NOTES ON COQ

VAIBHAV KARVE

These notes were last updated August 27, 2018. These notes started with my reading of *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Indictive Constructions* by *Yves Bertot* and *Pierre Castéran*. The associated site for the book (but not these notes) is www.labri.fr/perso/casteran/CoqArt/.

However, I soon moved on to a different resource – a book series called *Software Foundations*. Specifically, I am reading Volume 1: *Logical Foundations* written by Benjamin C. Pierce et. al.

1. A Brief Overview / Preface

(1) *Automated theorem provers* are tools to which you supply a proposition and they return either true or false. Examples of such tools are SAT solvers and model checkers.

(2) *Proof assistants* are hybrid tools that automate the routine aspects of building proofs while also depending on human guidance. Examples of proof assistants are Isabella, Coq and others.

(3) *Coq* is a proof assistant with which one can express specifications and develop programs that fulfill these specifications. It can develop proofs in higher-order logic. Coq has been in development since 1983, mostly at INRIA (which is a French national research lab).

(4) The name "Coq" refers to a rooster and is also a reference to the Calculus of Constructions (CoC) on which it is based. Also, the rooster is the national symbol of France. Also also, the name is a reference to Thierry Coquand, one of Coq's early developers.

(5) *Functional programming* refers to both a collection of programming idioms that can be used in any programming language and to a family of programming languages designed to emphasize these idioms including Haskell, OCaml, F#, Scala, Racket and Coq.

(6) The most basic tenet of functional programming is that computation should be pure, i.e. the only effect of execution should be to produce a result free from *side effects*. Side effects include I/O, assignments to mutable variables, redirecting pointers etc. Functional programming preserves concurrency and can therefore be parallelized.

2. Functional programming in Coq

(1) The specification language of Coq is *Gallina*. Gallina is a functional language.

(2) Functions are *first-class* values in a functional language i.e. functions can be passed as arguments to other functions, can be returned as results, included in data structures etc.

(3) Functional programming also supports algebraic data types, pattern matching and polymorphic type systems.

(4) The number of built-in features in Coq are extremely small.

(5) *Expressions* in Gallina are formed with constants and identifiers following a few construction rules. Every expression has a *type*. Type is given by a *declaration* and rules for combining expressions come from *typing rules*.

(6) In Coq, computations always terminate (the property of *strong normalization*). As a consequence, there exist computable functions that can be described in Coq but for which the computation cannot be performed by reduction mechanism.

(7) It is possible to express assertions or *propositions* about the values being manipulated.

(8) It is impossible to design a general algorithm that can build a proof of every true formula.

(9) A variation on Church's *typed λ-calculus* called *Calculus of Inductive Constructions* is used as the underlying formulation for Coq.

(10) The *Howard-Curry* isomorphism gives a relation between proofs and programs. It states that the relation between a program and its type is the same as the relation between a proof and the statement it proves. Thus, Coq verifies a proof by a type verification algorithm.

(11) In the Calculus of Constructions, every type is also a term and also has a type. For example, the proposition $3 \leq 7$ is the type of all proofs that 3 is smaller than 7 and it is at the same time a term of type `Prop`.

(12) A *predicate* makes it possible to build a parametric proposition. For example, "to be a prime number" is a predicate whose type is `nat`→`Prop`. "To be a sorted list" is a predicate of type `(list Z)`→`Prop` and the binary relation $\leq$ has type `Z`→`Z`→`Prop`. "To be a transitive relation on `Z`" is of type `(Z`→`Z`→`Prop)`→`Prop`. A polymorphic version of "to be transitive" will then be of type `(`A→A→`Prop)`→`Prop`, where A is any data type.

(13) Just as predicates can be converted to types, conversely, types can be converted to logical constraints (in terms of variables of other types). Thus, types that themselves depend on values (of possibly different types) and *dependent types*.

(14) An *extraction algorithm* can be used to convert a proof into a certified program in the *OCaml* language that can be compiled and run. The extraction algorithm replaces all logical statements with computations that need to be performed as a check of the validity of the proof.

(15) Example of defining a list sorting algorithm in Coq:
- A list of numbers will have type `list Z`.
- An empty list will be represented by `nil`.
- A list $[5, 2, 9]$ will be represented by `5::2::9::nil`.
- Adding an element n to a list l creates the list n`::`l.
- In order to define a new type we need two predicates defined by taking inspiration from the *Prolog* language. This is done by considering three clauses which together form an inductive definition. The clauses are:
  (a) the empty list is sorted,
  (b) every list with only one element is sorted,
  (c) if a list of form n`::`l is sorted and if $p \leq n$ then the list p`::`n`::`l is sorted.
- These clauses can be programmed in Prolog as follows:

```
Inductive sorted:list Z→ Prop:=
sorted0: sorted(nil)
sorted1: ∀z : Z, sorted(z::nil)
sorted2: ∀z1,z2 : Z, ∀l : list Z, z1 ≤ z2 ⟹ sorted(z2::l)
      ⟹ sorted(z1::z2::l)
```

(16) Enumerated types (recall that types are just sets of values) can be defined using the `Inductive` environment as follows:

```
Inductive day : Type :=
  | monday    : day
  | tuesday   : day
  | wednesday : day
  | thursday  : day
  | friday    : day
  | saturday  : day
  | sunday    : day.
```

(17) Functions can be defined using the `Definition` environment as follows:

```
Definition next_weekday (d:day) : day :=
  match d with
  | monday    => tuesday
  | tuesday   => wednesday
  | wednesday => thursday
  | thursday  => friday
  | friday    => monday
  | saturday  => monday
  | sunday    => monday
  end.
```

(18) Computations can be carried out using the `Compute` environment while types can be checked using the `Check` environment.

```
Compute next_weekend monday. (**This returns tuesday. *)
Check next_weekend.          (**This returns day -> day *)
```

(19) Expectations can be recorded using `Example` environment as follows:

```
Example. test1 : next_weekday saturdaty = next_weekday sunday. (** The name of this
    example is test1 *)
```

(20) This creates a goal that then needs a proof. The proof can be written as:

```
Proof. simpl. reflexivity. Qed.
```

where `simpl` means simplify both sides of the equation and `reflexivity` checks if LHS and RHS are equal.

(21) Notation can be defined by `Notation` environment.

```
Notation "x && y" : (andb x y).
```

(22) Modules can be started and ended as follows:

```
Module MyModuleName.
(** add content here*)
End MyModuleName.
```

then any definition `foo` in this module can be referenced as `MyModuleName.foo`.

(23) `Definition` works for most definitions that involve pattern-matching. However, for recursive definitions we use `Fixpoint`

```
Fixpoint factorial (n : nat): nat :=
  match n with
  | 0 => 1
  | S n2 => factorial n2
  end.
```

There are several things we need to note here –

- The `Fixpoint` environment works only as long as we decrease `n` in each subsequent pass to the function.

- At the same time, we cannot have something like `n' => factorial (n'-1)` because the recursion is accepted only if it is called on `n'`, not on `n'-1`.
- Furthermore, we cannot have something like `n'+1 => factorial(n')` because `n'+1` is not a good term for pattern-matching. Hence we are forced to use `S n'` where `S` is the successor function for natural numbers.

(24) Another example with two pattern-matches – we define the less-that-or-equal relation

```
Fixpoint leq (n m : nat) : bool :=
  match n with
  | 0 => true
  | S n2 => match m with
            | 0 => false
            | S m2 => leq n2 m2
            end
  end.
```

(25) To be continued...