus de notation" to be implemented in a more transparent way, through overloading or implicit coercions. We deal with this problem here by implicit synthesis of the category parameter and of the object parameters in order to write simply  $\mathbf{f} = \% \mathbf{H} \mathbf{g}$  for the equality of arrows  $\mathbf{f}$  and  $\mathbf{g}$ .

Here the reader may be puzzled at our seemingly too general type for arrow equality: the predicate Equal\_hom takes as arguments a Category C, objects a,b,c,d of C, and arrows (f:|(Hom a b)|) and (g:|(Hom c d)|). Since the only possible constructor for this equality is Build\_Equal\_hom, which requires the second arrow g to have the same type as the first one f, it might seem sufficient to restrict the type of Equal\_hom accordingly. However, this generality is needed, because we want to be able to state the equality of two arrows whose respective domains are not definitionally equal, but will be equal for certain instanciations of parameters. For instance, later on, the problem will arise when defining functor equality: we want to be able to write F(f) = G(f), which will force say F(A) and G(A) to be definitionally equal objects, but there is no way to specify F and G with type declarations such that F(A) = G(A). This would necessitate an extension of type theory with definitional constraints, which could be problematic with respect to decidability of definitional equality. This extension is not really needed if one takes care to write dependent equalities with sufficiently general types.

Section equal\_hom\_equiv.

```
Variable C : Category.
Variable a , b : (0b C).
Variable f : |(Hom a b)|.

Lemma Equal_hom_refl : f =%H f.

Variable c, d : (0b C).
Variable g : |(Hom c d)|.

Lemma Equal_hom_sym : f =%H g -> g =%H f.

Variable i, j : (0b C).
Variable h : |(Hom i j)|.

Lemma Equal_hom_trans : f =%H g -> g =%H h -> f =%H h.

End equal_hom_equiv.
```

## 4.3 Dual Categories

The dual category  $C^{\circ}$  of a category C has the same objects as C. Its arrows however are the opposites of the arrows of C, i.e.  $f^{\circ}: a \longrightarrow b$  is a morphism of  $C^{\circ}$  iff  $f: b \longrightarrow a$  is a morphism of C.

```
Section d_cat.
Variable C : Category.
Definition DHom := [a,b:(Ob C)](Hom b a).
   Composition is defined as expected: f^{\circ} \circ q^{\circ} = (q \circ f)^{\circ}. Identity arrows are invariant. We then
check the category laws.
Definition Comp_Darrow := [a,b,c:(Ob C)][df:|(DHom a b)|][dg:|(DHom b c)|] dg o df.
Lemma Comp_dual_congl : (Congl_law (Ob C) DHom Comp_Darrow).
Lemma Comp_dual_congr : (Congr_law (Ob C) DHom Comp_Darrow).
Definition Comp_Dual := (Build_Comp (Ob C) DHom Comp_Darrow
                          Comp_dual_congl Comp_dual_congr).
Lemma Assoc_Dual : (assoc_law (Ob C) DHom Comp_Dual).
Lemma Idl_Dual : (idl_law (Ob C) DHom Comp_Dual (id C)).
Lemma Idr_Dual : (idr_law (Ob C) DHom Comp_Dual (id C)).
   We write (Dual C) for the dual category of C.
Definition Dual := (Build_Category (Ob C) DHom Comp_Dual Id_Dual
                                     Assoc_Dual Idl_Dual Idr_Dual).
End Dcat.
```

#### 4.4 Category exercises

: Category.

We define epics, monos, and isos. As an exercise, we show that two initial objects are isomorphic. A morphism  $f: a \longrightarrow b$  is epi when for any two morphisms  $g, h: b \longrightarrow c$ , the equality  $f \circ g = f \circ h$  implies g = h.

```
Section epic_def.
```

Variable C

```
Structure isEpic [f:|(Hom a b)|] : Type :=
{Epic_l : (Epic_law f)}.
End epic_def.
Syntactic Definition IsEpic := isEpic | 3.
   A morphism f:b\longrightarrow a is monic when for any two morphisms g,h:c\longrightarrow b, the equality
g \circ f = h \circ f implies g = h.
Section monic_def.
Variable C : Category.
Variable a, b : (0b C).
Definition Monic_law := [f:|(Hom b a)|](c:(Ob C))(g,h:|(Hom c b)|)
                        (g \circ f) = %S (h \circ f) \rightarrow g = %S h.
Structure isMonic [f:|(Hom b a)|] : Type :=
{Monic_l : (Monic_law f)}.
End monic_def.
Syntactic Definition IsMonic := isMonic | 3.
   A morphism f is iso if there is a morphism f^{-1}1:b\longrightarrow a whith f^{-1}\circ f=Id_b and f\circ f^{-1}=Id_a.
Section iso_def.
Variable C : Category.
(f1 \circ f) = %S (Id b).
Variable a, b : (0b C).
Structure isIso [f:|(Hom a b)|] : Type :=
{inv_iso : |(Hom b a)|;
 Idl_inv : (Iso_law ? ? f inv_iso);
  Idr_inv : (Iso_law ? ? inv_iso f)}.
   We now say that two objects a and b are isomorphic (a \cong b) if they are connected by an iso
arrow.
Structure iso : Type :=
{Iso_mor : |(Hom a b)|;
 Prf_isIso : (isIso Iso_mor)}.
End iso_def.
Syntactic Definition IsIso := isIso | 3.
Syntactic Definition Inv_iso := inv_iso | 4.
Syntactic Definition Iso := iso | 1.
```

Now we say that object a is initial in Category C iff for any object b there exists a unique arrow in  $(Hom\ a\ b)$ .

```
Section initial_def.
Variable C : Category.
Definition At_most_1mor := [a,b:(0b\ C)](f,g:|(Hom\ a\ b)|) f =%S g.
Structure isInitial [a:(0b C)] : Type :=
         : (b:(0b C))|(Hom a b)|;
 UniqueI : (b:(0b C))(At_most_1mor a b)}.
End initial def.
Syntactic Definition IsInitial := isInitial | 1.
Syntactic Definition MorI := morI | 2.
   Dually we define when an object b is terminal in Category C: for any object a there exists a
```

unique arrow in  $(Hom\ a\ b)$ .

Section terminal\_def.

```
Variable C : Category.
```

```
Structure isTerminal [b:(0b C)] : Type :=
         : (a:(0b C))|(Hom a b)|;
 UniqueT : (a:(0b C))(At_most_1mor C a b)}.
```

End terminal\_def.

```
Syntactic Definition IsTerminal := isTerminal | 1.
Syntactic Definition MorT := morT | 2.
```

As an exercise we may prove easily that any two initial objects must be isomorphic:

```
Lemma I_unic : (C:Category)(I1,I2:(0b C))(IsInitial I1) -> (IsInitial I2) -> (Iso I1 I2).
```

We also prove that the property of being terminal is dual to that of being initial: an initial object in C is terminal in  $C^{\circ}$ .

```
Lemma Initial_dual : (C:(Category))(a:(Ob C))(IsInitial a) -> (isTerminal (Dual C) a).
```

We remark that these properties have been defined at the Type level. They could all be defined at the level Prop, except that in the case of IsIso, we could not extract the field inv\_iso of type Type.

## 4.5 The Category of Setoids

We now define the Category of Setoids with Maps as Homs. First we have to define composition and identity of Maps. The composition of two Maps is defined from the composition of their underlying functions; we have to check extensionality of the resulting function. We use the infix notation o%M.

```
Section mcomp.
```

The operator Comp\_map is just a function. We shall now "mapify" it, by proving that it is extensional in its two arguments, in order to get a Map composition operator.

```
Lemma Comp_map_congl : (Congl_law Setoid Map_setoid Comp_map).

Lemma Comp_map_congr : (Congr_law Setoid Map_setoid Comp_map).

Definition Comp_SET := (Build_Comp Setoid Map_setoid Comp_map_congr).
```

After checking the associativity of our composition operation, we define the identity Map from the identity function  $\lambda x.x$ , checking other category laws.

```
Lemma Assoc_SET : (assoc_law Setoid Map_setoid Comp_SET).
Section id_map_def.
Variable A : Setoid.
Definition Id_fun := [x:|A|]x.
Lemma Id_fun_map_law : (Map_law Id_fun).
Definition Id_map := (Build_Map Id_fun Id_fun_map_law).
End id_map_def.
```

```
Definition Id_SET := Id_map.

Lemma Idl_SET : (idl_law Setoid Map_setoid Comp_SET Id_SET).

Lemma Idr_SET : (idr_law Setoid Map_setoid Comp_SET Id_SET).

Now we have all the ingredients to form the Category of Setoids SET.

Definition SET := (Build_Category Setoid Map_setoid Comp_SET Id_SET Assoc_SET Idl_SET Idr_SET).
```

## 5 Functors

#### 5.1 Definition of Functor

Functors between categories C and D are defined in the usual way, with two components, a function from the objects of C to the objects of D, and a Map from Hom-sets of C to Hom-sets of D. Remark how type theory expresses in a natural way the type constraints of these notions, without arbitrary codings.

Section Functors.

```
Variable C, D: Category.
Variable FOb : (Ob C) -> (Ob D).
Variable FMap : (a,b:(Ob C))(Map (Hom a b) (Hom (FOb a) (FOb b))).
   Functors must preserve the Category structure, and thus verify the two laws: F(f \circ q) =
F(f) \circ F(g) and F(Id_a) = Id_{F(a)}.
Definition fcomp_law := (a,b,c:(0b\ C))(f:|(Hom\ a\ b)|)(g:|(Hom\ b\ c)|)
           (Ap (FMap a c) (f o g)) = %S ((Ap (FMap a b) f) o (Ap (FMap b c) g)).
Definition fid_law := (a:(0b C))(Ap (FMap a a) (Id a)) = %S (Id (F0b a)).
End Functors.
Syntactic Definition Fcomp_law := fcomp_law | 2.
Syntactic Definition Fid_law := fid_law | 2.
Structure Functor [C,D:Category] : Type :=
                : (0b C) -> (0b D);
                : (a,b:(0b C))(Map (Hom a b) (Hom (f0b a) (f0b b)));
  Prf_Fcomp_law : (Fcomp_law f0b FMap);
              : (Fid_law f0b FMap)}.
  Prf_Fid_law
```

As usual, we define some syntactical abbreviations. Thus F(a) will be written (FOb F a) and F(f) will be written (FMor F f).

We now define the Setoid of Functors. The equality of Functors is extensional equality on their morphism function component, written as infix =%F with appropriate type synthesis:

We now have all the ingredients to form the Functor Setoid.

```
Definition Functor_Setoid:= [C,D:Category](Build_Setoid (Functor C D) (Equal_Functor C D) (Equal_Functor_equiv C D)).
```

#### 5.2 Hom Functors

We give in this section an example of functor construction, with the family of Hom-Functors. Let C be a category and a an object of C. The functor  $Hom(a, -) : C \longrightarrow SET$  is defined by:

• for every object b of C,  $Hom(a, -)(b) = (Hom\ a\ b)$ 

```
Section funset.
```

```
Variable C : Category.
Variable a : (Ob C).
Definition FunSET_ob := [b:(Ob C)](Hom a b).
```

• for every  $f:b\longrightarrow c$ ,  $Hom(a,-)(f):(Hom\ a\ b)\longrightarrow (Hom\ a\ c)$  is a Map, mapping morphism  $g:a\longrightarrow b$  of C to morphism  $g\circ f$ .

```
Section funset_map_def.
Variable b, c : (0b C).
Section funset_mor_def.
Variable f : |(Hom b c)|.
```

```
Definition Funset_mor1 := [g:|(Hom a b)|] (g o f).

Lemma Funset_map_law1 : (Map_law Funset_mor1).

Definition Funset_mor := (Build_Map Funset_mor1 Funset_map_law1).

End funset_mor_def.

Lemma Funset_map_law : (map_law (Hom b c) (hom SET (Funset_ob b) (Funset_ob c)) Funset_mor).

Definition Funset_map := (build_Map (Hom b c) (hom SET (Funset_ob b) (Funset_ob c)) Funset_mor Funset_map_law).

End funset_map_def.

We check the functorial properties for $Hom(a, -)$ and define it, with notation (Funset_a).

Lemma Fun_comp_law : (fcomp_law C SET Funset_ob Funset_map).

Lemma Fun_id_law : (fid_law C SET Funset_ob Funset_map).

Definition funset := (Build_Functor C SET Funset_ob Funset_map Fun_comp_law Fun_id_law).

End funset.

Syntactic Definition Funset := funset | 1.
```

## 5.3 The Category of Categories

In this section we now reflect the theory upon itself: Categories may form the type of a category of Categories CAT, the arrows being Functors. All is really needed is to define Functor composition and Identity, and to prove a few easy lemmas exhibiting the Category structure of CAT.

The first step consists in defining the composition of two functors. We compose functors  $G: C \longrightarrow D$  and  $H: D \longrightarrow E$  to form a functor  $G \circ H: C \longrightarrow E$ , by composing separately their object functions and their morphism maps. We write o%F for this functor composition.

```
Section Comp_F.
```

```
Variable C, D, E : Category.
Variable G : (Functor C D).
Variable H : (Functor D E).

Definition comp_F0b := [a:(0b C)](F0b H (F0b G a)).

Section comp_functor_map.

Variable a, b : (0b C).
```

```
Definition comp_FMor := [f:|(Hom a b)|](FMor H (FMor G f)).
Lemma Comp_FMap_law : (Map_law comp_FMor).
Definition Comp_FMap := (Build_Map comp_FMor Comp_FMap_law).
 End comp_functor_map.
Lemma Comp_Functor_comp_law : (Fcomp_law comp_F0b Comp_FMap).
Lemma Comp_Functor_id_law : (Fid_law comp_F0b Comp_FMap).
Definition Comp_Functor := (Build_Functor C E comp_F0b Comp_FMap
                                          Comp_Functor_comp_law Comp_Functor_id_law).
End Comp_F.
Syntactic Definition Comp_F0b := comp_F0b | 3.
Syntactic Definition Comp_FMor := comp_FMor | 3.
Token "%F".
Grammar command command2 := [ command1($F) "o" "%F" command2($G) ] ->
                            [\$0 = <<(Comp_Functor ? ? \$F \$G)>>].
   As before, we construct a composition operator after checking the Congruence laws.
Lemma Comp_Functor_congl : (Congl_law Category Functor_setoid Comp_Functor).
Lemma Comp_Functor_congr : (Congr_law Category Functor_setoid Comp_Functor).
Definition Comp_CAT := (Build_Comp Category Functor_setoid Comp_Functor
                                      Comp_Functor_congl Comp_Functor_congr).
Lemma Assoc_CAT : (assoc_law Category Functor_setoid Comp_CAT).
   For every category C, we construct the identity functor Id_C from the identity function on
objects and the identity map on morphisms.
Section idCat.
Variable C : Category.
Definition Id_CAT_ob := [a:(0b C)]a.
Definition Id_CAT_map := Definition Id_CAT_map := [a,b:(0b C)](Id_map (Hom a b)).
Lemma Id_CAT_comp_law : (Fcomp_law Id_CAT_ob Id_CAT_map).
Lemma Id_CAT_id_law : (Fid_law Id_CAT_ob Id_CAT_map).
Definition Id_CAT := (Build_Functor C C Id_CAT_ob Id_CAT_map
```

End idCat.

```
Lemma Idl_CAT : (idl_law Category Functor_setoid Comp_CAT Id_CAT).

Lemma Idr_CAT : (idr_law Category Functor_setoid Comp_CAT Id_CAT).
```

We now have all the ingredients to recognize in CAT the structure of a Category. All we need to do is to take a second copy of the notion of Category, called Category'. The implicit universe adjustment mechanism will make sure that its Type refers to a bigger universe.

Note that here we make an essential use of the universes hierarchy: There is not a unique CAT, there is a family of  $CAT_i$ , and each  $CAT_i$  is a  $Category_j$  for i < j. Thus we do not have "small" and "large" categories, but "relatively small" categories. Thus the construction of CAT above is consistent with the analysis by Coquand[4] of paradoxes related to the category of categories.

It is to be remarked that this example justifies the mechanism called "universe polymorphism" defined by Harper and Pollack[6]. That is, with universe polymorphism, we could directly define CAT as a Category, without having to make an explicit copy of the notion, the copying being done implicitly for each occurrence of the name Category. Coq does not implement universe polymorphism at present, because this mechanism is rather costly in space, and seldom used in practice.

#### 5.4 Functor exercises

It is easy to check that a functor preserves the property of being iso, i.e.  $a \cong b \Longrightarrow F(a) \cong F(b)$ . Section functor\_prop.

```
Variable C, D: Category. Variable F: (Functor C D). Lemma Functor_preserves_iso: (a,b:(0b C))(Iso a b) -> (Iso (F0b F a) (F0b F b)). A functor F:C\longrightarrow D is said to be faithful if for any pair a,b of objects of C, and any pair f,g:a\longrightarrow b of morphisms, we have F(f)=F(g) only if f=g. Definition Faithful_law := (a,b:(0b C))(f,g:|(Hom a b)|) ((FMor F f) =%S (FMor F g)) -> (f =%S g). Structure isFaithful: Type := {Prf_faithful}: Faithful_law}.
```

A functor is said to be full if, for any pair a, b of objects of C, and any morphism  $h: F(a) \longrightarrow F(b)$ , there exists a morphism  $f: a \longrightarrow b$  such that h = F(f). According to the axiom of choice, there exists a function H such that for every morphism  $h: F(a) \longrightarrow F(b)$ , H(h) corresponds to f, i.e. h = F(H(h)).

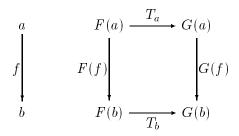
```
Definition Full_law := [H:(a,b:(0b C))|(Hom (F0b F a) (F0b F b))| \rightarrow |(Hom a b)|]
                        (a,b:(0b C))(h:|(Hom (F0b F a) (F0b F b))|)
                        h = %S (FMor F (H a b h)).
Structure isFull : Type :=
{\text{Full\_mor} : (a,b:(0b C))|(Hom (F0b F a) (F0b F b))|} \rightarrow {\text{|(Hom a b)|}};
  Prf_full : (Full_law Full_mor) }.
End functor_prop.
Syntactic Definition IsFaithful := isFaithful | 2.
Syntactic Definition IsFull := isFull | 2.
   These two properties are closed by composition:
Section comp_functor_prop.
Variable C, D, E : Category.
Variable F
                  : (Functor C D).
Variable G
                  : (Functor D E).
Lemma IsFaithful_comp : (IsFaithful F) -> (IsFaithful G) -> (IsFaithful (F o%F G)).
Lemma IsFull_comp : (IsFull F) -> (IsFull G) -> (IsFull (F o%F G)).
End comp_functor_prop.
```

# 6 The Functor Category

The type of Functors from Category C to Category D admits a Category structure. The corresponding arrows are called Natural Transformations.

#### 6.1 Natural Transformations

We now define Natural Transformations between two Functors F and G from C to D. A Natural Transformation T from F to G maps an object a of Category C to an arrow  $T_a$  from object F(a) to object G(a) in Category D such that the following naturality law holds:  $F(f) \circ T_b = T_a \circ G(f)$ .



Note that Natural Transformations are defined as functions, not as Maps, since objects are Types and not necessarily Setoids.

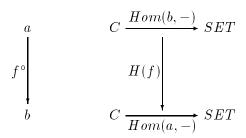
We now define  $\{F \Rightarrow G\}$ , the Natural Transformations Setoid between Functors F and G. Equality of natural transformations is also extensional. Thus, two natural transformations T and T' are said to be equal whenever their components are equal for an arbitrary object:  $\forall a. T_a = T'_a$ . As previously, we write =%NT for this equality.

```
Section setoid_nt.
```

Section nat\_transf.

#### 6.2 An example of Natural Transformation

Let C be any category, and  $f: a \longrightarrow a$  be b morphism of C. We define a natural transformation  $H(f): Hom(b, -) \longrightarrow Hom(a, -)$ , called the Yoneda map, as follows.



For every object c of C, H(f)(c) is a Map mapping a morphism  $h:b\longrightarrow c$  of C to the morphism  $f\circ h$ .

Section funset\_nt.

Variable C : Category.

Variable b, a : (0b C).

Variable f : |(Hom a b)|.

Section nth\_map\_def.

Variable c : (Ob C).

Definition NtH\_arrow := [h:|(Hom b c)|] f o h.

Lemma NtH\_map\_law : (Map\_law NtH\_arrow).

Definition NtH\_map := (Build\_Map NtH\_arrow NtH\_map\_law).

End nth\_map\_def.

We check the naturality of this map and define the natural transformation H(f), written as (NtH f).

```
 \label{lemma NtH_nt_law : (NT_law (FunSET b) (FunSET a) NtH_map). }
```

Definition ntH := (Build\_NT (FunSET b) (FunSET a) NtH\_map NtH\_nt\_law).

End funset\_nt.

Syntactic Definition NtH := ntH | 3.

#### 6.3 Constructing the Category of Functors

We now have all the tools to define the category of Functors from C to D. Objects are Functors, arrows are corresponding Natural Transformations Setoids.

We define the composition of two natural transformations T and T' as  $(T \circ_v T')_a = T_a \circ T'_a$ . The v subscript stands for "vertical", since we shall define later another "horizontal" composition.

```
Section cat_functor.
Variable C, D: Category.
Section compnt.
 Variable F, G, H : (Functor C D).
 Variable T
               : (NT F G).
 Variable T'
                : (NT G H).
Definition Comp_tau := [a:(Ob C)](ApNT T a) o (ApNT T' a).
Lemma Comp_tau_nt_law : (NT_law ? ? [a:(Ob C)](Comp_tau a)).
Definition CompV_NT := (Build_NT ? ? Comp_tau Comp_tau_nt_law).
End compnt.
Lemma CompV_NT_congl : (Congl_law (Functor C D) (NT_setoid C D) CompV_NT).
Lemma CompV_NT_congr : (Congr_law (Functor C D) (NT_setoid C D) CompV_NT).
Definition Comp_CatFunct := (Build_Comp (Functor C D) (NT_setoid C D)
                                            CompV_NT CompV_NT_congl CompV_NT_congr).
Lemma Assoc_CatFunct : (assoc_law (Functor C D) (NT_setoid C D) Comp_CatFunct).
   To every functor F, we associate an identity natural transformation Id_F defined as \lambda a. Id_{F(a)}:
Section id_catfunct_def.
 Variable F : (Functor C D).
 Definition Id_CatFunct_tau := [a:(0b C)](Id (F0b F a)).
Lemma Id_CatFunct_nt_law: (NT_law ? ? Id_CatFunct_tau).
Definition Id_CatFunct := (Build_NT ? ? Id_CatFunct_tau Id_CatFunct_nt_law).
 End id_catfunct_def.
Lemma Idl_CatFunct : (idl_law (Functor C D) (NT_setoid C D) Comp_CatFunct Id_CatFunct).
Lemma Idr_CatFunct : (idr_law (Functor C D) (NT_setoid C D) Comp_CatFunct Id_CatFunct).
   Having checked that we have all categorical properties, we may now define the functor category.
Definition CatFunct := (Build_Category (Functor C D) (NT_setoid C D) Comp_CatFunct
                               {\tt Id\_CatFunct\ Assoc\_CatFunct\ Idl\_CatFunct\ Idr\_CatFunct)}\;.
End cat_functor.
```

```
Token "%NTv".

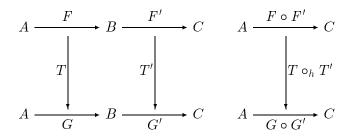
Grammar command command2 := [ command1($T1) "o" "%NTv" command2($T2) ] ->

[$0 = <<(CompV_NT ? ? ? ? $ $T1 $T2)>>].
```

# 7 The Interchange Law

In order to put to the test our categorical constructions, we prove the *interchange law*. This is one of the laws of 2-categories (categories whose arrows have themselves a category structure). This notion is used in theoretical computer science for the study of programming languages semantics and type theory.

Let A, B and C be categories,  $F, G: A \longrightarrow B$  and  $F', G': B \longrightarrow C$  be functors. We define the horizontal composition of natural transformations  $T: F \longrightarrow G$  and  $T': F' \longrightarrow G'$  as  $(T \circ_h T')_a = T'_{F(a)} \circ G'(T_a)$ . We check that  $T \circ_h T'$  is indeed a natural transformation from  $F \circ F'$  to  $G \circ G'$ .



Section horz\_comp.

Variable A, B, C : Category.

Variable F, G : (Functor A B).

Variable F', G' : (Functor B C).

Variable T : (NT F G).

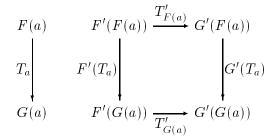
Variable T' : (NT F' G').

Definition Ast : (a:(0b A)) | (Hom (F0b (F o%F F') a) (F0b (G o%F G') a)) | := [a:(0b A)](ApNT T' (F0b F a)) o (FMor G' (ApNT T a)).

In order to prove the naturality of  $T \circ_h T'$ , we use the equality:

$$(T \circ_h T')_a = T'_{F(a)} \circ G'(T_a) = F'(T_a) \circ T'_{G(a)}$$

which follows from the naturality diagram of T' for the morphism  $T_a$ :



```
Lemma Ast_eq : (a:(0b A)) ((FMor F' (ApNT T a)) o (ApNT T' (F0b G a))) =%S  ((ApNT T' (F0b F a)) o (FMor G' (ApNT T a))).
```

Lemma Ast\_nt\_law : (NT\_law (F o%F F') (G o%F G') Ast).

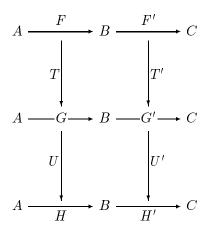
Definition CompH\_NT := (Build\_NT (F o%F F') (G o%F G') Ast Ast\_nt\_law).

End horz\_comp.

We shall write o%NTh for horizontal composition.

We shall now verify an important algebraic property, the *Interchange Law*, which links horizontal and vertical composition.

Let A, B, C be categories,  $F, G, H : A \longrightarrow B$ ,  $F', G', H' : B \longrightarrow C$  be functors, and  $T : F \longrightarrow G$ ,  $T' : F' \longrightarrow G'$ ,  $U : G \longrightarrow H$ ,  $U' : G' \longrightarrow H'$  be natural transformations.



Interchange Law:

$$(T \circ_h T') \circ_v (U \circ_h U') = (T \circ_v U) \circ_h (T' \circ_v U')$$

Section interchangelaw.

Variable A, B, C : Category.

Variable F, G, H : (Functor A B).

Variable T : (NT F G).

Variable T : (NT F G').

Variable U : (NT G H).

Variable U : (NT G' H').

Lemma InterChange\_law : ((T o%NTh T') o%NTv (U o%NTh U')) =%NT ((T o%NTv U) o%NTh (T' o%NTv U')).

End interchangelaw.

We end this section by showing that the horizontal composition of natural transformations is associative. Here lurks a small difficulty.

Given Categories A, B, C, Functors F, G from A to B and Functors F', G' from B to C, the horizontal composition of Natural Transformation T from F to G and Natural Transformation T' from F' to G' yields a Natural Transformation  $T \circ T'$  from  $F \circ F'$  to  $G \circ G'$ . Expressing the associativity of this horizontal composition operation would need to identify, as types, say  $(F \circ F') \circ F''$  and  $F \circ (F' \circ F'')$ . But here we run into a problem. Although these two terms are equal in the sense of Functor equality, they are *not* definitionally equal, and thus we are unable to even write the statement of associativity of horizontal composition: it does not typecheck.

In order to circumvent this problem, we need to define a less constrained equality =%NTH between natural transformations as follows.

```
Definition EqualH_NT := [C,D:Category][F,G:(Functor C D)]
                        [F',G':(Functor C D)][T:(NT F G)][T':(NT F' G')]
                        (a:(0b C)) (ApNT T a) = "H (ApNT T' a).
Token "=%NTH".
Grammar command command1 := [ command0($T) "=%NTH" command0($T') ] ->
                            [$0 = <<(EqualH_NT ? ? ? ? ? $T $T')>>].
Section assoc_horz_comp.
Variable A,B,C,D: Category.
Variable F,G
                 : (Functor A B).
Variable F',G'
                 : (Functor B C).
Variable F'',G'': (Functor C D).
Variable T
                 : (NT F G).
Variable T'
                 : (NT F' G').
Variable T''
                : (NT F'', G'').
Lemma CompH_NT_assoc : ((T o%NTh T') o%NTh T'') =%NTH (T o%NTh (T' o%NTh T'')).
End assoc_horz_comp.
```

# 8 Yoneda's Embedding

We now construct a functor  $Y: C^{\circ} \longrightarrow CatFunct(C, SET)$  as follows:

- for every object a of C, Y(a) = Hom(a, -)
- for every morphism  $f^{\circ}: b \longrightarrow a$  of  $C^{\circ}$ ,  $Y(f^{\circ}) = H(f)$

Section yoneda\_functor.

```
Variable C : Category.
Section funy_map_def.
Variable a, b : (Ob C).
```

End funy\_map\_def.

Lemma FunY\_comp\_law : (fcomp\_law (Dual C) (CatFunct C SET) (funSET C) FunY\_map).

Lemma FunY\_id\_law : (fid\_law (Dual C) (CatFunct C SET) (funSET C) FunY\_map).

Lemma: Y is full and faithful.

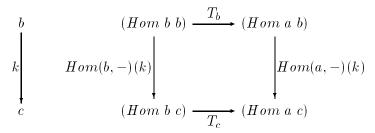
Lemma Y\_full : (IsFull FunY).

Lemma Y\_faithful : (IsFaithful FunY).

End yoneda\_functor.

This result may be obtained as corollary of a more general result known as Yoneda's lemma. We give here a direct proof. Let  $f, g: a \longrightarrow b$  be morphisms of  $C, Y(f^{\circ}) = Y(g^{\circ})$  implies that  $\forall c: C, \forall k: b \longrightarrow c, f \circ k = g \circ k$ . For  $k = Id_b$ , we obtain f = g (and thus  $f^{\circ} = g^{\circ}$ ). This shows that Y is faithful.

We now must show that Y is full. Let  $T: Y(b) \longrightarrow Y(a)$  (i.e.  $Hom(b, -) \longrightarrow Hom(a, -)$ ) be a natural transformation between hom-functors, we are looking for a morphism  $f: a \longrightarrow b$  such that  $T = Y(f^{\circ})$ . We know that  $T_b: (Hom\ b\ b) \longrightarrow (Hom\ a\ b)$  is a Map. We define  $f = T_b(Id_b)$ . Let us check that  $T = Y(f^{\circ})$ . This is equivalent to  $\forall c: C, \forall k: b \longrightarrow c, T_c(k) = T_b(Id_b) \circ k$ . This last assertion is a particular case of the naturality property of T:



Thus after simplification  $\forall g: b \longrightarrow b, T_c(g \circ k) = T_b(g) \circ k$ . Taking  $g = Id_b$ , we get  $T_c(k) = T_b(Id_b) \circ k$ . We conclude that Y is full.

This result shows that any natural transformation between hom-functors  $T: Hom(b, -) \longrightarrow Hom(a, -)$  is completely determined by a unique morphism in  $a \longrightarrow b$  (which is actually  $T_b(Id_b)$ ).

# 9 Conclusion

The development shown in this paper is but a tiny initial fragment of the theory of categories. However, it is quite promising, in that the power of dependent types and inductive types (or at least  $\Sigma$ -types) is put to full use; note in particular the dependent equality between morphisms of possibly non-convertible types.

We also point out that the syntactic facilities offered by the new version V5.10 of Coq are a first step in the right direction: the user may define his own notations through extensible grammars, types which are implicitly known by dependencies are synthesised automatically, and the macrodefinition facility (so-called Syntactic Definition) allows a certain amount of high-level notation.

In order to show how crucial these tools are, we give below the statement of the interchange law without syntactic abbreviations:

We are thus closing the gap with standard mathematical practice, although some supplementary overloading mechanisms are obviously still lacking in order to implement the usual "abus de notation".

From the point of view of user comfort of our proof assistant, two main difficulties have been encountered. The first one is due to the fact that Coq does very few automatic unfoldings of constants. We are too often required to make explicit manual unfoldings before being allowed to apply a lemma. The second difficulty is due to the lack of non-trivial automatic theorem proving tools. For instance, this development would benefit from efficient multi-relations rewriting tactics. These tactics ought to be extensible enough to allow the user specification of rewriting strategies, and generic enough to be usable in other developments. Such tactics have already been written for LCF [11] and NuPRL [9]. Their adaptation to Coq is currently under study.

This logical reconstruction of the basics of category theory follows initial attempts by R. Dyckhoff[5] in Martin-Löf type theory. It shows that intentional type theory is sufficient for developing this kind of mathematics, and we may thus hope to develop more sophisticated notions such as adjunction, which so far have been formally pursued only in extensional type theory[2]. Burstall and Rydeheard[12] have implemented a substantial number of concepts and constructions of category theory in SML (an ML dialect). The essential difference with our approach is that they do not include in their formalisation the properties (such as equations deriving from diagrams) of their categorical constructions. Thus they cannot mechanically check that their constructions have the intended properties. This exhibits the essential expressivity increase from a functional programming language with simple types to a type theory with dependent types, whose Curry-Howard interpretation includes the verification of predicate calculus conditions.

The above axiomatisation may indeed be pursued to include a significant segment of category theory. Thus A. Saïbi shows in [13] how to define adjunction and limits, develops standard constructions such as defining limits from equalisers and products, and shows the existence of left adjunct functors under the conditions of Freyd's theorem.

# References

- [1] P. Aczel. "Galois: A Theory Development Project." Turin workshop on the representation of mathematics in Logical Frameworks, January 1993.
- [2] J. A. Altucher and P. Panangaden. "A Mechanically Assisted Constructive Proof in Category Theory." In proceedings of CADE 10, Springer-Verlag LNCS no?, 1990.
- [3] R. Asperti and G. Longo. "Categories, Types, and Structures." MIT Press, 1991.
- [4] T. Coquand. "An analysis of Girard's paradox." Proceedings of LICS, Cambridge, Mass. July 1986, IEEE Press.
- [5] R. Dyckhoff. "Category theory as an extension of Martin-Löf type theory." Internal Report CS 85/3, Dept. of Computational Science, University of St. Andrews, Scotland.
- [6] R. Harper and R. Pollack. "Type checking with universes." Theoretical Computer Science 89, 1991.
- [7] M. Hofmann. "Elimination of extensionality in Martin-Löf type theory." Proceedings of work-shop TYPES'93, Nijmegen, May 1993. In "Types for Proofs and Programs", Eds. H. Barendregt and T. Nipkow, LNCS 806, Springer-Verlag 1994.
- [8] G. Huet. "Initiation à la Théorie des Catégories." Notes de Cours, DEA Paris 7, Nov. 1985.
- [9] P. B. Jackson. "Enhancing the NuPRL proof development system and applying it to computational abstract algebra." Ph.D. dissertation, Cornell University, Ithaca, NY, 1995.
- [10] S. Mac Lane. "Categories for the working mathematician". Springer-Verlag, 1971.
- [11] L. C. Paulson. "A higher order implementation of rewriting." Science of Comuter Programming, 3:119-149, 1983.
- [12] D. E. Rydeheard and R. M. Burstall. "Computational Category Theory". Prentice Hall, 1988.
- [13] A. Saïbi. "Une axiomatisation constructive de la théorie des catégories." Rapport de Recherche, en préparation.