These notes cover Coq basics and induction (Software Foundations chapters 3-4).

Topics that were briefly mentioned but not explored during class are highlighted in yellow. Subjectively "interesting" topics are also highlighted in yellow.

1 Coq overview

Coq is a system for writing programs in a formal specification and verifying that programs are correct with respect to their specification. The heart of Coq is a type-checking algorithm which verifies that programs compile to their specification.

Coq actually contains three languages:

- The Gallina language is the "specification language" of Coq. It is an expressive functional language formalizing both programs and proofs. Types, functions, and proofs are all defined in Gallina.
- The vernacular language contains commands for the Coq proof assistant.
  Its commands include: `Theorem`, `Proof`, `Qed`, etc.
- The tactics language is used to build proofs.
  Its commands include: `reflexivity`, `intros`, etc.

The Coq introduction explains the purpose of Coq and its components:
`https://coq.inria.fr/refman`

2 Coq language

2.1 Defining types (`Inductive`)

Types may be thought of as sets of elements.

Coq has inductive datatypes. Inductive datatypes may contain basic data values, as well as constructors taking other data values. Examples:

- Trivial enumerated type.

  ```
  Inductive bool : Type :=
    | true : bool
    | false : bool.
  ```

  The `bool` type contains two elements: `true` and `false`. It is called an "enumerated type" because it simply enumerates a finite set of elements.

- Simple recursive type.

  ```
  Inductive nat : Type :=
    | O : nat
    | S : nat -> nat.
  ```

  The `nat` type represents the natural numbers.

  `O` represents the value zero.

  `S` represents the successor of a natural number. It is a constructor, taking a `nat` and returning a `nat`.

> Intuitively, this means "natural numbers may be defined as either zero or the successor of another natural number".

Comparison with types in other languages

Coq's inductive datatypes are similar to algebraic datatypes in other languages (ML, Haskell, Swift, etc). Algebraic datatypes are a kind of composite types: types are composed to form new aggregate types.

So far, we've studied **sum types** (also known as tagged/disjoint unions). The values of a sum type are grouped into several "cases". Each case has its own constructor, which takes a specified number of arguments with specified types. (The name "sum type" comes from the fact that the number of elements in a sum type is equal to the sum of the number of elements of each of its component types.)

Enumerated types are a trivial form of sum types, where each case constructor takes no arguments. This is analogous to "enums" in C or Java:

```c
// C. Enumerated "boolean" type.
enum Bool {
    false = 0,
    true = 1
};
```

```java
// Java. Enumerated "weekday" type.
enum Weekday {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
};
```

Many languages support more complex sum types that have constructors with associated arguments.

```haskell
-- Haskell.
-- Recursive "natural number" type.
data Nat = Zero | Succ Nat
```

```swift
// Swift. Polymorphic "optional" enum type.
enum Optional<T> {
  case none
  case some(T)
}
```

Types with an uncountable set of elements

One question asked during class is: how is it possible to represent an uncountable set of elements as a type (e.g. real numbers)?

Coq `Reals` library defines real numbers as follows:

```
(* 'R' is an arbitrary type.
   There is no a priori information about the
   nature or number of instances of 'R'. *)
Parameter R : Set.

(* Define the constants 0 and 1. *)
Parameter R0 : R.
Parameter R1 : R.

(* Define functions on real numbers. *)
Parameter Rplus : R -> R -> R.
```

```
(* Define axioms on real number functions. *)
Axiom Rplus_comm : forall r1 r2:R, r1 + r2 = r2 + r1.

(* Define injection from ‘N‘ to ‘R‘. *)
Fixpoint INR (n:nat) : R :=
  match n with
  | O => R0
  | S O => R1
  | S n => INR n + R1
  end.
```

References:

- https://coq.inria.fr/distrib/current/stdlib/Coq.Reals.Rdefinitions.html
- https://coq.inria.fr/distrib/current/stdlib/Coq.Reals.Raxioms.html

2.2 Defining functions (`Definition` and `Fixpoint`)

Functions in Coq are pure, producing no side effects. Functions always return the same output given the same inputs, making it easier to reason about their behavior.

Non-recursive functions

Non-recursive functions are declared with the `Definition` keyword:

```
Definition negb (b:bool) : bool :=
  match b with
    | true => false
    | false => true
  end.
```

`match` statements are used for pattern matching, or destructuring inductive types into their constituent cases/patterns. This style of pattern matching is similar to other languages with algebraic datatypes.

In the function `negb`, `match` destructures the value `b` of type `bool` into two patterns (`true` and `false`). A clause is provided for each pattern.

Recursive functions

Recursive functions are declared with the `Fixpoint` keyword:

```
Fixpoint evenb (n:nat) : bool :=
  match n with
    | O        => true
    | S O      => false
    | S (S n') => evenb n'
  end.
```

In the pattern `S (S n')`, the argument of the inner `S` constructor is bound to the name `n'`. `evenb` is then called recursively with `n'`.

In Coq, it is not possible to write infinitely recursive functions. It is important that functions terminate for theorem proving. Thus, Coq's termination checker allows only "primitive recursion": recursive calls must take syntactic subterms of original arguments. This is one way to avoid the halting problem.

However, not all recursive functions can be easily expressed in a way that satisfies primitive recursion (e.g. mergesort). A workaround for this is explained in-depth at `http://adam.chlipala.net/cpdt/html/GeneralRec.html`.

To summarize the approach: one must choose a well-founded relation and prove that all recursive calls are performed on arguments that are "less" according to the well-founded relation.

Pattern matching details

The wildcard pattern "_" can be used to match any pattern/subpattern, without binding any names.

```
Definition is_two (n:nat) : bool :=
  match n with
    | S (S O) => true
    | _       => false (* Wildcard acts like a "default" clause here. *)
  end.

Fixpoint minus (n m:nat) : nat :=
  match n, m with
    | O, _  => O                (* 0 - m = 0 *)
    | _, O  => n                (* n - 0 = n *)
    | S n', S m' => minus n' m' (* (n+1) - (m+1) = n - m *)
end.
```

The Coq type checker ensures that patterns are exhaustively matched in `match` statements. Inexhaustive pattern matching results in an error:

```
Fixpoint evenb_fail (n:nat) : bool :=
  match n with
    | O   => true
    | S O => false
  end.
(* Error: Non exhaustive pattern-matching: no clause found for pattern S (S _) *)
```

2.3 Writing proofs (`Theorem`)

The "tactics" language in Coq is used to write proofs.

Proofs in Coq start with the declaration of a theorem. At each stage of a proof development, one has a list of goals to prove. Initially, the list consists only of the theorem itself. After some tactics are applied, the list of goals contains the subgoals generated by the tactics.

Example:

```
Theorem plus_id : forall n m : nat, (* Quantifiers. *)
  n = m -> n + n = m + m. (* Goal. *)
Proof.
  (* Move universal quantifiers ‘n‘ and ‘m‘ into context. *)
  intros n m.
  (* Move the hypothesis into context. ‘H : n = m‘ *)
  intros H.
  (* Rewrite goal using hypothesis. *)
  rewrite H.
  reflexivity. Qed.
```

Useful tactics

- `reflexivity`.

  Use to prove that something equals itself (the reflexive property $x = x$).

  `reflexivity` performs some simplification, so it can work as long as both sides of the equality evaluate to the same term.

- `simpl`.

  Simplify a term via syntactic rewriting.

  Unlike `reflexivity`, `simpl` never fails. If simplification is not possible, `simpl` is a no-op. In general, there is no need to call `simpl` before `reflexivity` at the end of a proof (as `reflexivity` also performs simplification).

- `intros`.

  Move quantifiers/hypotheses in the goal into the current context of assumptions.

  See the proof above for an example.

- `unfold`.

  Replace a function with its right-hand side (body). This is analogous to inlining.

  One use of `unfold` is to enable further syntactic simplification via `simpl` or `reflexivity` after unfolding.

- `rewrite`.

  If it is known that two terms are equal, use `rewrite` to transform one term into the other.

  In the proof above, `rewrite H` is used to transform all instances of the left-hand side of `H` (i.e. `n`) into the right-hand side (i.e. `m`).

  Use `rewrite <- H` to transform in the opposite direction (transform all instances of the RHS into the LHS).

  By itself, `rewrite` is not very smart about where it applies rewriting. For example, in the proof below, there are three instances of `+`, so simply using `rewrite Nat.add_comm` will transform the wrong terms. Instead, it's possible to apply `rewrite` to a particular subexpression:

  ```
  Require Import PeanoNat.
  Theorem plus_rearrange : forall n m p q : nat,
    (n + m) + (p + q) = (m + n) + (p + q).
  ```

```
Proof.
  intros n m p q.
  (* Rewrites ‘n + m‘ into ‘m + n‘ in the LHS. *)
  rewrite (Nat.add_comm n m).
  reflexivity.  Qed.
```

- `destruct`.

  Use to perform case analysis on a term. This is useful when the type of term is known, but its value is not. `destruct` generates subgoals for each constructor of the term.

```
Theorem negb_involutive : forall b : bool,
  negb (negb b) = b.
Proof.
  intros b.
  destruct b. (* Decompose ‘b‘ into its possible cases: ‘true‘ and ‘false‘. *)
  - reflexivity.
  - reflexivity. Qed.
```

- `induction`.

  Use to prove a theorem via induction.

  `induction` generates subgoals for each constructor of the term, just like `destruct`. However, for inductive constructors (e.g. `S n` for the `nat` type), `induction` also adds an inductive hypothesis to the context.

```
Theorem plus_n_O : forall n : nat, n = n + 0.
Proof.
  intros n. induction n as [| n' IHn'].
  - (* n = 0 *)
    (* Trivial subgoal: 0 = 0 + 0 *)
    reflexivity.
  - (* n = S n' *)
    (* IHn' : n' = n' + 0 *)
    (* Subgoal: S n' = S n' + 0 *)
    simpl. rewrite <- IHn'. reflexivity. Qed.
```

- `assert`.

  Create a local proof in the current context. The local proof can refer to terms in outer contexts, and can be used in the parent context after it is proved.

  This is analogous to creating a nested closure in other languages. Closures are capable of capturing variables from outer contexts.

```
Theorem mult_0_plus : forall n m : nat,
  (0 + n) * m = n * m.
Proof.
  intros n m.
  (* Creates a local proof called ‘H‘.
     The proof for ‘H‘ is defined within the curly braces. *)
```

```
assert (H: 0 + n = n). { reflexivity. }
rewrite -> H.
reflexivity. Qed.
```

Overviews of commonly used tactics:

- `https://pjreddie.com/coq-tactics`
- `http://www.cs.cornell.edu/courses/cs3110/2018sp/a5/coq-tactics-cheatsheet.html`

Comprehensive list of tactics: `https://coq.inria.fr/refman/coq-tacindex.html`.

3 Coq tips

Useful vernacular commands

- `Check`: show the type of a term.

```
Check nat.
(* nat : Set *)
Check Nat.add.
(* Nat.add : nat -> nat -> nat *)
```

- `Compute`: evaluate a given term.

```
Compute S (S O).
(* = 2 : nat *)
Compute 2 * 3 + 4.
(* = 10 : nat *)
```

- `Search`: displays the name and type of all objects (theorems, current hypotheses) in the current context related to the given term. This is useful for finding relevant theorems when writing a proof.

```
Search Nat.add.
(* plus_O_n: forall n : nat, 0 + n = n
   plus_n_O: forall n : nat, n = n + 0
   ... *)
```

Useful proof handling commands

- `Admitted`. This command gives up the current proof and declares the initial goal as an axiom. This can be used to "assert" that a theorem is correct so that it can be used in other proofs. The admitted theorem can be proven later.
- `Abort`. This command gives up the current proof and switches to the previous proof development or Coq toplevel.

Comprehensive list of vernacular commands:
`https://coq.inria.fr/refman/proof-engine/vernacular-commands.html`.

Coq tools

The following tools come bundled with distributions of Coq.

- `coqtop`: the Coq "toplevel", an interactive REPL.
- `coqc`: the Coq compiler. Compiles Coq `.v` source files and produces `.vo` files, which can be imported by other Coq source files.
- `coqdoc`: Coq documentation tool. Produces `html`/LaTeX files based on Coq source files.

4 Related reading

Curry-Howard correspondence

The Curry-Howard correspondence is a direct relationship between mathematical proofs and computer programs.

It states that the type of a function is analogous to a logical theorem, where the argument types of the function are hypotheses and the return type is the conclusion. Thus, a program to compute that function is the same as a proof of that theorem. Types are analogous to propositions.

This bridge between logic and programming is the foundation for interactive theorem provers like Coq. The mechanisms for "programming" and "proving" are closely related.

References:

- `https://en.wikipedia.org/wiki/Curry-Howard_correspondence`
- `https://www.seas.upenn.edu/~cis500/cis500-f14/sf/ProofObjects.html`

Calculus of Constructions

The Calculus of Constructions (CoC) is a type theory created by Thierry Coquand. It is a higher-order typed lambda calculus, supporting polymorphism, type constructors, and dependent types. The CoC serves as a basis for Coq.

References:

- `https://en.wikipedia.org/wiki/Lambda_calculus`
- `https://en.wikipedia.org/wiki/Calculus_of_constructions`
- `https://en.wikipedia.org/wiki/Lambda_cube`

5 Trivia

- "Coq" is named after the Calculus of Constructions (CoC) upon which it is based, as well as the French word for "rooster". Similarly, "gallina" means "hen" in Italian and Spanish.