9 Constructive Category Theory

Gérard Huet and Amokrane Saïbi

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 V6.1[4]. This paper gives the full transcript of the Coq axiomatization.

In this note we develop one possible axiomatization 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: Σ-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.

This research was partially supported by ESPRIT Basic Research Action "TYPES."

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

```
Section Orderings.

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 Partial_equivalence : Prop :=
 {Prf_trans : Transitive;
  Prf_sym   : Symmetric}.

Structure Equivalence : Prop :=
 {Prf_refl   :  Reflexive;
  Prf_pequiv :> Partial_equivalence}.

End Orderings.
```

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).
```

We now describe new syntactic facilities we used in the definitions above.

• To every declared object is associated the list of the positions of its *implicit arguments* [14]. These implicit arguments correspond to the arguments which can be deduced from the following ones. For instance the first argument `U` of `Reflexive` is implicit since it can be deduced from the type of `R:(Relation U)`. Thus when one applies these functions to arguments, one should omit the implicit ones. They are then automatically replaced by symbols "?" to be synthesized from the context. This way we use type synthesis to elide information which is implicit from the context. Thus it is sufficient to write `(Reflexive R)`.

The mechanism of synthesis of implicit arguments is not complete, so we have sometimes to give explicitly certain implicit arguments of an application. The syntax is *i* `!t` where *i* is the position of an implicit argument and `t` is its corresponding explicit term. We can also give all the arguments of an application, we have then to write `(!c t1...tn)`. For instance, we could write in a more explicit fashion `(Reflexive 1!U R)` or equivalently `(!Reflexive U R)`.

• Remark that in Coq Σ-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 `Partial_equivalence` into its constituent fields. It defines also a constructor `Build_Partial_equivalence` to build an object of type `Partial_equivalence` 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.

• The symbol `:>` in the definition of `Equivalence`, declares the constant `Prf_pequiv` as an *implicit coercion* [15] between the *classes* `Equivalence` and `Partial_equivalence`. The declared coercions are composed to build new coercions. Appropriate coercions are automatically inserted into ill-typed terms to make them well-typed. For instance, the term `(f e)` where `f:Partial_equivalence -> nat` and `e:Equivalence`, is transformed into `(f (Prf_pequiv e))`. Coercions are not always record projections, they may be any function with an appropriate type; they are then declared using the dedicated command `Coercion`. Remark also that coercions may have parameters, like the coercion `Prf_pequiv` after the end of the section.

```
Prf_pequiv : (U:Type)(R:(Relation U))(Equivalence R)
             -> (Partial_equivalence R)
```

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 [8]. This terminology is due to R. Burstall.

3.1 The Setoid Structure

```
Structure Setoid : Type :=
 {Carrier   :> Type;
  Equal     :  (Relation Carrier);
  Prf_equiv :> (Equivalence Equal)}.

Coercion Build_Setoid : Equivalence >-> Setoid.
```

The constant `Carrier` is a coercion from `Setoid` to the abstract class `SORTCLASS`, the class of sorts (`Type` and `Prop`). These coercions are used in the situations where sorts are expected, like in λ-abstractions and dependent types.

To have a more pleasing notation, we first introduce a concrete syntax declaration allowing to parse `(Carrier A)` as `|A|`:

```
Grammar command command1 := [ "|" command0($s) "|"]
                              -> [$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` with the precedence 2, the parameter `A` being synthesised by type-checking.

```
Infix 2 "=%S" Equal.
```

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

The last extracted field is the proof that the equality of a Setoid is an equivalence relation.

3.2 An Example

As example of the preceding notions, let us define the Setoid of natural numbers. The type of its elements can be directly inductively defined.

```
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. We get this proof 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.

```
Lemma Eq_Nat_equiv : (Equivalence Eq_Nat).

Definition Set_of_nat : Setoid := Eq_Nat_equiv.
```

In the last definition, we imposed the type of the constant. Since its inferred type (i.e., `(Equivalence Eq_Nat)`) is different from the declared one (i.e., `Setoid`), the system inserts an implicit coercion `Build_Setoid`.

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:

```
Structure PSetoid : Type :=
 {PCarrier  :> Type;
  Coherence :  (Relation PCarrier);
  Prf_PER   :> (Partial_equivalence Coherence)}.

Coercion Build_PSetoid : Partial_equivalence >-> PSetoid.

Definition Total := [A:PSetoid][x:A](Coherence x x).
```

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. Thanks to implicit coercions, we can write `A->B` instead of `|A|->|B|`. Remark also the use of generic equality in `Map_law` below.

```
Section maps.

Variables A, B : Setoid.

Definition Map_law := [f:A->B]
    (x,y:A) x =%S y -> (f x) =%S (f y).

Structure Map : Type := Build_Map
 {Ap   :> A->B;
  Pres :> (Map_law Ap)}.

Coercion Build_Map : Map_law >-> Map.
```

A Map `m` over `A` and `B` is thus similar to a pair, packing a function `(Ap m)` (of type `|A|->|B|`) with the proof `(Pres m)` that this function respects equality. The constant `Ap` is also a coercion from `Map` to `FUNCLASS`, the class of functions. It allows us to write `(f x)` where `f:Map` instead of `(Ap f x)`.

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

```
Definition Ext := [f,g:Map](x:A) (f x) =%S (g x).

Lemma Ext_equiv : (Equivalence Ext).

Definition Map_setoid : Setoid := Ext_equiv.

End maps.
```

We write `f =%M g` for $f = g$.

```
Infix 2 "=%M" Ext.
Infix Assoc 6 "==>" Map_setoid.
```

This last command allows us to write `A==>B`, with appropriate precedence level, for the Setoid of Maps between Setoids `A` and `B`. It associates to the right.

A (curried) binary application `Map2` between Setoids `A`, `B` and `C` is then simply a Map between `A` and `B==>C`. We prove that a binary function verifying the congruence laws for both arguments is equivalent to a binary application.

```
Section fun2_to_map2.

Variable A, B, C : Setoid.

Definition Map2 := (Map A (B ==> C)).

Variable f : A -> B -> C.

Definition Map2_congl_law := (b1,b2:B)(a:A)
                             (b1 =%S b2) -> (f a b1)
                             =%S  (f a b2).

Definition Map2_congr_law := (a1,a2:A)(b:B)
                             (a1 =%S a2) -> (f a1 b)
                             = %S (f a2 b).

Definition Map2_cong_law := (a1,a2:A)(b1,b2:B)
                            (a1 =%S a2) -> (b1 =%S b2)
                            ->  (f a1 b1) =%S (f a2 b2).

Hypothesis pgcl : Map2_congl_law.
Hypothesis pgcr : Map2_congr_law.

Lemma Map2_map_law1 : (a:A)(Map_law (f a)).

Definition Map2_map1 : A -> (B==>C)
                     := [a:A] (Map2_map_law1 a).

Lemma Map2_map_law2 : (Map_law Map2_map1).

Definition Build_Map2 : Map2 := (Build_Map Map2_map_law2).
```

```
End fun2_to_map2.
Section prop_map2.

Variable A, B, C : Setoid.
Variable f       : (Map2 A B C).

Definition Ap2 := [a:A][b:B] ((f a) b).

Lemma Prf_map2_congl : (Map2_congl_law Ap2).

Lemma Prf_map2_congr : (Map2_congr_law Ap2).

Lemma Prf_map2_cong : (Map2_cong_law Ap2).

End prop_map2.

Coercion Ap2 : Map2 >-> FUNCLASS.

Identity Coercion Map2_Map : Map2 >-> Map.
```

We end this section by defining a generic `Ap2`, denoting the application function associated with a (curried) binary application, useful for what follows. The `Identity Coercion` builds an identity function between the specified classes and declares it as a coercion.

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. We write `Hom` as infix `-->`.

```
Section cat.

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

Infix 6 "-->" Hom.
```

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

```
Variable Op_comp : (a,b,c:Ob)(Map2 a-->b b-->c a-->c).

Definition Cat_comp := [a,b,c:Ob][f:a --> b][g:b --> c]
                       ((Op_comp a b c) f g).

Infix Assoc 6 "o" Cat_comp.
```

Composition is assumed to be associative:

```
Definition Assoc_law := (a,b,c,d:Ob)(f:a --> b)(g:b --> c)
                        (h:c --> d)
                        (f o (g o h)) =%S ((f o g) o h).
```

The final component of a category is, for every object `a`, an arrow in `a --> a` which is an identity for composition:

```
Variable Id : (a:Ob) a-->a.

Definition Idl_law := (a,b:Ob)(f:a --> b)
                      ((Id ?) o f) =%S f.

Definition Idr_law := (a,b:Ob)(f:b --> a) f =%S
                      (f o (Id ?)).

End cat.
```

We are now able to define synthetically a Category:

```
Structure Category : Type :=
 {Ob      :> Type;
  Hom     :  Ob -> Ob -> Setoid;
  Op_comp :  (a,b,c:Ob)(Map2 (Hom a b) (Hom b c) (Hom a c));
  Id      :  (a:Ob)(Hom a a);
  Prf_ass :  (Assoc_law Op_comp);
  Prf_idl :  (Idl_law Op_comp Id);
  Prf_idr :  (Idr_law Op_comp Id)}.
```

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

```
Definition Comp := [C:Category](Cat_comp (!Op_comp C)).

Infix 6 "-->" Hom.
Infix Assoc 6 "o" Comp.
```

Remark that we now use the infix notation `o` in the context of a local Category parameter `C`. It must be noticed that Infix notations 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.

Variable A    : Type.
Variable H    : A -> A -> Setoid.
Variable Cfun : (a,b,c:A)(H a b) -> (H b c) -> (H a c).

Definition Congl_law := (a,b,c:A)(f,g:(H b c))(h:(H a b))
           f =%S g -> (Cfun h f) =%S (Cfun h g).

Definition Congr_law := (a,b,c:A)(f,g:(H a b))(h:(H b c))
           f =%S g -> (Cfun f h) =%S (Cfun g h).

Definition Cong_law := (a,b,c:A)(f,f':(H a b))(g,g':(H b c))
           f =%S f' -> g =%S g' -> (Cfun f g)
           =%S (Cfun f' g').

Hypothesis pcgl : Congl_law.
Hypothesis pcgr : Congr_law.

Variable a, b, c : A.
```

```
Definition Build_Comp := (Build_Map2 (!pcgl a b c)
                         (!pcgr a b c)).

End composition_to_operator.
```

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:

```
Section cat_cong.

Variable C : Category.

Lemma Prf_congl : (Congl_law (!Comp C)).

Lemma Prf_congr : (Congr_law (!Comp C)).

Lemma Prf_cong : (Cong_law (!Comp C)).

End cat_cong.
```

4.2 Hom Equality

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 `f =%H g` for the equality of arrows `f` and `g`.

```
Inductive Equal_hom [C:Category;a,b:C;f:a --> b]
  : (c,d:C)(c --> d) -> Prop := Build_Equal_hom
  : (g:a --> b) f =%S g -> (Equal_hom C a b f a b g).

Infix Assoc 6 "=%H" Equal_hom.
```

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:a --> b)` and `(g: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 : C.
Variable f    : a --> b.

Lemma Equal_hom_refl : f =%H f.

Variable c, d : C.
Variable g    : c --> d.

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

Variable i, j : C.
Variable h    : 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° 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° iff $f : b \longrightarrow a$ is a morphism of C.

```
Section d_cat.

Variable C : Category.

Definition DHom := [a,b:C]b --> a.
```

Composition is defined as expected: $f^\circ \circ^\circ g^\circ = (g \circ f)^\circ$. Identity arrows are invariant. We then check the category laws.

```
Definition Comp_Darrow := [a,b,c:C][df:(DHom a b)]
                          [dg:(DHom b c)] dg o df.

Lemma Comp_dual_congl : (Congl_law Comp_Darrow).

Lemma Comp_dual_congr : (Congr_law Comp_Darrow).

Definition Comp_Dual
 := (Build_Comp Comp_dual_congl Comp_dual_congr).

Lemma Assoc_Dual : (Assoc_law Comp_Dual).

Lemma Idl_Dual : (Idl_law 2!DHom Comp_Dual (!Id C)).

Lemma Idr_Dual : (Idr_law 2!DHom Comp_Dual (!Id C)).
```

We write `(Dual C)` for the dual category of `C`.

```
Definition Dual
 := (Build_Category Assoc_Dual Idl_Dual Idr_Dual).

End d_cat.

Syntactic Definition DHOM := (!Hom (Dual ?)).
```

The combinator `DHOM` 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 `DHOM` by `(!Hom (Dual ?))`.

4.4 Category Exercises

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_monic_def.

Variable C    : Category.
Variable a, b : C.

Definition Epic_law := [f:a --> b](c:C)(g,h:b --> c)
                       (f o g) =%S (f o h) -> g =%S h.

Structure Epic : Type :=
 {Epic_mor   :  a --> b;
  Prf_isEpic :> (Epic_law Epic_mor)}.

Coercion Build_Epic : Epic_law >-> Epic.
```

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 $g = h$.

```
Definition Monic_law := [f:b --> a](c:C)(g,h:c --> b)
                        (g o f) =%S (h o f) -> g =%S h.

Structure Monic : Type :=
 {Monic_mor   :  b --> a;
  Prf_isMonic :> (Monic_law Monic_mor)}.

Coercion Build_Monic : Monic_law >-> Monic.

End epic_monic_def.
```

Two opposite morphisms $f : a \longrightarrow b$ and $f^{-1} : b \longrightarrow a$ are iso if $f^{-1} \circ f = Id_b$ and $f \circ f^{-1} = Id_a$.

```
Section iso_def.

Variable C : Category.

Definition RIso_law := [a,b:C][f:a --> b][f1:b --> a]
                       (f1 o f) =%S (Id b).

Variable a, b : C.

Definition AreIsos := [f:a --> b][f1:b --> a]
                      (RIso_law f f1)/\ (RIso_law f1 f).
```

We now say that two objects a and b are isomorphic ($a \cong b$) if they are connected by an isomorphism.

```
Structure Iso : Type :=
 {Iso_mor : a --> b;
  Inv_iso : b --> a;
  Prf_Iso :> (AreIsos Iso_mor Inv_iso)}.

Coercion Build_Iso : AreIsos >-> Iso.
```

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*). We should explicitely provide the function giving this unique morphism.

```
Section initial_def.

Variable C : Category.

Definition IsInitial := [a:C][h:(b:C) a --> b](b:C)
                        (f:a --> b) f =%S (h b).

Structure Initial : Type :=
 {Initial_ob    :  C;
  MorI          : (b:C) Initial_ob --> b;
  Prf_isInitial :> (IsInitial MorI)}.

Coercion Build_Initial : IsInitial >-> Initial.
```

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

```
Lemma I_unic : (i1,i2:Initial)(Iso (Initial_ob i1)
               (Initial_ob i2)).
End initial_def.
```

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.

Definition IsTerminal := [b:C][h:(a:C) a --> b](a:C)
                         (f:a--> b) f =%S (h a).

Structure Terminal : Type :=
  {Terminal_ob    :  C;
   MorT           :  (a:C) a --> Terminal_ob;
   Prf_isTerminal :> (IsTerminal MorT)}.

Coercion Build_Terminal : IsTerminal >-> Terminal.

End terminal_def.
```

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°.

```
Lemma Initial_dual : (C:Category)(a:C)(h:(b:C) a --> b)
                     (IsInitial h) ->
                     (IsTerminal 1!(Dual C) h).

Coercion Initial_dual : IsInitial >-> IsTerminal.
```

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.

Variable A, B, C : Setoid.
Variable f       : A ==> B.
Variable g       : B ==> C.

Definition Comp_fun := [x:A](g (f x)).

Lemma Comp_fun_map_law : (Map_law Comp_fun).

Definition Comp_map : A ==> C := Comp_fun_map_law.

End mcomp.

Infix Assoc 6 "o%M" Comp_map.
```

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 Comp_map).

Lemma Comp_map_congr : (Congr_law Comp_map).

Definition Comp_SET
 := (Build_Comp Comp_map_congl Comp_map_congr).
```

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

```
Lemma Assoc_SET : (Assoc_law 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 : A ==> A := Id_fun_map_law.

End id_map_def.

Definition Id_SET := Id_map.

Lemma Idl_SET : (Idl_law Comp_SET Id_SET).

Lemma Idr_SET : (Idr_law Comp_SET Id_SET).
```

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

```
Definition SET := (Build_Category Assoc_SET Idl_SET
                   Idr_SET).

Syntactic Definition MAP := (!Hom 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 funct_def.

Variable C, D : Category.

Section funct_laws.

Variable FOb  : C -> D.
Variable FMap : (a,b:C)(Map (a --> b) ((FOb a)
                 --> (FOb b))).
```

Functors must preserve the Category structure, and thus verify the two laws: $F(f \circ g) = F(f) \circ F(g)$ and $F(Id_a) = Id_{F(a)}$.

```
Definition Fcomp_law := (a,b,c:C)(f:a --> b)(g:b --> c)
```

```
   ((FMap a c) (f o g)) =%S (((FMap a b) f) o
   ((FMap b c) g)).

Definition Fid_law := (a:C)((FMap a a) (Id a)) =%S
                      (Id (FOb a)).

End funct_laws.

Structure Functor : Type :=
  {FOb           :> C -> D;
   FMap          :  (a,b:C)(Map (a --> b) ((FOb a)
                    -->  (FOb b)));
   Prf_Fcomp_law :  (Fcomp_law FMap);
   Prf_Fid_law   :  (Fid_law  FMap)}.
```

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

```
Definition FMor := [F:Functor][a,b:C][f:a --> b]
                   ((FMap F a b) 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:

```
Definition Equal_Functor := [F,G:Functor](a,b:C)(f:a --> b)
                            (FMor F f) =%H (FMor G f).

End funct_def.

Infix 2 "=%F" Equal_Functor.

Lemma Equal_Functor_equiv : (C,D:Category)
                           (Equivalence (!Equal_Functor C D)).
```

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

```
Definition Functor_setoid
  := [C,D:Category](Build_Setoid (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 : C.

Definition FunSET_ob := [b:C] 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 : C.

Section funset_mor_def.

Variable f : b --> c.

Definition FunSET_mor1 := [g:a --> b] (g o f).

Lemma FunSET_map_law1 : (Map_law FunSET_mor1).

Definition FunSET_mor : (FunSET_ob b) ==> (FunSET_ob c)
                                        := FunSET_map_law1.

End funset_mor_def.

Lemma FunSET_map_law : (Map_law FunSET_mor).

Definition FunSET_map : (b --> c)
  ==> (MAP (FunSET_ob b) (FunSET_ob c)) := 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 FunSET_map).

Lemma Fun_id_law : (Fid_law FunSET_map).

Definition FunSET := (Build_Functor Fun_comp_law Fun_id_law).

End funset.
```

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 `○%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_FOb := [a:C](H (G a)).

 Section comp_functor_map.

 Variable a, b : C.

 Definition Comp_FMor := [f:a --> b](FMor H (FMor G f)).

 Lemma Comp_FMap_law : (Map_law Comp_FMor).
```

```
 Definition Comp_FMap : (Map (a --> b) ((Comp_FOb a) -->
                        (Comp_FOb b))) := Comp_FMap_law.

 End comp_functor_map.

Lemma Comp_Functor_comp_law : (Fcomp_law Comp_FMap).

Lemma Comp_Functor_id_law : (Fid_law Comp_FMap).

Definition Comp_Functor
  := (Build_Functor Comp_Functor_comp_law
      Comp_Functor_id_law).

End Comp_F.

Infix Assoc 6 "o%F" Comp_Functor.
```

As before, we construct a composition operator after checking the Congruence laws.

```
Lemma Comp_Functor_congl
  : (Congl_law 2!Functor_setoid Comp_Functor).

Lemma Comp_Functor_congr
  : (Congr_law 2!Functor_setoid Comp_Functor).

Definition Comp_CAT
  := (Build_Comp Comp_Functor_congl Comp_Functor_congr).

Lemma Assoc_CAT : (Assoc_law 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:C]a.
```

```
Definition Id_CAT_map := [a,b:C](Id_map (a --> b)).

Lemma Id_CAT_comp_law : (Fcomp_law 3!Id_CAT_ob Id_CAT_map).

Lemma Id_CAT_id_law : (Fid_law 3!Id_CAT_ob Id_CAT_map).

Definition Id_CAT
  := (Build_Functor Id_CAT_comp_law Id_CAT_id_law).

End idCat.
```

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.

```
Structure Category' : Type :=
  {Ob'      :> Type;
   Hom'     :  Ob' -> Ob' -> Setoid;
   Opcomp'  :  (a,b,c:Ob')(Map2 (Hom' a b) (Hom' b c)
               (Hom' a c));
   Id'      :  (a:Ob')(Hom' a a);
   Prf_ass' :  (Assoc_law Opcomp');
   Prf_idl' :  (Idl_law Opcomp' Id');
   Prf_idr' :  (Idr_law Opcomp' Id')}.

Definition CAT : Category' := (Build_Category'
                                Assoc_CAT Idl_CAT Idr_CAT).
```

Note that here we make an essential use of the universe's 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 [5] 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 [7]. 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., $a \cong b$ implies $F(a) \cong F(b)$.

```
Section functor_prop.

Variable C, D : Category.

Lemma Functor_preserves_iso : (F:(Functor C D))(a,b:C)
                              (Iso a b)->(Iso (F a)(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 := [F:(Functor C D)](a,b:C)
                           (f,g:a --> b)((FMor F f)
                           =%S (FMor F g)) -> (f =%S g).

Structure Faithful : Type :=
 {Faithful_functor :> (Functor C D);
  Prf_isFaithful   :> (Faithful_law Faithful_functor)}.

Coercion Build_Faithful : Faithful_law >-> Faithful.
```

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 := [F:(Functor C D)][H:(a,b:C)((F a)
                       --> (F b)) -> (a --> b)]
                       (a,b:C)(h:(F a) --> (F b))
                       h =%S (FMor F (H a b h)).

Structure Full : Type :=
 {Full_functor :> (Functor C D);
  Full_mor     : (a,b:C)((Full_functor a)
                 --> (Full_functor b)) -> (a --> b);
  Prf_isFull   :> (Full_law Full_mor)}.
```

```
Coercion Build_Full : Full_law >-> Full.

End functor_prop.
```

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 : (Faithful_law F) -> (Faithful_law G)
                         -> (Faithful_law (F o%F G)).

Variable F1      : (a,b:C)((F a) --> (F b)) -> (a --> b).
Variable G1      : (a,b:D)((G a) --> (G b)) -> (a --> b).

Lemma IsFull_comp : (Full_law F1) -> (Full_law G1) ->
                    [H1=[a,b:C][h:(G (F a))
                    --> (G (F b))](F1 (G1 h))]
                    (Full_law 3!(F o%F G) H1).

End comp_functor_prop.
```

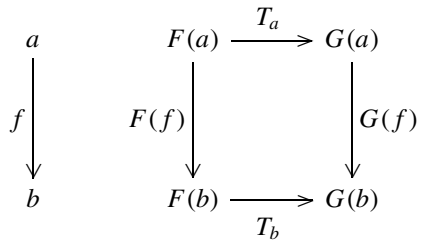
In the definition of `IsFull_comp`, `H1` is a local constant.

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 the type of objects are not given as a Setoid.

```
Section nt_def.

Variable C, D : Category.
Variable F, G : (Functor C D).

Definition NT_law := [T:(a:C)(F a) --> (G a)] (a,b:C)
                     (f:a --> b)((FMor F f) o (T b))
                     =%S ((T a) o (FMor G f)).

Structure NT : Type :=
  {ApNT       :> (a:C)(F a) --> (G a);
   Prf_NT_law :> (NT_law ApNT)}.

End nt_def.

Coercion Build_NT : NT_law >-> NT.
```

We now define 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: $\forall a \cdot T_a = T'_a$. As previously, we write `=%NT` for this equality.

```
Section setoid_nt.

Variable C, D : Category.
Variable F, G : (Functor C D).

Definition Equal_NT := [T,T':(NT F G)](a:C)(T a) =%S (T' a).
```

```
Lemma Equal_NT_equiv : (Equivalence Equal_NT).

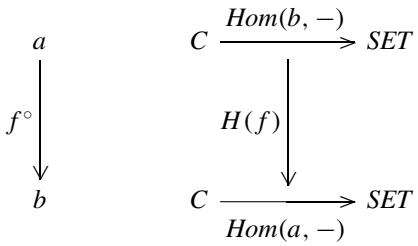
Definition NT_setoid : Setoid := Equal_NT_equiv.

End setoid_nt.

Infix 2 "=%NT" Equal_NT.
```

6.2 An Example of Natural Transformation

Let C be any category, and $f : a \longrightarrow a$ be a 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 : C.
Variable f    : a --> b.

Section nth_map_def.

Variable c : C.

Definition NtH_arrow := [h:b --> c] f o h.

Lemma NtH_map_law : (Map_law NtH_arrow).
```

```
Definition NtH_map : (MAP ((FunSET b) c) ((FunSET a) c))
                     := 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)`.

```
Lemma NtH_nt_law : (NT_law 3!(FunSET b) 4!(FunSET a)
                    NtH_map).

Definition NtH : (NT (FunSET b) (FunSET a)) := NtH_nt_law.

End funset_nt.
```

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_setoid F G).
Variable T'      : (NT_setoid G H).

Definition Comp_tau := [a:C] (T a) o (T' a).

Lemma Comp_tau_nt_law : (NT_law [a:C](Comp_tau a)).

Definition CompV_NT : (NT_setoid F H) := Comp_tau_nt_law.
```

```
End compnt.

Lemma CompV_NT_congl : (Congl_law CompV_NT).

Lemma CompV_NT_congr : (Congr_law CompV_NT).

Definition Comp_CatFunct
   := (Build_Comp CompV_NT_congl CompV_NT_congr).

Lemma Assoc_CatFunct : (Assoc_law Comp_CatFunct).
```

To every functor F, we associate an identity natural transformation Id_F defined as $\lambda a \cdot Id_{F(a)}$:

```
Section id_catfunct_def.

Variable F : (Functor C D).

Definition Id_CatFunct_tau := [a:C](Id (F a)).

Lemma Id_CatFunct_nt_law: (NT_law Id_CatFunct_tau).

Definition Id_CatFunct : (NT_setoid F F)
                         := Id_CatFunct_nt_law.

End id_catfunct_def.

Lemma Idl_CatFunct : (Idl_law Comp_CatFunct Id_CatFunct).

Lemma Idr_CatFunct : (Idr_law Comp_CatFunct Id_CatFunct).
```

Having checked that we have all categorical properties, we may now define the functor category.

```
Definition CatFunct
  :=(Build_Category Assoc_CatFunct Idl_CatFunct
      Idr_CatFunct).

End cat_functor.
```

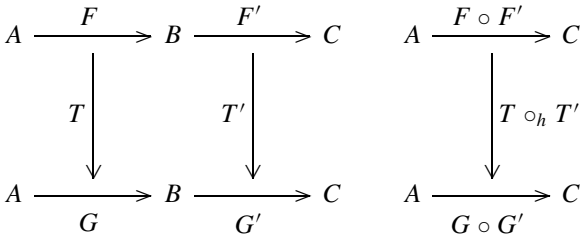
```
Infix Assoc 6 "o%NTv" CompV_NT.

Syntactic Definition NT1 := (!Hom (CatFunct ? ?)).
```

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.

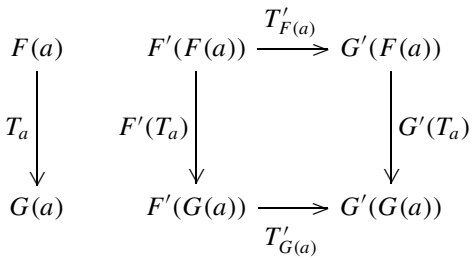
Variable A, B, C : Category.
Variable F, G    : (Functor A B).
Variable F', G'  : (Functor B C).
Variable T       : (NT F G).
Variable T'      : (NT F' G').

Definition Ast : (a:A)((F o%F F') a) --> ((G o%F G') a) :=
           [a:A](T' (F a)) o (FMor G' (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:A) ((FMor F' (T a)) o (T' (G a))) =%S
                     ((T' (F a)) o (FMor G' (T a))).

Lemma Ast_nt_law : (NT_law Ast).

Definition CompH_NT : (NT (F o%F F') (G o%F G'))
                    := Ast_nt_law.

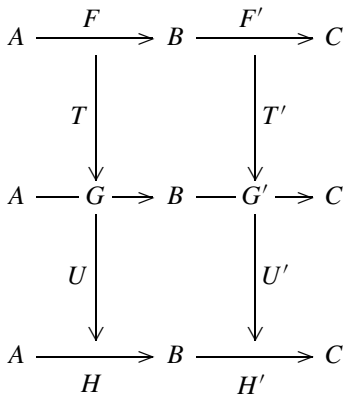
End horz_comp.
```

We shall write `o%NTh` for horizontal composition.

```
Infix Assoc 6 "o%NTh" CompH_NT.
```

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 F', G', H' : (Functor B C).
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,F',G':
                        (Functor C D)]
                        [T:(NT F G)]
                        [T':(NT F' G')]
                        (a:C)
                        (T a) =%H (T' a).
```

```
Infix 2 "=%NTH" EqualH_NT.

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°, $Y(f^{\circ}) = H(f)$

```
Section yoneda_functor.

Variable C : Category.

Section funy_map_def.

Variable a, b : C.

Lemma FunY_map_law : (Map_law 2!(NT1 (FunSET a)
                     (FunSET b)) (!NtH C a b)).

Definition FunY_map : (MAP (DHOM a b) (NT1 (FunSET a)
                      (FunSET b))) := FunY_map_law.
```

```
End funy_map_def.

Lemma FunY_comp_law : (Fcomp_law 1!(Dual C) FunY_map).

Lemma FunY_id_law : (Fid_law 1!(Dual C) FunY_map).

Definition FunY := (Build_Functor FunY_comp_law FunY_id_law).
```

Lemma : Y is full and faithful.

```
Lemma Y_full : [f=[a,b:C][h:(NT1 (FunSET a) (FunSET b))]
               ((h a) (Id a))] (Full_law 3!FunY f).

Lemma Y_faithful : (Faithful_law 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:

$$\begin{array}{ccc} b & & (Hom\ b\ b) \xrightarrow{T_b} (Hom\ a\ b) \\ \Big\downarrow k & & \Big\downarrow Hom(b,-)(k) \qquad\qquad \Big\downarrow Hom(a,-)(k) \\ c & & (Hom\ b\ c) \xrightarrow[T_c]{} (Hom\ a\ c) \end{array}$$

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 Σ-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 V6.1 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, the macro-definition facility (so-called `Syntactic Definition`) and the implicit coercions allow a certain amount of high-level notation. For this initial development of Category Theory, the implicit coercions are not strictly necessary, and in the initial version of this work, this facility was lacking. But further developments [14] showed that this facility was absolutely necessary for scaling up.

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

```
(Equal_NT A C (Comp_Functor A B C F F')
  (Comp_Functor A B C H H')
          (CompV_NT A C (Comp_Functor A B C F F')
            (Comp_Functor A B C G G')
                    (Comp_Functor A B C H H')
                       (CompH_NT A B C F G F' G' T T')
                     (CompH_NT A B C G H G' H' U U'))
          (CompH_NT A B C F H F' H' (CompV_NT A B F G H T U)
                    (CompV_NT B C F' G' H' T' U'))))
```

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 [12] and NuPRL [10]. Their adaptation to Coq is currently under study.

This logical reconstruction of the basics of category theory follows initial attempts by R. Dyckhoff [6] 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 [13] 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 [14] how to define adjunction and limits, develops standard constructions such as defining limits from equalisers and products, and shows the existence of left adjoint 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, LNAI 449, pp. 500–513, Springer-Verlag 1990.

[3] R. Asperti and G. Longo. "Categories, Types, and Structures." MIT Press, 1991.

[4] B. Barras and al. "The Coq Proof Assistant Reference Manual, version V6.1." Technical Report in Preparation, 1997.

[5] T. Coquand. "An analysis of Girard's paradox." Proceedings of LICS, Cambridge, Mass. July 1986, IEEE Press.

[6] 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.

[7] R. Harper and R. Pollack. "Type checking with universes." *Theoretical Computer Science* 89, 1991.

[8] M. Hofmann. "Elimination of extensionality in Martin-Löf type theory." Proceedings of workshop TYPES'93, Nijmegen, May 1993. In "Types for Proofs and Programs," Eds. H. Barendregt and T. Nipkow, LNCS 806, Springer-Verlag 1994.

[9] G. Huet. “Initiation à la Théorie des Catégories.” Notes de Cours, DEA Paris 7, Nov. 1985.

[10] P. B. Jackson. “Enhancing the NuPRL proof development system and applying it to computational abstract algebra.” Ph.D. dissertation, Cornell University, Ithaca, NY, 1995.

[11] S. Mac Lane. “Categories for the working mathematician.” Springer-Verlag, 1971.

[12] L. C. Paulson. “A higher order implementation of rewriting.” *Science of Computer Programming,* 3:119–149, 1983.

[13] D. E. Rydeheard and R. M. Burstall. “Computational Category Theory.” Prentice Hall, 1988.

[14] A. Saïbi. “Algèbre constructive en théorie des types. Outils génériques de modélisation et de démonstration. Application à la théorie des catégories.” PhD Thesis, Université Paris VI, March 1999.

[15] A. Saïbi. “Typing algorithm in type theory with inheritance.” In Proceedings 24th POPL, pp. 292–301, ACM Press, Paris 1997.