Lecture 5: Quantifier Rules

Stefano M. Nicoletti

1 Lecture 5: Quantifier Rules

1.1 Aim of the lecture

We review the formalizations from Lecture 4 and introduce the natural-deduction rules for ∀ and ∃. We first complete examples with functions and nested quantifiers. We then study how to construct and use universal and existential proofs, including the side conditions that make the four rules valid. Finally, we combine quantifiers with the propositional connectives.

Four recurring forms provide a useful review:

```
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
```

1.2 Functions inside formulas

Terms built with functions can occur inside predicates, relations, and quantified statements.

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

#check Curious (motherOf hypatia)
#check ∀ x : Thing, Student x → Knows x (motherOf x)
#check ∃ x : Thing, Student x ∧ Reads (motherOf x)
```

The parentheses show that we first build `motherOf x` and then use that term as the argument of a predicate or relation.

1.3 Nested quantifiers

Quantifier order determines which choices may depend on others.

```
variable (Thing : Type)
variable (Person : Thing → Prop)
variable (Knows : Thing → Thing → Prop)

-- Every person knows someone.
#check ∀ x : Thing, Person x →
  ∃ y : Thing, Person y ∧ Knows x y

-- Someone knows every person.
#check ∃ x : Thing, Person x ∧
  ∀ y : Thing, Person y → Knows x y

-- Nobody knows every person.
#check ∃¬ x : Thing, Person x ∧
  ∀ y : Thing, Person y → Knows x y
```

In the first statement, the witness `y` may change when `x` changes. The second statement requires one fixed `x` who knows every relevant `y`.

1.4 Universal introduction, '∀I'

$$\frac{A(x)}{\forall y\, A(y)}\ \forall \mathrm{I}$$

Universal quantifier introduction, ∀I

To prove `∀ y : Thing, A y`, introduce an arbitrary `x`. The goal becomes `A x`. Prove it without using any special property of `x`; the conclusion then holds for every object in the domain.

```
theorem lecture05_forall_intro
    (Thing : Type)
    (Person Curious : Thing → Prop) :
    ∀ y : Thing, Person y → Curious y → Curious y := by
  intro x
  intro hXPerson
  intro hXCurious
  exact hXCurious
```

Only the first `intro` applies `∀I`. The following two introduce the assumptions of the implications.

1.4.1 The arbitrariness condition

The object introduced by `∀I` cannot be a particular object selected because of a special property. From `hHypatiaCurious : Curious hypatia`, for example, we cannot

conclude `∀ y : Thing, Curious y`. After `intro x`, the goal would be `Curious x`, whereas the assumption proves only `Curious hypatia`.

Formally, the variable used in the derivation must not occur free in any open assumption. In `Curious x`, `x` is free. In `∀ y : Thing, Curious y`, `y` is bound. Lean enforces this condition through scope: `intro x` creates a fresh local variable inside the universal goal.

If a source proof deliberately reuses an existing name, Lean may display the older variable as `x†`:

```
x† : Thing
hXCurious : Curious x†
x : Thing
⊢ Curious x
```

The dagger only marks automatic renaming. `Curious x†` and `Curious x` remain different types.

1.5 Universal elimination, ‘∀E‘

$$\frac{\forall x\, A(x)}{A(t)}\ \forall\mathrm{E}$$

Universal quantifier elimination, ∀E

Given `∀ x : Thing, A x`, choose a term `t` and apply the universal proof to it. This yields `A t`.

```
theorem lecture05_forall_elim
    (Thing : Type)
    (Student Reads : Thing → Prop)
    (marta : Thing)
    (hEveryStudentReads : ∀ x : Thing, Student x → Reads x)
    (hMartaStudent : Student marta) :
    Reads marta := by
  have hIfMartaStudentThenReads := hEveryStudentReads marta
  have hMartaReads := hIfMartaStudentThenReads hMartaStudent
  exact hMartaReads
```

Applying the universal assumption to `marta` produces `Student marta → Reads marta`, which can then be applied to `hMartaStudent`.

The chosen term need not be a simple name. It can be built by a function. If `motherOf : Thing → Thing` and `hEverythingCurious : ∀ x : Thing, Curious x`, then `hEverythingCurious (motherOf hypatia)` proves `Curious (motherOf hypatia)`. Lean checks the term’s type and handles bound variables without accidental capture.

1.6 Existential introduction, ‘∃I‘

$$\frac{A(t)}{\exists x\, A(x)}\ \exists\mathrm{I}$$

Existential quantifier introduction, ∃I

To prove `∃ x : Thing, A x`, choose a witness `t` and prove `A t`. Lean’s constructor is `Exists.intro`.

```
theorem lecture05_exists_intro
    (Thing : Type)
    (Reads : Thing → Prop)
    (marta : Thing)
    (hMartaReads : Reads marta) :
    ∃ x : Thing, Reads x := by
  apply Exists.intro marta
  exact hMartaReads
```

Choosing `marta` turns the existential goal into `Reads marta`, which is already available as an assumption.

The witness can also be a compound term. To prove `∃ x : Thing, Curious x`, we may choose `motherOf hypatia` when we have a proof of `Curious (motherOf hypatia)`.

1.7 Existential elimination, ‘∃E‘

$$\frac{\exists x\, A(x) \qquad \begin{array}{c}\overline{A(y)}^{\,1}\\ \vdots\\ B\end{array}}{B}\ \exists\mathrm{E}\,{}^{1}$$

Existential quantifier elimination, ∃E

Given `∃ x : Thing, A x`, we know that a witness exists but may not assume its identity. To derive `B`, introduce an arbitrary witness `y` together with `A y`, and derive a conclusion that does not depend on the identity of `y`.

```
theorem lecture05_exists_elim
    (Thing : Type)
    (FiledReport : Thing → Prop)
    (InvestigationStarts : Prop)
    (hSomeoneFiledReport : ∃ x : Thing, FiledReport x)
    (hReportStartsInvestigation :
      (x : Thing) → FiledReport x → InvestigationStarts) :
    InvestigationStarts := by
  apply Exists.elim hSomeoneFiledReport
```

```
  intro y
  intro hYFiledReport
  have hIfYFiledThenInvestigationStarts :=
    hReportStartsInvestigation y
  have hInvestigationStarts :=
    hIfYFiledThenInvestigationStarts hYFiledReport
  exact hInvestigationStarts
```

The `x` in the second assumption is a bound placeholder. We instantiate it with the local witness `y`, obtaining `FiledReport y → InvestigationStarts`, and then apply that implication to `hYFiledReport`.

The conclusion is fixed before the existential is opened and contains no free occurrence of `y`. Lean enforces this independence through the scope of `y` and the type of `Exists.elim`: the witness cannot escape the local derivation unless it is packaged inside another existential statement.

1.8 Combining the rules

Quantifier rules can be combined in one proof.

```
theorem lecture05_exists_elim_combined
    (Thing : Type)
    (HasUmbrella StaysDry : Thing → Prop)
    (hSomeoneHasUmbrella : ∃ x : Thing, HasUmbrella x)
    (hUmbrellaStaysDry : ∀ x : Thing, HasUmbrella x → StaysDry x) :
    ∃ x : Thing, StaysDry x := by
  apply Exists.elim hSomeoneHasUmbrella
  intro y
  intro hYHasUmbrella
  apply Exists.intro y
  have hIfYHasUmbrellaThenStaysDry := hUmbrellaStaysDry y
  have hYStaysDry := hIfYHasUmbrellaThenStaysDry hYHasUmbrella
  exact hYStaysDry
```

Here `Exists.elim` opens the first existential, applying `hUmbrellaStaysDry` to `y` uses `∀E`, and `Exists.intro y` uses the same object as the witness of the new existential.

1.8.1 Universal quantification and conjunction

Every student who reads takes notes and understands. We introduce an arbitrary student, apply both universal assumptions, and construct the conjunction.

```
theorem lecture05_forall_conjunction_combined
    (Thing : Type)
    (Student Reads TakesNotes Understands : Thing → Prop)
    (hReadersTakeNotes :
      ∀ x : Thing, Student x → Reads x → TakesNotes x)
    (hReadersUnderstand :
      ∀ x : Thing, Student x → Reads x → Understands x) :
    ∀ x : Thing, Student x → Reads x → TakesNotes x ∧ Understands x := by
```

```
  intro x
  intro hXStudent
  intro hXReads
  apply And.intro
  · have hIfXStudentThenReadingImpliesNotes := hReadersTakeNotes x
    have hIfXReadsThenTakesNotes :=
      hIfXStudentThenReadingImpliesNotes hXStudent
    have hXTakesNotes := hIfXReadsThenTakesNotes hXReads
    exact hXTakesNotes
  · have hIfXStudentThenReadingImpliesUnderstanding := hReadersUnderstand x
    have hIfXReadsThenUnderstands :=
      hIfXStudentThenReadingImpliesUnderstanding hXStudent
    have hXUnderstands := hIfXReadsThenUnderstands hXReads
    exact hXUnderstands
```

1.8.2 Existential quantification and disjunction

Someone visits Rome or Athens. We open the existential and consider both cases of the disjunction. Each case supplies a witness who sees a museum.

```
theorem lecture05_exists_or_combined
    (Thing : Type)
    (VisitsRome VisitsAthens SeesMuseum : Thing → Prop)
    (hSomeoneVisitsACity : ∃ x : Thing, VisitsRome x ∨ VisitsAthens x)
    (hRomeVisitorsSeeMuseum : ∀ x : Thing, VisitsRome x → SeesMuseum x)
    (hAthensVisitorsSeeMuseum : ∀ x : Thing, VisitsAthens x → SeesMuseum x) :
    ∃ x : Thing, SeesMuseum x := by
  apply Exists.elim hSomeoneVisitsACity
  intro y
  intro hYVisitsRomeOrAthens
  cases hYVisitsRomeOrAthens with
  | inl hYVisitsRome =>
      apply Exists.intro y
      have hIfYVisitsRomeThenSeesMuseum := hRomeVisitorsSeeMuseum y
      have hYSeesMuseum := hIfYVisitsRomeThenSeesMuseum hYVisitsRome
      exact hYSeesMuseum
  | inr hYVisitsAthens =>
      apply Exists.intro y
      have hIfYVisitsAthensThenSeesMuseum := hAthensVisitorsSeeMuseum y
      have hYSeesMuseum := hIfYVisitsAthensThenSeesMuseum hYVisitsAthens
      exact hYSeesMuseum
```

1.8.3 Nested quantifiers

The existential witness may depend on the person introduced by the universal quantifier. We open the existential only after fixing that person.

```
theorem lecture05_nested_quantifiers_combined
    (Thing : Type)
    (Person : Thing → Prop)
    (Knows : Thing → Thing → Prop)
```

```
    (hEveryPersonKnowsSomeone :
      ∀ x : Thing, Person x →
        ∃ y : Thing, Person y ∧ Knows x y) :
    ∀ x : Thing, Person x →
      ∃ y : Thing, Knows x y ∧ Person y := by
  intro x
  intro hXPerson
  have hIfXPersonThenKnowsSomeone := hEveryPersonKnowsSomeone x
  have hXKnowsSomeone := hIfXPersonThenKnowsSomeone hXPerson
  apply Exists.elim hXKnowsSomeone
  intro y
  intro hYPersonAndXKnowsY
  apply Exists.intro y
  apply And.intro
  · exact hYPersonAndXKnowsY.right
  · exact hYPersonAndXKnowsY.left
```

1.8.4 Existential quantification and negation

A witness submitted but received no confirmation. The universal rule produces a confirmation for that witness, and the negation produces `False`.

```
theorem lecture05_exists_negation_combined
    (Thing : Type)
    (Submitted ReceivedConfirmation : Thing → Prop)
    (hSubmissionGetsConfirmation :
      ∀ x : Thing, Submitted x → ReceivedConfirmation x)
    (hCounterexample :
      ∃ x : Thing, Submitted x ∧ ¬ReceivedConfirmation x) :
    False := by
  apply Exists.elim hCounterexample
  intro y
  intro hYSubmittedAndNoConfirmation
  have hYSubmitted := hYSubmittedAndNoConfirmation.left
  have hYNoConfirmation := hYSubmittedAndNoConfirmation.right
  have hIfYSubmittedThenConfirmation := hSubmissionGetsConfirmation y
  have hYReceivedConfirmation := hIfYSubmittedThenConfirmation hYSubmitted
  have hContradiction := hYNoConfirmation hYReceivedConfirmation
  exact hContradiction
```

1.8.5 Existential quantification and biconditional

We open the existential, instantiate the biconditional at its witness, and use `Iff.mp` to extract the required direction. The same object witnesses the goal.

```
theorem lecture05_exists_iff_combined
    (Thing : Type)
    (Square HasFourEqualSides : Thing → Prop)
    (hEquivalence :
      ∀ x : Thing, Square x ↔ HasFourEqualSides x)
    (hSquareExists : ∃ x : Thing, Square x) :
```

```
    ∃ x : Thing, HasFourEqualSides x := by
  apply Exists.elim hSquareExists
  intro y
  intro hYSquare
  apply Exists.intro y
  have hEquivalenceForY := hEquivalence y
  have hForwardDirection := Iff.mp hEquivalenceForY
  have hYHasFourEqualSides := hForwardDirection hYSquare
  exact hYHasFourEqualSides
```

In each example, we first handle the outer constructor of the goal, make local witnesses explicit, and then apply the propositional rules.

1.9 Working strategy

When reading a quantified formula, ask:

- Does the goal begin with ∀? Introduce an arbitrary object with `intro`.
- Is there a universal assumption? Apply it to a term from the domain.
- Does the goal begin with ∃? Choose a witness with `Exists.intro`.
- Is there an existential assumption? Open it with `Exists.elim` without presupposing the witness's identity.
- Are several quantifiers nested? Check their order and dependencies.

This strategy extends the one used for propositional connectives. Inspect the main constructor of the goal for an introduction rule and compound assumptions for useful elimination rules.

1.10 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, Robert Y. Lewis, and Floris van Doorn, *Logic and Proof*, "Natural Deduction for First Order Logic": https://leanprover.github.io/logic_and_proof_lean3/natural_deduction_for_first_order_logic.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/.