Lab Session
September 27, 2017
Introduction to Coq

Olivier Hermant

1 Introduction

1.1 Presentation

Coq is a proof-assistant that relies on the Calculus of Inductive Constructions (with universes). This is a very powerful type system for λ-terms, that guarantees that all typable λ-terms are strongly normalisable (*i.e.* every β-reduction sequence is finite). Many constructs have been added to the core λ-abstraction and application, in particular the so called *inductive types* that are very convenient to express structures as lists, trees, ...[1]

It is possible to express, almost natively, natural numbers (unary as well as binary), lists, vectors, matrices, ... Coq is a proof-assistant that can also be used as a programming language: we can define almost every function[2], there is only two obstacles to this use:

- the efficiency of the code. Compared to a C, Python, Java or OCaml program, the same program written in Coq is very slow. One of the major advantages is the richness of the type system that allows us to specify programs with extremely refined types and to state and prove properties on the function we are writing. Notice that this feature, necessary for a proof-assistant, is the main reason for the loss of performance since the program carries all the long those proofs inside it. We have to choose on which side we are: proof assistant *vs.* programming language.

- as already said, Coq's type system has limitations on the expressivity (structurally decreasing inductive arguments, for instance) which sometimes are natural, and sometimes a burden.

[1] as well, it allows a stronger theoretical power. For instance, in the Calculus of Constructions (or CoC – hence the name Coq) it is impossible to prove *from the inside* that `0` is different from 1.

[2] although the language is obviously not Turing-complete (remember: all the function that one can express in Coq are terminating/strongly normalisable), only very vicious functions cannot be expressed.

We will first look at propositional logic, then at predicate logic (first-order logic). Notice that in the Calculus of Constructions, types are themselves terms, so they also can be abstracted over (through λ-abstraction) and give them as arguments to functions, like in the polymorphic identity:

$$\lambda\alpha : \texttt{Type}.\lambda x : \alpha.x$$

or in the dependent array type:

$$\lambda n : \texttt{Type}.\texttt{array n}$$

where `array n` is the type of arrays of length `n`. We can also have polymorphic dependent arrays, etc, etc. There is (almost) no limits to fun with dependent types.[3]

1.2 How to start Coq

Use the `coqide` IDE, that can be launched as a command line or with the menu. It is also possible to interact directly with Coq through a command line interface by typing `coqtop` in a terminal.

If you prefer using `emacs` (which I can understand), the `proofgeneral` major mode has been installed. It will not be described in details in the sequel. To use it, you must, in a terminal, type `proofgeneral`. The `emacs` text editor will then arise, it is entirely customized for Coq. Type your code and save it with the "`.v`" Coq extension (for instance `first_order_logic.v`). Once your file will be given the extension, additional menu options will appear.

You need to follow those steps:

1. define the symbols you will use.
2. define the proposition to prove.
3. write the proof.
4. ask Coq to check that this proof is correct, that is to say to check that the proof (which is a λ-term, even if it does not appear under this form) is well-typed.

1.3 How to use Coq

1.3.1 Commands and their Interpretation

All Coq commands end with a dot "`.`". Our first command, `Check`, is used to check the type of constants and definitions. For instance `Check 3.` (including the capital letter and the dot) is a valid command.

After that you need to start the interpreter so as to evaluate the command.

[3] What should be surprizing is not the *possibility* to do that, but the fact that the resulting system is still consistent. We will not go that far.

- in `coqide`:
  - write the command in the left window.
  - Go to the `Navigation` menu, `Forward` item.
  - the `Check 3.` instruction will be overlined in green, indicating that `Check 3.` has been checked. For (named) definitions and theorems, it means that the definition has been saved and can be used from now on.
  - the lower right window will display Coq's answer, `3 : nat`, meaning that `3` has type `nat`. You will also see a warning message, since this particular command should be used through the menu in `coqide`. For this: select the character `3` in the left window, and go to the menu item `Queries -> Check`. You will see the answer in the lower right window.
- with `emacs`, go to the `Proof-General` menu and then to `Next Step`. This starts the interpreter and executes the first (and unique) command. If everything went well, executing the `Check 3.` instruction will have the following effect:
  - the window of `emacs` will be divided in two frames.
  - the instruction `Check 3.` will be overlined in blue, this indicates that `Check 3.` has been checked. For (named) definitions and theorems, it means that the definition has been saved and can be used from now on.
  - in the bottom frame you will see the answer of Coq, that must be `3 : nat`. Coq has recognized `3` and informs you that it has the type `nat`.

For more fun, you can type `Check nat.`: yes, as we saw, the type `nat` is itself a term of the Calculus of Inductive Constructions ! You can as well check the type of `nat`, the type of the type of `nat`, etc, etc. There is no limit to fun[4] in the *calculus of inductive constructions with universes*.[5]

1.3.2 Forming propositions

A proposition can be composed of:

[4]in fact, there is a transfinite limit which is the ordinal ω

[5]Notice that the typing rule `Type: Type` seems quite incestuous. It is. Girard in 1972 and Miquel in 2000 have shown that with this rule we can encode paradoxes as for instance the one of the set that contains all the sets that do not belong to themselves, or the one of the madman who is painting a ceil. In the real world, Coq has a hiearchy of universes and hides it under the carpet. Formally, we have $\texttt{Type}_i$: $\texttt{Type}_{i+1}$ but the index i is not displayed on the screen, unless you use the command `Set Printing Universes.` or use the `View` menu of `CoqIde`.

- proposition symbols `A`, `B`, `C`, ... If we want to use them, we must defined them though the instruction `Variable name: type`. In our case, to define a *proposition* symbol `A`, we must write `Variable A: Prop` (the type of propositions is called `Prop`), we can declare many variables at the same time with `Variable A B C: Prop.`
- other propositions linked by logical connectives. Here is the way to write them in Coq:

$\wedge$	$\vee$	$\Rightarrow$	$\neg$
`/\`	`\/`	`->`	`~`

The proposition $A \Rightarrow B \Rightarrow A$ will then be written as: `A -> (B -> A)` or, more simply `A -> B -> A`.

1.3.3 State a Theorem or a Lemma

This is doable with the keyword `Theorem` (resp. `Lemma`):

```
Theorem name_of_theorem : proposition_to_prove.
```

For instance: `Theorem identity : A -> A.` (Obivously - or not - `Variable A: Prop.` must have been defined previously). Lemmas are stated in the same way.

1.3.4 Prove a Theorem or a Lemma

Once the theorem is stated, we must prove it. After Coq has checked that the statement of the theorem is well-formed, you will see the following text appear:

```
1 subgoal

  ============================
   A -> A
```

This is the proposition you shall prove. The bar ======= represents the "turnstyle" $\vdash$ of the sequent. So you must prove, (in natural deduction) the sequent: $\vdash A \Rightarrow A$.

For this, as you surely know now, we only need to apply the $\Rightarrow$-intro rule and then the axiom rule. In Coq, $\Rightarrow$-intro is just `intro.` Try it.

Now you are left with the sequent $H : A \vdash A$ (Coq has automatically named `H` the hypothesis that you have pushed on the left of the sequent). In Coq notation:

```
1 subgoal

  H : A
  ============================
   A
```

Now you just give the instruction to use the hypothesis `H`, through the instruction `apply H.` or directly the instruction to apply the axiome rule through the instruction `assumption.` Once Coq says `proof completed` or `No more subgoals.`, you must use the instruction `Qed.` (latin abbreviation which french equivalent is CQFD). You can also use the instruction `Save.`, there is no difference here. The effect of both instructions is to ask Coq to check the proof, which implies in particular the production of the associated λ-term. If you are curious, you can see the λ-term with the instruction `Print name_of_theorem.` (proofgeneral) or the corresponding option of the `Queries` menu.

When you encounter several subtrees (for instance, after having applied the $\wedge$-intro rule, two premises must be proved), Coq asks you to prove them one after the other. Once a subtree is proved, it switches automatically to the next subtree.

It is possible to choose the subtree you want to prove first through the instruction `Focus.` but be aware that you will anyway need to eventually prove all subtrees.

1.3.5 Generalizing a Bit

As a more advanced feature, we can use the *higher-order* nature of Coq, which means that we can universally (and existentially) quantify over virtually anything. So, if we want to give more generality to our theorem, we can say:

```
Theorem gen_id : forall C: Prop, C -> C.
```

We now have a generic theorem that shows that for any *proposition* `C`, the proposition `C -> C` is provable. The proof requires now *two* `intro.` rules[6], plus the axiom rule. In consequence, there is no more need to introduce the variable `A` before stating the theorem. This was not possible in first-order logic. Do not forget to save the proof of the theorem with `Save.` or `Qed.`

The genericity of this theorem allows to apply it to any proposition. For instance, if we reconsider our `Goal A -> A.` (of theorem `identity` above), we can now prove it by simply applying theorem `gen_id` *via* the instruction `apply gen_id.`

2 Proofs in propositional logic

2.1 The rules

Here is the table that gives the match between the rules of natural deduction and Coq's instructions:

	$\wedge$	$\vee$	$\Rightarrow$	$\neg$
Introduction:	`split`	`left / right`	`intro`	`intro`
Elimination:	`elim`	`elim`	`apply`	`elim`

[6]the first introduces the proposition symbol `C0`, and it corresponds to the $\forall$-intro rule, but in a logic much stronger than first-order logic: for this we need at least second order logic.

The instructions to appy the elimination rules are used together with the name of the hypotesis (so `elim H.` or `apply H.` for instance). We must explicit the proposition, for which we want to eliminate the main connective. Notice also that some rules do not correspond directly to a rule in natural deduction, but they are equivalent to them.

For instance, when proving to prove the sequent $A \wedge B \vdash C$, and when we can apply the rule `elim` (that is to say $\wedge$-elim) on the conjunction $\wedge$, we get the following proof:

$$\dfrac{\wedge\text{-elim2}\ \dfrac{\dfrac{}{A \wedge B \vdash A \wedge B}}{A \wedge B \vdash B} \qquad \dfrac{\wedge\text{-elim1}\ \dfrac{\dfrac{}{A \wedge B \vdash A \wedge B}}{A \wedge B \vdash A} \qquad \dfrac{\vdots}{A \wedge B \vdash A \Rightarrow (B \Rightarrow C)}}{A \wedge B \vdash B \Rightarrow C}\ \Rightarrow\text{-elim}}{A \wedge B \vdash C}\ \Rightarrow\text{-elim}$$

This way, we go from a proof of $A \wedge B \vdash C$ to a proof of $A \wedge B \vdash A \Rightarrow (B \Rightarrow C)$. After making the two necessary `intro` we are left to prove the sequent $A \wedge B, A, B \vdash C$. We have a proof with two cuts, admittedly,[7] but much more easy to manipulate.

When we use the instruction `elim` on a disjonction, the constructed proof-tree is the following:

$$\dfrac{\dfrac{\vdots}{A \vee B \vdash A \Rightarrow C} \qquad \dfrac{\vdots}{A \vee B \vdash B \Rightarrow C} \qquad \dfrac{}{A \vee B \vdash A \vee B}}{A \vee B \vdash C}$$

After making `intro` in the two subtrees to be proved, we obtain the "usual" rule $\vee$-elim:

$$\dfrac{\dfrac{\dfrac{\vdots}{A \vee B, A \vdash C}}{A \vee B \vdash A \Rightarrow C} \qquad \dfrac{\dfrac{\vdots}{A \vee B, B \vdash C}}{A \vee B \vdash B \Rightarrow C} \qquad \dfrac{}{A \vee B \vdash A \vee B}}{A \vee B \vdash C}$$

The instruction `intros` makes as many `intro` rules as possible. It is also possible to introduce external formulæ through a *cut*[8]. In theory, you can always avoid this, but maybe it will be handy at some point, so here is the command:

```
assert F.
```

where `F` is any formula, makes two things. It puts `F` in the context, so you have more hypothesis, and it creates another branch in the proof tree, where you must prove `F` (No miracle and no free lunch !).

[7]of course they can be eliminated. By the way, after eliminating those two cuts, we are left with a proof where *both* $\wedge$-elim rules are at the very top.

[8]technically it is a $\Rightarrow$-elim / $\Rightarrow$-intro cut

2.2 Propositions to Prove

Prove the following propositions:

1. $A \Rightarrow B \Rightarrow A$
2. $(A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C$
3. $(A \Rightarrow B) \Rightarrow (C \Rightarrow A) \Rightarrow C \Rightarrow B$
4. $A \Rightarrow (A \vee B)$
5. $A \Rightarrow B \Rightarrow (A \wedge B)$
6. $(A \Rightarrow B) \Rightarrow (C \Rightarrow B) \Rightarrow ((A \vee C) \Rightarrow B)$
7. $\neg(A \vee B) \Rightarrow (\neg A \wedge \neg B)$
8. $(A \vee (B \vee C)) \Rightarrow ((A \vee B) \vee C)$
9. $(A \Rightarrow B) \Rightarrow (A \wedge C) \Rightarrow (B \wedge C)$
10. $\neg\neg\neg A \Rightarrow \neg A$
11. $(A \wedge (B \vee C)) \Rightarrow ((A \wedge B) \vee (A \wedge C))$
12. $(A \vee (B \wedge C)) \Rightarrow ((A \vee B) \wedge (A \vee C))$
13. $(\neg A \wedge \neg B) \Rightarrow (A \vee B) \Rightarrow \bot$

Hints: $\bot$ is called `False` in Coq. Sometimes, it is better to come with a proof-tree written on a paper sheet and to try to make it understand to Coq. But sometimes, you also can let Coq guide you.

3 Proofs in Predicate Logic

3.1 The Rules

Here are the quantifier rules:

	$\forall$	$\exists$
Introduction	`intro`	`exists v`
Elimination	`apply`	`elim`

Usually, when using the $\forall$-elim rule, one must give the term that will instantiate the quantified variable. Coq cannot guess those variables automatically in the general case. For this, we must explicitely give the term v (just as in the $\exists$-intro case), like this:

```
apply (H v).
```

Notice that the parentheses do matter. In some case, we can want to chain the $\forall$-elimination rules. Then it is enough to say `elim (H v)`, or `elim (H v w)`, if we need to instantiate two universally quantified variables.

Also, take care that all the variables and constants you introduce in the term `v` with `exists v` or `apply (H v)` **must be already defined**, either in the current context, at the left of the turnstyle, or externally. Coq forces you to use *only existing* things, which is the only proper way to be formal. Since we quantify over natural numbers, there is obvious candidates for terms of this type.

Now, you must define your *predicates*. That is to say, if you want to prove a formula of the type $\forall x P(x) \Rightarrow P(0)$, you must explicitely say that `P` is a predicate having one input parameter, for instance a natural number. the type of `P` will then be `nat -> Prop`, or `Set -> Prop` or `Type -> Prop`.[9] This choice does not really matter at our level. The statement of the theorem is therefore:

```
Theorem abc: forall P: nat -> Prop, (forall x: nat, (P x)) -> (P 0).
```

When we need to define a predicate that has two variables, it will then have the type `nat -> nat -> Prop`, or the slightly different type `nat*nat -> Prop` (where * denotes the cartesian product).

The proof, on the other side, functions as previously. Notice that Coq handles automatically variable renaming in case there is a conflict for freshness, as it can be the case in the $\forall$-intro and $\exists$-elim rules.

3.2 The formulas to prove

Prove the following formulas:

1. $[\forall x \forall y R(x,y)] \Rightarrow \forall x \forall y R(y,x)$
2. $[\forall x (P(x) \wedge Q(x))] \Rightarrow [(\forall x P(x)) \wedge (\forall x Q(x))]$
3. $(\exists y \forall x P(x,y)) \Rightarrow (\forall x \exists y P(x,y))$
4. $((\forall x P(x)) \vee Q) \Rightarrow \forall x (P(x) \vee Q)$ (beware! P is a unary predicate (a predicate with one variable) and Q, a 0-ary predicate, *i.e.* without variable).
5. $[\forall x (P(x) \wedge Q)] \Rightarrow ((\forall x P(x)) \wedge Q)$ (with the same notations as above).
6. $\neg(\exists x P(x)) \Rightarrow \forall x(\neg P(x))$
7. $(\forall x \neg P(x)) \Rightarrow \neg \exists x (P(x))$
8. $[\exists x(\neg P(x))] \Rightarrow \neg(\forall x P(x))$
9. the sequent number 9 of the Homework assignment (Section 3.2).

[9] or even `Prop -> Prop`. There is no limit to fun in the Calculus of Inductive Constructions

4 Excluded-middle – harder

Coq is a constructive framework, so it cannot prove $A \vee \neg A$ for all formulas A. This is why, if we want to use that controversial principle, we must add it as an axiom:

```
Axiom excluded_middle : forall A:Prop, A \/ ~A.
```

A (nonlogical) axiom is used just like a lemma of a theorem, but we do not need to give a proof for it.[10] Assume that we want to show:

```
Theorem excluded_imp: forall B:Prop, (B -> B) \/ ~ (B -> B).
```

It is sufficient to call `intros. apply excluded_middle.`, and Coq will directly instantiate the propositions A of the excluded-middle axiom with $B \Rightarrow B$. We could have been more precise in giving the explicit instance of the excluded-middle we want to apply: `apply (excluded_middle (B0 -> B0)).` in order to indicate that we want to instantiate the excluded-middle law to $B_0 \Rightarrow B_0$. Of course, the theorem `excluded_imp` can be proved without the help of the excluded-middle, this is left as an additional exercise.

Generally speaking, the excluded-middle law allows to "temporarily store" formulas in the hypothesis of the sequent, while we are focusing on other formulas on the right of the sequent. For instance, if we want to show the sequent $\Gamma \vdash B$, we can begin by doing this:

$$\dfrac{\text{Ax.}\ \dfrac{}{\Gamma, B \vdash B} \qquad \dfrac{\vdots}{\Gamma, \neg B \vdash B} \qquad \dfrac{}{\Gamma \vdash B \vee \neg B}\ \text{excluded-middle}}{\Gamma \vdash B}\ \vee\text{-elim}$$

And now we are left to prove the sequent $\Gamma, \neg B \vdash B$: we are back to a proof of B, *but* we have *more* information in the hypothesis, and this can be helpful. A pattern that we often meet in those kind of proofs is the following:

$$\dfrac{\text{Ax.}\ \dfrac{}{\Gamma, B \vdash B} \qquad \dfrac{\dfrac{\dfrac{\text{Ax.}\ \dfrac{}{\Delta, \neg B \vdash \neg B} \quad \dfrac{\vdots}{\Delta, \neg B \vdash B}}{\Delta, \neg B \vdash C}\ \neg\text{-elim}}{\vdots}}{\Gamma, \neg B \vdash B} \qquad \dfrac{}{\Gamma \vdash B \vee \neg B}\ \text{excluded-middle}}{\Gamma \vdash B}\ \vee\text{-elim}$$

Now we must prove the sequent $\Delta, \neg B \vdash B$, while initially we had to prove the sequent $\Gamma \vdash B$. Since Γ is included into Δ, we have more information in Δ and we have more weapons to mak this proof.

[10]this is the very essence of axioms.

Prove with the help of the axiom the following formulas:

1. $(\neg B \Rightarrow \neg A) \Rightarrow (A \Rightarrow B)$
2. $\neg\neg A \Rightarrow A$
3. $(A \Rightarrow B) \Rightarrow (\neg A \vee B)$
4. $\neg(\forall x P(x)) \Rightarrow \exists x(\neg P(x))$
5. $\exists y \forall x(P(y) \Rightarrow P(x))$ (the "drinker's principle").

Conversely, we now consider *Peirce's law*:

```
Axiom peirce_law : forall A B:Prop, ((A -> B) -> A) -> A.
```

Show the excluded-middle law starting from this law (so, Peirce's law should be an axiom and the excluded-middle law should be a theorem). Do the converse: prove this law starting from the excluded-middle axiom.

5 Program writing in Coq

We now come to a more interesting part of the lab session, discussing the programming language nature of Coq, and how we can reason on them. We first need to define datatypes. Generally, we use inductive datatypes. For instance, to define lists:

```
Inductive my_list: Set :=
  | nil: my_list
  | cons: nat -> my_list -> my_list.
```

This corresponds to the definition of the datatype "`my_list`" in a *functional programming language* with strong types, in ML style, similar to what OCaml can look like.

1. define the list `[1;2;3]`, that you can call `trois`.

We can now define a function that returns the head of a list, by *pattern matching*:

```
Definition head (l:my_list) : nat :=
  match l with nil => 0
    | cons t q => t
  end.
```

2. define on the same model a function `tail` that returns the *tail* of a list.

We can now prove assertions on our code, for instance:

```
Theorem head_of_trois_is_1 : (head trois) = 1.
```

3. Prove this theorem. For this (almost) trivial proof, it can be useful to `unfold` the definitions for `head` and `trois`, the rest will be done automatically. For this, you can use Coq's following basic tactics:
   - `auto` solves automatically certain easy goals.
   - `unfold` expands the definition given as argument.
   - Refer to Coq's standard library for more tactics.

Here is the way to define the length function on a list:

```
Fixpoint length (l:my_list) : nat :=
  match l with nil => 0
    | cons m l1 => S (length l1)
  end.
```

This is a *recursive* definition, hence the keyword `Fixpoint`. Notice that Coq checks thoroughly that your recursive call obeys to the structurally decreasing argument pattern: the recursive function call is done on a smaller argument (`l1`) than the initial argument (`l` which is equal, by pattern matching, to `cons m l1`: it strictly contains `l1` and therefore is structurally bigger). Coq displays a message in this sense. This point can be hard to cope with in some cases, contrary to what happens in "usual" programming languages. But, by doing this, Coq ensures that the function length is terminating, and in particular, has no infinite recursive calls.

4. Prove the following theorems:

   ```
   Theorem length_1: length trois = 3.
   Theorem length_2: length (tail trois) = 2.
   ```

5. more difficult: prove that if a list is not empty, then its length is equal to the length of its tail plus one. At this point of this lab session, you may find useful to:
   - make *case distinctions*
   - use predefined theorems of Coq's standard library: you must include by hand the files, with the command `Require Import Arith.` (in case we need theorems that are located in the library `Arith`).
   - use higher-level tactics (such as `auto`, `eauto`, or even `omega` and `ring`) in order to avoid showing "trivial" results such as 1 is different of —0—, or to replace manually `2+2` by `4` (which is a pain).
   - refer to Coq's documentation (see the links below).

6. if you still have time left, you can define usual functions on lists such as: reversing a list (hint: define a more general function, with two arguments, first), concatenating two lists, etc, etc, and prove properties on those functions. For instance: the length of the reverse of a list is equal to the length of the original list (difficult); the reverse of the reverse of a list is the same as the original list (difficult), then length of the concatenation of two lists is the sum of the lengths of the two lists (easy), etc, etc.

6 Extracting programs from proofs

Lastly, we explore how, from a Coq proof of a theorem, we can *extract* the corresponding program (technically, one says that the program is a *universal realizer* of the theorem. Unfortunately, we lack of time to explore this fundamental aspect of proof assistants. This feature exploits heavily the Curry-Howard correspondence. Indeed:

- consider a proof of some proposition of the type "there exists an x such that BLAH". This proof *computes* the witness `t` and proves then BLAH.
- from a purely programming point of view, the proof contains an *algorithm* that builds `t`: all those informations are contained into the proof.
- therefore, we must be able to extract a program that corresponds to this proof.

In fact, the proof-term associated to the ∃-intro rule is a pair:

$$\exists\text{-intro}\ \frac{\Gamma \vdash \pi : \{t/x\}A}{\Gamma \vdash \langle t, \pi \rangle : \exists x A}$$

A proof of an existential property *contains*, in substance, a witness t. However, this witness can be hidden: a proof of the sequent $\Gamma \vdash \exists x A$ can begin with something else than an ∃-intro rule. In particular, there may be cuts, and the cut-elimination procedure (otherwise said, the computation done by the λ-term when it β-reduces) is the process that computes effectively this witness.

Of course, if in Coq it is essential to have a *proof* that the computed witness `t` respects property BLAH, in a normal programming language, we do not care (we blindly trust the programmer, or Coq in our case) and we can drop the proof, to leave only the effective information on the computation of `t`.

6.1 Examples

A trivial example is the following theorem:

```
Lemma ziro: {m: nat | m=0}.
exists 0.
auto.
```

```
Qed.
Recursive Extraction ziro.
```

Here, the construction `{m: nat | m=0}` is used instead of the more usual statement `exists m: nat, m=0`. We *must* do it that way, otherwise we will not be able to extract the program. The theoretical reason for that is that the type `Prop` is impredicative and therefore does not have strong eliminations[11] and this impose us to work not in `Prop` but in `Set` (or in `Type`) to perform the extraction. In those types, the way to write the existential quantifier is this very one, with curly brackets.

To sum up and go back to the example, we extract a constant function with zero argument, which value is 0, which exactly corresponds to the proof.

A more interesting example is this one:

```
Theorem minus_2: forall n: nat, n > 3 -> { m: nat | m < n /\ m > 0}.
```

You can try to do a proof of this theorem, and then extract a program. You will need at least Coq library `Lt`, and more probably the whole library `Arith`, as well as a case analysis on `n` (through the instruction `case` or through an application of the induction principle).

Further problems:

1. for the moment, the extraction is towards a *new copy* of natural numbers in OCaml (the base type is copied). We want a function that computes on OCaml's `int` type instead. Can you modify the extraction to do that?

2. find more interesting theorems, for which we can extract an algorithm. For instance, euclidian division (quite difficult, induction needed, do not cheat), or any other theorem with an interesting computational content.

3. lastly, you can play coqoban

6.2 Proofs and Specifications

The possibility to extract a program from a proof of a certain formula has its roots in the very formula: it describes *exactly* what the program is supposed to do and this specification is sufficiently detailed for us to make the proof and then, turn the proof into a program. We then know, since the proof has been checked (and may be mechanically checked by other proof assistants), that the program fits exactly to its specifications. In the previous example, we produce a program that takes as input a natural number (different from zero) and that outputs another natural number that is smaller than the input. For instance, it can be the input minus one, but it could also be the quotient of the euclidian division of the input by `2`, `3` or any number greater than `2`, or just the number zero. Those different programs correspond to different proofs that can be given to this theorem: there is many valid proofs, some complicated, some other not, and they give rise to different algorithms. The extracted program will *not* check that its argumnt is greater than 3, or that the result is greater, than 0, which means that a user can use that program in the *wrong* way, outside the specification zone.

The inverse operation is *much more* used: the goal is, given an existing program, to specify it sufficiently precisely (in particular on the assumption on the inputs of the program), to be able to transform it into a *proof* of its own specification. We will discuss that later in this lecture.

7 Further documentation

An extensive book on Coq: the Coq'Art book. The exercises of the book are available online.

In particular, for the pure "logical" proofs, you can have a look at the following exercises

The complete documentation is available on Coq's website. You can fruitfully exploit Coq's FAQ and tutorial.

[11] The type `Prop` only has the second projection of the pairs $\langle t, A\rangle$ associated to the $\exists$-intro rule, because it is an impredicative type ($\forall A.A$ is a proposition quantifying over all the propositions), and allowing to have the first projection of the pair would make the Calculus of Inductive Constructions inconsistent: we could encode all Girard's paradoxes.