Computer-certified Proofs - Interim Report

Ismail Suleman

Contents

1 Introduction **1**

2 Summary of topics covered so far **2**
2.1 Overview of the Coq system 2
2.1.1 Language 2
2.1.2 Type Hierarchy 2
2.2 A brief study of example code 2
2.3 Basic Gallina Commands 5
2.4 `->`, `forall`, `Set`, `Prop`, `Type` 6
2.4.1 `Set` 7
2.4.2 `Prop` 7
2.4.3 The Dependent Product 8
2.4.4 Law of Excluded Middle 9

3 Miscellaneous proofs and exercises **10**

4 Plan of future work **14**

1 Introduction

Coq is a formal proof assistant which provides an environment in which one can:

1. Write specifications of programs that can then be converted into code in another programming language eg Ocaml
2. Write mathematical theorems and proofs of those theorems (with the Coq compiler verifying the proofs)

Both of these aspects can be combined to create certified programs, where properties of the program can be formally verified.

In this project, I will be learning how to use Coq and then applying my knowledge to rewrite proofs of theorems in the Coq standard library in a more human-readable fashion.

2 Summary of topics covered so far

2.1 Overview of the Coq system

2.1.1 Language

Coq is designed principally as an interactive language. This means that commands are passed in sequence to a running `coqtop` program, which prints the output of the commands as they are executed. This is in contrast to other programming languages (eg Java, Haskell) which are designed to be compiled into an executable.

The language that Coq is given commands in is called *Gallina*. Note that each command starts with a capital letter and ends with a full stop (eg `Print` 5.), in a similar fashion to written languages, such as English.

2.1.2 Type Hierarchy

In most programming languages, we can define variables which have a type. In a similar fashion, in Coq we can declare variables, which may or may not be assigned a value. Variables in Coq have a type, and types are considered objects in their own right, and have a type themselves. Thus, there is an infinite hierarchy of types in Coq as detailed in figure 1.

There are 2 main sections of this hierarchy:

- Elements of the type `Set`. This is where terms and types related to programs reside.
- Elements of the type `Prop`. This is where terms and types related to propositions and proofs reside.

Both `Set` and `Prop` are of type `Type`(0). `Type`(0) is of type `Type`(1) and in general, `Type`(`i`) is of type `Type`(`j`) where `i` $<$ `j`. Also note that the Coq compiler by default abbreviates `Type`(`i`) to just `Type`, which means that the Coq compiler will say that `Type` is of type `Type`.

The type hierarchy is infinite in order to ensure that:

1. every element has a type
2. there are no loops in the type hierarchy

2.2 A brief study of example code

To give a brief understanding of Coq, an example of some code I have written is given here, and then explained in detail.

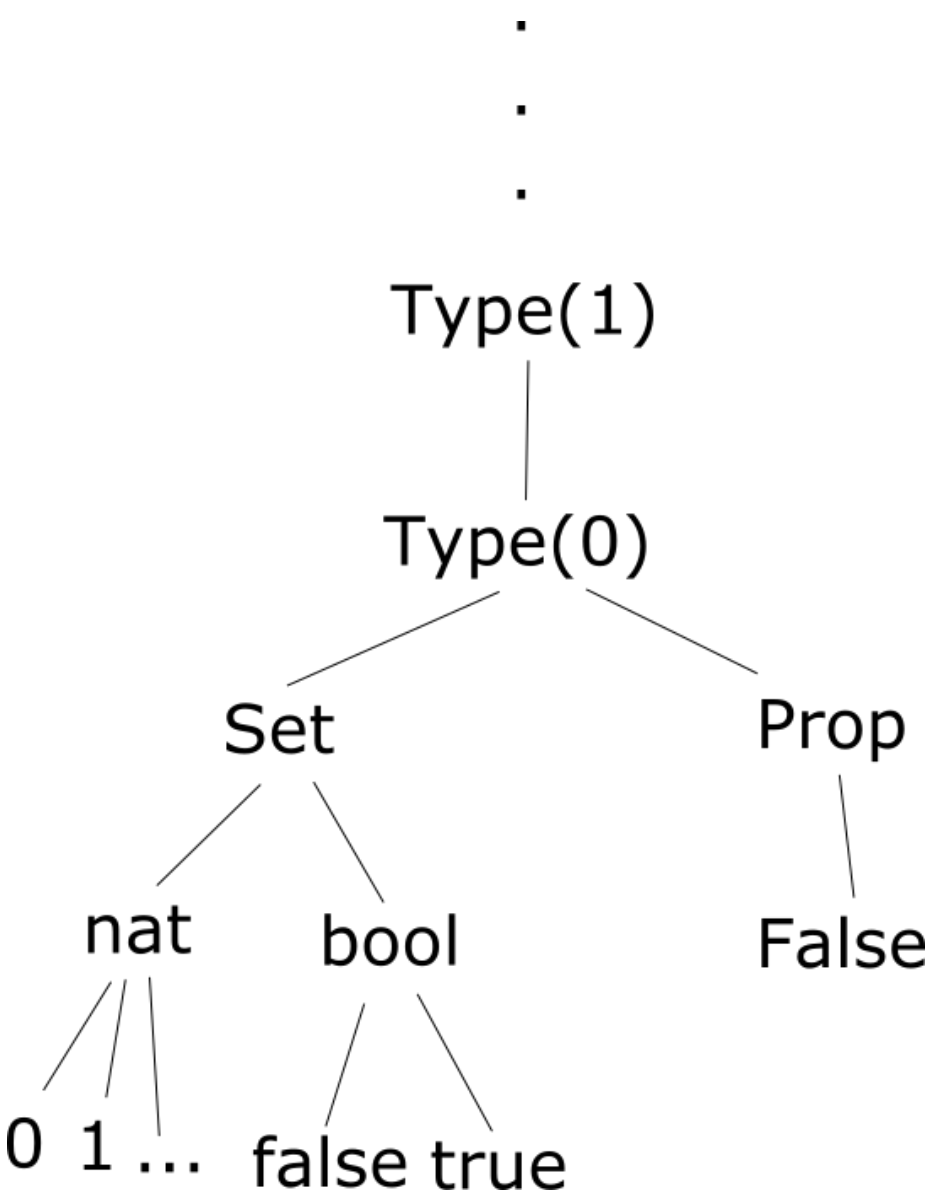

Figure 1: Hierarchy of types in Coq

```
(* part of exercise 4.3, p93 *)
Section A_declared.
  Variables (A: Set) (P Q: A → Prop) (R: A → A → Prop).

  Theorem all_perm: (forall a b: A, R a b) → forall a b: A, R b a.
  Proof (fun H a b ⇒ H b a).

  Theorem all_perm': (forall a b: A, R a b) → forall a b: A, R b a.
  Proof.
    intros H a b.
    apply H.
  Qed.
End A_declared.
```

The first line of this example,

```
(* Exercise 4.3, p93 *)
```

is a comment. Comments begin with `(*` and end with $*$).

```
Section A_declared.
```

Here we start a new section, which is later closed by the command `End A_declared.`. Sections are a way of organising proofs in a structured fashion. Note that sections are not the same as modules in Coq (which are not being covered here).

```
Variables (A: Set) (P Q: A → Prop) (R: A → A → Prop).
```

Here 4 variables, A, P, Q, and R are declared without an associated value. Variable A has type `Set`, P and Q have type `A → Prop` (ie a function that takes an element of type `A` and returns an element of type `Prop`) and R has type `A → A → Prop` (ie a function that takes 2 elements of type `A` and returns an element of type `Prop`).

```
Theorem all_perm: (forall a b: A, R a b) → forall a b: A, R b a.
```

A *theorem* is an element of type `Prop`, also known as a proposition. Proving a theorem is then a matter of finding an element whose type is the theorem itself.

Here we are defining a variable called `all_perm`, which has a type `(forall a b: A, R a b) → forall a b: A, R b a`, which itself is of type `Prop`. This theorem states

that if a binary predicate holds for all possible inputs `a` and `b` (where both inputs are of the same type), then the order in which the inputs `a` and `b` are supplied to the predicate doesn't affect whether or not the predicate holds, as it holds for all inputs (of the correct type).

```
Proof (fun H a b ⇒ H b a).
```

Here we give a proof of the preceding theorem, or in other words, we construct an element whose type is that of the preceding theorem. The element constructed here is a function which takes as input:

- `H` - a proof of the theorem `forall a b: A, R a b`, or equivalently, a function that takes as input 2 elements of the type `A` and returns an element of the type `R a b`
- `a` - an element of the type `A`
- `b` - an element of the type `B`

and then, considering `H` as a function, invokes `H` on `b` and `a` (in that order).

Proofs of theorems can be constructed in 2 ways:

1. By directly constructing the element that is the proof
2. By using *tactics* which provide a more user-friendly approach to constructing the element that constitutes the proof

`fun H a b ⇒ H b a` is an example of directly constructing the proof. Later on in the example, the theorem `all_perm'` is defined. This theorem is the same theorem as `all_perm`, but the proof of the theorem has been constructed using tactics. By comparison with the direct proof of `all_perm`, it can be seen how the tactics correspond to constructing different aspects of the proof term:

- `intros H a b.` corresponds to specifying that we have a function that takes 3 arguments ie `fun H a b ⇒ body_of_function_still_to_be_determined`
- `apply H.` specifies that the body of this function consists of simply applying the function H.

2.3 Basic Gallina Commands

Some basic Gallina commands are presented in this section:

Print Display some information on screen about the supplied term

> `Print 5.` Parameter
>
> Create a global declaration of a variable. Only the type of the variable is specified. No value can be assigned to the variable within or after the command.
>
> `Parameter Q: Prop.` Definition

Create a global definition of a variable. This is the same as the **Parameter** command but a value is assigned to the variable. A type can be specified explicitly or inferred.

`Definition x := false.`

`Definition y: bool := false` Variable

Create a local declaration of a variable. Essentially, this behaves the same as **Parameter** but the variable is only available inside the current section.

`Variable A: Prop.` Let

Create a local definition of a variable. Essentially, this behaves the same as **Definition** but the variable is only available inside the current section.

`Let x: nat := 5.` Section / End

Start / end a section. Note that sections can be nested. Within sections, one can create and use local and global variables. Outside a section, any global variables that used locally declared variables become functions that take as input values that take the place of the locally declared variables (as shown in the following example).

```
Section test.
  Variable X: nat.
  Definition x_plus_two: nat := X + 2.
End test.

Print x_plus_two.
(* This prints x_plus_two = fun X : nat ⇒  X + 2
      : nat → nat *)
```

Hypothesis

Synonym for **Variable**. Theorem / Proof

Declare an element of a given type, with a proof block following that constructs an element of that type. Generally used when writing theorems in Coq, with a proof of the theorem.

See section 2.2 for example usage and further explanation.

2.4 `->`, `forall`, `Set`, `Prop`, `Type`

In this section, some fundamental concepts in Coq will be described in more detail.

2.4.1 Set

The Set type, as discussed briefly in section 1, has as elements types related to program specifications. Examples of such types of type Set are:

- bool which has only 2 elements of this type, true and false
- nat which contains all natural numbers
- Function types. These are the types of functions that take any number of inputs and output an element whose type has Set as its type. Some examples of function types are bool → bool (a function that takes a boolean and returns a boolean) and nat → bool → nat (a function that takes a natural number and a boolean and returns a natural number).

2.4.2 Prop

Also discussed briefly in section 1, the Prop type has as its elements terms and types related to propositions and proofs. Examples of such types are shown in the following code:

```
Variables (A B: Prop).
Definition example_proof (a: A): A := a.
```

- A and B are called *propositions*. Propositions are elements that are of type Prop.
- example_proof is a *proof*. A proof is an element whose type is a proposition. In this case, the type of example_proof is A → A. Just as function types work within the Set type, the type A → A has type Prop. Note that in several places in Coq, the same Gallina syntax is used for constructing elements of the type Set and of the type Prop. Coq could have also been created using different syntax for Set and Prop, but using the same syntax emphasises the relationship between programs and proofs (known as the Curry-Howard correspondence).

2.4.2.1 Inhabited types

In maths, there are many propositions that can be written down which are syntactically valid, but are not true. In Coq, we can write propositions which are not true eg A → ¬A (note that ¬ in Gallina means negation). Syntactically, such propositions are valid, but there exist no proofs of them. In Coq, this means that there exist no elements which have that proposition as their type. Thus the notion of whether or not a proposition holds or not in Coq is equivalent to the notion of whether or not the type is inhabited (ie there exists an element with that type) or not.

A simple example of this is the term `False`, which is a proposition. There are no elements of type `False`, which is another way of saying that `False` is not provable (unless we have a contradiction).

2.4.2.2 Predicates and Connectives

Also directly related to the type `Prop` are *predicates* and *connectives*.

- Predicates are functions that take any number of inputs whose type is of type `Set` and return a proposition. An example of a predicate is: `Variable H: nat → Prop`. The function `H` takes as input a number and returns a proposition. An example of such a function is a function is_prime, which would return a proposition asserting that the input number is a prime number.
- Connectives are functions that take any number of inputs which are propositions and return a proposition. An example of a connective is `^` or `and`, which comes built-in in Coq. `and` is a function of type `Prop → Prop → Prop` and returns a proposition which is the logical conjunction of the two input propositions.

2.4.3 The Dependent Product

The dependent product is one of the main features that gives Coq a lot of expressiveness. The essential idea behind the dependent product is that the type of a function can depend on the input values. To make this clearer, here is a simple example:

```
Variables (A: Set) (B: A → Type).

Variable C: forall a: A, B a.
```

In this example, we have the following terms:

- `A` is an element of type `Set` (so `A` could be eg `nat` or `bool`)
- `B` is a function mapping elements of `A` to a type. `B` can also be thought of as a family of types indexed by elements of type `A`
- `C` is a function that takes one argument a. `C` is an example of the dependent product. The type of `C` depends on the value that it is passed. Note that `B` is an arbitrary function that returns a type.

Dependent products are used throughout Coq, for writing program specifications, propositions and proofs.

In section 2.2, the example code discussed included an example of the dependent product:

```
Variables (A: Set) (P Q: A → Prop) (R: A → A → Prop).
Hypothesis all_perm: (forall a b: A, R a b) → forall a b: A, R b a.
```

In this example, the function `R` is a binary predicate. `R` is then used in two dependent products in the type of `all_perm`. Note that `all_perm` itself can be seen as simply a function that takes as input a function that has a dependent type and returns a function that has a dependent type.

2.4.4 Law of Excluded Middle

In section 2.4.2.1, it was said that a proof of a proposition is an element which has the proposition as its type. This means that, in Coq, to prove a proposition, it is not allowed to simply prove that the negation of the proposition leads to a contradiction. Rather, a direct proof of the proposition must be constructed.

A consequence of this is that various mathematical theorems cannot be proved within Coq, which are considered provably true in classical logic. The fundamental example of this is the *law of excluded middle*:

```
Theorem p_or_not_p: forall P: Prop, (P ∨ ¬P).
(* Note that ∨ is the disjunction operator aka 'or' *)
```

In Coq, it is impossible to construct a proof of this theorem. Note that it is possible to construct a proof of the double-negation of this theorem:

```
Theorem double_neg_a_or_not_a: forall A: Prop, ¬¬(A ∨ ¬A).
Proof.
  intros A H.
  apply H.
  right.
  intros a.
  apply H.
  left.
  assumption.
Qed.
(* This constructs a proof which is the following:
   fun (A : Prop) (H : ¬(A ∨ ¬A) ⇒
   H (or_intror (fun a : A ⇒ H (or_introl a))) *)
```

This then also indicates that the following theorem can not be proved in Coq:

```
Theorem double_neg: forall A: Prop, ¬¬A → A.
```

This theorem can be thought of as stating that if the negation of a theorem is false, then the theorem itself must be true. This is essentially proof by contradiction, a concept rejected by the constructive logic, *The Calculus of Constructions*, which Coq uses, as proof by contradiction does not directly construct a proof of a theorem, but instead shows that the converse leads to a contradiction.

3 Miscellaneous proofs and exercises

The following exercises are from the book *Interactive Theorem Proving and Program Development* by Yves Bertot and Pierre Castéran. The proofs of the theorems were constructed by me.

```
(* Exercise 4.3, p93 *)
Section A_declared.
  Variables (A: Set) (P Q: A → Prop) (R: A → A → Prop).

  Theorem all_perm: (forall a b: A, R a b) → forall a b: A, R b a.
  Proof (fun H a b ⇒ H b a).

  Theorem all_imp_dist: (forall a: A, P a → Q a) → (forall a: A, P a) →
    forall a: A, Q a.
  Proof (fun H0 H1 a ⇒ H0 a (H1 a)).

  Theorem all_delta: (forall a b: A, R a b) → forall a: A, R a a.
  Proof (fun H a ⇒ H a a).

End A_declared.

(* Exercise 4.4, p96 *)
Section Ex4_4.
  Definition id: forall A: Set, A → A := fun _ a ⇒ a.

  Definition diag: forall A B: Set, (A → A → B) → A → B :=
    fun _ _ f a ⇒ f a a.

  Definition permute: forall A B C: Set, (A → B → C) → B → A → C :=
    fun _ _ _ f b a ⇒ f a b.

  Require Import ZArith.
  Definition f_nat_Z: forall A: Set, (nat → A) → Z → A :=
    fun _ f z ⇒ f (Zabs_nat z).

End Ex4_4.
```

```
31
(* Exercise 4.5, p97 *)
33 Section Ex4_5.
  Theorem all_perm2: forall (A: Type) (P: A → A → Prop), (forall x y: A, P x y)
    → forall x y: A, P y x.
35  Proof (fun _ _ f x y ⇒ f y x).

37  Theorem all_perm2_2: forall (A: Type) (P: A → A → Prop), (forall x y: A, P x
    y) → forall x y: A, P y x.
  Proof.
39    intros A P f x y.
    apply f.
41  Qed.

43  Theorem resolution: forall (A: Type) (P Q R S: A → Prop), (forall a: A, Q a →
    R a → S a) → (forall b: A, P b → Q b) → (forall c: A, P c → R c → S c).
  Proof (fun _ _ _ _ _ f g c P_c R_c ⇒ f c (g c P_c) R_c).
45
  Theorem resolution2: forall (A: Type) (P Q R S: A → Prop), (forall a: A, Q a →
     R a → S a) → (forall b: A, P b → Q b) → (forall c: A, P c → R c → S c).
47  Proof.
    intros A P Q R S f g c P_c R_c.
49    apply f.
    apply g. exact P_c.
51    exact R_c.
  Qed.
53
End Ex4_5.
```

```
  (* Exercise 5.3 *)
2 Section Ex_5_3.
  Theorem not_false: ¬False.
4  Proof.
    unfold not; intros f; assumption.
6  Qed.

8  Theorem modus_ponens: forall P Q: Prop, P → (P → Q) → Q.
  Proof (fun P Q p f ⇒ f p).
10
  Theorem triple_neg_is_equal_to_single_neg: forall P: Prop, ¬¬¬P → ¬P.
12  Proof.
    intros P f p.
14    apply f.
    intros g.
16    exact (g p).
  Qed.
18  (* fun (P : Prop) (f : ¬ ¬ ¬ P) (p : P) ⇒ f (fun g : ¬ P ⇒ g p)
     : forall P : Prop, ¬ ¬ ¬ P → ¬ P*)
```

```
20
  Theorem not_not_not_P_and_P_implies_Q: forall P Q: Prop, ¬¬¬P → P → Q.
22  Proof.
    intros P Q H p.
24    exact (False_ind Q (H (fun f: ¬P ⇒ f p))).
  Qed.
26
  Theorem question4: forall P Q: Prop, (P → Q) → ¬Q → ¬P.
28  Proof.
    intros P Q H nQ p.
30    exact (nQ (H p)).
  Qed.
32
  Theorem question5: forall P Q R: Prop, (P → Q) → (P → ¬Q) → P → R.
34    intros P Q R H H0 p.
    exact (False_ind R (H0 p (H p))).
36  Qed.

38 End Ex_5_3.


40
(* Exercise 5.5 *)
42 Section Ex_5_5.
  Theorem a_b_c_d_or: forall (A: Set) (a b c d: A), a=c ∨ b=c ∨ c=c ∨ d=c.
44  Proof.
    intros A a b c d.
46    right. right. left. reflexivity.
  Qed.
48 End Ex_5_5.

50 (* Exercise 5.6 *)
Section Ex_5_6.
52  Theorem conj_dist: forall A B C: Prop, A ∧ (B ∧ C) → (A ∧ B) ∧ C.
  Proof.
54    intros A B C H.
    elim H.
56    intros a b_and_c.
    elim b_and_c.
58    split. split.
    assumption. assumption. assumption.
60  Qed.

62  Theorem imp_conj: forall A B C D: Prop, (A → B) ∧ (C → D) ∧ A ∧ C → B ∧
      D.
  Proof.
64    intros A B C D H.
    elim H. intros H0 H1. elim H1. intros H2 H3. elim H3. intros a c.
66    exact (conj (H0 a) (H2 c)).
  Qed.
68
```

```
  Theorem not_true_and_false: forall A: Prop, ¬(A ∧ ¬A).
70  intros A H.
    elim H.
72  intros a neg_a.
    exact (neg_a a).
74 Qed.

76 Theorem disj_dist: forall A B C: Prop, A∨ (B∨ C) → (A∨ B)∨ C.
   Proof.
78  intros A B C H.
    elim H.
80  - left. left. assumption.
    - intros b_or_c.
82    elim b_or_c.
      + left. right. assumption.
84    + right. assumption.
   Qed.
86
   Theorem double_neg_a_or_not_a: forall A: Prop, ¬¬(A ∨ ¬A).
88 Proof.
    intros A H.
90  apply H.
    right.
92  intros a.
    apply H.
94  left.
    assumption.
96 Qed.
   (* fun (A : Prop) (H : ¬(A ∨ ¬A) ⇒
98    H (or_intror (fun a : A ⇒ H (or_introl a))) *)

100 Theorem a_or_b_and_not_a_gives_b: forall A B: Prop, (A ∨ B) ∧ ¬A → B.
   Proof.
102  intros A B H.
    elim H. intros a_or_b not_a.
104 elim a_or_b.
    - intros a.
106   elim not_a.
      assumption.
108 - intros b. assumption.
   Qed.
110 (* fun (A B : Prop) (H : (A ∨ B) ∧ ¬ A) ⇒
and_ind
112 (fun (a_or_b : A ∨ B) (not_a : ¬ A) ⇒
    or_ind (fun a : A ⇒ False_ind B (not_a a)) (fun b : B ⇒ b) a_or_b)
     H
114   : forall A B : Prop, (A ∨ B) ∧ ¬ A → Bfun (A B : Prop) (H : (A
    ∨ B) ∧ ¬ A) ⇒
and_ind
116 (fun (a_or_b : A ∨ B) (not_a : ¬ A) ⇒
```

```
    or_ind (fun a : A ⇒ False_ind B (not_a a)) (fun b : B ⇒ b) a_or_b)
     H
118      : forall A B : Prop, (A ∨ B) ∧ ¬ A → B*)
End Ex_5_6.
120
(* Exercise 5.10 *)
122 Section Ex_5_10.
  Require Import Arith.
124   Theorem plus_permute2: forall n m p: nat, n+m+p = n+p+m.
  Proof.
126     intros n m p.
    pattern (n + p + m) at 1. rewrite ← plus_assoc.
128     pattern (p + m) at 1. rewrite plus_comm.
    rewrite plus_assoc.
130     reflexivity.
  Qed.
132 End Ex_5_10.
```

4 Plan of future work

So far, I have studied the book *Interactive Theorem Proving and Program Development* mentioned in section 3 upto and including chapter 5. I still need to study chapter 6 in order to gain an understanding of inductive data types.

Once this has been completed, I will be rewriting proofs of some theorems found in the standard library, in a more human-readable fashion. Examples of areas of the standard library I may work on are areas concerning basic arithmetic, such as the `Arith` and `ZArith` modules.

Taking into account university exams and the deadline for the final report, a provisional timetable for the rest of the project is as follows:

Task	Date
Study Chapter 6 of the book	16 January - end of January
Work on the standard library	start of February - end of May
Complete writing the final report	April