Category Theory in Coq

Alexandra Carvalho

Diploma Thesis

Supervised by Amílcar Sernadas and Paulo Mateus

May 1998

Abstract

Herein we formalize a segment of category theory using the implementation of Calculus of Inductive Construction in Coq. Adopting the axiomatization proposed by Huet and Saïbi we start by presenting basic concepts, examples and results of category theory in Coq. Next we define adjunction and cocartesian lifting and establish some results using the Coq proof assistant. Finally we remark that the axiomatization proposed by Huet and Saïbi is not good when dealing with the equality for objects.

Acknowledgments

Special thanks to:

Professor Amílcar Sernadas for his enthusiasm in supervising this thesis and for being always a good source of ideas;

Professor Cristina Sernadas for her advice and assistance;

Paulo for his support and motivation during many stages of this work;

Sara for her company, patience and being always willing to help;

Carlos, Jaime and Nuno for our fruitful talks;

All section 84 for the excellent working environment;

My parents for their patience;

Mia for being my furry friend.

This work was partially supported by the PRAXIS XXI Program and FCT, as well as by PRAXIS XXI Projects 2/2.1/MAT/262/94 SitCalc, PCEX/P/MAT/46/96 ACL plus 2/2.1/TIT/1658/95 LogComp, and ESPRIT IV Working Groups 22704 ASPIRE and 23531 FIREworks.

Contents

Notation

We use the following notation to denote variables.

Categories	`c`, `d`, `e`
Functors	`fF`, `fG`, `fH`, `fF'`, `fG'`, `fH'`
Morphisms	`f`, `g`, `h` and `v`, `u`, `w`
Natural Transformations	`nt`, `nt'`
Presetoids	`p`, `p'`, `p''`
Propositional Formulae	`pf`, `pf'`, `pf''`
Propositional Signatures	`sig`, `sig'`
Propositional Symbols	`ps`, `ps'`, `ps''`
Propositional Theories	`pt`, `pt'`, `pt''`
Setoids	`s`, `s'`, `s''`
Sets	`a`, `a'`, `a''`
Sets of Propositional Formulae	`gamma`, `gamma'`
Valuations	`val`

Chapter 1

Introduction

Constructive type theory has been shown to be adequate for representing categorical reasoning. In this work we use Calculus of Inductive Constructions as implemented in Coq V6.1 to formalize a segment of category theory. We follow the axiomatization proposed by Huet and Saïbi (see [HS95]) where objects are modeled as types and hom-sets as hom-setoids.

We start this work by first presenting the axiomatization of the notion of category proposed by Huet and Saïbi. Afterwards we define some examples to illustrate the previous axiomatization. Finally we present basic concepts and results related to this notions.

In the second part of this work we define adjunction situation in Coq. Adjunction was already defined in Coq by Saïbi (see for instance [Saï95]) however we choose an alternative definition following [AHS90]. After we present some examples of adjunction situation. We end this chapter showing two results concerning adjunctions. First we prove that a functor G has left adjoint iff for any object X the comma category $X \downarrow G$ has initial object. Second we prove that the left adjoint of a functor is unique up to natural isomorphism.

In the third part of this work we define cocartesian lifting in Coq. As usual we provide some examples and show that the codomain of a cocartesian lifting is unique up to isomorphism. In addition we remark that the axiomatization proposed by Huet and Saïbi is not good when dealing with the equality for objects.

During these three last chapters we just give the statement of the lemmas without the proof scripts. In appendix we present a proof in detail explaining all the tactics used. We also present all the Coq source code and the trace of proofs in appendix.

We assume that the reader is familiar with the basics of category theory and considering that the Coq notation is similar to the standard mathematical notation, we only explain the semantics of some Coq commands that are less intuitive (for more details see [PM96]).

Chapter 2

A Crash Introduction to Coq

The Coq notation is similar to the usual mathematical notation. It is however worthwhile to make two remarks. First, the universal quantification is denoted by parenthesis, so for instance `(x:T)(P x)` stands for $\forall_{x \in T} P(x)$. Second, the functional abstraction is denoted by square brackets, so for instance `[x:T](f x)` stands for $\lambda x \in T.f(x)$.

2.1 Binary Relations

In this section we define binary relations that are central to define the category theory in Coq. We start by introducing some Coq commands whose semantics may not be trivial.

When we apply arguments to a term it is common that some arguments can be determined by other arguments. So, for the sake of simplicity, we would like to apply only the latter ones. The Coq system allows us to do this by calling the implicit arguments mode.

```
Implicit Arguments On.
```

The Coq section is a modular mechanism to organize the source. All notions defined in the body of a section can be used outside, with the small difference that we have to parameterize this notions by the variables on which they depend. In the sequel, we present the section where we define binary relations and related properties.

```
Section BinRel.

Variable t: Type.

Definition Relation:= t→t→Prop.

Variable r: Relation.

Definition Reflexive:= (x:t)(r x x).
```

```
Definition Transitive:= (x,y,z:t)(r x y)→(r y z)→(r x z).

Definition Symmetric:= (x,y:t)(r x y)→(r y x).
```

The macro `Structure` generates an inductive definition with one constructor and defines the projection functions for each field. It also defines a constructor `Build_`*ident* where *ident* stands for the name of the `Structure`.

```
Structure Equivalence:  Prop:= {
  Prf_refl  : Reflexive;
  Prf_trans : Transitive;
  Prf_sym   : Symmetric
}.
```

In this case the projections are `Prf_refl`, `Prf_trans` and `Prf_sym`, thus for instance, given an `Equivalence equiv` its proof of reflexivity is `(Prf_refl equiv)`. To build an `Equivalence` from its constituents we use the constructor `Build_Equivalence`.

```
End BinRel.
```

2.2 Setoids

To define a category as general as possible the objects and the morphisms can not be sets, or else we are only defining small categories. One possible axiomatization of category theory in Coq that solves this problem was proposed by Huet and Saïbi (see [HS95]). In this work we adopt this solution and so we start by defining the structure `Setoid`.

2.2.1 The Setoid Structure

Setoids are triples composed of a type `Carrier`, a relation `Equal` over `Carrier` and a proof that `Equal` is an equivalence relation. It is usual in mathematics to overload the notation when the context is clear. The Coq system allows us to overload the notation by using coercions. In a `Structure` a coercion is defined by the symbol >. In our case, when we declare a `Setoid s`, Coq treats `s` as a `Setoid` or as its `Carrier`, depending of the context.

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

All operators in Coq are prefix however it is more natural for some operators to be infix. The `Infix` command defines a prefix operator, like `Equal`, as infix.

We start to declare =%S as a new token, since it is not predefined. Thus for each s1 and s2 in a Setoid s, s1 =%S s2 stands for (Equal s1 s2).

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

The last field of a Setoid is a proof that its equality is an equivalence relation. Thus it is trivial to obtain the corollaries of reflexivity, symmetry and transitivity.

```
Lemma Equal_refl:  (s:Setoid)(s1:s) s1 =%S s1.

Lemma Equal_sym:  (s:Setoid)(s1,s2:s) s1 =%S s2 → s2 =%S s1.

Lemma Equal_trans:  (s:Setoid)(s1,s2,s3:s)
  s1 =%S s2 → s2 =%S s3 → s1 =%S s3.
```

2.2.2 The Setoid of Maps between two Setoids

As proposed in [HS95] the morphisms between two objects in a category constitute a setoid. Thus, the concept of map between two setoids is the base to define the composition of a category.

A mapoid between two setoids s and s' is a map between the (Carrier s) and the (Carrier s') provided that this map preserves the equality of the setoid. The coercion that we defined before on the Setoid allows us to write Map as a map between s and s'.

```
Section Mapoids.

Variables s,s':  Setoid.

Definition MapLaw:= [f:s→s']
  (s1,s2:s) s1 =%S s2 → (f s1) =%S (f s2).

Structure Mapoid:  Type:= {
  Map      :> s→s';
  Prf_pres :  (MapLaw Map)
}.
```

The notion of mapoid is needed to define the composition but it is not sufficient. The composition as a binary operator receives two morphisms and returns the composite morphism. A mapoid receives only a setoid, hence the codomain of the composition mapoid must be a setoid of mapoids. This is the traditional *currying* transformation commonly used in functional programming.

To define the setoid of mapoids we have to give an equality between two mapoids and check that it is an equivalence relation. We say that two mapoids are equal iff they are extensionally equal.

Definition `Ext:= [f,g:Mapoid](s1:s) (f s1) =%S (g s1).`

Lemma `Ext_equiv: (Equivalence Ext).`

Now that we have defined mapoids and an equality relation over mapoids that is an equivalence, we can define the setoid of the mapoids between two setoids.

Definition `SetoidMapoid: Setoid:= (Build_Setoid Ext_equiv).`

End `Mapoids.`

We write `s ==> s'` for the setoid of mapoids between the setoids `s` and `s'`.

```
Token "==>".
Infix Assoc 6 "==>" SetoidMapoid.
```

2.2.3 Binary Mapoids

Given three setoids `s`, `s'` and `s''`, a binary mapoid is a mapoid between the setoid `s` and the setoid of the mapoids between the setoids `s'` and `s''`.

Section `BinaryMapoids.`

```
Variable s,s',s'': Setoid.
```

Definition `BinMapoid:= (Mapoid s s'==>s'').`

Remark that if the morphisms between two objects constitute a setoid the composition must be a binary mapoid.

Until the end of this section we sketch a few results that we use later on in some lemmas and definitions. We intend now to prove that from a binary map `f` we can obtain a binary mapoid if `f` holds the congruence laws for the equality of the setoid.

```
Variable f: s->s'->s''.
```

Definition `BinMapConglLaw:= (s1',s2':s')(s1:s)`
`s1' =%S s2' -> ((f s1) s1') =%S ((f s1) s2').`

Definition `BinMapCongrLaw:= (s1,s2:s)(s1':s')`
`s1 =%S s2 -> ((f s1) s1') =%S ((f s2) s1').`

Definition `BinMapCongLaw:= (s1,s2:s)(s1',s2':s')`
`s1 =%S s2 -> s1' =%S s2' -> ((f s1) s1') =%S ((f s2) s2').`

Hypothesis `pcgl: BinMapConglLaw.`

Hypothesis `pcgr: BinMapCongrLaw.`

Lemma `f_pres: (s1:s)(MapLaw (f s1)).`

Definition `Mapf: s→(s'==⇒s''):=`
`[s1:s](Build_Mapoid (f_pres s1)).`

Lemma `Mapf_pres: (MapLaw Mapf).`

Definition `Build_BinMapoid: BinMapoid:= (Build_Mapoid Mapf_pres).`

End `BinaryMapoids.`

Conversely, from a binary mapoid `f` we can obtain a binary map that holds the congruence laws for the equality of the setoid.

Section `CongBinMaps.`

```
Variable s,s',s'': Setoid.
```

```
Variable f: (BinMapoid s s' s'').
```

Definition `BinMap:= [s1:s][s1':s']((f s1) s1').`

Lemma `BinMap_congl: (BinMapConglLaw BinMap).`

Lemma `BinMap_congr: (BinMapCongrLaw BinMap).`

Lemma `BinMap_cong: (BinMapCongLaw BinMap).`

End `CongBinMaps.`

2.3 Categories

2.3.1 The Category Structure

We are finally ready to define category. The objects of a general category have type `Type` and the morphisms are a family of setoids indexed by their domain and codomain. In the sequel we use hom-setoids to denote the setoids of this family.

Section `CatLaws.`

```
Variable ob: Type.
```

```
Variable hom: ob→ob→Setoid.
```

As we said before we define the composition as a binary mapoid.

```
Variable comp_mapoid: (x,y,z:ob)
  (BinMapoid (hom x y) (hom y z) (hom x z)).
```

However, the associativity and the identity laws of the composition are defined over a binary map. We make use of `BinMap`, defined in the last section, to extract a binary map from a binary mapoid. Note that this map is congruent for the equality of the morphisms.

```
Definition Comp_map:= [x,y,z:ob][f:(hom x y)][g:(hom y z)]
  (BinMap (comp_mapoid x y z) f g).
```

For simplicity, we write `f o g` for `(Comp_map f g)` and we say that this infix operator is associative.

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

Remark that the symbol `o` is not used in the usual way (its arguments are in the inverse order).

In addition we have to assure that this composition map is associative.

```
Definition AssocLaw:=
  (x,y,z,w:ob)(f:(hom x y))(g:(hom y z))(h:(hom z w))
  (f o (g o h)) =%S ((f o g) o h).
```

Finally we have to define the identity that holds the identity laws for composition.

```
Variable id: (x:ob)(hom x x).

Definition IdlLaw:= (x,y:ob)(f:(hom x y)) ((id x) o f) =%S f.

Definition IdrLaw:= (x,y:ob)(f:(hom x y)) f =%S (f o (id y)).

End CatLaws.
```

Now we are able to define the category structure.

```
Structure Category: Type:= {
  Ob         :> Type;
  Hom        :  Ob->Ob->Setoid;
  CompMapoid :  (x,y,z:Ob)
                (BinMapoid (Hom x y) (Hom y z) (Hom x z));
  Id         :  (x:Ob)(Hom x x);
  Prf_assoc  :  (AssocLaw CompMapoid);
```

```
  Prf_idl     :  (IdlLaw CompMapoid Id);
  Prf_idr     :  (IdrLaw CompMapoid Id)
}.
```

As we shall see later, we use very frequently the composition as a binary map. So, in order to have a lighter notation, we present the following definition.

Definition `CompMap:= [c:Category](Comp_map (CompMapoid 1!c)).`

The exclamation mark is used whenever we want to explicitly give an implicit argument. The number that appears before the exclamation mark is the number of the implicit argument. We can see the list of implicit arguments with their respective numbers by typing the command `Print`.

We write `f o g` for `(CompMap f g)` and in addition we say that this infix operator is associative.

```
Infix Assoc 6 "o" CompMap.
```

Remark that grammar definitions inside a section disappear when the section is closed. Thus this last new rule does no conflict with the previous one defined inside of the section `CatLaws`.

By the results of the previous section to build the `CompMapoid` of a category we have to give a binary map and check that it holds the congruence laws. What we present next is a usual procedure to define the composition in any category from such a map. We shall use systematically this procedure from now on for every category definition.

Section `CatComp.`

```
Variable ob:  Type.

Variable hom:  ob->ob->Setoid.

Variable compmap:  (x,y,z:ob)(hom x y)->(hom y z)->(hom x z).
```

Definition `ConglLaw:= (x,y,z:ob)(f,g:(hom y z))(h:(hom x y))`
`  f =%S g -> (compmap h f) =%S (compmap h g).`

Definition `CongrLaw:= (x,y,z:ob)(f,g:(hom x y))(h:(hom y z))`
`  f =%S g -> (compmap f h) =%S (compmap g h).`

Definition `CongLaw:= (x,y,z:ob)(f,f':(hom x y))(g,g':(hom y z))`
`  f =%S f' -> g =%S g' -> (compmap f g) =%S (compmap f' g').`

Hypothesis `pcgl:  CongllLaw.`

Hypothesis `pcgr:  CongrLaw.`

```
Variable x,y,z: ob.
```

Definition `Build_CompMapoid:`
```
  (BinMapoid (hom x y) (hom y z) (hom x z)):=
  (Build_BinMapoid (pcgl 1!x 2!y 3!z) (pcgr 1!x 2!y 3!z)).
```

End `CatComp.`

Now we check that the composition map of a category respects the congruence laws. These laws are trivial to obtain with the results of the last section but still they are important to obtain some proofs in the future.

Section `CatCong.`

```
Variable c: Category.
```

Lemma `CompMap_congl: (ConglLaw (CompMap 1!c)).`

Lemma `CompMap_congr: (CongrLaw (CompMap 1!c)).`

Lemma `CompMap_cong: (CongLaw (CompMap 1!c)).`

End `CatCong.`

2.3.2 The Dual Category

A dual category $\mathtt{c}^{op}$ of a category `c` has the same objects of `c` but its morphisms are opposite. That is, if `f:(Hom c1 c2)` is a morphism in `c` then `f:(Hom c2 c1)` is a morphism in $\mathtt{c}^{op}$.

```
Variable c: Category.
```

Definition `DHom:= [c1,c2:c](Hom c2 c1).`

The composition is defined as expected. If (f o g) is a morphism in `c` then (g o f) is a morphism in $\mathtt{c}^{op}$. We then present the congruence laws to build the composition and check the associativity law.

Definition `DCompMap:= [c1,c2,c3:c]`
```
  [df:(DHom c1 c2)][dg:(DHom c2 c3)] dg o df.
```

Lemma `DCompMap_congl: (ConglLaw DCompMap).`

Lemma `DCompMap_congr: (CongrLaw DCompMap).`

Definition `CompDual:`

```
  (c1,c2,c3:c)(BinMapoid (DHom c1 c2)
                         (DHom c2 c3)
                         (DHom c1 c3)):=
  (Build_CompMapoid DCompMap_congl DCompMap_congr).
```

Lemma `Dual_assoc: (AssocLaw CompDual).`

The identity of $\mathtt{c}^{op}$ is the identity of `c`. After checking the identity laws for composition we are able to build the dual category.

Lemma `Dual_idl: (IdlLaw 2!DHom CompDual (Id 1!c)).`

Lemma `Dual_idr: (IdrLaw 2!DHom CompDual (Id 1!c)).`

Definition `Dual:= (Build_Category Dual_assoc Dual_idl Dual_idr).`

End `DualCat.`

2.3.3 The Category Setoid

As expected, the objects of the category Setoid are setoids and the morphisms are mapoids. Since we have already defined the setoid of mapoids our work is partially done. Hence we start by defining the composition. The composition of two mapoids is the composition of their respective maps. We only have to check that this composite map is a mapoid.

Section `CompositionMapoid.`

```
Variable s,s',s'': Setoid.

Variable f: (Mapoid s s').

Variable g: (Mapoid s' s'').
```

Definition `Comp_Map:= [s1:s](g (f s1)).`

Lemma `Comp_Map_pres: (MapLaw Comp_Map).`

End `CompositionMapoid.`

Definition `SetoidComp:`
```
  (s,s',s'':Setoid)(s==>s')->(s'==>s'')->(s==>s''):=
  [s,s',s'':Setoid][sm:s==>s'][sm':s'==>s'']
  (Build_Mapoid (Comp_Map_pres sm sm')).
```

To build the composition mapoid of a category, we have to give a composition map, `SetoidComp`, and check that it verifies the congruence laws.

```
Lemma SetoidComp_congl: (ConglLaw SetoidComp).

Lemma SetoidComp_congr: (CongrLaw SetoidComp).

Definition Comp_SETOID:
  (s,s',s'':Setoid)(BinMapoid s==>s' s'==>s'' s==>s''):=
  (Build_CompMapoid SetoidComp_congl SetoidComp_congr).

Lemma Assoc_Setoid: (AssocLaw Comp_SETOID).
```

The next step is to define the identity as a mapoid. We only have to check that the identity map is a mapoid.

```
Section Setoid_Id.

Variable s: Setoid.

Definition Id_Map:= [s1:s]s1.

Lemma Id_Map_pres: (MapLaw Id_Map).

Definition Id_SETOID: (Mapoid s s):= (Build_Mapoid Id_Map_pres).

End Setoid_Id.

Lemma Idl_Setoid: (IdlLaw Comp_SETOID Id_SETOID).

Lemma Idr_Setoid: (IdrLaw Comp_SETOID Id_SETOID).
```

With the laws of associativity and identity for composition we can define the category Setoid.

```
Definition SETOID: Category:=
  (Build_Category Assoc_Setoid Idl_Setoid Idr_Setoid).
```

2.3.4 The Category Presetoid

The objects of the category Presetoid are preorders and the morphisms are monotonous mapoids between preorders. A preorder is a reflexive and transitive binary relation between setoids. Since we are dealing with setoids and not sets we have to impose that two equal elements in a setoid must be related. To define the morphisms we have to build a setoid of monotonous mapoids. Hence we have to give an equality for monotonous mapoids and show that it is an equivalence relation. Obviously the equality provided is the equality between mapoids.

```
Section Setoid_Presetoid.

Structure PreOrder: Type:= {
  S              :> Setoid;
  Rel            :  (Relation S);
  Prf_presequal  :  (s1,s2:S)(s1 =%S s2)->(Rel s1 s2);
  Prf_po_refl    :  (Reflexive Rel);
  Prf_po_trans   :  (Transitive Rel)
}.

Variable p,p': PreOrder.

Definition IsMonotonous:= [f:(Mapoid p p')]
  (p1,p2:p)(Rel p1 p2)->(Rel (f p1) (f p2)).

Structure MonMapoid: Type:= {
  Mon_Mapoid :> (Mapoid p p');
  Prf_ismon  :  (IsMonotonous Mon_Mapoid)
}.

Definition EqualMonMapoid:= [f,g:MonMapoid](Ext f g).

Lemma EqualMonMapoid_equiv: (Equivalence EqualMonMapoid).

Definition Setoid_MonMapoid: Setoid:=
  (Build_Setoid EqualMonMapoid_equiv).

End Setoid_Presetoid.
```

The next step is to define the composition. The composition of two monotonous mapoids is the composition of their mapoids. In addition we have to check that the composite mapoid is monotonous.

```
Section CompositionMonMapoid.

Variable p,p',p'': PreOrder.

Variable f: (MonMapoid p p').

Variable g: (MonMapoid p' p'').

Definition Comp_MonMap:= [p1:p](g (f p1)).

Lemma Comp_MonMap_pres: (MapLaw Comp_MonMap).

Definition Comp_MonMapoid: (Mapoid p p''):=
```

```
  (Build_Mapoid Comp_MonMap_pres).
```

Lemma `Comp_MonMapoid_ismon: (IsMonotonous Comp_MonMapoid).`

End `CompositionMonMapoid.`

Definition `PresetoidComp:`
```
  (p,p',p'':PreOrder)(Setoid_MonMapoid p p')→
                     (Setoid_MonMapoid p' p'' )→
                     (Setoid_MonMapoid p p''):=
  [p,p',p'':PreOrder]
  [sm:(Setoid_MonMapoid p p')][sm':(Setoid_MonMapoid p' p'')]
  (Build_MonMapoid (Comp_MonMapoid_ismon sm sm')).
```

To build the composition mapoid of a category, we have to give a composition map, `PreSetoidComp`, and check that it verifies the congruence laws.

Lemma `PresetoidComp_congl: (ConglLaw PresetoidComp).`

Lemma `PresetoidComp_congr: (CongrLaw PresetoidComp).`

Definition `Comp_PRESETOID:`
```
  (p,p',p'':PreOrder)(BinMapoid (Setoid_MonMapoid p p')
                                (Setoid_MonMapoid p' p'')
                                (Setoid_MonMapoid p p'')):=
  (Build_CompMapoid PresetoidComp_congl PresetoidComp_congr).
```

Lemma `Assoc_Presetoid: (AssocLaw Comp_PRESETOID).`

We define the identity as the identity mapoid, that is obviously monotonous.

Section `Presetoid_Id.`

`Variable p: PreOrder.`

Definition `Id_MonMap:= [p1:p]p1.`

Lemma `Id_MonMap_pres: (MapLaw Id_MonMap).`

Definition `Id_MonMapoid: (Mapoid p p):=`
```
  (Build_Mapoid Id_MonMap_pres).
```

Lemma `Id_MonMapoid_ismon: (IsMonotonous Id_MonMapoid).`

Definition `Id_PRESETOID: (MonMapoid p p):=`
```
  (Build_MonMapoid Id_MonMapoid_ismon).
```

End Presetoid_Id.

Lemma Idl_Presetoid: (IdlLaw Comp_PRESETOID Id_PRESETOID).

Lemma Idr_Presetoid: (IdrLaw Comp_PRESETOID Id_PRESETOID).

Finally with the laws of associativity and identity for composition we can build the category Presetoid.

Definition PRESETOID: Category:=
  (Build_Category Assoc_Presetoid Idl_Presetoid Idr_Presetoid).

2.3.5 The Category Set

The objects of this category are the `Set`'s. The morphisms must constitute a setoid so we have to define the setoid of maps between `Set`'s. In order to define such setoid we must provide an equality relation for the maps. We say that two maps are equal iff they are extensional equal.

Section Setoid_Set.

Variable a,a': Set.

Definition EqualFunction:= [f,g:a→a'](a1:a)(f a1)=(g a1).

Lemma EqualFunction_equiv: (Equivalence EqualFunction).

Definition Setoid_Function: Setoid:=
  (Build_Setoid EqualFunction_equiv).

End Setoid_Set.

The composition is clearly the composition of maps.

Section CompositionFunction.

Variable a,a',a'': Set.

Variable f: a→a'.

Variable g: a'→a''.

Definition Comp_Function: a→a'':= [a1:a](g (f a1)).

End CompositionFunction.

Definition SetComp:

```
  (a,a',a'':Set)(Setoid_Function a a')→
                (Setoid_Function a' a'')→
                (Setoid_Function a a''):=
  [a,a',a'':Set]
  [sm:(Setoid_Function a a')][sm':(Setoid_Function a' a'')]
  (Comp_Function sm sm').
```

To build the composition mapoid of a category, we have to give a composition map, `SetComp`, and check that it verifies the congruence laws.

Lemma `SetComp_congl: (ConglLaw SetComp).`

Lemma `SetComp_congr: (CongrLaw SetComp).`

Definition `Comp_SET:`

```
  (a,a',a'':Set)(BinMapoid (Setoid_Function a a')
                           (Setoid_Function a' a'')
                           (Setoid_Function a a'')):=
  (Build_CompMapoid SetComp_congl SetComp_congr).
```

Lemma `Assoc_Set: (AssocLaw Comp_SET).`

Finally it only remains to define the identity, that is the identity map.

Section `Set_Id.`

`Variable a: Set.`

Definition `Id_SET: a→a:= [a1:a]a1.`

End `Set_Id.`

Lemma `Idl_Set: (IdlLaw Comp_SET Id_SET).`

Lemma `Idr_Set: (IdrLaw Comp_SET Id_SET).`

With all the framework above we are able to define the category Set.

Definition `SET: Category:=`

```
  (Build_Category Assoc_Set Idl_Set Idr_Set).
```

2.3.6 The Category PTh

Herein we define the category of propositional theories. A propositional theory is a pair $\langle \Sigma, \Gamma \rangle$ where Σ is a set, called signature, and Γ is a subset of L_Σ (the language of propositional formulae that can be written with symbols of Σ) closed for the semantic entailment. The elements of Σ are called propositional symbols and the elements of Γ are called theorems.

A morphism between (propositional) theories $\sigma : \langle\Sigma_1, \Gamma_1\rangle \to \langle\Sigma_2, \Gamma_2\rangle$ is a map between Σ_1 and Σ_2 such that $\sigma^\wedge(\Gamma_1) \subseteq \Gamma_2$. The extension $\sigma^\wedge$ of σ to the power of L_{Σ_1} is canonically established by replacing in each formula each symbol p of Σ_1 by $\sigma(p)$.

Section `Setoid_PTh.`

Section `Objects.`

To define objects we assume as given a signature that we called `sig`.

```
Variable sig:  Set.
```

We now have to define the set of theorems that is a subset of the language of `sig`. Hence we start by defining the language of `sig`, `Lsig`. This definition is obviously an inductive definition. Until now we have only used inductive definitions with a single constructor (macro `Structure`). However to define `Lsig` we need three constructors. The command `Inductive` is the primitive way to define inductive definitions with as many constructors as we want. In order to define an inductive type with `Inductive` we must provide the name of the constructors and their respective types. In the case of `Lsig`, `id`, `imp` and `no` are the constructors.

```
Inductive Lsig:  Type :=
  id   :  sig→Lsig
  | imp:  Lsig→Lsig→Lsig
  | no :  Lsig→Lsig.
```

The satisfaction of a propositional formula, by a valuation, is inductively defined in the structure of the propositional formulae. Whenever we want to define inductive objects using the inductive construction of their arguments we must use the command `Fixpoint`. In this case to define `SatPF` we take advantage of the inductive definition of `Lsig`.

```
Fixpoint SatPF[val:sig→Prop; pf:Lsig]:  Prop:=
  Case pf of
    [ps:sig] (val ps)
    [pf1,pf2:Lsig] (SatPF val pf1)→(SatPF val pf2)
    [pf1:Lsig] ¬(SatPF val pf1)
  end.
```

The `Case` operator matches the value `pf` with the various constructors of its inductive type. Thus when `pf` is `(id ps)` it returns `(val ps)`, when `pf` is `(imp pf1 pf2)` it returns `(SatPF val pf1)→(SatPF val pf2)` and when `pf` is `(no pf1)` it returns `¬(SatPF val pf1)`.

We say that a valuation satisfies a set of propositional formulae whenever it satisfies all the formulae of the set. Actually a set of propositional formulae is a subset of `Lsig`. To define subsets of `Lsig` we use an unary relation, `PLsig`, over `Lsig`.

```
Definition PLsig:= Lsig→Prop.
```

```
Definition SatSet:  (sig→Prop)→PLsig→Prop:=
  [val:sig→Prop][gamma:PLsig]
  (pf:Lsig)(gamma pf)→(SatPF val pf).
```

A formula `pf` of `Lsig` is a semantic consequence of a subset `gamma` of `Lsig` iff it is satisfied for all valuations that satisfy `gamma`.

```
Definition Entailment:  PLsig→Lsig→Prop:=
  [gamma:PLsig][pf:Lsig]
  (val:sig→Prop)(SatSet val gamma)→(SatPF val pf).
```

```
End Objects.
```

Now it only remains to define what is a set closed for the semantic entailment. We say that `gamma` is closed for the semantic entailment iff for any formula `pf` that is entailed by `gamma` belongs to `gamma`.

```
Inductive GammaClose[sig:Set; gamma:(PLsig sig)]:  Prop:=
  Build_TS: ((pf:(Lsig sig))(Entailment gamma pf)→(gamma pf))→
  (GammaClose sig gamma).
```

Finally, the objects of a propositional theory are composed by a `Signature` and a set `Gamma` of formulae in `(Lsig Signature)` that is closed for the semantic entailment.

```
Structure PTh:  Type:= {
  Signature :> Set;
  Gamma     :  (PLsig Signature);
  Prf_close :  (GammaClose Gamma)
}.
```

We now define the extension of a map between signatures to the power of the language of their respective signatures.

```
Section Morphisms.
```

```
Variable sig,sig':Set.
```

```
Fixpoint Extension[f:sig→sig'; pf:(Lsig sig)]:  (Lsig sig'):=
  Case pf of
    [ps:sig] (id (f ps))
```

```
    [pf1,pf2:(Lsig sig)] (imp (Extension f pf1)
                              (Extension f pf2))
    [pf1:(Lsig sig)] (no (Extension f pf1))
  end.

End Morphisms.
```

The morphisms between propositional theories are maps that hold the `InclusionLaw`.

```
Variable pt,pt':PTh.

Definition InclusionLaw:= [f:pt→pt']
  (pf:(Lsig pt))((Gamma 1!pt) pf)→
               ((Gamma 1!pt') (Extension f pf)).

Structure MorphismPTh:  Type:= {
  Application   :> pt→pt';
  Prf_inclusion :  (InclusionLaw Application)
}.
```

With all the framework presented above we are able to define the setoid of the morphisms between propositional theories. Since this morphisms are maps, that hold the `InclusionLaw`, the equality is obviously the equality between maps. As we show next this relation is an equivalence and so we can build the setoid of morphisms between propositional theories.

```
Definition EqualMorphismPTh:=[f,g:MorphismPTh]
  (ps:pt)(f ps)=(g ps).

Lemma EqualMorphismPTh_equiv:  (Equivalence EqualMorphismPTh).

Definition Setoid_MorphismPTh:  Setoid:=
  (Build_Setoid EqualMorphismPTh_equiv).

End Setoid_PTh.
```

The composition of two morphisms between propositional theories is the composition of their respective maps. To check that this composition is a morphism between propositional theories we have to show that it respects the `InclusionLaw`. For this purpose we start by an auxiliary lemma where we prove that the extension of a composition is the composition of the extended maps.

```
Section CompositionMorphismPTh.

Variable pt,pt',pt'':  PTh.
```

```
Variable f: (MorphismPTh pt pt').

Variable g: (MorphismPTh pt' pt'').

Definition Comp_Application:= [ps:pt](g (f ps)).

Lemma CompExtension: (pf:(Lsig pt))(f':pt→pt')(g':pt'→pt'')
  (Extension [ps:pt](g' (f' ps)) pf)==
  (Extension g' (Extension f' pf)).

Lemma Comp_Application_inclusion:
  (InclusionLaw Comp_Application).

End CompositionMorphismPTh.

Definition PThComp:
  (pt,pt',pt'':PTh)(Setoid_MorphismPTh pt pt')→
                  (Setoid_MorphismPTh pt' pt'')→
                  (Setoid_MorphismPTh pt pt''):=
  [pt,pt',pt'':PTh]
  [sm:(Setoid_MorphismPTh pt pt')]
  [sm':(Setoid_MorphismPTh pt' pt'')]
  (Build_MorphismPTh (Comp_Application_inclusion sm sm')).
```

As usual we can build the composition mapoid with the proves that the composition map `PThComp` holds the congruence laws.

```
Lemma PThComp_congl: (ConglLaw PThComp).

Lemma PThComp_congr: (CongrLaw PThComp).

Definition Comp_PTH:
  (pt,pt',pt'':PTh)(BinMapoid (Setoid_MorphismPTh pt pt')
                              (Setoid_MorphismPTh pt' pt'')
                              (Setoid_MorphismPTh pt pt'')):=
  (Build_CompMapoid PThComp_congl PThComp_congr).

Lemma Assoc_PTh: (AssocLaw Comp_PTH).
```

The identity morphism is obviously the identity map. We only have to check that it holds the `InclusionLaw`. To check this we start by showing that the extension of the identity map is the identity.

```
Section PTh_Id.

Variable pt: PTh.
```

Definition `Id_Application:= [ps:pt]ps.`

Lemma `IdExtension:`
```
  (pf:(Lsig pt))(Extension [ps:pt]ps pf)==pf.
```

Lemma `Id_Application_inclusion:  (InclusionLaw Id_Application).`

Definition `Id_PTH: (MorphismPTh pt pt):=`
```
  (Build_MorphismPTh Id_Application_inclusion).
```

End `PTh_Id.`

Lemma `Idl_PTh:  (IdlLaw Comp_PTH Id_PTH).`

Lemma `Idr_PTh:  (IdrLaw Comp_PTH Id_PTH).`

Finally with the laws of associativity and identity for composition we are able to build the category PTh.

Definition `PTH: Category:=`
```
  (Build_Category Assoc_PTh Idl_PTh Idr_PTh).
```

2.4 Functors

A functor is a pair of maps, one for the objects and another for the morphisms. The first is a map in `Type` and the second is a mapoid, since the morphisms constitute a setoid. These maps must preserve the composition and the identity.

Section `FunctorDef.`

```
Variable c,d:  Category.
```

Section `FunctorLaws.`

```
Variable fF0:  c→d.

Variable fF1:  (c1,c2:c)
  (Mapoid (Hom c1 c2) (Hom (fF0 c1) (fF0 c2))).
```

Definition `FCompLaw:= (c1,c2,c3:c)(f:(Hom c1 c2))(g:(Hom c2 c3))`
```
  ((fF1 c1 c3) (f o g)) =%S (((fF1 c1 c2) f) o ((fF1 c2 c3) g)).
```

Definition `FIdLaw:= (c1:c)`
```
  ((fF1 c1 c1) (Id c1)) =%S (Id (fF0 c1)).
```

```
End FunctorLaws.

Structure Functor: Type:= {
  F0         :> c->d;
  F1         :  (c1,c2:c)
                (Mapoid (Hom c1 c2) (Hom (F0 c1) (F0 c2)));
  Prf_comp   :  (FCompLaw F1);
  Prf_id     :  (FIdLaw F1)
}.
```

We can not make two coercions simultaneously for `F0` and `F1` because they are both functions (the `Coq` system does not allow it). Thus we choose to make a coercion for `F0`.

To simplify the syntax we define `FMor` that returns the image of a morphism `f` by a functor `F`. In `FMor`, the arguments `c1` and `c2` are implicit and that is not the case for `F1`.

```
Definition FMor:= [fF:Functor][c1,c2:c][f:(Hom c1 c2)]
  ((F1 fF c1 c2) f).

End FunctorDef.
```

2.5 Isomorphisms and Initial and Terminal Objects

We say that two objects `c1` and `c2` are isomorphic whenever there are two morphisms, `IsoMor:(Hom c1 c2)` and `InvIso:(Hom c2 c1)`, such that one is the inverse of the other.

```
Section IsoDef.

Variable c: Category.

Variable c1,c2: c.

Definition InverseLaw:= [c1,c2:c][f:(Hom c1 c2)][g:(Hom c2 c1)]
  (g o f) =%S (Id c2).

Definition IsoLaw:= [f:(Hom c1 c2)][g:(Hom c2 c1)]
  (InverseLaw f g)∧(InverseLaw g f).

Structure Iso: Type:={
  IsoMor  : (Hom c1 c2);
  InvIso  : (Hom c2 c1);
  Prf_iso : (IsoLaw IsoMor InvIso)
}.
```

```
End IsoDef.
```

We say that an object `ObI` is initial in a category `c` if there is a family of morphisms `MorI:(c2:c)(Hom ObI c2)` such that for every `c2` in `c` any morphism `g:(Hom ObI c2)` belongs to the source ⟨`ObI`, λ `c2:c.(MorI c2)`⟩. See for instance [AHS90] for more details in sources.

```
Section InitialDef.

Variable c: Category.

Definition InitialLaw:= [c1:c][f:(c2:c)(Hom c1 c2)]
  (c2:c)(g:(Hom c1 c2)) (f c2) =%S g.

Structure Initial: Type:= {
  ObI         :> c;
  MorI        :  (c2:c)(Hom ObI c2);
  Prf_initial :  (InitialLaw MorI)
}.

End InitialDef.
```

Terminal objects are defined similarly to initial objects, using a sink instead of a source.

```
Section TerminalDef.

Variable c: Category.

Definition TerminalLaw:= [c2:c][f:(c1:c)(Hom c1 c2)]
  (c1:c)(g:(Hom c1 c2)) (f c1) =%S g.

Structure Terminal: Type:= {
  ObT          :> c;
  MorT         :  (c1:c)(Hom c1 ObT);
  Prf_terminal :  (TerminalLaw MorT)
}.

End TerminalDef.
```

2.6 Some Exercises

Herein we show some lemmas. First we obtain three basic results with respect to the concepts that we defined above. This results are already established by Huet and Saïbi in [HS95] and are presented here only for satisfying the curiosity of the reader about the articulation of these concepts. Next we check the presentation lemma that is a powerful lemma that we shall use later on for showing some results.

2.6.1 Basic Results

We start to prove that initial objects are unique up to isomorphism.

Lemma `Two_ObI_Iso: (c:Category)(i1,i2:(Initial c))(Iso i1 i2).`

We now show that an initial object in a category is terminal in the dual category.

Lemma `Initial_Dual: (c:Category)(c1:c)(i:(c2:c)(Hom c1 c2))`
```
  (InitialLaw i)→(TerminalLaw 1!(Dual c) i).
```

Finally we show that functors preserve isomorphisms.

Lemma `F_Preserve_Iso: (c,d:Category)(fF:(Functor c d))`
```
  (c1,c2:c)(Iso c1 c2)→(Iso (fF c1) (F c2)).
```

2.6.2 The Presentation Lemma

We start by defining the closure of a set for semantic entailment and the inclusion of a set in another set. We also define the set of images, by the extension of a map `f`, of a set (given a set `gamma` we want the set `f^(gamma)`).

Definition `Closure: (sig:Set)(PLsig sig)→(Lsig sig)→Prop:=`
```
  [sig:Set][gamma:(PLsig sig)][pf:(Lsig sig)]
  (Entailment gamma pf).
```

Definition `Inclusion: (sig:Set)(PLsig sig)→(PLsig sig)→Prop:=`
```
  [sig:Set][gamma1,gamma2:(PLsig sig)]
  (pf:(Lsig sig))(gamma1 pf)→(gamma2 pf).
```

Inductive `ExtensionSet`
```
  [sig,sig':Set;f:sig→sig';gamma:(PLsig sig)]: (PLsig sig'):=
  Build_ES: (pf:(Lsig sig))(gamma pf)→
  (ExtensionSet sig sig' f gamma (Extension f pf)).
```

Before presenting the presentation lemma we check three properties of the semantic entailment, the monotony, the idempotency and the structurality condition. Actually we only prove half of the idempotency (the inclusion in the other direction is trivial and not necessary to show the presentation lemma).

Lemma `Monotony:`
```
  (sig:Set)(gamma1,gamma2:(PLsig sig))
  (Inclusion gamma1 gamma2)→
  (Inclusion (Closure gamma1) (Closure gamma2)).
```

Lemma `IdemPotency:`

```
(sig:Set)(gamma:(PLsig sig))
(Inclusion (Closure (Closure gamma)) (Closure gamma)).
```

Lemma `Structurality:`

```
(sig,sig':Set)(f:sig→sig')(gamma:(PLsig sig))
(pf':(Lsig sig'))
(ExtensionSet f [pf:(Lsig sig)](Closure gamma pf) pf')→
(Closure [pf1':(Lsig sig')](ExtensionSet f gamma pf1') pf').
```

Lemma `PresentationLemma:`

```
(sig,sig':Set)(f:sig→sig')
(gamma:(PLsig sig))(gamma':(PLsig sig'))
(((pf:(Lsig sig))(gamma pf)→
                (Closure gamma' (Extension f pf)))↔
 ((pf:(Lsig sig))(Closure gamma pf)→
                (Closure gamma' (Extension f pf)))).
```

Chapter 3

Adjunctions

In this section we present the concept of adjunction in `Coq`. We also define some examples and results that relate this definition with other concepts of category theory. The adjunction was already defined in `Coq` by Saïbi (see [Saï95]). We implement an equivalent but different definition of adjunction. The adjunction that we define in this section is the one given in [AHS90]:

Let C and D be categories and $F : C \to D$ and $G : D \to C$ be functors. We say that F is left adjoint of G iff

- *there is a natural transformation $\eta : id_C \to GoF$,*
  $\eta = \{\eta_X : X \to G(F(X))\}_{X \in |C|}$*;*
- *for all $f : X \to G(A)$ in C there is only one morphism $g : F(X) \to A$ in D such that $G(g) o \eta_X = f$.*

We say that η is the unit of the adjunction $\langle F, G, \eta \rangle$.

3.1 The Adjunction Structure

We start by defining natural transformation. Given two categories `c` and `d` and two functors `fF:(Functor c d)` and `fG:(Functor c d)` a natural transformation `NT` from `fF` to `fG` is a family $\{$`(NTMap c1):(Hom (fF c1) (fG c1))`$\}_{\texttt{c1:c}}$ that holds the `NTLaw`.

```
Section NTDef.

Variable c,d:  Category.

Variable fF,fG: (Functor c d).

Definition NTLaw:= [nt:(c1:c)(Hom (fF c1) (fG c1))]
  (c1,c2:c)(f:(Hom c1 c2))
  ((FMor fF f) o (nt c2)) =%S ((nt c1) o (FMor fG f)).
```

```
Structure NT: Type:= {
  NTMap      :> (c1:c)(Hom (fF c1) (fG c1));
  Prf_ntlaw  :  (NTLaw NTMap)
}.

End NTDef.
```

For the specific case of an adjunction the natural transformation is between an identity functor and a composite functor. Hence we begin by defining the identity functor `IdFunctor` for a category `c`. We call `IdF0` the map for the objects and `IdF1` the mapoid for the morphisms.

```
Section IdFunct.

Variable c: Category.

Definition IdF0:= [c1:c]c1.

Section IdMor.

Variable c1,c2: c.

Definition IdFMor:= [f:(Hom c1 c2)]f.

Lemma IdFMor_pres: (MapLaw IdFMor).

Definition IdF1:
  (Mapoid (Hom c1 c2) (Hom (IdF0 c1) (IdF0 c2))):=
  (Build_Mapoid IdFMor_pres).

End IdMor.

Lemma IdF1_comp: (!FCompLaw c c IdF0 IdF1).

Lemma IdF1_id: (!FIdLaw c c IdF0 IdF1).

Definition IdFunctor: (Functor c c):=
  (Build_Functor IdF1_comp IdF1_id).

End IdFunct.
```

Next we check that the composition of two functors is a functor. We call the composite functor by `CompFunctor`, the map for the objects by `CompF0` and the mapoid for the morphisms `CompF1`.

```
Section CompFunct.
```

```
Variable c,d,e: Category.

Variable fG: (Functor c d).

Variable fH: (Functor d e).

Definition CompF0:= [c1:c](fH (fG c1)).

Section CompMor.

Variable c1,c2:c.

Definition CompFMor:= [f:(Hom c1 c2)](FMor fH (FMor fG f)).

Lemma CompFMor_pres: (MapLaw CompFMor).

Definition CompF1:
  (Mapoid (Hom c1 c2) (Hom (CompF0 c1) (CompF0 c2))):=
  (Build_Mapoid CompFMor_pres).

End CompMor.

Lemma CompF1_comp: (FCompLaw CompF1).

Lemma CompF1_id: (FIdLaw CompF1).

Definition CompFunctor: (Functor c e):=
  (Build_Functor CompF1_comp CompF1_id).

End CompFunct.
```

Finally we can define adjunction. Given two categories `c` and `d` and two functors `fF:(Functor c d)` and `fG:(Functor d c)` we say that `fF` is left adjoint of `fG` whenever we can find a natural transformation `unit` from `(IdFunctor c)` to `(CompFunctor fF fG)` that holds the universal property of the adjunction. We split the universal property into two properties, one dealing with commutation, `AdjCommuteLaw`, and other dealing with uniqueness, `AdjUniqueLaw`.

```
Section AdjunctionDef.

Variable c,d: Category.

Variable fF: (Functor c d).

Variable fG: (Functor d c).

Section AdjunctionLaws.
```

```
Variable unit:  (NT (IdFunctor c) (CompFunctor fF fG)).

Definition Commute_c:=
  [x:c][a:d][f:(Hom x (fG a))][g:(Hom (fF x) a)]
  ((unit x) o (FMor fG g)) =%S f.

Variable g:  (x:c)(a:d)(f:(Hom x (fG a)))(Hom (fF x) a).

Definition Unique_d:=
  [x:c][a:d][f:(Hom x (fG a))][g':(Hom (fF x) a)]
  (Commute_c f g') → (g f) =%S g'.

Definition AdjCommuteLaw:=
  (x:c)(a:d)(f:(Hom x (fG a)))(Commute_c f (g f)).

Definition AdjUniqueLaw:=
  (x:c)(a:d)(f:(Hom x (fG a)))(g':(Hom (fF x) a))
  (Unique_d f g').

End AdjunctionLaws.

Structure Adjunction:  Type:= {
  unit        : (NT (IdFunctor c) (CompFunctor fF fG));
  g           : (x:c)(a:d)(f:(Hom x (fG a)))(Hom (fF x) a);
  Prf_commute : (AdjCommuteLaw unit g);
  Prf_unique  : (AdjUniqueLaw unit g)
}.

End AdjunctionDef.
```

3.2 The Adjunction between Setoid and Presetoid

In this section we intend to prove that the forgetful functor from PRESETOID to SETOID has left and right adjoint. We start by defining this forgetful functor, FForgetfulPS. We call F0ForgetfulPS to the map of the objects and F1ForgetfulPS to the mapoid of the morphisms.

```
Section F_PRESETOID_SETOID.

Variable p,p':  PRESETOID.

Definition F0ForgetfulPS: PRESETOID→SETOID := [p:PRESETOID]p.

Definition FMapForgetfulPS:
  (Hom p p')→(Hom (F0ForgetfulPS p) (F0ForgetfulPS p')):=
```

```
  [f:(MonMapoid p p')](Mon_Mapoid f).
```

Lemma `FMapForgetfulPS_pres: (MapLaw FMapForgetfulPS).`

Definition `F1ForgetfulPS:`
```
  (Mapoid (Hom p p')
          (Hom (F0ForgetfulPS p) (F0ForgetfulPS p'))):=
  (Build_Mapoid FMapForgetfulPS_pres).
```

End `F_PRESETOID_SETOID.`

Lemma `F1ForgetfulPS_comp: (FCompLaw F1ForgetfulPS).`

Lemma `F1ForgetfulPS_id: (FIdLaw F1ForgetfulPS).`

Definition `FForgetfulPS: (Functor PRESETOID SETOID):=`
```
  (Build_Functor F1ForgetfulPS_comp F1ForgetfulPS_id).
```

Next we define the functor `FEqualRel`, the candidate for left adjoint of `FForgetfulPS`. The functor `FEqualRel` maps each setoid `s` into a preorder `POEqual` corresponding to a pair having `s` and the equality relation of the setoid `s`. Obviously any mapoid between two preorders, that are image of `POEqual`, is a monotonous mapoid. The preservation of the relation is just the functionality condition for mapoids.

Section `F_SETOID_PRESETOID.`

Section `FEqualRelPO.`

`Variable s: SETOID.`

Lemma `Equal_presequal: (s1,s2:s)(s1 =%S s2)→(s1 =%S s2).`

Lemma `Equal_po_refl: (Reflexive (!Equal s)).`

Lemma `Equal_po_trans: (Transitive (!Equal s)).`

Definition `EqualPO: PreOrder:=`
```
  (Build_PreOrder Equal_presequal Equal_po_refl Equal_po_trans).
```

End `FEqualRelPO.`

Definition `F0EqualRel: SETOID→PRESETOID:=`
```
  [s:SETOID](EqualPO s).
```

Section `FEqualRelMonMapoid.`

```
Variable s,s': SETOID.

Variable f: (Mapoid (F0EqualRel s) (F0EqualRel s')).

Lemma f_ismon: (IsMonotonous f).

Definition MonMapf:
  (MonMapoid (F0EqualRel s) (F0EqualRel s')):=
  (Build_MonMapoid f_ismon).

End FEqualRelMonMapoid.

Variable s,s': SETOID.

Definition FMapEqualRel:
  (Hom s s')→(Hom (F0EqualRel s) (F0EqualRel s')):=
  [f:(Mapoid s s')](MonMapf f).

Lemma FMapEqualRel_pres: (MapLaw FMapEqualRel).

Definition F1EqualRel:
  (Mapoid (Hom s s') (Hom (F0EqualRel s) (F0EqualRel s'))):=
  (Build_Mapoid FMapEqualRel_pres).

End F_SETOID_PRESETOID.

Lemma F1EqualRel_comp: (FCompLaw F1EqualRel).

Lemma F1EqualRel_id: (FIdLaw F1EqualRel).

Definition FEqualRel: (Functor SETOID PRESETOID):=
  (Build_Functor F1EqualRel_comp F1EqualRel_id).
```

To define the adjunction we must provide a natural transformation between `(IdFunctor SETOID)` and `(CompFunctor FEqualRel FForgetfulPS)`. The natural transformation `NTSETOID` associates each setoid with its identity.

```
Section NT_SETOID.

Variable s: SETOID.

Definition NTSetoidMap:
  ((IdFunctor SETOID) s)→
  ((CompFunctor FEqualRel FForgetfulPS) s):=
  (Id_Map 1!s).

Lemma NTSetoidMap_pres: (MapLaw NTSetoidMap).
```

Definition `NTSetoidMapoid:`
```
  (Mapoid ((IdFunctor SETOID) s)
          ((CompFunctor FEqualRel FForgetfulPS) s)):=
  (Build_Mapoid NTSetoidMap_pres).
```

End `NT_SETOID.`

Lemma `NTSetoidMapoid_ntlaw:`
```
  (NTLaw 1!SETOID 2!SETOID NTSetoidMapoid).
```

Definition `NTSETOID:`
```
  (NT (IdFunctor SETOID) (CompFunctor FEqualRel FForgetfulPS)):=
  (Build_NT NTSetoidMapoid_ntlaw).
```

Finally we have to show the universal property of the adjunction. This is, given a setoid `s` and a presetoid `p`, for each morphism `f:(Hom s (FForgetfulPS p))`, we must provide a morphism `g:(Hom (FEqualRel s) p)` that holds `AdjCommuteLaw` and `AdjUniqueLaw`. Obviously the candidate for `g` is `f`.

Section `Adj_SETOID.`

```
Variable s:  SETOID.

Variable p:  PRESETOID.

Variable f:  (Hom s (FForgetfulPS p)).
```

Lemma `f_ismon:  (!IsMonotonous (FEqualRel s) p f).`

Definition `g:  (MonMapoid (FEqualRel s) p):=`
```
  (Build_MonMapoid f_ismon).
```

End `Adj_SETOID.`

Lemma `g_commute:  (AdjCommuteLaw NTSETOID g).`

Lemma `g_unique:  (AdjUniqueLaw NTSETOID g).`

Definition `AdjSETOID: (Adjunction FEqualRel FForgetfulPS):=`
```
  (Build_Adjunction g_commute g_unique).
```

Next we define the functor `FTotalRel`, the candidate for right adjoint of `FForgetfulPS`. The functor `FTotalRel` maps each setoid `s` into a preorder corresponding to a pair having `s` and the total relation over `s`, that we call `Total`. It is trivial to check that mapoids are always monotonous with respect to total relations.

```
Section F_SETOID_PRESETOID.

Section FTotalRelPO.

Variable s: SETOID.

Inductive Total: (Relation (Carrier s)):=
  Build_Total: (s1,s2:s)(Total s1 s2).

Lemma Total_presequal: (s1,s2:s)(s1 =%S s2)→(Total s1 s2).

Lemma Total_po_refl: (Reflexive Total).

Lemma Total_po_trans: (Transitive Total).

Definition TotalPO: PreOrder:=
  (Build_PreOrder Total_presequal Total_po_refl Total_po_trans).

End FTotalRelPO.

Definition F0TotalRel: SETOID→PRESETOID:=
  [s:SETOID](TotalPO s).

Section FTotalRelMonMapoid.

Variable s,s': SETOID.

Variable f: (Mapoid (F0TotalRel s) (F0TotalRel s')).

Lemma f_ismon: (IsMonotonous f).

Definition MonMapf:
  (MonMapoid (F0TotalRel s) (F0TotalRel s')):=
  (Build_MonMapoid f_ismon).

End FTotalRelMonMapoid.

Variable s,s': SETOID.

Definition FMapTotalRel:
  (Hom s s')→(Hom (F0TotalRel s) (F0TotalRel s')):=
  [f:(Mapoid s s')](MonMapf f).

Lemma FMapTotalRel_pres: (MapLaw FMapTotalRel).

Definition F1TotalRel:
```

```
  (Mapoid (Hom s s') (Hom (F0TotalRel s) (F0TotalRel s'))):=
  (Build_Mapoid FMapTotalRel_pres).

End F_SETOID_PRESETOID.

Lemma F1TotalRel_comp:  (FCompLaw F1TotalRel).

Lemma F1TotalRel_id:  (FIdLaw F1TotalRel).

Definition FTotalRel:  (Functor SETOID PRESETOID):=
  (Build_Functor F1TotalRel_comp F1TotalRel_id).
```

To define the adjunction we must provide a natural transformation between `(IdFunctor PRESETOID)` and `(CompFunctor FForgetfulPS FTotalRel)`. The natural transformation `NTPRESETOID` associates each preorder with its identity.

```
Section NT_PRESETOID.

Variable p:  PRESETOID.

Definition NTPresetoidMap:
  ((IdFunctor PRESETOID) p)->
  ((CompFunctor FForgetfulPS FTotalRel) p):=
  (Id_MonMap 1!p).

Lemma NTPresetoidMap_pres:  (MapLaw NTPresetoidMap).

Definition NTPresetoidMapoid:
  (Mapoid ((IdFunctor PRESETOID) p)
          ((CompFunctor FForgetfulPS FTotalRel) p)):=
  (Build_Mapoid NTPresetoidMap_pres).

Lemma NTPresetoidMapoid_ismon:
  (IsMonotonous NTPresetoidMapoid).

Definition NTPresetoidMonMap:
  (MonMapoid ((IdFunctor PRESETOID) p)
             ((CompFunctor FForgetfulPS FTotalRel) p)):=
  (Build_MonMapoid NTPresetoidMapoid_ismon).

End NT_PRESETOID.

Lemma NTPresetoidMonMap_ntlaw:
  (NTLaw 1!PRESETOID 2!PRESETOID NTPresetoidMonMap).

Definition NTPRESETOID:
  (NT (IdFunctor PRESETOID)
```

```
    (CompFunctor FForgetfulPS FTotalRel)):=
  (Build_NT NTPresetoidMonMap_ntlaw).
```

Finally we have to show the universal property of the adjunction. This is, given a presetoid `p` and a setoid `s`, for each morphism `f:(Hom p (FTotalRel s))`, we must provide a morphism `g:(Hom (FForgetfulPS p) s)` that holds `AdjCommuteLaw` and `AdjUniqueLaw`. Obviously the candidate for `g` is `f`.

```
Section Adj_PRESETOID.

Variable p:  PRESETOID.

Variable s:  SETOID.

Variable f:  (Hom p (FTotalRel s)).

Lemma MonMapoidf_pres:  (MapLaw (Mon_Mapoid f)).

Definition g:  (Mapoid (FForgetfulPS p) s):=
  (Build_Mapoid MonMapoidf_pres).

End Adj_PRESETOID.

Lemma g_commute:  (AdjCommuteLaw NTPRESETOID g).

Lemma g_unique:  (AdjUniqueLaw NTPRESETOID g).

Definition AdjPRESETOID: (Adjunction FForgetfulPS FTotalRel):=
  (Build_Adjunction g_commute g_unique).
```

3.3 The Adjunction between Set and PTh

Herein we show that the forgetful functor from `PTH` to `SET` has right adjoint. First we define the forgetful functor, `FForgetfulPT`. This functor maps any propositional theory in its corresponding signature with `F0ForgetfulPT` and, maps any morphism between propositional theories in its corresponding map with `F1ForgetfulPT`.

```
Section F_PTH_SET.

Variable pt,pt':  PTH.

Definition F0ForgetfulPT: PTH→SET:= [pt:PTH]pt.

Definition FMapForgetfulPT:
  (Hom pt pt')→(Hom (F0ForgetfulPT pt) (F0ForgetfulPT pt')):=
```

```
  [f:(MorphismPTh pt pt')](Application f).
```

Lemma `FMapForgetfulPT_pres:  (MapLaw FMapForgetfulPT).`

Definition `F1ForgetfulPT:`
```
  (Mapoid (Hom pt pt')
          (Hom (F0ForgetfulPT pt) (F0ForgetfulPT pt'))):=
  (Build_Mapoid FMapForgetfulPT_pres).
```

End `F_PTH_SET.`

Lemma `F1ForgetfulPT_comp:  (FCompLaw F1ForgetfulPT).`

Lemma `F1ForgetfulPT_id:  (FIdLaw F1ForgetfulPT).`

Definition `FForgetfulPT: (Functor PTH SET):=`
```
  (Build_Functor F1ForgetfulPT_comp F1ForgetfulPT_id).
```

Now we have to give the candidate for right adjoint of `FForgetfulPT`. We propose the functor `FLanguage` from `SET` to `PTH` that maps any set `a` in the propositional theory constituted by `a` and the language of `a`, `L`. A map between propositional theories where the set of theorems is the language of its signatures verifies clearly the `InclusionLaw`.

Section `F_SET_PTH.`

Section `FLanguagePTh.`

`Variable a:  SET.`

Definition `L: (Lsig a)→Prop:= [pf:(Lsig a)]True.`

Lemma `L_close:  (GammaClose L).`

Definition `LPTh:  PTh:= (Build_PTh L_close).`

End `FLanguagePTh.`

Definition `F0Language:  SET→PTH:= [a:SET](LPTh a).`

Section `FLanguageMorphism.`

`Variable a,a':  SET.`

`Variable f:  (LPTh a)→(LPTh a').`

Lemma `f_inclusion:  (InclusionLaw f).`

```
Definition Morphismf:
  (MorphismPTh (LPTh a) (LPTh a')):=
  (Build_MorphismPTh f_inclusion).

End FLanguageMorphism.

Variable a,a': SET.

Definition FMapLanguage:
  (Hom a a')→(Hom (F0Language a) (F0Language a')):=
  [f:a→a'](Morphismf f).

Lemma FMapLanguage_pres: (MapLaw FMapLanguage).

Definition F1Language:
  (Mapoid (Hom a a') (Hom (F0Language a) (F0Language a'))):=
  (Build_Mapoid FMapLanguage_pres).

End F_SET_PTH.

Lemma F1Language_comp: (FCompLaw F1Language).

Lemma F1Language_id: (FIdLaw F1Language).

Definition FLanguage: (Functor SET PTH):=
  (Build_Functor F1Language_comp F1Language_id).
```

The next step is to give a natural transformation from `(IdFunctor PTH)` to `(CompFunctor FForgetfulPT FLanguage)`. The natural transformation `NTPTH` associates to each propositional theory its identity.

```
Section NT_PTH.

Variable pt: PTH.

Definition NTPThApplication:
  ((IdFunctor PTH) pt)→
  ((CompFunctor FForgetfulPT FLanguage) pt):=
  (Id_Application 1!pt).

Lemma NTPThApplication_inclusion:
  (InclusionLaw NTPThApplication).

Definition NTPThMorphismPTh:
  (MorphismPTh ((IdFunctor PTH) pt)
               ((CompFunctor FForgetfulPT FLanguage) pt)):=
```

```
  (Build_MorphismPTh NTPThApplication_inclusion).
```

End `NT_PTH.`

Lemma `NTPThMorphismPTh_ntlaw:`
```
  (NTLaw 1!PTH 2!PTH NTPThMorphismPTh).
```

Definition `NTPTH:`
```
  (NT (IdFunctor PTH) (CompFunctor FForgetfulPT FLanguage)):=
  (Build_NT NTPThMorphismPTh_ntlaw).
```

Finally we are able to define the adjunction. We only have to find, given a propositional theory `pt`, a set `a` and a morphism `f:(Hom pt (FLanguage a))`, a morphism `g:(Hom (FForgetfulPT pt) a)` that holds the universal property. It is clear that the candidate for `g` is `f`.

Section `Adj_PTH.`

```
Variable pt:  PTH.

Variable a:  SET.

Variable f:  (Hom pt (FLanguage a)).
```

Definition `g:  (FForgetfulPT pt)→a:= (Application f).`

End `Adj_PTH.`

Lemma `g_commute:  (AdjCommuteLaw NTPTH g).`

Lemma `g_unique:  (AdjUniqueLaw NTPTH g).`

Definition `AdjPTH: (Adjunction FForgetfulPT FLanguage):=`
```
  (Build_Adjunction g_commute g_unique).
```

3.4 Adjunction vs Initial in Comma Category

Herein we intend to show the result that relates left adjoints with initial objects in comma category:

Let C and D be categories and $G : D \to C$ be a functor. Then, G has left adjoint iff $X \downarrow G$ has initial object for any $X \in |C|$.

We start by defining the comma category. If `x` is an object of `c` and `fG` a functor from `d` to `c`, the category `x↓fG` has as objects all pairs `Codom` and `Arrow`, where `Codom:d` and `Arrow:(Hom x (fG Codom))`, and as morphisms from `v` to `u` all those arrows `MorComma:(Hom (Codom v) (Codom u))` in `d` such that `(v o (FMor fG MorComa)) =%S u`. To this property we call `CommaCommuteLaw`. To define the morphisms we have to build a setoid of comma morphisms. Hence we have to give an equality for comma morphisms and show that it is an equivalence relation. The equality provided is the equality between the corresponding arrows in `d`.

```
Section CommaDef.

Variable c,d:  Category.

Variable fG: (Functor d c).

Variable x:  c.

Section Setoid_Comma.

Structure ObjectComma:  Type:= {
  Codom :  d;
  Arrow :> (Hom x (fG Codom))
}.

Variable v,u:  ObjectComma.

Definition CommaCommuteLaw:=
  [c1,c2,c3:c][v:(Hom c1 c2)][u:(Hom c1 c3)][w:(Hom c2 c3)]
  (v o w) =%S u.

Structure MorphismComma:  Type:= {
  MorComma    :> (Hom (Codom v) (Codom u));
  Prf_commute :  (CommaCommuteLaw v u (FMor fG MorComma))
}.

Definition EqualMorphismComma:= [g,h:MorphismComma]
  (MorComma g) =%S (MorComma h).

Lemma EqualMorphismComma_equiv:
  (Equivalence EqualMorphismComma).

Definition Setoid_MorphismComma:  Setoid:=
  (Build_Setoid EqualMorphismComma_equiv).

End Setoid_Comma.
```

As usual after the definition of objects and the setoid of morphisms we have to define the composition. The composition of two `MorphismComma` is the composition of the corresponding arrows `MorComma`. We only have to check that the composite arrow holds `CommaCommuteLaw`.

```
Section CompositionMorphismComma.

Variable v,u,w:  ObjectComma.

Variable g:  (MorphismComma v u).

Variable h:  (MorphismComma u w).

Lemma Comp_MorComma_commute:
  (CommaCommuteLaw v w (FMor fG (g o h))).

End CompositionMorphismComma.

Definition CommaComp:
  (v,u,w:ObjectComma)(Setoid_MorphismComma v u)→
                     (Setoid_MorphismComma u w)→
                     (Setoid_MorphismComma v w):=
  [v,u,w:ObjectComma]
  [sm:(Setoid_MorphismComma v u)][sm':(Setoid_MorphismComma u w)]
  (Build_MorphismComma (Comp_MorComma_commute sm sm')).
```

To build the composition mapoid we have to show that the composition map `CommaComp` holds the congruence laws.

```
Lemma CommaComp_congl:  (ConglLaw CommaComp).

Lemma CommaComp_congr:  (CongrLaw CommaComp).

Definition Comp_Comma:
  (v,u,w:ObjectComma)
  (BinMapoid (Setoid_MorphismComma v u)
             (Setoid_MorphismComma u w)
             (Setoid_MorphismComma v w)):=
  (Build_CompMapoid CommaComp_congl CommaComp_congr).

Lemma Assoc_Comma:  (AssocLaw Comp_Comma).
```

The final step is to define the identity. The identity in comma category is defined by the identity arrow in `d` that clearly holds `CommaCommuteLaw`.

```
Section Comma_Id.

Variable v:  ObjectComma.
```

```
Lemma Id_commute:
  (CommaCommuteLaw v v (FMor fG (Id 1!d (Codom v)))).

Definition Id_Comma:  (MorphismComma v v):=
  (Build_MorphismComma Id_commute).

End Comma_Id.

Lemma Idl_Comma:  (IdlLaw Comp_Comma Id_Comma).

Lemma Idr_Comma:  (IdrLaw Comp_Comma Id_Comma).
```

Provided with the laws of associativity and identity for composition we can define the comma category.

```
Definition COMMA: Category:=
  (Build_Category Assoc_Comma Idl_Comma Idr_Comma).

End CommaDef.
```

With the comma category defined we are able to show the result that relates left adjoints with initial objects in comma category. To state this lemma we have to have a way to say that there is a left adjoint for a functor `fG` and there is an initial object in `x↓fG` for any object `x`. Since we only can express an existence of an adjunction by `Adjunction`, that requires two functors, we define first `HasLeftAdjoint`. The proposition `(HasLeftAdjoint fG)` states that there is a functor that is a left adjoint of `fG`. We use the same reasoning to define `HasInitialForAnyX`.

```
Section HasDef.

Variable c,d:  Category.

Variable fG: (Functor d c).

Definition HasLeftAdjoint:  Prop:=
  (ExT [fF:(Functor c d)]
  (ExT [aFG:(Adjunction fF fG)] True)).

Definition HasInitialForAnyX: Prop:=
  (ExT [i:(x:c)(Initial (COMMA fG x))] True).

End HasDef.
```

The envisage lemma is simply stated by,

```
Lemma AdjInitialComma:  (c,d:Category)(fG:(Functor d c))
```

```
(HasLeftAdjoint fG)↔(HasInitialForAnyX fG).
```

3.5 Left Adjoint Unique up to Natural Isomorphism

In this section we want to show that the left adjoint of a functor is unique up to natural isomorphism:

Let C and D be categories and $F, F' : C \to D$ and $G : D \to C$ be functors, such that both F and F' are left adjoints of G. Then, there is a natural isomorphism α from F to F', i.e., there is a natural transformation $\alpha : F \to F'$ where α_X is an isomorphism for each $X \in |C|$.

We are talking about natural isomorphism but until now it has not been defined. So this is the first thing that we do.

```
Section NatIsoDef.

Variable c,d:  Category.

Variable fF,fF':(Functor c d).

Definition NTIsoLaw:= [nt:(NT fF fF')][nt':(NT fF' fF)]
  (c1:c)(IsoLaw 1!d (nt c1) (nt' c1)).

Structure NTIso:  Type:= {
  IsoNT     : (NT fF fF');
  InvIso    : (NT fF' fF);
  Prf_ntiso : (NTIsoLaw IsoNT InvIso)
}.

End NatIsoDef.
```

The result of left adjoint unique up to natural isomorphism is simply stated by the lemma `NTIsoLeftAdjoints`.

```
Lemma NTIsoLeftAdjoints:
  (c,d:Category)(fG:(Functor d c))(fF,fF':(Functor c d))
  (Adjunction fF fG)→(Adjunction fF' fG)→(NTIso fF fF').
```

Chapter 4

Cocartesian Liftings

In this section we define in `Coq` the concept of cocartesian lifting that is given in [BW90]:

Let C and D be categories and $F : C \to D$ be a functor. Let X be an object in C, A be an object in D and $f : F(X) \to A$ be a morphism in D. Then $u : X \to Y \in Mor_C$ is called cocartesian lifting by F for X and f iff

- $F(Y) = A$;
- $F(u) = f$;
- *for any* $v : X \to Z \in Mor_C$ *and* $g : F(Y) \to F(Z) \in Mor_D$ *such that*
  - $g \; o \; F(u) = F(v)$;

  there is only one morphism $w : Y \to Z \in Mor_C$ *such that*
  - $F(w) = g$;
  - $w \; o \; u = v$.

We also define examples of cocartesian lifting and show that the codomain of cocartesian lifting is unique up to isomorphism.

4.1 The Cocartesian Lifting Structure

To define cocartesian lifting we have to check, among other things, that an object and a morphism are image by a functor of another object and morphism. In order to assert that an object is image of another we need an equality for objects. Considering that in a category objects have sort `Type` and that in a functor the map for objects is a map in `Type` (and thus preserves the equality in `Type`), we conclude that the envisaged equality is the equality in `Type`. Having this in mind we define the predicate `IsImageF0`. Remark that in `Coq` the token `==` is the infix representation of the `Type` equality `eqT` (for more details see [PM96]).

Definition `IsImageF0:=`
```
  [c,d:Category][fF:(Functor c d)][c1:c][d1:d] (fF c1)==d1.
```

The first problem in considering this equality with our category implementation is that we are not able to show that if two objects are equal then there is a morphism between them (at least the identity morphism should exist). We can solve this problem by changing the Coq definition of category. In this case we have to say that there is an identity between two equal objects. However this solution is not easy to implement and it is out of the scope of this work since all the previous work would have to be redone. Instead of changing the category definition we introduce a global variable that will do the job of the identity between equal objects. We call `ident` to this global variable that in Coq is declared by the command `Parameter`.

```
Structure Identity:  Type:= {
  Ident     :> (c:Category)(c1,c2:c)(prf:(c1==c2))(Hom c1 c2);
  IdentId   :  (c:Category)(c1:c)(prf:(c1==c1))
               (Ident c c1 c1 prf)=%S(Id c1);
  IdentComp :  (c:Category)(c1,c2,c3:c)
               (prf:(c1==c2))(prf':(c2==c3))(prf'':(c1==c3))
               ((Ident c c1 c2 prf)o(Ident c c2 c3 prf'))=%S
               (Ident c c1 c3 prf'')
}.

Parameter ident:  Identity.
```

To simplify the syntax we define `identity`. In `identity`, the arguments `c`, `c1` and `c2` are implicit and that is not the case for `ident`.

Definition `identity:=`
```
  [c:Category][c1,c2:c][prf:(c1==c2)](ident c c1 c2 prf).
```

Remark that `Ident`, as we define, must hold some properties for ensuring that it is the identity modulo equality in `Type`. This properties are stated by `IdentId` and `IdentComp`.

With respect to the equality of morphisms we may think that the setoid equality will be enough, however this is not the case. Why? After having a candidate u for cocartesian lifting by F for f and X we must check that f is image of u by F. However the codomain of $F(u)$ is $F(Y)$ and the codomain of f is A. Even if we consider that $F(Y)$ and A are equal we can not use the equality of the setoid (that only compares morphisms with the same domain and codomain). Hence we have to extend the equality of the hom-setoid in such a way that we can compare morphisms with different domains and codomains. We define a new equality, `EqualHom`, that extends `Equal` and takes into account the new definition `identity`.

Definition EqualHom:

```
(c:Category)(c1,c2:c)(Hom c1 c2)→(c3,c4:c)(Hom c3 c4)→Prop:=
[c:Category][c1,c2:c][f:(Hom c1 c2)][c3,c4:c][g:(Hom c3 c4)]
c1==c3 ∧ c2==c4 ∧
(prf:(c1==c3))(prf':(c2==c4))
((f o (identity prf'))=%S((identity prf) o g)).
```

We write `f =%H g` to denote `(EqualHom f g)`.

```
Token "=%H".
Infix Assoc 6 "=%H" EqualHom.
```

Before presenting cocartesian lifting we still want to establish some results that will help us clear up the idea about `identity`. We start by checking that `EqualHom` is an equivalence relation.

Lemma `EqualHom_refl: (c:Category)(c1,c2:c)(f:(Hom c1 c2))`
`(f =%H f).`

Lemma `EqualHom_trans: (c:Category)(c1,c2,c3,c4,c5,c6:c)`
`(f:(Hom c1 c2))(g:(Hom c3 c4))(h:(Hom c5 c6))`
`(f =%H g)→(g =%H h)→(f =%H h).`

Lemma `EqualHom_sym: (c:Category)(c1,c2,c3,c4:c)`
`(f:(Hom c1 c2))(g:(Hom c3 c4))`
`(f =%H g)→(g =%H f).`

Next we prove that `Id` and `identity` are in the same equivalence class relative to `EqualHom`.

Lemma `Ident_Id: (c:Category)(c1,c2:c)(prf:(c1==c2))`
`(identity prf) =%H (Id c1).`

We also establish the relation between `EqualHom` and `Equal`.

Lemma `EqualEqualHom_Equiv: (c:Category)(c1,c2:c)`
`(f,g:(Hom c1 c2))(f =%H g)↔(f =%S g).`

By the definition of `identity` we may think that given two proofs of equality between objects we obtain two different morphisms. However we show that this is not the case.

Lemma `Ident_Proofs: (c:Category)(c1,c2:c)(prf,prf':(c1==c2))`
`(identity prf)=%S(identity prf').`

Finally we check the laws of the identity `identity` for composition.

```
Lemma IdentL: (c:Category)(c1,c2,c3:c)(prf:(c1==c2))
  (f:(Hom c2 c3)) ((identity prf) o f) =%H f.

Lemma IdentR: (c:Category)(c1,c2,c3:c)(prf:(c2==c3))
  (f:(Hom c1 c2)) f =%H (f o (identity prf)).
```

Remark that if we have two morphisms such that the codomain of one is equal to the domain of the other we should be able to compose them. We can define this composition by,

Definition CompHom:=[c:Category][c1,c2,c3,c4:c][prf:(c2==c3)]
[f:(Hom c1 c2)][g:(Hom c3 c4)] ((f o (identity prf)) o g).

However to define cocartesian lifting we do not have to deal with this kind of composition. Note that both this composition and the usual composition are congruent for the new equality `EqualHom`. We do not present these results because they are not needed but they are in the appendix.

Before cocartesian lifting it only remains to define the predicate `IsImageF1` that is true whenever a morphism is image of another by a functor.

```
Definition IsImageF1:=
  [c,d:Category][fF:(Functor c d)][c1,c2:c][d1,d2:d]
  [u:(Hom c1 c2)][f:(Hom d1 d2)] (FMor fF u) =%H f.
```

Finally we can define cocartesian lifting. Given two categories `c` and `d`, a functor `fF:(Functor c d)`, an object `x` in `c`, an object `a` in `d` and a morphism `f:(Hom (fF x) a))` in `c` , we say that `u:(Hom x y)` is a cocartesian lifting by `fF` for `f` and `x` iff we can show `aImagey` and `fImageu` and we can find a morphism `w:(Hom y z)` that holds the universal property. For simplicity we split the universal property into two properties, one representing the commutation, `CoCartCommuteLaw`, and other representing the uniqueness, `CocartUniqueLaw`.

```
Section CoCartesianLiftDef.

Variable c,d:  Category.

Variable fF: (Functor c d).

Variable x:  c.

Variable a:  d.

Variable f:  (Hom (fF x) a).

Section CoCartesianLiftLaws.

Variable y:  c.
```

```
Variable u: (Hom x y).

Hypothesis aImagey: (IsImageF0 fF y a).

Hypothesis fImageu: (IsImageF1 fF u f).

Definition Commute_d:=
  [z:c][v:(Hom x z)][g:(Hom (fF y) (fF z))]
  ((FMor fF u) o g) =%S (FMor fF v).

Variable w:
  (z:c)(v:(Hom x z))(g:(Hom (fF y) (fF z)))(prf:(Commute_d v g))
  (Hom y z).

Definition Commute_c:=
  [z:c][v:(Hom x z)][w:(Hom y z)]
  (u o w) =%S v.

Definition Unique_w:=
  [z:c][v:(Hom x z)][g:(Hom (fF y) (fF z))]
  [prf:(Commute_d v g)][w':(Hom y z)]
  ((FMor fF w') =%S g)∧(Commute_c v w')→((w prf) =%S w').

Definition CoCartCommuteLaw:=
  (z:c)(v:(Hom x z))(g:(Hom (fF y) (fF z)))(prf:(Commute_d v g))
  ((FMor fF (w prf)) =%S g)∧(Commute_c v (w prf)).

Definition CoCartUniqueLaw:=
  (z:c)(v:(Hom x z))(g:(Hom (fF y) (fF z)))(prf:(Commute_d v g))
  (w':(Hom y z))(Unique_w prf w').

End CoCartesianLiftLaws.

Structure CoCartLift: Type:= {
  y            :> c;
  u            :> (Hom x y);
  Prf_aImagey  :  (IsImageF0 fF y a);
  Prf_fImageu  :  (IsImageF1 fF u f);
  w            :  (z:c)(v:(Hom x z))(g:(Hom (fF y) (fF z)))
                  (prf:(Commute_d u v g))(Hom y z);
  Prf_commute  :  (CoCartCommuteLaw w);
  Prf_unique   :  (CoCartUniqueLaw w)
}.

End CoCartesianLiftDef.
```

4.2 The Cocartesian Lifting from Setoid to Presetoid

Herein we give an example of cocartesian lifting from `SETOID` to `PRESETOID`. We use, and we do not present again, the definitions of these two categories as well as the definition of the forgetful functor with respect to them.

Given an object `s` in `SETOID`, an object `p` in `PRESETOID` and a morphism `f:(Hom (FOForgetfulPS p) s)` in `SETOID` the candidate for Y is the preorder `p'`= ⟨s,R⟩. The relation `R` is the least reflexive and transitive closure of {(`(Map f p1)`,`(Map f p2)`): (p1,p2) ∈ `(Rel 1!p)`} that contains the equality of the setoid `s`.

```
Section CoCart_PRESETOID_SETOID.

Variable s:  SETOID.

Variable p:  PRESETOID.

Variable f:  (Hom (FOForgetfulPS p) s).

Inductive R: (Relation (Carrier s)):=
  Refl   : (s1:(Carrier s))(R s1 s1)
  | Trans : (s1,s2,s3:(Carrier s))(R s1 s2)→(R s2 s3)→(R s1 s3)
  | Pres  : (s1,s2:(Carrier s))(s1 =%S s2)→(R s1 s2)
  | Image : (s1,s2:(Carrier (FOForgetfulPS p)))
            (Rel s1 s2)→(R (Map f s1) (Map f s2)).

Lemma R_presequal:  (s1,s2:(Carrier s))(s1 =%S s2)→(R s1 s2).

Lemma R_po_refl:  (Reflexive R).

Lemma R_po_trans:  (Transitive R).

Definition p':  PreOrder:=
  (Build_PreOrder R_presequal R_po_refl R_po_trans).
```

Next we have to find the candidate for u. The candidate is `f`. It is very easy to check that the mapoid `f` is monotonous with respect to the preorders `p` and `p'`.

```
Lemma f_ismon:  (!IsMonotonous p p' f).

Definition u:  (MonMapoid p p'):= (Build_MonMapoid f_ismon).
```

With `p'` and `u` defined we have to show that `s` is the image of `p'` and `f` is the image of `u`, by the functor `FForgetfulPS`.

```
Lemma sImgp':  (IsImageFO FForgetfulPS p' s).
```

Lemma `fImgu: (IsImageF1 FForgetfulPS u f).`

Finally it only remains to find the morphism w that holds the universal property. This is, for any object `p''` in `PRESETOID` and any morphisms `v:(Hom p p'')` in `PRESETOID` and `g:(Hom (FOForgetfulPS p') (FOForgetfulPS p''))` in `SETOID` such that the commutation `prf` holds, we have to find the morphism w that respects the properties `CoCartCommuteLaw` and `CoCartUniqueLaw`. The candidate for w is `g`. To define the morphism `w` we only have to check that `g` is monotonous with respect to the preorders `p'` and `p''`.

```
Section PropUniversal.

Variable p'': PRESETOID.

Variable v: (Hom p p'').

Variable g: (Hom (FOForgetfulPS p') (FOForgetfulPS p'')).

Variable prf:
  (!Commute_d PRESETOID SETOID FForgetfulPS p p' u p'' v g).

Lemma g_ismon: (!IsMonotonous p' p'' g).

Definition w: (MonMapoid p' p''):= (Build_MonMapoid g_ismon).

End PropUniversal.
```

After we prove that `w` holds `CoCartCommuteLaw` and `CocartUniqueLaw` we are able to define the cocartesian lifting, that we call `CoCartPRESETOID`.

```
Lemma w_commute:
  (!CoCartCommuteLaw PRESETOID SETOID FForgetfulPS p p' u w).

Lemma w_unique:
  (!CoCartUniqueLaw PRESETOID SETOID FForgetfulPS p p' u w).

Definition CoCartPRESETOID:
  (!CoCartLift PRESETOID SETOID FForgetfulPS p s f):=
  (Build_CoCartLift sImgp' fImgu w_commute w_unique).

End CoCart_PRESETOID_SETOID.
```

4.3 The Cocartesian Lifting from Set to PTh

In this section we present another example of cocartesian lifting, this time from `SET` to `PTH`.

Given an object `a` in `SET`, an object `pt` in `PTH` and a morphism `f:(Hom (FOForgetfulPT pt) a)` in `SET` the candidate for Y is the propositional theory `pt'`$=\langle$a,G$\rangle$. The set of theorems `G` is the closure of the set of the images, by the extension of the map `f`, of the set of theorems of the propositional theory `pt`.

```
Section CoCart_PTH_SET.

Variable a: SET.

Variable pt: PTH.

Variable f: (Hom (FOForgetfulPT pt) a).

Inductive Gamma_a: (PLsig a):=
  Build_Gamma_a: (pf:(Lsig (Signature pt)))
  (Gamma 1!pt pf)→(Gamma_a (Extension f pf)).

Definition G: (Lsig a)→Prop:= (Closure Gamma_a).

Lemma G_close: (GammaClose G).

Definition pt': PTh:=(Build_PTh G_close).
```

Next we have to give the candidate for the cocartesian lifting. It is clear that the candidate is `f`. We only have to prove that `f` holds the `InclusionLaw`.

```
Lemma f_inclusion: (!InclusionLaw pt pt' f).

Definition u: (MorphismPTh pt pt'):=
  (Build_MorphismPTh f_inclusion).
```

With `pt'` and `u` defined we start by checking that `a` is image of `pt'` and that `f` is image of `u`, by the functor `FForgetfulPT`.

```
Lemma aImgpt': (IsImageFO FForgetfulPT pt' a).

Lemma fImgu: (IsImageF1 FForgetfulPT u f).
```

Finally it only remains to find the morphism w that holds the universal property. This is, given an object `pt''` in `PTH` a morphism `v:(Hom pt pt'')` in `PTH` and a morphism `g:(Hom (FOForgetfulPT pt') (FOForgetfulPT pt''))` in `SET` such that the commutation `prf` holds, we have to find the morphism w that respects the properties `CoCartCommuteLaw` and `CoCartUniqueLaw`. The candidate for w is `g`. To define the morphism `w` we only have to check that `g` holds the `InclusionLaw`.

```
Section PropUniversal.

Variable pt'': PTH.

Variable v: (Hom pt pt'').

Variable g: (Hom (F0ForgetfulPT pt') (F0ForgetfulPT pt'')).

Variable prf:
  (!Commute_d PTH SET FForgetfulPT pt pt' u pt'' v g).

Lemma g_inclusion: (!InclusionLaw pt' pt'' g).

Definition w: (MorphismPTh pt' pt''):=
  (Build_MorphismPTh g_inclusion).

End PropUniversal.
```

After we check that `w` is the unique morphism in `PTH` that commutes the diagram in `SET` we are able to build the cocartesian lifting form `SET` to `PTH`, that we call `CoCartPTH`.

```
Lemma w_commute:
  (!CoCartCommuteLaw PTH SET FForgetfulPT pt pt' u w).

Lemma w_unique:
  (!CoCartUniqueLaw PTH SET FForgetfulPT pt pt' u w).

Definition CoCartPTH:
  (!CoCartLift PTH SET FForgetfulPT pt a f):=
  (Build_CoCartLift aImgy fImgu w_commute w_unique).

End CoCart_PTH_SET.
```

4.4 Codomain of Cocartesian Lifting Unique up to Isomorphism

In this section we want to show that the codomain of cocartesian lifting is unique up to isomorphism:

Let C and D be categories and $F : C \to D$ be a functor. Let X be an object in C, A be an object in D and $f : F(X) \to A$ be a morphism in D. If the morphisms $u : X \to Y$ and $u' : X \to Y'$ in C are cocartesian liftings by F for

f and X then Y is isomorphic to Y'.

In Coq this lemma can be simply stated by,

Lemma `IsoCoCart:`
```
  (c,d:Category)(fF:(Functor c d))(x:c)(a:d)(f:(Hom (fF x) a))
  (ccl,ccl':(CoCartLift f))(Iso (y ccl) (y ccl')).
```

Chapter 5

Concluding Remarks

We achieved to define adjunctions and some heavy categories, like the comma category and the category of propositional theories, as well as results concerning these definitions without any trouble. We conclude that the category axiomatization proposed by Huet and Saïbi is good whenever we are defining concepts that do not refer explicitly the equality between objects. This is not the case of the cocartesian lifting where we have to check that an object is image of another. Considering that in a category objects have sort `Type` and that in a functor the map for objects is a map in `Type` we were forced to compare objects with the equality in `Type`. The problem in considering this equality is that with our category implementation we are not able to obtain an identity between equal objects. So we either change the category definition or we compensate this limitation artificially. We chose the second solution, providing an identity morphism between equal objects, since we did not want to loose the previous work. For further work we may consider developing a new definition of category in `Coq` where we can compare objects with a given relation, rather than comparing them with the equality in `Type`. We remark that along the way we were able to provide examples of cocartesian lifting easily.

With this incursion in the `Coq` system we conclude that `Coq` can be used more as a proof checker than as a proof assistant. Even using the `Hint` command that is supposed to help the automatization of the proofs we were not able to automatize very simple reasoning.

Bibliography

[AHS90] J. Adámek, H. Herrlich, and G. Strecker. *Abstract and concrete categories: the joy of cats.* John Wiley & Sons, 1990.

[Ano96] Anonymous. *The Coq Proof Assistant: The Standard Library.* INRIA Rocquencourt, first edition, December 1996.

[BW90] Michael Barr and Charles Wells. *Category Theory for Computing Science.* Prentice-Hall International Series in Computer Science. Prentice-Hall, 1990.

[Coq97] Thierry Coquand. Course notes in typed lambda calculus. Technical report, Chalmers University, 1997.

[HKPM96] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. *The Coq Proof Assistance: A Tutorial.* INRIA Rocquencourt, first edition, December 1996.

[HS95] Gérard Huet and Amokrane Saïbi. Constructive category theory. Technical report, INRIA Rocquencourt, 1995.

[Lan71] Saunders Mac Lane. *Categories for the Working Mathematician.* Springer-Verlag, 1971.

[PM96] Christine Paulin-Mohring. *The Coq Proof Assistant: Reference Manual V6.1.* INRIA Rocquencourt, first edition, December 1996.

[Saï95] Amokrane Saïbi. Théorie constructive des catégories: Draft. Technical report, INRIA Rocquencourt, 1995.

[Sel92] Jonathan P. Seldin. Coquand's calculus of constructions: A mathematical foundation for a proof development system. *Formal Aspects of Computing*, 4:425–441, 1992.