An experiment in writing formal mathematics in Coq

Carlos Simpson
carlos@math.unice.fr
CNRS, Laboratoire J.A. Dieudonne
Universite de Nice-Sophia Antipolis

Introduction

We present here a small attempt at writing proofs in a formal language. Many different efforts in this direction have been going on for quite some time. For example the *Journal of Formalized Mathematics* has been publishing articles verified in the MIZAR system since at least 1989. What we do here is by no means as much as what has been done in the MIZAR system [?], the HOL system [?], the Isabelle system [?], or in Coq [?]; and indeed most of what we do is contained in a Coq contribution dating from 1995. Nonetheless, it still seems necessary to do some work toward developing a mathematical style which would be convenient and easy to use for the ordinary mathematician. We try to make a contribution in this direction by presenting a proof development wherein the definitions and proofs are written in a style which is precise yet easy to understand. The reader will judge whether this has been successful.

We start from an axiomatization of set theory which is close to what Benjamin Werner constructed [?]. I suppose that if you replace our parameter Ens by the Ens which he constructed, then one should be able to prove the things which we suppose as axioms, relatively easily from his paper. However, we don't treat that here, rather we just start with some reasonable-sounding axioms.

Then we proceed with the standard development of ZFC set theory. Our definition of pair is slightly different from the standard one, designed to make the proofs of the main lemmas about pairs a little easier and more transparent.

Our goal is to introduce functions and prove some basic properties. We also introduce the set Naturals and prove some basic properties. See below for a summary of parameters, axioms, definitions and theorems (only after this summary does the actual Coq file start).

Most of what we do was already done by Guillaume Alexandre in his Coq contribution "An axiomatisation of intuitionistic Zermelo-Fraenkel set theory" (zf, 1995) [?], indeed we start with the same parameters and similar axioms. We use classical logic (which is easier) though: our parameter Ens is in Type. It is also worth noting that everything we do here is also already known in the context of the MIZAR system [?]. It also seems to have been done (again in a much more complete way) in the "ZF" logic in the Isabelle system [?]. Thus it is extremely unlikely that there is anything new in the present work. Rather, our purpose is twofold: first, to create some publicity for all of these efforts, basically by showing that anybody with a solid knowledge of mathematical foundations can begin to write mathematics in a theorem-proving system; and second to suggest a style of writing that might allow for ease of further advancement toward the ultimate goal of writing modern high-level research mathematics in a theorem-proving environment.

For example, the main difference between the present work and Alexandre's file [?] is in the writing style: we don't use the Coq proof engine but rather construct explicitly the proof terms, depending often on the Section mechanism. The file compiles in about 6 seconds using the native code version 6.2.3 on our Linux server.

This style of proof development can be thought of as an attempt to implement some of the ideas discussed in the TPHOLs '99 volume (for example the article by Markus Wenzel about Isar [?]).

Our functions `CHOICE` and `REPLACEMENT` are defined on all properties `P: Ens ->Prop`; however they satisfy the required properties `CHOICE_pr` and `REPLACEMENT_pr1`, `REPLACEMENT_pr2`, only when `P` holds for one element (in the case of `CHOICE`) or when `P` is bounded (in the case of `REPLACEMENT`). Since all other constructions are based on these (plus a few everywhere-defined constructions in the parameters) we gain the beneficial situation that all of our constructions are defined as `Ens->Ens`; no constructions depend on proofs. This can be contrasted with the "setoid" approach. Our constructions have the required properties only when the arguments have the required properties. Thus for example we have a construction `(EV f x)` defined for all sets `f` and `x`: however, it represents what one might call the evaluation of the function f on the element x, only when `(IN x (Domain f))`.

The system of axioms which we assume was just invented on the spur of the moment; it is basically ZFC but there are probably several redundancies. Also there is a certain interaction between this ZFC and the ambient type-theoretic language of Coq, which seems to mean that we are probably implicitly assuming the existence of one or maybe two inaccessible cardinals. See Werner's paper "Sets in Types, Types in Sets" for the discussion that underlies our justification for doing what we are doing. A possible perspective of the present work is that one could envision coding this first half of Werner's paper, giving a construction Type -¿ Ens with the properties he describes. This might be useful.

A short course in Coq

This section is addressed to readers who are unfamiliar with type-theoretical theorem proving systems (as I was until a few months ago). We try to explain rapidly the basic elements of how to write classical mathematics according to one interpretation of things.

This information was obtained from two sources: André Hirschowitz whom I warmly thank for explaining much of the basics; and the Coq system reference manual [?]. The present explanation is of course redundant with respect to [?]; however it might be useful because we concentrate on those aspects of Coq which we actually use.

Names of objects are strings of letters and numerals and the symbol `_` which takes the place of a space; actual spaces have meaning (they separate different objects).

All lines begin with a capitalized command and end with a period.

`Parameter` means we introduced something. `Definition` means we give a definition.

All objects are "terms". Some terms are also "Types". When a term is a type, it can have member terms. If `A` is a term of type `B` we write `A:B`.

A particular term is `Prop`. It is a type whose members are themselves types known as "propositions". If `P` is a proposition (written `P : Prop`) then a term `p` of type `P` (written `p:P`) denotes a "proof" of the proposition `P`.

If `A` and `B` are types then there is a type denoted `A -> B`. The terms of type `A -> B` are functions from `A` to `B`; and they can be applied to terms of type `A` to obtain terms of type `B`. Application is written as juxtaposition separated by a space and delimited by parenthesis. Thus if `f: A-> B` and if `a: A` then `(f a) : B`.

We work with the type `Prop`, and with one other type which we introduce in the first line of the file, called `Ens`. The member terms of `Ens` are thought of as "sets". However, an `X: Ens` is not itself a type. Its member elements are other `Y: Ens` which satisfy the property `(IN Y X)` which we introduce in the second line of the file.

The method described in the above paragraph has already been used by Guillaume Alexandre [?].

The main symbols which are used are parentheses and square brackets. Parentheses are sometimes used to delimit expressions in the usual way. The other use is to write the statement "for all". Suppose `P` is a property on `Ens`, i.e. `P: Ens-> Prop`. This could be coded by saying

```
Parameter P : Ens-> Prop.
```

or if we are in a section, equivalently

```
Variable P : Ens -> Prop.
```

Then the statement "for all X in Ens, P of X" is written:

```
Definition example1 : Prop := (X: Ens)(P X).
```

This line of code actually says that we define a new object, which is a term of type Prop (i.e. a proposition), which says that for all `X: Ens, (P X)`.

Square brackets mean "maps to". Thus we could write

```
Definition example2 : Ens -> Prop := [X: Ens] (P X).
```

In this example, example2 is actually equal to P and in fact Coq treats them as interchangeable. A slightly more nontrivial example would be the "constant property":

```
Parameter Q : Prop.
Definition example3 : Ens-> Prop := [X: Ens] Q.
```

This object is of type `Ens-> Prop`. If `Y: Ens` then the value `(example3 Y)` is equal to `Q` (i.e. interchangeable with `Q` by Coq).

Functions of several variables : if `A`,`B` and `C` are types then a term `f: A-> B-> C` means a term of type `A-> (B -> C)`; these parentheses can be omitted. If `f` is such a term and if `a: A` and `b: B` then `(f a)` is of type `(B-> C)`, so we can form `((f a) b)` which is of type `C`. Again the inner parentheses can be omitted and this can be written as `(f a b)`.

Caution : statements like `(a:A)` are not themselves propositions, i.e. we cannot write `(a:A):Prop`. This is the basic reason why we treat sets which are just terms and not types, and we introduce a separate inclusion relation for sets denoted `(IN X Y)`. If `X: Ens` and `Y: Ens` then `(IN X Y) : Prop`.

If `P: Prop` then we can issue the statement

```
Axiom axiom1 : P.
```

This introduces a term denoted axiom1 which is a proof of `P`.

One can note that proofs and other terms are manipulated by exactly the same symbols `(a:A)` and `[a:A]`. If `P` is a proposition then `(p:P) Q` means `P` implies `Q` ; if `Q` doesn't depend on `p` then it can also be written `P -> Q`. For other types `A,B`, the phrase `(a:A) B` means the type of functions which to `a:A` associate a term of type `B`. If `B` is independent of `a` then this is equivalent to the type `A-> B` (they are two ways of writing the same thing i.e. interchangeable for Coq). On the other hand, `B` could be a complex formula which depends on `a:A`. In this case, `(a:A) B` is the type, whose member terms are "sections" associating to each a a term of type "B of a". This is best understood when it arises in practice.

If one issues the statement

```
Theorem theorem1 : P.
```

this means that we claim a theorem whose proof is denoted theorem1, of type `P` i.e. `theorem1` is a proof of proposition `P`. This statement should be followed by a "proof" which can have many forms in the full version of Coq. In the very restricted part of Coq which we use here, all proofs are written as

```
Proof p.
```

where `p` is an already-existing term (which could be a new composite of old terms) of type `P`. Once this is done, the fact that there is a proof named `theorem1`, of the proposition `P`, is remembered but the structure of the old proof `p` is forgotten (this is called "proof irrelevance"). This is good because it prevents the machine from getting bogged down by remembering long proofs.

The last piece of machinery which needs to be explained is the notion of section. A section is started with the command

```
Section section_name.
```

and ended with the command

```
End section_name.
```

Within a section, one can introduce local variables (i.e. variables which are forgotten when the section is closed) with the command

```
Variable X : A.
```

which introduces a term `X` of type `A`; the phrase

```
Hypothesis p: P.
```

is equivalent to `Variable p: P.` and is (by convention) used when `P` is a proposition.

The command

```
Local A:= B.
```

defines a local variable denoted `A` (a redefinition of `B`). This is contrasted with `Definition` which, when used in a section, defines a global object which is remembered after the section.

The command

```
Remark step1 : P.
```

is the same as `Theorem` but it is local; it has to be followed by `Proof ... .` and, fortunately, its name will be forgotten when the section is closed. This allows us to use the names `step1`, `step2`, etc. in every section.

The command

```
Fact step1 : P.
```

is like `Remark` or `Theorem` but it is semi-local, it persists in the next higher section but is then forgotten.

The command `Lemma` is equivalent to `Theorem` (not local). These have to have unique names because their names cannot be reused.

A section will typically close with a `Lemma`; then right after the section is closed, the `Lemma` is redefined as a `Theorem`. The theorem takes into account the local variables which were defined in the section. (in reality, Coq remembers the lemma but changes its meaning to take into account the local variables; it is convenient to rewrite this as a theorem so that the new meaning is visible to the user). This is all best understood when it comes up in the file.

A simple example would be to redefine our example example2 from above:

```
Parameter P: Ens-> Prop.
Section example2_section.
Variable X : Ens.
Definition example2bis := (P X).
End example2_section.

Definition example2ter : Ens-> Prop :=
example2bis.
```

Advanced remark

A feature which is sometimes annoying and sometimes useful is that a `Lemma` or other global object defined in a section, does not necessarily "depend" on all of the `Variables` which were introduced in the section, but only on the `Variables` which actually go into its definition. For example, one CANNOT define the constant proposition of `example3` above by the command

```
Parameter Q : Prop.
Section example3_section.
Variable X: Ens.
Definition example3bis := Q.
End example3_section.

Definition example3ter : Ens->Prop :=
example3bis.
```

This last line generates an error: `example3bis` is of type `Prop` not of type `Ens-> Prop` because the variable `X` was not effectively used in the definition of `example3bis` inside the section.

This behavior can occasionally be used to good effect, for example to define several global objects within the same section; and at other times it requires that we define constant objects by hand.

Record types

The Coq language provides very sophisticated techniques for defining types by inductive definitions. We don't use these techniques here; however there is a special case which we use, the case of "Record" types. We use this in a limited case, to represent conjugation of propositions. Suppose we have several properties

```
Parameter P_1 : Ens -> Prop.
Parameter P_2 : Ens -> Prop.
Parameter P_3 : Ens -> Prop.
```

If we want to define the property "P_1 and P_2 and P_3", a conveniant approach is to use the following command:

```
Record ourconjugate [X: Ens] : Prop := {
conjugate_1 : (P_1 X);
conjugate_2 : (P_2 X);
conjugate_3 : (P_3 X)
}.
```

This is read as saying that the property (ourconjugate X) is defined as the logical and of the properties (P_1 X), (P_2 X), and (P_3 X).

This has the effect of introducing five new objects denoted ourconjugate, conjugate1, conjugate2, conjugate3 and Build_ourconjugate. (The names ourconjugate and conjugate_1 etc. are of course arbitrary; the system takes the assigned name ourconjugate and adds the prefix Build_ to obtain the fifth object.) The first object is a property ourconjugate : Ens -> Prop.

The best way to illustrate how the remaining objects work is by example: suppose we have X: Ens and suppose we have lemmas p1 : (P_1 X), p2 : (P_2 X), and p3 : (P_3 X). Then

```
(Build_ourconjugate X p1 p2 p3)
```

is a term of type (ourconjugate X), thus we can write

```
Theorem example4 : (ourconjugate X).
Proof (Build_ourconjugate X p1 p2 p3).
```

On the other hand, suppose we know c: (ourconjugate X). Then we obtain terms (conjugate_1 X c) : (P_1 X) etc., so we can write for example

```
Theorem example5 : (P_3 X).
Proof (conjugate_3 X c).
```

All in all, the `Record` construction, used in the limited way we do here, is just a replacement for successively applying "and". It seems to be more natural and gives a better guide to the proof development; specially when there are 3 or more terms there is no question about parenthetisation of the iterated "and"s; so we adopt the "Record" point of view rather than using "and".

Instructions

To be totally explicit, here are instructions for compiling the present file. The first thing to do is to get the Coq system installed on your computer; see the website [?]. This generates, in particular, two programs named "coqc" (batch-compiler) and "coqtop" (interactive top-level). The batch-compiler will compile any file named "filename.v" and tries to create the output file "filename.vo", giving applicable error messages when it stops. The same effect can be had in the coqtop shell by typing "Compile Module filename." One could also type "Load filename"; this gives the error messages but not the output file "filename.vo". The "coqtop" shell is an interactive shell where the user can use "tactics" to interactively prove theorems. This is useful for testing things out; refer to [?] for details.

To get the applicable file out of the present paper, copy the `.tex` source file to a "filename.v". There are two options: look for the line

```
(********* the actual Coq file starts here &&&******)
```

below in the source file, which is where the file starts. It is hidden from Tex by an `\invisible{  }` command. (The full summary of axioms and theorems is put in between the `\invisible{  }` command and the start of the actual Coq file.) The last line of the file is designed to be a comment for Coq. Another option is to simply remove the first character of the `.tex` file; then the whole discussion which you are currently reading is a comment, thus invisible, to Coq.

Extended example

Here we extract from the full file, some first examples of axioms and a couple of theorems, to give an idea of how things work. The reader should then look directly at the main file which is attached at the end of the `.tex` source file.

```
Parameter Ens : Type.

Parameter IN : Ens -> Ens -> Prop.

Definition SUB := [A,B: Ens] (C: Ens) ((IN C A) -> (IN C B)).

Record EQ  [A: Ens; B: Ens] : Prop := {
EQ_forwards : (SUB A B);
EQ_backwards : (SUB B A)
}.

(*** the following is a system theorem which we import ***)
Theorem false_implies_everything : (P : Prop) (False -> P).
Proof False_ind.

Definition DOESNT_EXIST := [P: Ens -> Prop]
((A : Ens)(not (P A))).

Definition EXISTS := [P : Ens -> Prop] (not (DOESNT_EXIST P)).

Parameter EmptySet : Ens.
Axiom EmptySet_pr : (DOESNT_EXIST ([X : Ens] (IN X EmptySet))).

Section EXISTS_th1_proof.
Variable P: Ens -> Prop.
Variable X : Ens.
Hypothesis p: (P X).
Hypothesis h : (DOESNT_EXIST P).

Remark step1 : (not (P X)).
Proof (h X).

Lemma EXISTS_th1_proof_Out : False.
Proof (step1 p).

End EXISTS_th1_proof.
```

```
Theorem EXISTS_th1 :
(P : Ens -> Prop)
(X : Ens )
(p : (P X))
(EXISTS P) .
Proof EXISTS_th1_proof_Out.


Section subset_reflexive.
Variable A:Ens.
Variable B:Ens.
Hypothesis h:(IN B A).

Lemma subset_reflexive_Out : (IN B A).
Proof h.

End subset_reflexive.

Theorem SUB_refl : (A:Ens) (SUB A A) .
Proof subset_reflexive_Out.

Section equality_reflexive.
Variable A:Ens.

Remark step1 : (SUB A A).
Proof (SUB_refl A).

Lemma equality_reflexive_Out : (EQ A A).
Proof (Build_EQ A A step1 step1).

End equality_reflexive.

Theorem EQ_refl : (A:Ens) (EQ A A).
Proof equality_reflexive_Out.

Section equality_symmetric.
Variable A:Ens.
Variable B:Ens.
```

```
Hypothesis h: (EQ A B).

Remark stepfor : (SUB A B).
Proof (EQ_forwards A B h).

Remark stepback : (SUB B A).
Proof (EQ_backwards A B h).

Lemma equality_symmetric_Out : (EQ B A).
Proof (Build_EQ B A stepback stepfor).

End equality_symmetric.

Theorem EQ_symm : (A,B: Ens) (EQ A B) -> (EQ B A).
Proof equality_symmetric_Out.


Section SUB_trans_proof.
Variable A: Ens.
Variable B: Ens.
Variable C: Ens.
Hypothesis h: (SUB A B).
Hypothesis k : (SUB B C).

Variable X : Ens.
Hypothesis m: (IN X A).

Remark step1 : (IN X B).
Proof (h X m).

Lemma SUB_trans_proof_Out : (IN X C).
Proof (k X step1).

End SUB_trans_proof.

Theorem SUB_trans :
(A,B,C: Ens)
(h:(SUB A B))
```

```
(k: (SUB B C))
(SUB A C).
Proof SUB_trans_proof_Out.

Section EQ_trans_proof.
Variable A: Ens.
Variable B: Ens.
Variable C: Ens.
Hypothesis h: (EQ A B).
Hypothesis k : (EQ B C).

Remark step1 : (SUB A B).
Proof (EQ_forwards A B h).

Remark step2 : (SUB B A).
Proof (EQ_backwards A B h).

Remark step3 : (SUB B C).
Proof (EQ_forwards B C k).

Remark step4 : (SUB C B).
Proof (EQ_backwards B C k).

Remark step5 : (SUB A C).
Proof (SUB_trans A B C step1 step3).

Remark step6 : (SUB C A).
Proof (SUB_trans C B A step4 step2).


Lemma EQ_trans_proof_Out :(EQ A C).
Proof (Build_EQ A C step5 step6).

End EQ_trans_proof.

Theorem EQ_trans:
(A,B,C: Ens)
(h: (EQ A B))
```

```
(k: (EQ B C))
(EQ A C).
Proof EQ_trans_proof_Out.



Section emptyset_empty.

Variable X: Ens.
Hypothesis h : (IN X EmptySet).

Lemma emptyset_empty_Out : False.
Proof (EmptySet_pr X h).

End emptyset_empty.

Theorem EmptySet_th1 :
(X: Ens)
(h: (IN X EmptySet))
False .
Proof emptyset_empty_Out.


Section emptyset_subset_everything.
Variable A: Ens.

Section emptyset_subset_everything_2.
Variable B: Ens.
Hypothesis h: (IN B EmptySet).

Remark step1 : False . Proof (EmptySet_th1 B h).

Lemma emptyset_subset_everything_2_Out : (IN B A) .
Proof (false_implies_everything (IN B A) step1).

End emptyset_subset_everything_2.

Remark step2 :
```

```
(B: Ens) (h: (IN B EmptySet)) (IN B A) .
Proof emptyset_subset_everything_2_Out.

Lemma emptyset_subset_everything_Out :
(SUB  EmptySet A).
Proof step2.

End emptyset_subset_everything.

Theorem EmptySet_th2 : (A : Ens) (SUB EmptySet A) .
Proof
emptyset_subset_everything_Out.

Definition Its_empty := [A : Ens] (X : Ens) (not (IN X A) ).

Section its_empty_implies_equals_emptyset_proof.
Variable A: Ens.
Hypothesis k: (Its_empty A).

Section ieieep1.
Variable X: Ens.
Hypothesis i: (IN X A).

Remark step1_1 : False .
Proof (k X i).

Lemma ieieep1_Out : (IN X EmptySet).
Proof (false_implies_everything (IN X EmptySet) step1_1).

End ieieep1.

Remark step1 : (SUB A EmptySet).
Proof ieieep1_Out.

Section ieieep2.
Variable X: Ens.
Hypothesis i: (IN X EmptySet).
```

```
Remark step2_1 : False.
Proof (EmptySet_th1 X i).

Lemma ieieep2_Out : (IN X A).
Proof (false_implies_everything (IN X A) step2_1).

End ieieep2.

Remark step2 : (SUB EmptySet A).
Proof ieieep2_Out.

Lemma ieieep_Out :(EQ A EmptySet).
Proof (Build_EQ A EmptySet step1 step2).

End its_empty_implies_equals_emptyset_proof.

Theorem its_empty_implies_equals_emptyset :
(A: Ens)
(k: (Its_empty A))
(EQ A EmptySet).
Proof ieieep_Out.

Theorem Its_empty_emptyset : (Its_empty EmptySet).
Proof EmptySet_th1.

Definition Its_nonempty := [A : Ens] (not (Its_empty A)).

Section Its_nonempty_proof.
Variable X : Ens.
Variable A : Ens.
Hypothesis  h : (IN X A).
Hypothesis  h2 : (Its_empty A).

Lemma Its_nonempty_proof_Out : False.
Proof (h2 X h).

End Its_nonempty_proof.
```

```
Theorem Its_nonempty_th1 :
(X,A: Ens)
(h : (IN X A))
(Its_nonempty A).
Proof Its_nonempty_proof_Out.
```

Summary

We now give a listing of all of the parameters and axioms, most of the definitions, and some of the theorems. This summary was generated by a simple c program from the Coq source file, then pruned by hand. The Coq source file itself is viewable only in the .tex source file; this is because it is somewhat long and is not best viewed printed out in postscript. For those who insist on having a paper version, directly printing that part of the source file will be more efficient than the "verbatim" tex command.

In case that might be useful, we have included the *full* summary in the source file, just before the beginning of the full Coq file.

```
Parameter Ens : Type .

Parameter IN : Ens -> Ens -> Prop .

Definition SUB :=
   [A, B : Ens]
   (C : Ens)
   ((IN C A) -> (IN C B)).

Record EQ [A : Ens ; B : Ens] : Prop := {
   EQ_forwards : (SUB A B);
   EQ_backwards : (SUB B A)
}.

Axiom excluded_middle :
   (P : Prop)
```

```
   (not (not P))
   ->
   P .

Theorem false_implies_everything :
   (P : Prop)
   (False -> P) .

Definition DOESNT_EXIST :=
   [P : Ens -> Prop]
   ((A : Ens) (not (P A))).

Definition EXISTS :=
   [P : Ens -> Prop]
   (not (DOESNT_EXIST P)).

Parameter CHOICE : (P : Ens -> Prop) Ens .

Axiom CHOICE_pr :
   (P : Ens -> Prop)
   (h : (EXISTS P))
   (P (CHOICE P)) .

Parameter EmptySet : Ens .

Axiom EmptySet_pr :
   (DOESNT_EXIST ([X : Ens] (IN X EmptySet))) .

Definition Bounded_By :=
   [A : Ens]
   [P : Ens -> Prop]
   (X : Ens)
   ((P X) -> (IN X A)).

Definition Bounded :=
   [P : Ens -> Prop]
   (EXISTS ([A : Ens] (Bounded_By A P))).
```

```
Parameter union : (A, B : Ens) Ens .

Axiom union_pr1 :
   (A, B : Ens)
   (SUB A (union A B)) .

Axiom union_pr2 :
   (A, B : Ens)
   (SUB B (union A B)) .

Axiom union_pr3 :
   (A, B, C : Ens)
   (h : (SUB A C))
   (k : (SUB B C))
   (SUB (union A B) C) .

Record nested_IN [A, B, C : Ens] : Prop := {
   nested_IN_ab : (IN A B);
   nested_IN_bc : (IN B C)
}.

Definition b_in_Union_a :=
   [A, B : Ens]
   (EXISTS ([C : Ens] (nested_IN B C A))).

Axiom Union_bounded :
   (F : Ens)
   (Bounded (b_in_Union_a F)) .

Parameter Singleton : Ens -> Ens .

Axiom Singleton_pr1 :
   (X : Ens)
   (IN X (Singleton X)) .

Axiom Singleton_pr2 :
   (X : Ens)
   (Y : Ens)
```

```
   (h : (IN Y (Singleton X)))
   (EQ X Y) .

Axiom Extensionality :
   (X, Y, Z : Ens)
   (h : (EQ X Y))
   (k : (IN Y Z))
   (IN X Z) .

Axiom Powerset_bounded :
   (X : Ens)
   (Bounded ([Y : Ens] (SUB Y X))) .

Parameter REPLACEMENT : (P : Ens -> Prop) Ens .

Axiom REPLACEMENT_pr1 :
   (P : Ens -> Prop)
   (b : (Bounded P))
   (X : Ens)
   ((P X) -> (IN X (REPLACEMENT P))) .

Axiom REPLACEMENT_pr2 :
   (P : Ens -> Prop)
   (b : (Bounded P))
   (X : Ens)
   ((IN X (REPLACEMENT P)) -> (P X)) .

Definition Zero :=
   EmptySet.

Definition Next : Ens -> Ens :=
   [X : Ens]
   (union X (Singleton X)).

Record Contains_NN [X : Ens] : Prop := {
   Contains_NN_zero : (IN Zero X);
   Contains_NN_nexts :
     (A : Ens)(h : (IN A X))(IN (Next A) X)
```

```
}.

Axiom Infinity_exists :
   (EXISTS Contains_NN) .

(****** the axioms are now finished ********)

Theorem EXISTS_th1 :
   (P : Ens -> Prop)
   (X : Ens)
   (p : (P X))
   (EXISTS P) .

Theorem SUB_refl :
   (A : Ens)
   (SUB A A) .

Theorem EQ_refl :
   (A : Ens)
   (EQ A A) .

Theorem EQ_symm :
   (A, B : Ens)
   (EQ A B)
   ->
   (EQ B A) .

Definition NEQ :=
   [A, B : Ens]
   (not (EQ A B)).

Theorem NEQ_symm :
   (A, B : Ens)
   (k : (NEQ A B))
   (NEQ B A) .

Theorem SUB_trans :
   (A, B, C : Ens)
```

```
   (h : (SUB A B))
   (k : (SUB B C))
   (SUB A C) .

Theorem EQ_trans :
   (A, B, C : Ens)
   (h : (EQ A B))
   (k : (EQ B C))
   (EQ A C) .

Record StrictSUB [A, B : Ens] : Prop := {
   StrictSUB_sub : (SUB A B);
   StrictSUB_neq : (not (EQ A B))
}.


Theorem EmptySet_th1 :
   (X : Ens)
   (h : (IN X EmptySet))
   False .

Theorem EmptySet_th2 :
   (A : Ens)
   (SUB EmptySet A) .

Definition Its_empty :=
   [A : Ens]
   (X : Ens)
   (not (IN X A)).

Definition Its_nonempty :=
   [A : Ens]
   (not (Its_empty A)).

Theorem Its_nonempty_th1 :
   (X, A : Ens)
   (h : (IN X A))
   (Its_nonempty A) .
```

```
Theorem Nothing_strictsub_Zero :
   (X : Ens)
   (not (StrictSUB X Zero)) .

Theorem Bounded_th1 :
   (P : Ens -> Prop)
   (A : Ens)
   (h : (X : Ens) ((P X) -> (IN X A)))
   (Bounded P) .

Theorem Bounded_th2 :
   (P : Ens -> Prop)
   (A : Ens)
   (h : (Bounded_By A P))
   (Bounded P) .



Record Restriction [P : Ens -> Prop; A, X : Ens] : Prop
:= {
   remaining_property : (P X);
   the_restriction : (IN X A)
}.

Theorem Restriction_bounded :
   (P : Ens -> Prop)
   (A : Ens)
   (Bounded (Restriction P A)) .

Definition Set_Of :=
   [A : Ens]
   [P : Ens -> Prop]
   (REPLACEMENT (Restriction P A)).


Theorem Set_Of_th1 :
   (A : Ens)
```

```
   (P : Ens -> Prop)
   (X : Ens)
   (h : (IN X A))
   (k : (P X))
   (IN X (Set_Of A P)) .

Theorem Set_Of_th2 :
   (A : Ens)
   (P : Ens -> Prop)
   (X : Ens)
   (h : (IN X (Set_Of A P)))
   (P X) .

Theorem Set_Of_th4 :
   (A : Ens)
   (P : Ens -> Prop)
   (SUB (Set_Of A P) A) .

Definition intersection :=
   [A, B : Ens]
   (Set_Of A ([X : Ens] (IN X B))).

Theorem intersection_th1 :
   (A, B : Ens)
   (SUB (intersection A B) A) .

Theorem intersection_th2 :
   (A, B : Ens)
   (SUB (intersection A B) B) .

Theorem intersection_th3 :
   (A, B, X : Ens)
   (i : (IN X A))
   (j : (IN X B))
   (IN X (intersection A B)) .

Theorem intersection_th4 :
   (A, B, C : Ens)
```

```
   (s : (SUB C A))
   (t : (SUB C B))
   (SUB C (intersection A B)) .

Theorem Singleton_th2 :
   (X : Ens)
   (Its_nonempty (Singleton X)) .

Theorem union_th1 :
   (A, B : Ens)
   (X : Ens)
   (h : (IN X A))
   (IN X (union A B)) .

Theorem union_th2 :
   (A, B : Ens)
   (X : Ens)
   (h : (IN X B))
   (IN X (union A B)) .

Theorem union_th3 :
   (A, B : Ens)
   (X : Ens)
   (h : (not (IN X A)))
   (k : (not (IN X B)))
   (not (IN X (union A B))) .

Theorem union_th4 :
   (P : Prop)
   (A, B, X : Ens)
   (i : (IN X (union A B)))
   (a : ((IN X A) -> P))
   (b : ((IN X B) -> P))
   P .

Theorem union_refl :
   (A : Ens)
   (EQ A (union A A)) .
```

```
Theorem union_symm :
   (A, B : Ens)
   (EQ (union A B) (union B A)) .

Definition has_leq_0_elements :=
   Its_empty.

Theorem zero_has_leq_0_elements :
   (has_leq_0_elements Zero) .

Definition has_geq_1_elements :=
   [A : Ens]
   (not (has_leq_0_elements A)).

Theorem has_geq_1_elements_th1 :
   (A : Ens)
   (X : Ens)
   (k : (IN X A))
   (has_geq_1_elements A) .

Theorem Singletons_have_geq_1_elements :
   (X : Ens)
   (has_geq_1_elements (Singleton X)) .

Definition has_leq_1_elements :=
   [A : Ens]
   (X, Y : Ens)
   (k : (IN X A))
   (l : (IN Y A))
   (EQ X Y).

Theorem has_leq_0_elements_implies_leq_1 :
   (A : Ens)
   (q : (has_leq_0_elements A))
   (has_leq_1_elements A) .

Theorem Singletons_have_leq_1_elements :
```

```
   (A : Ens)
   (has_leq_1_elements (Singleton A)) .

Definition has_geq_2_elements :=
   [A : Ens]
   (not (has_leq_1_elements A)).

Theorem has_geq_2_elements_th1 :
   (A : Ens)
   (X, Y : Ens)
   (k : (IN X A))
   (l : (IN Y A))
   (m : (not (EQ X Y)))
   (has_geq_2_elements A) .

Definition Doubleton :=
   [X : Ens]
   [Y : Ens]
   (union (Singleton X) (Singleton Y)).

Theorem Doubleton_th1 :
   (X, Y : Ens)
   (IN X (Doubleton X Y)) .

Theorem Doubleton_th2 :
   (X, Y : Ens)
   (IN Y (Doubleton X Y)) .

Theorem Doubleton_th3 :
   (X, Y : Ens)
   (A : Ens)
   (iXA : (IN X A))
   (iYA : (IN Y A))
   (SUB (Doubleton X Y) A) .

Theorem Doubleton_th4 :
   (X, Y : Ens)
   (Z : Ens)
```

```
   (k : (IN Z (Doubleton X Y)))
   (m : (not (EQ Z X)))
   (n : (not (EQ Z Y)))
   False .

Theorem Doubleton_th4_a :
   (X, Y : Ens)
   (Z : Ens)
   (k : (IN Z (Doubleton X Y)))
   (m : (not (EQ Z X)))
   (EQ Z Y) .

Theorem Doubleton_th4_b :
   (X, Y : Ens)
   (Z : Ens)
   (k : (IN Z (Doubleton X Y)))
   (n : (not (EQ Z Y)))
   (EQ Z X) .

Theorem distinct_Doubletons_have_geq_2_elements :
   (X, Y : Ens)
   (n : (not (EQ X Y)))
   (has_geq_2_elements (Doubleton X Y)) .

Theorem Big_Extensionality :
   (P : Ens -> Prop)
   (X, Y : Ens)
   (p : (P X))
   (e : (EQ X Y))
   (P Y) .

Theorem Substitute :
   (X, Y : Ens)
   (P : Ens -> Prop)
   (e : (EQ X Y))
   (p : (P Y))
   (P X) .
```

```
Theorem counting_lem1 :
   (X, Y : Ens)
   (m : (has_leq_0_elements X))
   (n : (has_geq_1_elements Y))
   (not (EQ X Y)) .

Theorem counting_lem2 :
   (X, Y : Ens)
   (m : (has_leq_1_elements X))
   (n : (has_geq_2_elements Y))
   (not (EQ X Y)) .

Definition First : Ens -> Ens :=
   [X : Ens]
   (Singleton X).

Definition Second : Ens -> Ens :=
   [X : Ens]
   (Doubleton Zero (Singleton X)).

Definition PAIR : Ens -> Ens -> Ens :=
   [X, Y : Ens]
   (Doubleton (First X) (Second Y)).

Theorem First_uniqueness :
   (A, B : Ens)
   (m : (EQ (First A) (First B)))
   (EQ A B) .

Theorem Singleton_uniqueness :
   (A, B : Ens)
   (m : (EQ (Singleton A) (Singleton B)))
   (EQ A B) .

Record pairwise_EQ [A, B, C, D : Ens] : Prop := {
   pairwise_EQ_ac : (EQ A C);
   pairwise_EQ_bd : (EQ B D)
}.
```

```
Theorem Doubleton_uniqueness :
   (P : Ens -> Prop)
   (A, B, C, D : Ens)
   (pA : (P A))
   (nB : (not (P B)))
   (pC : (P C))
   (nD : (not (P D)))
   (e : (EQ (Doubleton A B) (Doubleton C D)))
   (pairwise_EQ A B C D) .

Theorem Second_has_geq_2_elements :
   (A : Ens)
   (has_geq_2_elements (Second A)) .

Theorem Second_uniqueness :
   (A, B : Ens)
   (k : (EQ (Second A) (Second B)))
   (EQ A B) .

Theorem PAIR_uniqueness :
   (A, B, C, D : Ens)
   (e : (EQ (PAIR A B) (PAIR C D)))
   (pairwise_EQ A B C D) .

Theorem PAIR_uni_ac :
   (A, B, C, D : Ens)
   (e : (EQ (PAIR A B) (PAIR C D)))
   (EQ A C) .

Theorem PAIR_uni_bd :
   (A, B, C, D : Ens)
   (e : (EQ (PAIR A B) (PAIR C D)))
   (EQ B D) .

Definition is_a_pair :=
   [M : Ens]
   (EXISTS ([A : Ens]
```

```
      (EXISTS ([B : Ens] (EQ M (PAIR A B)))))).

Theorem a_pair_is_a_pair :
   (A, B : Ens)
   (is_a_pair (PAIR A B)) .

Theorem a_pair_is_a_pair_th2 :
   (A, B, U : Ens)
   (i : (EQ (PAIR A B) U))
   (is_a_pair U) .

Definition PR1 :=
   [M : Ens]
   (CHOICE ([A : Ens]
      (EXISTS ([B : Ens] (EQ M (PAIR A B)))))).

Theorem PAIR_proj1_pr1 :
   (M : Ens)
   (p : (is_a_pair M))
   (EXISTS ([B : Ens] (EQ M (PAIR (PR1 M) B)))) .

Definition PR2 :=
   [M : Ens]
   (CHOICE ([B : Ens] (EQ M (PAIR (PR1 M) B)))).

Theorem PAIR_proj_th1 :
   (M : Ens)
   (p : (is_a_pair M))
   (EQ M (PAIR (PR1 M) (PR2 M))) .

Theorem PAIR_proj_th2 :
   (X, Y : Ens)
   (pairwise_EQ X Y (PR1 (PAIR X Y)) (PR2 (PAIR X Y))) .

Theorem PAIR_proj_th3 :
   (X, Y : Ens)
   (EQ X (PR1 (PAIR X Y))) .
```

```
Theorem PAIR_proj_th4 :
   (X, Y : Ens)
   (EQ Y (PR2 (PAIR X Y))) .

Theorem PAIR_proj_uni :
   (M : Ens)
   (m : (is_a_pair M))
   (N : Ens)
   (n : (is_a_pair N))
   (e : (EQ (PR1 M) (PR1 N)))
   (f : (EQ (PR2 M) (PR2 N)))
   (EQ M N) .

Definition Powerset :=
   [X : Ens]
   (REPLACEMENT ([Y : Ens] (SUB Y X))).

Theorem Powerset_pr1 :
   (X, Y : Ens)
   (h : (SUB Y X))
   (IN Y (Powerset X)) .

Theorem Powerset_pr2 :
   (X, Y : Ens)
   (i : (IN Y (Powerset X)))
   (SUB Y X) .

Theorem Powerset_th1 :
   (A : Ens)
   (IN Zero (Powerset A)) .

Theorem Powerset_th2 :
   (A, X, Y : Ens)
   (x : (IN X (Powerset A)))
   (y : (IN Y (Powerset A)))
   (IN (union X Y) (Powerset A)) .

Theorem Powerset_th3 :
```

```
   (A, X, Y : Ens)
   (x : (IN X (Powerset A)))
   ((y : (IN Y (Powerset A))))
   (IN (intersection X Y) (Powerset A)) .

Theorem Powerset_th4 :
   (A, X : Ens)
   (x : (IN X A))
   (IN (Singleton X) (Powerset A)) .

Theorem Powerset_th5 :
   (A, X, Y : Ens)
   (x : (IN X A))
   (y : (IN Y A))
   (IN (Doubleton X Y) (Powerset A)) .

Theorem Powerset_th6 :
   (A, X : Ens)
   (x : (IN X A))
   (IN (Second X) (Powerset (Powerset A))) .

Theorem Powerset_th7 :
   (A, B : Ens)
   (a : (SUB A B))
   (SUB (Powerset A) (Powerset B)) .

Definition Element_of_NN :=
   [Y : Ens]
   (X : Ens)
   (k : (Contains_NN X))
   (IN Y X).

Theorem Zero_element_of_NN :
   (Element_of_NN Zero) .

Theorem Nexts_element_of_NN :
   (A : Ens)
   (h : (Element_of_NN A))
```

```
   (Element_of_NN (Next A)) .

Definition Set_containing_NN :=
   (CHOICE Contains_NN).

Theorem Set_containing_NN_pr :
   (Contains_NN Set_containing_NN) .

Theorem Bounded_NN_pre :
   (EXISTS ([A : Ens] (Bounded_By A Element_of_NN))) .

Theorem Bounded_NN :
   (Bounded Element_of_NN) .

Definition Naturals :=
   (REPLACEMENT Element_of_NN).

Theorem Naturals_pr1 :
   (X : Ens)
   (Element_of_NN X)
   ->
   (IN X Naturals) .

Theorem Naturals_pr2 :
   (X : Ens)
   (IN X Naturals)
   ->
   (Element_of_NN X) .

Theorem Zero_in_Naturals :
   (IN Zero Naturals) .

Theorem Nexts_in_Naturals :
   (A : Ens)
   (h : (IN A Naturals))
   (IN (Next A) Naturals) .

Theorem Contains_NN_Naturals :
```

```
   (Contains_NN Naturals) .

Theorem Naturals_th1 :
   (X : Ens)
   (h : (Contains_NN X))
   (SUB Naturals X) .

Theorem anything_sub_next_of_itself :
   (A : Ens)
   (SUB A (Next A)) .

Theorem anything_in_next_of_itself :
   (A : Ens)
   (IN A (Next A)) .

Definition naturals_induction_nexts :=
   [P : Ens -> Prop]
   (A : Ens)
   (h : (IN A Naturals))
   (k : (P A))
   (P (Next A)).

Theorem naturals_inductionA :
   (P : Ens -> Prop)
   (IH_zero : (P Zero))
   (IH_nexts : (naturals_induction_nexts P))
   (B : Ens)
   (p : (IN B Naturals))
   (P B) .

Definition in_then_strictsub :=
   [N : Ens]
   (X : Ens)
   (IN X N)
   ->
   (StrictSUB X N).

Definition not_in_itself :=
```

```
   [X : Ens]
   (not (IN X X)).

Record naturals_main_ind_hyp [N : Ens] : Prop := {
   naturals_main_ind1 : (in_then_strictsub N);
   naturals_main_ind2 : (not_in_itself N)
}.

Theorem some_things_strictsub_their_nexts :
   (A : Ens)
   (h : (not_in_itself A))
   (StrictSUB A (Next A)) .

Theorem in_then_strictsub_Zero :
   (X : Ens)
   (IN X Zero)
   ->
   (StrictSUB X Zero) .

Theorem zero_not_in_itself :
   (not_in_itself Zero) .

Theorem naturals_main_ind_for_Zero :
   (naturals_main_ind_hyp Zero) .

Theorem naturals_main_induction_next :
   (A : Ens)
   (h : (naturals_main_ind_hyp A))
   (naturals_main_ind_hyp (Next A)) .

Theorem naturals_main_induction_next_1 :
   (naturals_induction_nexts naturals_main_ind_hyp) .

Theorem naturals_main_th1 :
   (B : Ens)
   (i : (IN B Naturals))
   (naturals_main_ind_hyp B) .
```

```
Theorem naturals_main_th2_rewrite :
   (B : Ens)
   (i : (IN B Naturals))
   (A : Ens)
   (j : (IN A B))
   (StrictSUB A B) .


Theorem naturals_main_th3_rewrite :
   (B : Ens)
   (i : (IN B Naturals))
   (not (IN B B)) .

Definition Domain_prop :=
   [R : Ens]
   [X : Ens]
   (EXISTS ([Y : Ens] (IN (PAIR X Y) R))).

Definition Range_prop :=
   [R : Ens]
   [Y : Ens]
   (EXISTS ([X : Ens] (IN (PAIR X Y) R))).

Definition EV :=
   [R : Ens]
   [X : Ens]
   (CHOICE ([Y : Ens] (IN (PAIR X Y) R))).

Theorem EV_pr :
   (R : Ens)
   (X : Ens)
   (p : (Domain_prop R X))
   (IN (PAIR X (EV R X)) R) .


Theorem Domain_prop_bounded :
   (R : Ens)
   (Bounded (Domain_prop R)) .
```

```
Theorem Range_prop_bounded :
   (R : Ens)
   (Bounded (Range_prop R)) .

Definition Domain :=
   [R : Ens]
   (REPLACEMENT (Domain_prop R)).

Theorem Domain_pr1 :
   (R : Ens)
   (X : Ens)
   (p : (Domain_prop R X))
   (IN X (Domain R)) .

Theorem Domain_pr2 :
   (R : Ens)
   (X : Ens)
   (i : (IN X (Domain R)))
   (Domain_prop R X) .

Theorem EV_th1 :
   (R : Ens)
   (X : Ens)
   (i : (IN X (Domain R)))
   (IN (PAIR X (EV R X)) R) .

Theorem Domain_th1 :
   (R : Ens)
   (X, Y : Ens)
   (i : (IN (PAIR X Y) R))
   (IN X (Domain R)) .

Definition Range :=
   [R : Ens]
   (REPLACEMENT (Range_prop R)).

Theorem inverse_EV_th1 :
```

```
   (R : Ens)
   (Y : Ens)
   (i : (IN Y (Range R)))
   (IN (PAIR (inverse_EV R Y) Y) R) .

Theorem Range_th1 :
   (R : Ens)
   (X, Y : Ens)
   (i : (IN (PAIR X Y) R))
   (IN Y (Range R)) .

Theorem EV_in_range :
   (R, X : Ens)
   (i : (IN X (Domain R)))
   (IN (EV R X) (Range R)) .

Definition is_a_relation :=
   [R : Ens]
   (X : Ens)
   (i : (IN X R))
   (is_a_pair X).

Definition well_definedness :=
   [F : Ens]
   (X, Y, Z : Ens)
   (i : (IN (PAIR X Y) F))
   (j : (IN (PAIR X Z) F))
   (EQ Y Z).

Record is_a_function [F : Ens] : Prop := {
   function_is_a_relation : (is_a_relation F);
   function_well_definedness : (well_definedness F)
}.

Theorem function_sub :
   (F : Ens)
   (f : (is_a_function F))
   (G : Ens)
```

```
   (s : (SUB G F))
   (is_a_function G) .

Theorem function_th1 :
   (F : Ens)
   (f : (is_a_function F))
   (X, Y : Ens)
   (i : (IN (PAIR X Y) F))
   (EQ Y (EV F X)) .

Definition Extensionally_equivalent :=
   [F, G : Ens]
   (X : Ens)
   (i : (IN X (Domain F)))
   (j : (IN X (Domain G)))
   (EQ (EV F X) (EV G X)).

Theorem Extensionality_for_functions :
   (F, G : Ens)
   (f : (is_a_function F))
   (g : (is_a_function G))
   (d : (EQ (Domain F) (Domain G)))
   (e : (Extensionally_equivalent F G))
   (EQ F G) .

Theorem function_evaluation :
   (F, X, Y : Ens)
   (f : (is_a_function F))
   (i : (IN (PAIR X Y) F))
   (EQ Y (EV F X)) .

Definition restricted_to :=
   [A, F : Ens]
   (Set_Of F ([M : Ens] (IN (PR1 M) A))).

Theorem restricted_to_th5 :
   (A, F : Ens)
   (EQ (intersection A (Domain F))
```

```
          (Domain (restricted_to A F))) .

Theorem restricted_to_th9 :
   (A, F : Ens)
   (f : (is_a_function F))
   (X : Ens)
   (i : (IN X A))
   (j : (IN X (Domain F)))
   (EQ (EV F X) (EV (restricted_to A F) X)) .

Theorem empty_function_function :
   (is_a_function Zero) .

Theorem empty_function_domain :
   (EQ Zero (Domain Zero)) .

Record in_cartesian [A, B, P : Ens] : Prop := {
   in_cartesian_pair : (is_a_pair P);
   in_cartesian_pr1 : (IN (PR1 P) A);
   in_cartesian_pr2 : (IN (PR2 P) B)
}.

Theorem in_cartesian_bounded :
   (A, B : Ens)
   (Bounded (in_cartesian A B)) .

Definition Cartesian :=
   [A, B : Ens]
   (REPLACEMENT (in_cartesian A B)).

Theorem Cartesian_construction :
   (A, B : Ens)
   (X, Y : Ens)
   (i : (IN X A))
   (j : (IN Y B))
   (IN (PAIR X Y) (Cartesian A B)) .
```

```
Theorem Cartesian_pair_th1 :
   (A, B, X, Y : Ens)
   (i : (IN (PAIR X Y) (Cartesian A B)))
   (IN X A) .

Theorem Cartesian_pair_th2 :
   (A, B, X, Y : Ens)
   (i : (IN (PAIR X Y) (Cartesian A B)))
   (IN Y B) .

Theorem relations_in_cartesian :
   (R : Ens)
   (r : (is_a_relation R))
   (SUB R (Cartesian (Domain R) (Range R))) .

Theorem cartesian_is_relation :
   (A, B : Ens)
   (is_a_relation (Cartesian A B)) .

Theorem covering_th1 :
   (A, B, F, G : Ens)
   (f : (is_a_function F))
   (g : (is_a_function G))
   (u : (EQ (Domain F) (union A B)))
   (i : (EQ (restricted_to A F) (restricted_to A G)))
   (j : (EQ (restricted_to B F) (restricted_to B G)))
   (Extensionally_equivalent F G) .

Theorem covering_th2 :
   (A, B, F, G : Ens)
   (f : (is_a_function F))
   (g : (is_a_function G))
   (u : (EQ (Domain F) (union A B)))
   (v : (EQ (Domain G) (union A B)))
   (i : (EQ (restricted_to A F) (restricted_to A G)))
   (j : (EQ (restricted_to B F) (restricted_to B G)))
   (EQ F G) .
```

```
Theorem glueing_th4 :
   (F, G : Ens)
   (EQ (Domain (union F G))
          (union (Domain F) (Domain G))) .

Theorem glueing_th5 :
   (F, G : Ens)
   (f : (is_a_function F))
   (g : (is_a_function G))
   (e : (Extensionally_equivalent F G))
   (is_a_function (union F G)) .

(****** the following are for composition of functions
or more generally relations ************************)

Record Comp_inter [F, G, X, Y, Z : Ens] : Prop := {
   Comp_inter_xy : (IN (PAIR X Y) F);
   Comp_inter_yz : (IN (PAIR Y Z) G)
}.

Record Comp_prop [F, G, X : Ens] : Prop := {
   Comp_pair : (is_a_pair X);
   Comp_intern :
   (EXISTS ([Y : Ens] (Comp_inter F G (PR1 X) Y (PR2 X))))
}.

Definition Comp_INTER :=
   [F, G, X : Ens]
   (CHOICE ([Y : Ens] (Comp_inter F G (PR1 X) Y (PR2 X)))).

Theorem Comp_INTER_th1 :
   (F, G, X : Ens)
   (c : (Comp_prop F G X))
   (IN (PAIR (PR1 X) (Comp_INTER F G X)) F) .

Theorem Comp_INTER_th2 :
   (F, G, X : Ens)
   (c : (Comp_prop F G X))
```

```
   (IN (PAIR (Comp_INTER F G X) (PR2 X)) G) .

Theorem composition_bounded_by :
   (F, G : Ens)
   (Bounded_By (Cartesian (Domain F) (Range G))
      (Comp_prop F G)) .

Definition COMP :=
   [F, G : Ens]
   (REPLACEMENT (Comp_prop F G)).

Theorem COMP_th2 :
   (F, G : Ens)
   (SUB (COMP F G) (Cartesian (Domain F) (Range G))) .

Definition Comp_INTERp :=
   [F, G, X, Y : Ens]
   (Comp_INTER F G (PAIR X Y)).

Theorem COMP_th3p :
   (F, G, X, Y : Ens)
   (i : (IN (PAIR X Y) (COMP F G)))
   (IN (PAIR X (Comp_INTERp F G X Y)) F) .

Theorem COMP_th4p :
   (F, G, X, Y : Ens)
   (i : (IN (PAIR X Y) (COMP F G)))
   (IN (PAIR (Comp_INTERp F G X Y) Y) G) .

Theorem composition_th2 :
   (F, G : Ens)
   (f : (is_a_function F))
   (g : (is_a_function G))
   (is_a_function (COMP F G)) .

Theorem COMP_th5 :
   (F, G, X, Y, Z : Ens)
   (i : (IN (PAIR X Y) F))
```

```
   (j : (IN (PAIR Y Z) G))
   (IN (PAIR X Z) (COMP F G)) .

Theorem composition_th3 :
   (F, G, X : Ens)
   (i : (IN X (Domain F)))
   (j : (IN (EV F X) (Domain G)))
   (IN (PAIR X (EV G (EV F X))) (COMP F G)) .

Theorem composition_th4 :
   (F, G, X : Ens)
   (f : (is_a_function F))
   (g : (is_a_function G))
   (i : (IN X (Domain F)))
   (j : (IN (EV F X) (Domain G)))
   (EQ (EV (COMP F G) X) (EV G (EV F X))) .

Theorem COMP_assoc :
   (F, G, H : Ens)
   (EQ (COMP F (COMP G H)) (COMP (COMP F G) H)) .
```

References

[1] G. Alexandre. An axiomatisation of intuitionistic Zermelo-Fraenkel set theory: see
`http://coq.inria.fr/contribs-eng.html`.

[2] M. Wenzel. Article in *TPHOLs '99, Nice*, Springer LECTURE NOTES IN COMPUTER SCIENCE.

[3] B. Werner. Sets in Types, Types in Sets:
`http://pauillac.inria.fr/ werner/publis/zfc.ps.gz`

[4] B. Werner. An encoding of Zermolo-Fraenkel Set Theory in Coq: see
`http://coq.inria.fr/contribs-eng.html`.

SOME OF THE MAIN THEOREM PROVING SYSTEMS THAT I HAVE HEARD ABOUT ARE LISTED BELOW.

[5] Coq system reference manual:
http://coq.inria.fr/doc/main.html.

[6] Coq system:
http://coq.inria.fr/.

[7] Formal Methods web page at Oxford (this has a very complete listing of items on the web concerning formal methods):
http://www.afm.sbu.ac.uk/.

[8] ACL2 system:
http://www.cs.utexas.edu/users/moore/acl2/acl2-doc.html.

[9] HOL system:
http://www.afm.sbu.ac.uk/archive/formal-methods/hol.html.

[10] IMPS system (this seems to be out of commission now)—old address :
ftp://math.harvard.edu/imps/imps_html/imps.html.

[11] Isabelle system :
http://www.cl.cam.ac.uk/Research/HVG/Isabelle/.

[12] LEGO system :
http://www.dcs.ed.ac.uk/home/lego/.

[13] Mizar system :
http://mizar.uw.bialystok.pl/.

[14] *Journal of Formalized Mathematics* :
http://mizar.uw.bialystok.pl/JFM/.

[15] Nuprl system:
http://www.cs.cornell.edu/Info/Projects/NuPrl/nuprl.html.