Introduction to Coq (Lecture notes)

- **Formal methods**: mathematically rigorous techniques for specification, development, analysis and verification of programs. This includes for example, type systems, formal verification, program modeling, static analysis, etc.
- **Formal verification**: Proving the correctness (w.r.t. a specification, requirement) in (a) machine-checked way(s). Tools that can be used for this purpose: automated theorem provers (e.g., SMT-solvers, SAT-solvers), proof checkers, interactive theorem provers (e.g, Coq, Isabelle, Lean, Agda, ACL2, Metamath).

Formal verification is used for two purposes:

1. Do programs conform to their specification?
2. Are proofs about the specifications correct?

Coq

- Developed in 1984, by Coquand and Huet
- It is based on the Calculus of Inductive Constructions
- Maintained by Inria, in France
- Implemented in OCaml
- It is a dependently typed functional programming language

To implement machine-checked proofs with Coq about program properties, we need the following:

- Program + execution model
  - A functional program in Coq, or
  - Any program of a given language, and the semantics of the language to execute the program.
- Specifications, for example
  - A property (e.g., array indices are in-bound)
  - Pre- and postconditions
  - Semantics preservation (e.g., in case of compiler correctness, optimisations, refactorings)
- Theorems and proof scripts

After implementing the abovementioned items, Coq generates a proof term, which is easily (type-)checkable, resulting in a *machine-checked* proof.

Basic concepts

- `Set` : A basic type of types.
- `S : nat -> nat` : A function that takes a natural number, and returns one.
- `Inductive` : Definition of a type. The values of this type **only** can be constructed by its constructors.
- `match ... with | ctor_1 => ... | ... | ctor_n => ... end` : Pattern matching expession, which inspects which constructor was used to construct a value.
- `Compute` : This command reduces a term to its normal form. Alternatively, `Eval compute in <term>` (`compute` can be changed to other evaluation strategies, such as `cbn`, `lazy`, and others).
- `nil : forall A : Type, list A` : This is a dependent function type.

- `Set` vs `Type`: `Type` is actually an indexed universe of types, i.e., `Type 0 : Type 1`, `Type 1 : Type 2`, etc. `Set` $\approx$ `Type 0`, and `Set` is used to express data types.
- `Module` : Used for scoping and separating definitions, theorems, proofs.
- `{A : Type}` : `A` is an implicit parameter, Coq tries to infer it automatically from the context. E.g., `cons : forall {A : Set}, A -> list A -> list A`. The character `@` can be used to make implicit parameters explicit, such as `@cons nat 1 nil`.
- In Coq, one can define notations to simplify displaying expressions. E.g., `Notation "a + b" := (add a b)`
- Anonymous functions can be defined with the `fun` keyword, e.g., `fun x : nat => S x`
- `Qed` : If a proof is finished, we close it with this statement to save the proof.
- `Admitted` : If a theorem is closed with this statement, it becomes an axiom (i.e., a proposition that is assumed to be true; however, it is not proved).
- `Definition`, `Theorem`, `Lemma`, `Example`, `Proposition`, `Corollary` : These statements all introduce a new definition or proof, their meaning is the same.

Dependent types: vectors

```
Inductive Vec (A : Type) : nat -> Type :=
| vnil : Vec A 0
| vcons {n : nat} (a : A) (v : Vec A n) : Vec A (S n).
```

Here, the `nat` parameter of the type cannot be given on the left hand side of `:`, because it is different in the constructors (`0` in `vnil`, and `S n` in `vcons`). Dependent functions can also be defined:

```
Fixpoint vappend {A : Type} {n1 n2 : nat}
     (ls1 : Vec A n1) (ls2 : Vec A n2) : Vec A (n1 + n2) :=
match ls1 with
| vnil => ls2
| vcons a ls1' => vcons a (vappend ls1' ls2)
end.
```

In this case, we need to make sure that the types align with the operations. In this case, the addition `(n1 + n2)` aligns with the constructors, because:

- in case of `vnil`, `n1 = 0`, thus `ls2 : Vec A (0 + n2)` is correctly typed.
- in case of `vcons`, `n1 = S n1'`, and `vappend ls1' ls2 : Vec A (n1' + n2)`, therefore `vcons a (vappend ls1' ls2) : Vec A (S (n1' + n2)) = Vec A (S n1' + n2)` is correctly typed.

Q: can we define a `filter` function for vectors? A: Only if we know how many elements satisfied the given property in the vector.

Dependent types: equality

Equality (=) is just a simple dependent type in Coq.

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
| eq_refl : eq x x.
```

Terms of this type can be seen as proofs of the terms being indentical. We can construct instances of such dependent types (i.e., proof terms) by writing dependent functions; however, this is usually not too easy. To make the construction of proof terms easier, Coq provides a proof mode and a tactic language (Ltac).

Q: Is there a way to do ``hole programming'' in Coq? A: Yes, with the module `Program`.

Induction

After defining an inductive type, Coq automatically generates an induction/recursion principle for it. To apply the principle, the given goal has to be matched against it. To prove a property by induction we need to prove it for the base cases, and supposing that the property holds for recursive subterms, we need to prove it for the given term. For example, if we want to prove `P` for any list `l`, then:

1. We prove that `P` holds for the empty list (`P nil`).
2. Supposing that `P` holds for some list `l`, we have to prove that for any value `v`, `P` holds for `v :: l` (`forall l : list A, P l -> forall v : A, P (v :: l)`).

With induction, we can also define relations (e.g., less-or-equal-than, `le`). Since these relations are defined inductively, we can use their induction principle to prove properties about them.

When doing induction, it is crucial to get the induction hypothesis right. If too many variables are introduced to the list of hypotheses, the induction hypotheses will be stated for fixed variables instead of universally quantified ones.

Equalities

There are multiple concepts of equality in Coq:

- Two terms are convertible (i.e., they compute to the same normal form). Typically denoted with ≡ (triple bar).
- Two terms are equal, i.e., there is a proof of them being in `eq`. Typically denoted with = (two bars).
- Boolean equality, which is defined for specific types, e.g., `Nat.eqb : nat -> nat -> bool`, `Bool.eqb : bool -> bool -> bool`, etc.

Evaluation happens during type checking because of the dependency between types and terms, for example to check that a term has type `3 + 4 = 7`, Coq needs to reduce `3 + 4` to `7` first (`3 + 4` is convertible to `7`).

Maps

Functions can be used to express total maps. Maps can be used to represent a simple memory state while executing a simple imperative program: it contains which variable is bound to which value. To reason about equality of maps (functions, in general) functional extensionality might be needed: `(forall x, f x = g x) -> f = g`. Function extensionality is an axiom in Coq but usually considered safe to use.

One can represent total maps in Coq using just pure functions. As opposed to other data structures, like a list of key-value pairs, one advantage to this approach is that we can use functional extensionality to check whether two maps are equal, greatly simplifying proofs. To begin, we define a function type mapping strings to some generic type `A`:

```
Definition total_map (A : Type) := string -> A.
```

To construct a total map, we define a function that returns an empty map:

```
Definition t_empty {A : Type} (v : A) : total_map A :=
  (fun _ => v).
```

Note that, since the map is meant to be *total*, keys that have not been added yet will be mapped to a default value, here represented by `v`. To populate the map with key-value pairs we make use of an update function, which takes a map `m`, string `x`, and value `v`, and yields a new map that satisfies `m x = v`

```
Definition t_update {A : Type} (m : total_map A)
                    (x : string) (v : A) :=
  fun x' => if String.eqb x x' then v else m x'.
```

A simple imperative language: Imp

We introduce a simple imperative language to study proving program properties. It includes arithmetic and boolean expressions, and simple commands:

```
a ::= x | n | a1 + a2 | a1 - a2 | a1 * a2
b ::= true | false | a1 = a2 | a1 /= a2 | a1 <= a2 | a1 > a2 | not b | b1
and b2
c ::= skip | x := a | c1; c2 | if b then c1 else c2 end | while b do c end
```

The arithmetic and boolean expressions can be interpreted by a simple recursive function that essentially maps the syntactical operations to their semantical counterpart between natural numbers/booleans in Coq.

For the simple expression language, we can already prove some properties, for example the correctness of optimisations (e.g., expressions like `0 + a` are replaced by just `a`). During the proofs, we use induction, and as many case separations as many pattern matching was in the optimization function. Sometimes, a lot of subgoals are solved by the same tactics, these can be automated.

To make it simpler to write programs with the syntax above formalised in Coq, we can define `Notation`s and `Coercion`s (automatically converting elements of one type to another).

We can define another semantics for arithmetic/boolean expressions, in a big-step, relational style by defining derivation rules (which will be the constructors of an inductive type). For example:

```
 a1 ==> n1           a2 ==> n2
---------------------------------
      a1 + a2 ==> n1 + n2
```

Note that, in this rule the `+` on the right-hand side is `Nat.add`, the addition between natural numbers in the meta language (Coq).

Why is it advantageous to have different semantics?

- A relational model can more easily express recursion (since `Fixpoint`s need to always terminate in Coq, this is harder with functions).
- A relational model comes with an induction principle, thus induction on the structure of the relation can be used to prove properties.
- A functional model can effectively evaluate concrete programs/expressions.

If there are multiple semantics for the same language, it is important to assure that they are equivalent:

- `aeval a = n -> a ==> n`: this direction is proved by induction on the structure of `a`. The derivation `a ==> n` can be proved by applying the derivation rules (constructors).
- `a ==> n -> aeval a = n`: this direction is proved by induction on `a ==> n`.

For the commands, we only define a big-step semantics, and we highlight two rules (for the rest, we refer to the code). From now, `st` denotes a total map (a state that assigns natural numbers to variables).

If the loop condition evaluates to false, the loop terminates, and does not modify the state:

```
     beval b st = false
---------------------------- E_WhileFalse
st -[while b do c end]-> st
```

If the loop condition evaluates to true, the body of the loop needs to be executed, followed by executing the loop again. Note that, in this rule, the semantics of the loop is used also in the premise. For this reason, when we prove properties structural induction on the command is not usable, since there will not be an inductio hypothesis for the loop evaluation for the aforementioned premise.

```
beval b st = true     st -[c]-> st'     st' -[while b do c end]-> st''
--------------------------------------------------------------------------
E_WhileTrue
                 st -[while b do c end]-> st''
```

Evaluation. To evaluate commands, we apply the constructors. In case of loops (see second rule above), and sequence, the intermediate state (`st'`) cannot be inferred automatically, because it does not occur in the conclusion of the rule. We can define it by hand with `apply E_WhileTrue with (st' := <definition of st'>)` or with partial function application `apply (E_WhileTrue _ _ _`

`<definition of st'>`. If there are multiple constructors that match the goal, we have to choose the suitable one.

Determinism of the semantics. We prove the determinism of the semantics of commands:

```
forall st st' st'' c,
st -[c]-> st' ->
st -[c]-> st'' ->
st' = st''
```

We prove this theorem by induction on `st -[c]-> st'`. In all subgoals, we check how the second derivation `st -[c]-> st''` has been carried out by `inversion`. In case of conditionals, and loops, in two cases, we get to a contradiction, if different constructors were used in the first and second derivation (which leads to the contradicting `beval b st = true` and `beval b st = false` hypotheses). In the other cases, the induction hypotheses can be used on the second derivations. When evaluating a sequence or a loop, then we have to make sure, that the induction hypotheses are applied in the order of the evaluation.

Hands-on sessions

Covered tactics:

- `intros`: introduces the universally quantified variables, and the left-hand sides of implications to the list of hypotheses.
- `simpl (in ...)`: simplifies the terms in the goal/hypotheses by computation.
- `reflexivity`: This tactic is equivalent to `apply eq_refl`, and solves goals shaped like `x = x`.
- `exact`: This tactic solves the goal, if it matches exactly the given hypothesis or type.
- `apply`: This tactic matches *the conclusion* of the given theorem against the goal, and replaces it with the premises of the theorem.
- `apply ... in`: This tactic matches *one of the premises* against the given hypothesis, and replaces it with the theorem's conclusion (and generates subgoals for the other premises).
- `inversion`: This tactic checks which constructors were used to construct a term.
- `rewrite (<-)`: With this tactic, equal terms can be replaced in the goal or in the hypotheses.
- `symmetry`: This tactic swaps the sides of the equality in the goal/given hypothesis.
- `subst`: This tactic substitutes all primitive equalities from the hypotheses, and removes the unused free variables.
- `assert`: With this tactic, a new proposition can be proved, which will be introduced to the list of hypotheses.
- `induction`: With this tactic, we can apply the induction principle of an inductive type to prove a goal.
- `change`: Replaces the goal with an equivalent one.
- `generalize dependent`, `revert`: These tactics are the opposite of `intros`, they move hypotheses back to the goal (as implications of universally quantified variables).
- `split`: This tactic is used on a goal shaped like `G1 /\ G2` and generates two subgoals of `G1` and `G2`.
- `discriminate`: This tactic looks for an equality hypothesis that cannot be satisfied (e.g., `1 = 0`).
- `rename ... into ..`: Changes the name of a hypothesis in the proof context. Use this to give new names to assumptions or modify the names of variables.

- `assumption` : This tactic attempts to find a hypothesis in the proof context context that exactly matches the goal.
- `contradiction` : Similar to `discriminate`. This tactic tries to find a hypothesis in the proof context that is logically equivalent to `False`.

A single goal can be focused with `*`, `+`, `-`, `{ ... }`, etc.

Automation

- `try tac`: tries to execute the given tactic `tac`, and does not do anything if it fails.
- `repeat tac`: tries to execute given tactic `tac` until it fails or until it succeeds but fails to make further progress. Note that while Gallina is guaranteed to terminate, `repeat` may not. If Coq gets stuck while running `repeat`, use the interrupt command to stop the infinite loop.
- `tac1; tac2`: after executing `tac1`, executes `tac2` for all generated subgoals.
- `tac;[tac1 | tac2 | tac3 |... ]`: `tac` generates n subgoals, and executes `tac1` for the first, `tac2` for the second, `tac3` for the third, etc.
- `lia`: linear integer arithmetics, automatically solves a provable goal about the equality of nats/integers.

Custom tactics can be defined with `Ltac`, for more details, we refer to the Ltac documentation.

Further reading

- Slides and Coq codes
- Software Foundations
- Certified programming with dependent types
- Coq'Art
- Coq reference manual