Lecture 4: Classical Logic and Quantifiers

Stefano M. Nicoletti

1 Lecture 4: Classical Logic and Quantifiers

1.1 Aim of the lecture

Lecture 3 connected Lean tactics with the introduction and elimination rules of natural deduction. This lecture extends that framework in two directions. We first introduce reductio ad absurdum as a classical rule. We then move from propositional logic to predicate logic and learn to represent objects, properties, functions, relations, and quantifiers.

The lecture will:

- distinguish constructive rules from classical principles and apply reductio ad absurdum;
- build a small first-order language in Lean;
- formalize statements containing ∀ and ∃.

1.2 From constructive rules to classical logic

The rules seen so far have a direct constructive reading: each proof specifies how to build the required term. To prove a conjunction we construct both components; to prove an implication we construct a function.

In Lean's constructive base logic, excluded middle `A ∨ ¬A` and double-negation elimination `¬¬A → A` are not automatically available. Lean does allow classical principles to be used locally and explicitly (Source: Theorem Proving in Lean 4, Classical Logic).

1.3 Reductio ad absurdum

Classical reductio ad absurdum proves `A` by showing that assuming `¬A` leads to `False`. In Lean, we apply it with `Classical.byContradiction`.

$$\dfrac{\dfrac{}{\neg A}\,1}{\vdots}$$
$$\dfrac{\bot}{A}\,1\ \text{RAA}$$

Reductio ad absurdum, RAA

The working pattern is: apply reductio, introduce ¬A, derive `False` using the available assumptions, and use exactly that contradiction.

```
theorem lecture04_reductio_ad_absurdum
    (Rains TakeUmbrella : Prop)
    (hNotRainsNotUmbrella : ¬Rains → ¬TakeUmbrella)
    (hTakeUmbrella : TakeUmbrella) :
    Rains := by
  apply Classical.byContradiction
  intro hNotRains
  have hNotTakeUmbrella := hNotRainsNotUmbrella hNotRains
  have hContradiction := hNotTakeUmbrella hTakeUmbrella
  exact hContradiction
```

The first assumption turns `¬Rains` into `¬TakeUmbrella`. The latter is a function from `TakeUmbrella` to `False`; applying it to `hTakeUmbrella` produces the required contradiction.

1.4 A limitation of propositional logic

In propositional logic, each complete statement is represented by a separate proposition. We might use `P` for "Hypatia is a philosopher," `Q` for "Hannah Arendt is a philosopher," and `R` for "Hypatia is curious." This hides the fact that `P` and `Q` attribute the same property to different objects. It also prevents us from directly expressing "every philosopher is curious" or "there is a curious philosopher."

Predicate logic exposes the objects we discuss and the properties or relations attributed to them.

1.5 The domain of discourse

Every formalization begins with a *domain*: the collection of objects under consideration. In this lecture we use one general type.

```
variable (Thing : Type)
```

`Thing` represents the domain of discourse. A term `x : Thing` is an element of that domain. This is the usual single-domain presentation of first-order logic (Source: Logic and Proof, First Order Logic in Lean).

Named objects are represented by terms of the domain:

```
variable (Thing : Type)
variable (hypatia hannah rome athens : Thing)
```

These declarations do not yet assert any proposition. They only introduce terms that will receive an interpretation.

1.6 Predicates: properties of objects

A one-place predicate takes an object and returns a proposition.

```
variable (Thing : Type)
variable (Person City Philosopher Curious : Thing → Prop)
variable (hypatia hannah : Thing)

#check Philosopher hypatia
#check Philosopher hannah
```

`Philosopher hypatia` and `Philosopher hannah` are different propositions, but they share the same predicate.

Natural language	Lean
Hypatia is curious	`Curious hypatia`
Hannah Arendt is curious	`Curious hannah`
Io is a moon of Jupiter	`MoonOfJupiter io`
It rains in Rome	`RainsIn rome`

1.7 Functions: constructing new terms

A function takes one or more objects from the domain and returns an object.

```
variable (Thing : Type)
variable (hypatia : Thing)
variable (motherOf : Thing → Thing)
variable (Curious : Thing → Prop)

#check motherOf hypatia
#check Curious (motherOf hypatia)
```

`motherOf hypatia : Thing` is a new term. Since its type is still `Thing`, it can be used as the argument of a predicate. Functions can be nested: `motherOf (motherOf hypatia)` denotes the mother of Hypatia's mother.

In first-order logic a function is total and returns exactly one result for each argument. A function returns an object; a predicate returns a proposition.

1.8 Relations between objects

A predicate with two or more arguments expresses a relation.

```
variable (Thing : Type)
variable (hypatia hannah athens : Thing)
variable (Knows Visits : Thing → Thing → Prop)

#check Knows hypatia hannah
#check Visits hypatia athens
```

Argument order matters. `Knows hypatia hannah` and `Knows hannah hypatia` represent different statements.

1.9 Local parameters and assumptions

The keyword `variable` introduces local parameters. A `section` limits their scope, and Lean adds to a theorem only the parameters actually used in its statement.

```
section PredicateLogicLanguage

variable (Thing : Type)
variable (hypatia hannah : Thing)
variable (Philosopher Curious : Thing → Prop)
variable (motherOf : Thing → Thing)
variable (Knows : Thing → Thing → Prop)

end PredicateLogicLanguage
```

Declaring `Philosopher : Thing → Prop` does not assert that anyone is a philosopher. A genuine assumption that Hypatia is a philosopher has the form `hHypatiaPhilosopher : Philosopher hypatia`. The vocabulary determines which propositions can be formulated; assumptions provide proofs of some of them.

1.10 Variables and quantifiers

A variable `x : Thing` represents an object that has not been specified. Quantifiers turn expressions containing such variables into statements:

- `∀` reads “for every”;
- `∃` reads “there exists at least one.”

Natural language	Lean
Every person is curious	`∀ x : Thing, Person x → Curious x`
Some person is curious	`∃ x : Thing, Person x ∧ Curious x`
Every philosopher is curious	`∀ x : Thing, Philosopher x → Curious x`
There is a curious philosopher	`∃ x : Thing, Philosopher x ∧ Curious x`

Writing `x : Thing` explicitly makes both the introduced variable and its domain visible.

1.11 Implication and conjunction under quantifiers

Two patterns occur repeatedly:

```
variable (Thing : Type)
variable (Philosopher Curious : Thing → Prop)

#check ∀ x : Thing, Philosopher x → Curious x
#check ∃ x : Thing, Philosopher x ∧ Curious x
```

In the universal formula we take an arbitrary thing and say: *if* it is a philosopher, then it is curious. The implication restricts the assertion to philosophers. In the existential formula we seek one witness that is *both* a philosopher *and* curious, so the properties are joined by a conjunction.

Consider the following examples:

Statement	Lean
Every student reads	`∀ x : Thing, Student x → Reads x`
Some student reads	`∃ x : Thing, Student x ∧ Reads x`
Every student who reads understands	`∀ x : Thing, Student x → Reads x → Understands x`
There is a student who reads and understands	`∃ x : Thing, Student x ∧ Reads x ∧ Understands x`

We can check the same formalizations in Lean:

```
variable (Thing : Type)
variable (Student Reads Understands : Thing → Prop)

#check ∀ x : Thing, Student x → Reads x
#check ∃ x : Thing, Student x ∧ Reads x
#check ∀ x : Thing, Student x → Reads x → Understands x
#check ∃ x : Thing, Student x ∧ Reads x ∧ Understands x
```

`#check` verifies that an expression is a well-formed proposition. It does not prove that proposition.

1.12 Sources and further reading

- Jeremy Avigad, Robert Y. Lewis, and Floris van Doorn, *Logic and Proof*, "Natural Deduction Rules": https://leanprover.github.io/logic_and_proof_lean3/nd_quickref.html.
- Lean developers, *The Lean Language Reference*, "Quantifiers": https://lean-lang.org/doc/reference/latest/Basic-Propositions/Quantifiers/.
- Jeremy Avigad, Robert Y. Lewis, and Floris van Doorn, *Logic and Proof*, "First Order Logic in Lean": https://leanprover.github.io/logic_and_proof_lean3/first_order_logic_in_lean.html.
- Jeremy Avigad, Leonardo de Moura, Soonho Kong, and Sebastian Ullrich, *Theorem Proving in Lean 4*, "Classical Logic": https://lean-lang.org/theorem_proving_in_lean4/Propositions-and-Proofs/.