Introduction to Type Theories — Anja Petković Komel

Lecture 5 - *June 26, 2025*

1 Martin-Löf Type Theory Principles in Rocq

We continue from last lecture by looking at Rocq tactics and seeing what type theory they are "hiding".

Some Rocq syntax:

- `Check x`: Displays the type of `x`.
- `About x`: Similar to check, `About` displays the type of `x` but also tells you which arguments are implicit.
- `Print x`: Displays what is defined, both the term and the type. If `x` is a defined proof, `Print x` will display the full term that the proof compiled to. Recalling Curry-Howard, the tactics in Rocq to build proofs are building a term of the type.

1.1 The Empty Type

`True` is the unit type and `False` is the empty type.

We can prove the ex falso principle which says that if we can derive the empty type, anything can be proven.

```
Definition exfalso_example : forall A : Type, False -> A.
Proof.
  intros A p.
  exfalso.
  exact p.
Defined.
```

Some Rocq tactics relating to the empty type are:

- `exfalso`:

  `exfalso` introduces `False` as the goal, hinging on the fact that if we can derive false we can prove anything. In this case, after `intros`, we have a proof of `False` in our context as `p`.

- `discriminate`:

  This tactic works if you have something clearly false in your context. If you do, it applies `exfalso` for you and proves the goal. Otherwise, it fails.

1.2 Variable Conversion

Consider the proof below:

```
(* variable conversion *)
Definition variable_conv : forall n : nat, 0 + n = n -> n = n.
Proof.
  intros n p.
  (* point 1 *)
  exact p.
Defined.
```

At point 1, we have a proof, `p`, that $0 + n = n$. When we invoke `exact p`, Rocq uses variable conversion to convert the type of the variable `p` in our context to the type $n = n$ in the goal, and the proof is defined. The variable conversion is $0 + n \doteq 0$. Instead of this, we could invoke `simpl in p` then `exact p`, but then Rocq is not using the variable conversion rule.

Another example of Rocq implicitly doing variable conversion is the user specifying a different type than is defined. If we have `banana 0 : 0 + 0 = 0 + 0`, then we `Check banana 0 : 0 = 0`, Rocq will do variable conversion and display `banana 0 : 0 = 0`.

1.3 Transport

Recall the transport rule, shown below.

$$\mathrm{tr}_{A,B} : \Pi_{x,y:A}(x =_A y) \to B(x) \to B(y)$$

Now, let's consider the type family of the `Vec` which will be used to show examples of how Rocq can use the transport rule implicitly and explicitly.

```
Inductive Vec (A : Type) : nat -> Type :=
    nil : Vec A 0
    | cons : A -> forall n : nat, Vec A n -> Vec A (S n).
```

Based on `Vec`, we want to define some basic operations like `concat` and `rev`.

1.3.1 Concat

We define `concat` below.

```
Definition concat {A : Type} {n m : nat}:
                                Vec A n -> Vec A m -> Vec A (n + m).
Proof.
    intros v u.
    induction v.
    (* point 1 *)
    - exact u.
    (* point 2 *)
    - exact (cons A a (n+m) IHv).
Defined.
```

At point 1, we have `u : Vec A m` and must construct a term for `Vec A (0 + m)`. Relating this to the definition of the transport rule above, $B(x)$ is `Vec A m`, and we are trying to construct $B(y)$ as `Vec A (0 + m)`. When `exact u` is invoked, Rocq implicitly uses the transport rule, and can automatically infer the other required premise: that `m = 0 + m`.

1.3.2 Rev

Following a similar approach, we attempt to define `rev` as well:

```
Definition rev {A : Type} {n m : nat} : Vec A n -> Vec A n.
Proof.
   intros v.
   destruct v.
   - exact (nil A).
   - exact (concat IHv (cons A a 0 (nil A))).
Defined.
```

But here, Rocq cannot implicitly apply the transport rule. We can observe that the `concat` produces type `Vec A (n+1)`, whereas our desired object type is actually `Vec A (S n)`.

Whilst they are equal as natural numbers, such two representations are not definitionally equal, so Rocq cannot infer them as such for the transport rule. These are different syntactic forms that requires a proof to establish their actual equality. Therefore, we use the built-in explicit transport rule in rocq, `eq_rect_r`, shown below.

```
About eq_rect_r.
(* eq_rect_r : forall (A : Type) (x : A) (P : A -> Type),
                P x -> forall y : A, y = x -> P y *)
```

The order of arguments is different than our definition above, but they are the same.

We can instantiate this with the type families from `Vec` and provide a proof that the terms are equal to use the transport rule explicitly.

```
Lemma foo : forall n : nat, S n = n + 1.

eq_rect_r (fun d => Vec A d)
          (concat IHv (cons A a 0 (nil A)))
          (foo n)
```

In this way, we can:

1. Start with a term of type `Vec A (n+1)`
2. Have a proof that `S n = n + 1`
3. Then transport will give us a term of type `Vec A (S n)` that satisfies our type requirement.

2 Universes

As of yet, our types are types and our terms are terms, and never the twain shall meet. It would be convenient to allow (some) types to be used as terms, and for this purpose we'll introduce the notion of a *universe*: a "type of types"; that is, a type whose terms can themselves be interpreted as types.

The definition of universe has been refined over time, and in lecture, Prof. Komel presented a very modern definition. For added context, in these notes we will present it somewhat differently, getting to the definition from lecture by iterating on an initial attempted definition.

Attempt 1. We define a type $\mathcal{U}$ whose terms are all the types:

$$\frac{}{\Gamma \vdash \mathcal{U}\ \text{type}} \qquad \frac{\Gamma \vdash A : \mathcal{U}}{A\ \text{type}} \qquad \frac{\Gamma \vdash A\ \text{type}}{\Gamma \vdash A : \mathcal{U}}$$

This setup does not work. Philosophically, it seems weird to have $\mathcal{U}$ be an element of itself –

$$\frac{\dfrac{}{\vdash \mathcal{U}\ \text{type}}}{\vdash \mathcal{U} : \mathcal{U}}$$

– and mathematically, this shows up as *Girard's paradox*, a type-theoretic version of Russell's paradox. Perforce, there can't be a set of all sets, nor a type of all types.

Attempt 2. As $\mathcal{U}$ can't be a type of *all* types, we go to the next best thing: a type of most of the types that we care about. Concretely, we'll ask it to contain 0, 1, $\mathbb{N}$, Π types, and identity types.

$$\frac{}{\Gamma \vdash \mathcal{U}\ \text{type}} \qquad \frac{\Gamma \vdash A : \mathcal{U}}{A\ \text{type}} \qquad \frac{}{\Gamma \vdash 0 : \mathcal{U}} \qquad \frac{}{\Gamma \vdash 1 : \mathcal{U}} \qquad \frac{}{\Gamma \vdash \mathbb{N} : \mathcal{U}}$$

$$\frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma, x : A \vdash B(x) : \mathcal{U}}{\Gamma \vdash \Pi_{(x:A)} B(x) : \mathcal{U}} \qquad \frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma \vdash x : A \qquad \Gamma \vdash y : A}{\Gamma \vdash x =_A y : \mathcal{U}}$$

This version ("Russell-style") no longer has paradoxes! However, there are reasons why it is still unsatisfactory:

- These rules force the elements of $\mathcal{U}$ to be types themselves. However, all we really need is for each element of $\mathcal{U}$ to *uniquely determine* a type, and this would give more flexibility in defining our type system.
- Before adding these rules, we had different syntactic classes of terms and of types. With Russell-style universes, the two are merged, which makes metatheory trickier.
- Philosophically, it seems odd to have abandoned our formula of introduction, elimination, and computation rules.

Attempt 3 (final version). We tweak the above by making the terms of $\mathcal{U}$ their own thing, which get coerced to types via a type family $\mathcal{T}$. We call the elements of $\mathcal{U}$ *codes* for types, to distinguish them from the types themselves. In another way, we can understand $\mathcal{T}$ as a tool to

transform a "encoding of type / type code" into a type.

$$\frac{}{\Gamma \vdash \mathcal{U}\ \text{type}} \qquad \frac{\Gamma \vdash A : \mathcal{U}}{\Gamma \vdash \mathcal{T}(A)\ \text{type}} \qquad \frac{}{\begin{array}{c}\Gamma \vdash \check{0} : \mathcal{U} \\ \Gamma \vdash \mathcal{T}(\check{0}) \doteq 0\ \text{type}\end{array}} \qquad \frac{}{\begin{array}{c}\Gamma \vdash \check{1} : \mathcal{U} \\ \Gamma \vdash \mathcal{T}(\check{1}) \doteq 1\ \text{type}\end{array}}$$

$$\frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma, x : \mathcal{T}(A) \vdash B(x) : \mathcal{U}}{\begin{array}{c}\Gamma \vdash \check{\Pi}(A, B) : \mathcal{U} \\ \Gamma \vdash \mathcal{T}(\check{\Pi}(A, B)) \doteq \Pi_{(x:\mathcal{T}(A))}\mathcal{T}(B(x))\ \text{type}\end{array}} \qquad \frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma \vdash x : \mathcal{T}(A) \qquad \Gamma \vdash y : \mathcal{T}(A)}{\begin{array}{c}\Gamma \vdash \check{eq}(A, x, y) : \mathcal{U} \\ \Gamma \vdash \mathcal{T}(\check{eq}(A, x, y)) \doteq (x =_{\mathcal{T}(A)} y)\ \text{type}\end{array}}$$

The formation rule gives us $\mathcal{U}$ type, and the elimination rule gives us $\mathcal{T}(A)$ type for $A : \mathcal{U}$. For ease of notation, I've written the introduction rules and computation rules together, just as a shorthand.

Aside. $\mathcal{T}(\cdot)$ is often called $\mathsf{El}(\cdot)$ in other sources.

This version (universe à la Tarski) supports all the same uses as our previous version, while fixing the issues we had, at the cost of being more verbose. However, we can recover exactly the syntax of the previous attempt using the notational convention that $\mathcal{T}$ is implicit, and that 0 is overloaded to mean both 0 and $\check{0}$, etc. Today, even when a type system is presented using Russell-style universes, its metatheory may be studied using a corresponding Tarski-style presentation.

2.1 A whole hierarchy of universes

Just one universe $\mathcal{U}$ can only contain *some* of the types, so type theories will typically include many universes. A common choice is a countable hierarchy $\mathcal{U}_0, \mathcal{U}_1, \mathcal{U}_2, \ldots$, where each $\mathcal{U}_k$ has codes for each of the preceding $\mathcal{U}_i$'s, $i < k$.

However, rather than fix any particular hierarchy of universes, we'll simply assume that there are "enough" universes, in a particular sense:

Assumption 2.1 (Rijke, 6.2.1, [2])**.** *For any finite collection of types-in-context*

$$\Gamma_1 \vdash A_1\ \textit{type} \qquad \Gamma_2 \vdash A_2\ \textit{type} \qquad \ldots \qquad \Gamma_n \vdash A_n\ \textit{type},$$

there exists a universe $(\mathcal{U}, \mathcal{T}(\cdot))$ *with codes*

$$\Gamma_i \vdash \check{A}_i : \mathcal{U} \qquad \Gamma_i \vdash \mathcal{T}(\check{A}_i) \doteq A_i\ \textit{type}$$

for each $i = 1, \ldots, i = n$.

Aside. This assumption is closely related to Grothendieck's *axiom of universes* (SGA4) from a set-theoretic context. The universes in Rocq, Agda, and Lean all satisfy this assumption.

As instances of this axiom, we get the following constructions on universes.

- Successor universe. Let $\mathcal{U}$ be a universe. We have types-in-context

$$A : \mathcal{U} \vdash \mathcal{T}(A) \text{ type} \qquad\qquad \vdash \mathcal{U} \text{ type}.$$

So there is a universe, which we'll call $\mathcal{U}^+$, containing (codes for) all the types in $\mathcal{U}$, as well as $\mathcal{U}$ itself.

To give a sense of this universe, consider the following type judgements in the successor universe:

$$\vdash \check{\mathcal{U}} : \mathcal{U}^+ \qquad\qquad \vdash \mathcal{T}^+(\check{\mathcal{U}}) \doteq \mathcal{U} \text{ type}$$

$$X : \mathcal{U} \vdash \check{\mathcal{T}}(X) : \mathcal{U}^+ \qquad\qquad X : \mathcal{U} \vdash \mathcal{T}^+(\check{\mathcal{T}}(X)) \doteq \mathcal{T}(X) \text{ type}$$

By applying this repeatedly, we can consider $(\mathcal{U}^+)^+$, etc.

- Join of universes. The join of two universes $\mathcal{U}$ and $\mathcal{V}$ is the universe $\mathcal{U} \sqcup \mathcal{V}$, by applying the postulate that there are always enough universes on the following two type families:

$$X : \mathcal{U} \vdash \mathcal{T}_{\mathcal{U}}(X) \text{ type} \qquad\qquad Y : \mathcal{V} \vdash \mathcal{T}_{\mathcal{V}}(Y) \text{ type}.$$

An intuitive way to view the join of universes is to imagine it as "lifting the levels of universes".

3 Bonus: MLTT vs CIC

In mainstream theorem provers, Agda is built on MLTT whilst Rocq and Lean is using CIC (Calculus of Inductive Constructions).

Prop One of the main differences between them is about the *Prop*. Rocq and Lean treat `Prop` as an impredicative sort, and they define their *sorts* (universes) as follows[1]:

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

Prop is special, in that CIC has special rules for dependent function types in Prop.

$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma, x : A \vdash B(x) : \text{ Prop}}{\Gamma \vdash \Pi_{x:A} B(x) : \text{ Prop}} \qquad\qquad \frac{\Gamma \vdash A : \text{ Type}_i \qquad \Gamma, x : A \vdash B(x) : \text{ Type}_i}{\Gamma \vdash \Pi_{x:A} B(x) : \text{ Type}_i}$$

Constructing $\prod_{(x:A)} B(x) : \text{Type}_i$ requires that $A : \text{Type}_i$, but Prop has no such restrictions: it includes dependent products out of every $\vdash A$ type. This property is called being *impredicative*.

Eliminating Existential Quantifiers There are three kinds of existential quantifiers in Rocq:

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

They behave differently:

- ex: Cannot only eliminated with `destruct` when proving something in `Prop`, maps into `Prop`.
- sig: Can always be eliminated with `destruct`, maps into `Prop`.
- sigT: Can always be eliminated with `destruct`, maps into `Type`. It is the usual Σ-type like in MLTT.

You can refer to the `fall_ex` and `fall_ex_ty` examples in the code handout. Sometimes a change of universe level could resolve lots of seemingly tricky issues.

References

[1] Christine Paulin-Mohring. "Introduction to the calculus of inductive constructions". In: *All about Proofs, Proofs for All* 55 (2015).

[2] Egbert Rijke. *Introduction to Homotopy Type Theory*. 2022. arXiv: 2212.11082 [math.LO]. URL: https://arxiv.org/abs/2212.11082.