# Lezione 2: Proposizioni, dimostrazioni e regole logiche

Stefano M. Nicoletti

## 1 Lezione 2: Proposizioni, dimostrazioni e regole logiche

Registrazione video della Lezione 2

### 1.1 Scopo della lezione

Questa lezione riprende il ruolo del kernel e del typechecker, poi introduce il modo in cui Lean legge proposizioni e dimostrazioni. Il punto centrale è la corrispondenza tra proposizioni e tipi: una proposizione `P : Prop` è un tipo, e una dimostrazione di `P` è un termine che abita quel tipo (Fonte: Theorem Proving in Lean 4, Propositions and Proofs).

Valuteremo esempi minimali per osservare, nel proof state, le regole di introduzione ed eliminazione dei connettivi logici e di alcuni costrutti specifici:

- implicazione `P → Q`;
- congiunzione `P ∧ Q`;
- disgiunzione `P ∨ Q`;
- negazione `¬P`;
- `False`.

### 1.2 Kernel e typechecker

Abbiamo visto che Lean permette di costruire dimostrazioni con tattiche, supporto di librerie, e tramite interfacce interattive. Questi strumenti aiutano l'utente, ma in ultima analisi, il kernel controlla l'oggetto formale prodotto e verifica che abbia il tipo richiesto (Fonte: de Moura and Ullrich, Lean 4).

Il typechecker controlla se un termine ha un certo tipo. Nelle dimostrazioni, il tipo atteso è la proposizione che vogliamo dimostrare. Un teorema viene accettato quando Lean riesce a controllare il termine di dimostrazione rispetto all'enunciato. Per questo, dal punto di vista del sistema, dimostrare significa costruire un termine ben tipato.

Questa prospettiva spiega perché il proof state è così utile. Il goal mostra il tipo che dobbiamo abitare. Le assunzioni mostrano i termini già disponibili nel contesto. Una tattica è un modo interattivo per trasformare questa situazione fino a chiudere tutti i goal.

## 1.3 Proposizioni come tipi

In Lean, `P : Prop` dice che `P` è una proposizione. Se nel contesto compare `hP : P`, allora `hP` è una dimostrazione di `P`. Non è solo un'etichetta informale: è un oggetto formale che può essere usato per costruire altre dimostrazioni.

L'implicazione è l'esempio più diretto. Un termine di tipo `P → Q` si comporta come una funzione: prende una dimostrazione di `P` e restituisce una dimostrazione di `Q`. Se abbiamo

```
hPQ : P → Q
hP : P
```

allora `hPQ hP` è una dimostrazione di `Q`.

## 1.4 Regole di introduzione ed eliminazione

Per ogni connettivo logico distinguiamo due domande.

- La regola di introduzione spiega come costruire una dimostrazione della proposizione composta.
- La regola di eliminazione spiega come usare una dimostrazione della proposizione composta per ottenerne delle componenti.

Guardare il goal significa chiedersi quale regola di introduzione può costruirlo. Guardare le assunzioni significa chiedersi quale regola di eliminazione può usarle. Questa terminologia segue la presentazione standard delle regole di deduzione naturale (Fonte: Logic and Proof, Natural Deduction Rules).

## 1.5 Implicazione

Per introdurre `P → Q`, assumiamo temporaneamente `P` e costruiamo `Q`.

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

La riga `intro hP` introduce l'assunzione temporanea `hP : P`. In questo esempio la conclusione `Q` è già disponibile come `hQ`, quindi possiamo chiudere il goal con `exact hQ`.

Per eliminare `P → Q`, applichiamo l'implicazione a una dimostrazione di `P`.

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

Qui `hPQ hP` è applicazione di funzione: da `P → Q` e `P` otteniamo `Q`. Siamo di nuovo al modus ponens!

## 1.6 Congiunzione

Una congiunzione `P ∧ Q` contiene entrambe le informazioni. Per introdurla, dobbiamo fornire una dimostrazione di entrambe le componenti (qui sotto le abbiamo inserite come assunzioni nell'enunciato).

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

La stessa regola `And.intro` può essere usata come tattica. Il goal `P ∧ Q` si divide in due subgoal, uno per `P` e uno per `Q`.

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

Per eliminare una congiunzione usiamo le sue componenti, e preleviamo il congiunto sulla sinistra/destra.

```
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 Disgiunzione

Una disgiunzione `P ∨ Q` contiene una delle due informazioni. Per introdurla, basta fornire uno dei due lati (con `Or.inl/inr`: injection left/right).

```
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
```

Per eliminare una disgiunzione dobbiamo ragionare per casi. Se abbiamo `P ∨ Q`, non sappiamo in generale quale lato sia disponibile (in altri termini, non sappiamo se la disgiunzione sia vera per via di P o di Q). Per ottenere una conclusione dobbiamo produrla in entrambi i rami: se riusciamo a concludere la stessa proposizione sia nel caso in cui sia `P` a essere vero, sia nel caso in cui lo sia `Q`, allora possiamo eliminare la disgiunzione. In questo caso, vogliamo concludere che `Q ∨ P` da `P ∨ Q`: dobbiamo quindi ragionare valutando i due casi, P è vero oppure Q è vero, e per entrambi dimostrare che riusciamo a concludere `Q ∨ P`.

```
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
```

La stessa struttura si può scrivere con `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 e negazione

`False` rappresenta uno stato contraddittorio. Se abbiamo una dimostrazione di `False`, possiamo chiudere qualunque goal.

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

In Lean la negazione è definita come implicazione verso `False`: `¬P` significa `P → False`. Per eliminare una negazione, la applichiamo a una dimostrazione positiva di `P`.

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

Per introdurre `¬P`, assumiamo temporaneamente `P` e deriviamo `False`. Il seguente esempio è lo schema di ragionamento del modus tollens. `intro hP` svolge la seguente funzione: per

dimostrare ¬P, assumiamo P e deriviamo `False` (una contraddizione). Osservate come cambia il goal:

```
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 Tattiche, comandi e scorciatoie

In questa lezione abbiamo usato soprattutto tattiche che corrispondono alle regole di introduzione ed eliminazione dei connettivi.

- `intro`: introduce un'assunzione temporanea quando il goal è un'implicazione o una negazione;
- `exact`: chiude il goal fornendo un termine del tipo richiesto;
- `have`: introduce un risultato intermedio nel contesto;
- `apply`: applica una regola, un teorema o un costruttore al goal corrente;
- `cases`: distingue i casi di una disgiunzione.

Abbiamo inoltre usato alcuni costruttori ed eliminatori espliciti:

- `And.intro`, `.left`, `.right` per la congiunzione;
- `Or.inl`, `Or.inr`, `Or.elim` per la disgiunzione;
- `False.elim` per usare una contraddizione.

## 1.10 Esercizi e soluzioni

Trovate i file con esercizi e soluzioni (`Exercises.lean` e `Solutions.lean`) sul sito di riferimento.

## 1.11 Fonti e letture

- Jeremy Avigad, Leonardo de Moura, Soonho Kong, Sebastian Ullrich, *Theorem Proving in Lean 4*, capitolo "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*, appendice "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.