NOTES ON COQ

VAIBHAV KARVE

These notes were last updated July 18, 2018. They are notes taken from 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/.

1. A Brief Overview

(1) *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.

(2) The specification language of Coq is *Gallina*.

(3) *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*.

(4) 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.

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

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

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

(8) 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.

(9) 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`.

(10) A *predicate* makes it possible to build a parametric proposition. For example, "to be a prime number" is a predicate whose type is `nat`$\rightarrow$`Prop`. "To be a

sorted list" is a predicate of type `(list Z)`$\rightarrow$`Prop` and the binary relation $\leq$ has type `Z`$\rightarrow$`Z`$\rightarrow$`Prop`. "To be a transitive relation on `Z`" is of type `(Z`$\rightarrow$`Z`$\rightarrow$`Prop)`$\rightarrow$`Prop`. A polymorphic version of "to be transitive" will then be of type `(`$A$$\rightarrow$$A$$\rightarrow$`Prop)`$\rightarrow$`Prop`, where A is any data type.

(11) 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*.

(12) 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.

(13) 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)
```

(14) To be continued...