How we interact with the type theories in Rocq

Anja Petković Komel

Introduction to type theories
OPLSS 2025

June 2025

My groupie photo

- Official Rocq documentation:
  `https://rocq-prover.org/docs`

Disclaimer: In this lecture we will not cover the full power of Rocq, but merely the basics.

Files and comments

- Rocq files are text files ending with `.v`
- One can type-check a Rocq file from the command line with rocq compile

  ```
  rocq c file.v
  rocq compile file.v
  ```

  or by using rcoq top, preferably from RocqIDE or your favorite editor (I use VSCode).
- `(* Comment *)`

Basics

- Coq commands end with a dot.
- Syntax for definitions

```
Definition <name> : <type> := <term>.
```

- We step through the proof script: checked part is colored green.
- Function types are written with $\rightarrow$.
- Dependent function types are written with `forall x : A, B`.
- Lambda abstraction syntax is `fun x : A ⇒ t`.

- Check the type of a term using `Check t`.
- Print the full term definition and its type using `Print t`.

Tactics

Rocq is designed to prove theorems in a way that is similar to reasoning with natural deduction – using **tactics**. One can first write the type of a definition/theorem/lemma and then write a proof and walk through the proof scripts.

- List of most commonly used tactics with examples: `https://pjreddie.com/coq-tactics/`.
- For more advanced tactics (and use cases) consult the official documentation.

Fixpoint vs. ordinary match

- Functions out of inductive types can defined with pattern matching using the `match` eliminator.
- Recursive definitions require the explicit `Fixpoint` keyword.

Defined vs. Qed

Concluding proof with `Defined` or `Qed`:

- Type checks/verifies the constructed proof.
- Creates the proof object in the environment.
- Checks for termination (if applicable).

Difference:

- `Defined` is transparent: can be unfolded.
- `Qed` is opaque: cannot be unfolded.

We can leave holes in the proof script: using the keyword `Admitted` and tactic `enough`.

MLTT vs CIC: eliminating existential quantifier

There are three kinds of existential quantifiers in Coq:

```
Print ex.
Print sig.
Print sigT.
```

- `ex`: Cannot be eliminated with `destruct`, maps into `Prop`.
- `sig`: Can be eliminated with `destruct`, maps into `Prop`.
- `sigT`: Can be eliminated with `destruct`, maps into `Type`, it is the usual sigma type like in MLTT.

MLTT vs CIC: Prop

Source: Christine Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. Bruno Woltzenlogel Paleo; David Delahaye. All about Proofs, Proofs for All, 55, College Publications, 2015, Studies in Logic (Mathematical logic and foundations), 978-1-84890-166-7. hal-01094195

$$\mathcal{S} \stackrel{\text{def}}{=} \{\mathsf{Prop}\} \cup \bigcup_{i \in \mathbb{N}} \{\mathsf{Type}_i\}$$

$$\frac{\Gamma, x : A \vdash B : \mathsf{Prop}}{\Gamma \vdash \Pi x : A, B : \mathsf{Prop}} \qquad \frac{\Gamma, x : A \vdash B : \mathsf{Type}_i \qquad \Gamma \vdash A : \mathsf{Type}_i}{\Gamma \vdash \Pi x : A, B : \mathsf{Type}_i}$$

MLTT vs CIC: Prop

$$\frac{\begin{array}{l} t : I\,pars\,t_1 \ldots t_p \\ y_1 \ldots y_p, x : I\,pars\,y_1 \ldots y_p \vdash P : s' \\ \{x_1 : A_1 \ldots x_n : A_n \vdash f : P[u_1/y_1, \ldots, u_p/y_p, (c\,x_1 \ldots x_n)/x]\}_c \end{array}}{\begin{array}{l} \textbf{match}\,t\,\textbf{as}\,x\,\textbf{in}\,I\,\_\,y_1 \ldots y_p\,\textbf{return}\,P \\ \textbf{with} \ldots |\,c\,x_1 \ldots x_n \Rightarrow f| \ldots \\ \textbf{end} : P[t_1/y_1, \ldots, t_p/y_p, t/x] \end{array}}$$

Type-checking conditions. The main restriction lies in the relation between the sort s of the inductive definition and the sort s' of the pattern-matching.

When s is Type, which means that we have a predicative inductive definition, then we can have any possible sorts s' for case analysis.

When s is Prop however, the question is a bit more tricky for several reasons:

- Prop is an impredicative sort, so uncareful elimination can easily introduce paradoxes;
- it is sometimes useful to add an axiom of proof irrelevance for propositions (which says that two different proofs of the same property can be considered as equal) so while it is good to be able to prove that for instance `true` $\neq$ `false`, a similar mechanism that will lead to two terms (representing proofs of) in $A \vee B$ that are provably different is less desirable;
- Prop is used for program extraction: any term in A : Prop is removed during extraction so should not be needed for computing the informative art, in a pattern-matching is done on a term in an inductive definition in Prop, but with the result being used for computing, then we need to be able to execute the match without executing the head, which is only feasible in specific cases.

For an inductive definition of sort Prop, the only elimination allowed is on the sort Prop itself. There are exceptions where any elimination is allowed: in the specific case where I is a *predicative* definition with only zero or one constructor with all its arguments A_i : Prop. The exception covers cases like absurdity, equality, conjunction of two propositions, accessibility...