Lecture 2: Propositions, Proofs, and Logical Rules

Stefano M. Nicoletti

1 Lecture 2: Propositions, Proofs, and Logical Rules

1.1 Aim of the Lecture

This lecture recalls the role of the kernel and the typechecker, then introduces the way Lean reads propositions and proofs. The central point is the correspondence between propositions and types: a proposition `P : Prop` is a type, and a proof of `P` is a term inhabiting that type (Source: Theorem Proving in Lean 4, Propositions and Proofs).

We will work through minimal examples in order to observe, in the proof state, the introduction and elimination rules for the logical connectives and for a few specific constructs:

- implication `P → Q`;
- conjunction `P ∧ Q`;
- disjunction `P ∨ Q`;
- negation ¬P;
- `False`.

1.2 Kernel and Typechecker

We have seen that Lean lets us build proofs with tactics, library support, and interactive interfaces. These tools help the user, but in the end the kernel checks the formal object that has been produced and verifies that it has the required type (Source: de Moura and Ullrich, Lean 4).

The typechecker checks whether a term has a certain type. In proofs, the expected type is the proposition that we want to prove. A theorem is accepted when Lean can check the proof term against the statement. From the system's point of view, proving means constructing a well-typed term.

This explains why the proof state is useful. The goal shows the type that we must inhabit. The assumptions show the terms already available in the context. A tactic is an interactive way to transform this situation until all goals are closed.

1.3 Propositions as Types

In Lean, `P : Prop` says that `P` is a proposition. If the context contains `hP : P`, then `hP` is a proof of `P`. It is not just an informal label: it is a formal object that can be used to build other

proofs.

Implication is the most direct example. A term of type `P → Q` behaves like a function: it takes a proof of `P` and returns a proof of `Q`. If we have

```
hPQ : P → Q
hP : P
```

then `hPQ hP` is a proof of `Q`.

1.4 Introduction and Elimination Rules

For each logical connective we distinguish two questions.

- The introduction rule explains how to construct a proof of the compound proposition.
- The elimination rule explains how to use a proof of the compound proposition to obtain its components.

Looking at the goal means asking which introduction rule can construct it. Looking at the assumptions means asking which elimination rule can use them. This terminology follows the standard presentation of natural deduction rules (Source: Logic and Proof, Natural Deduction Rules).

1.5 Implication

To introduce `P → Q`, we temporarily assume `P` and construct `Q`.

```
theorem lecture02_imp_intro (P Q : Prop) (hQ : Q) :
    P → Q := by
  intro hP
  exact hQ
```

The line `intro hP` introduces the temporary assumption `hP : P`. In this example the conclusion `Q` is already available as `hQ`, so we close the goal with `exact hQ`.

To eliminate `P → Q`, we apply the implication to a proof of `P`.

```
theorem lecture02_imp_elim (P Q : Prop) :
    (P → Q) → P → Q := by
  intro hPQ
  intro hP
  have hQ := hPQ hP
  exact hQ
```

Here `hPQ hP` is function application: from `P → Q` and `P` we obtain `Q`. This is modus ponens again.

1.6 Conjunction

A conjunction `P ∧ Q` contains both pieces of information. To introduce it, we must provide a proof of both components. In the next example they are given as assumptions in the statement.

```
theorem lecture02_and_intro (P Q : Prop) (hP : P) (hQ : Q) :
    P ∧ Q := by
  have hPAndQ := And.intro hP hQ
  exact hPAndQ
```

The same rule `And.intro` can be used as a tactic. The goal `P ∧ Q` splits into two subgoals, one for `P` and one for `Q`.

```
theorem lecture02_and_intro_with_apply (P Q : Prop) (hP : P) (hQ : Q) :
    P ∧ Q := by
  apply And.intro
  · exact hP
  · exact hQ
```

To eliminate a conjunction, we use its components and take the left or right conjunct.

```
theorem lecture02_and_elim_left (P Q : Prop) :
    P ∧ Q → P := by
  intro hPAndQ
  exact hPAndQ.left
```

```
theorem lecture02_and_elim_right (P Q : Prop) :
    P ∧ Q → Q := by
  intro hPAndQ
  exact hPAndQ.right
```

1.7 Disjunction

A disjunction `P ∨ Q` contains one of the two pieces of information. To introduce it, it is enough to provide one side, using `Or.inl` or `Or.inr`.

```
theorem lecture02_or_intro_left (P Q : Prop) :
    P → P ∨ Q := by
  intro hP
  exact Or.inl hP
```

```
theorem lecture02_or_intro_right (P Q : Prop) :
    Q → P ∨ Q := by
  intro hQ
  exact Or.inr hQ
```

To eliminate a disjunction, we reason by cases. If we have `P ∨ Q`, we do not generally know which side is available. To obtain a conclusion, we must produce it in both branches. In this

example, we want to conclude `Q ∨ P` from `P ∨ Q`, so we consider the case where `P` is true and the case where `Q` is true.

```
theorem lecture02_or_elim (P Q : Prop) :
    P ∨ Q → Q ∨ P := by
  intro hPOrQ
  cases hPOrQ with
  | inl hP =>
      exact Or.inr hP
  | inr hQ =>
      exact Or.inl hQ
```

The same structure can be written with `Or.elim`.

```
theorem lecture02_or_elim_with_or_elim (P Q : Prop) :
    P ∨ Q → Q ∨ P := by
  intro hPOrQ
  apply Or.elim hPOrQ
  · intro hP
    exact Or.inr hP
  · intro hQ
    exact Or.inl hQ
```

1.8 False and Negation

`False` represents a contradictory state. If we have a proof of `False`, we can close any goal.

```
theorem lecture02_false_elim (P : Prop) :
    False → P := by
  intro hFalse
  exact False.elim hFalse
```

In Lean, negation is defined as implication to `False`: `¬P` means `P → False`. To eliminate a negation, we apply it to a positive proof of `P`.

```
theorem lecture02_not_elim (P : Prop) :
    P → ¬P → False := by
  intro hP
  intro hNonP
  exact hNonP hP
```

To introduce `¬P`, we temporarily assume `P` and derive `False`. The next example is the reasoning pattern of modus tollens. The line `intro hP` has this role: to prove `¬P`, assume `P` and derive `False`. Observe how the goal changes.

```
theorem lecture02_not_intro (P Q : Prop) :
    (P → Q) → ¬Q → ¬P := by
  intro hPQ
  intro hNonQ
  intro hP
```

```
  have hQ := hPQ hP
  exact hNonQ hQ
```

1.9 Tactics, Commands, and Shortcuts

In this lecture we mostly used tactics corresponding to the introduction and elimination rules for the connectives.

- `intro`: introduces a temporary assumption when the goal is an implication or a negation;
- `exact`: closes the goal by providing a term of the required type;
- `have`: introduces an intermediate result into the context;
- `apply`: applies a rule, theorem, or constructor to the current goal;
- `cases`: distinguishes the cases of a disjunction.

We also used some explicit constructors and eliminators:

- `And.intro`, `.left`, `.right` for conjunction;
- `Or.inl`, `Or.inr`, `Or.elim` for disjunction;
- `False.elim` for using a contradiction.

1.10 Exercises and Solutions

The exercise and solution files (`Exercises.lean` and `Solutions.lean`) are available on the course website.

1.11 Sources and Further Reading

- Jeremy Avigad, Leonardo de Moura, Soonho Kong, Sebastian Ullrich, *Theorem Proving in Lean 4*, chapter "Propositions and Proofs": https://lean-lang.org/theorem_proving_in_lean4/Propositions-and-Proofs/.
- Jeremy Avigad, Robert Y. Lewis, Floris van Doorn, *Logic and Proof*, appendix "Natural Deduction Rules": https://leanprover.github.io/logic_and_proof_lean3/nd_quickref.html.
- Leonardo de Moura and Sebastian Ullrich, "The Lean 4 Theorem Prover and Programming Language": https://lean-lang.org/papers/lean4.pdf.