

Lezione 7: Induzione, semplificazione e verifica del software

Stefano M. Nicoletti

1 Lezione 7: Induzione, semplificazione e verifica del software

[Registrazione video della Lezione 7](#)

1.1 Scopo della lezione

Riprendiamo la prima dimostrazione per induzione della Lezione 6. Dimostriamo altre proprietà della somma, introduciamo `simp` e combiniamo induzione, riscrittura e semplificazione. Appliciamo poi gli stessi strumenti alla verifica di un algoritmo ricorsivo sulle liste.

1.2 Definizioni e risultati disponibili

Continuiamo a usare i naturali, la somma e il doppio definiti nella Lezione 6:

Naturale ha i costruttori `zero` e `succ`. La funzione `somma` (`m n : Naturale`) ricorre sul secondo argomento, mentre `doppio` (`n : Naturale`) ricorre sull'unico argomento.

Ricordare quale argomento viene analizzato non è un dettaglio puramente sintattico. La definizione di `somma` può scegliere un ramo del `match` quando conosce la forma del secondo argomento. Per questo `somma m zero` e `somma m (succ n)` si riducono direttamente, mentre `somma zero n` richiede un ragionamento per induzione quando `n` è ancora una variabile.

Sono già disponibili le equazioni computazionali e il risultato conclusivo della Lezione 6:

```
open Naturale
```

```
theorem lecture07_somma_zero_destra (m : Naturale) :  
  somma m zero = m := by  
  rfl
```

```
open Naturale
```

```
theorem lecture07_somma_succ_destra (m n : Naturale) :
```

```
somma m (succ n) = succ (somma m n) := by
rfl
```

open Naturale

```
theorem lecture07_doppio_succ (n : Naturale) :
  doppio (succ n) = succ (succ (doppio n)) := by
rfl
```

open Naturale

```
theorem lecture07_zero_somma (n : Naturale) :
  somma zero n = n := by
induction n with
| zero =>
  rfl
| succ n ih =>
  unfold somma
  rewrite [ih]
  rfl
```

I primi tre risultati descrivono direttamente il comportamento delle definizioni. L'ultimo, invece, è un teorema generale ottenuto per induzione. Questa distinzione sarà utile durante tutta la lezione: alcune trasformazioni sono computazioni, altre richiedono risultati già dimostrati.

1.3 Una seconda dimostrazione per induzione

Vogliamo ora dimostrare che un successore nel primo argomento può essere portato davanti alla somma:

$\text{somma} (\text{succ } m) n = \text{succ} (\text{somma } m n).$

La forma dell'enunciato può suggerire di fare induzione su m , perché il successore compare accanto a quella variabile. Dobbiamo però guardare la definizione della funzione. `somma` analizza il secondo argomento, e in entrambe le somme quel posto è occupato da n . È quindi n a impedire alla definizione di scegliere tra il caso `zero` e il caso `succ`. Facciamo induzione su n .

open Naturale

```
theorem lecture07_succ_somma (m n : Naturale) :
  somma (succ m) n = succ (somma m n) := by
induction n with
| zero =>
  rfl
| succ n ih =>
  unfold somma
  rewrite [ih]
  rfl
```

Nel caso base sostituiamo n con zero. Entrambi i lati si riducono alla stessa espressione e `rfl` conclude. Nel passo induttivo consideriamo `succ n`. `unfold somma` apre entrambe le somme e il goal contiene la stessa uguaglianza richiesta per n . L'ipotesi induttiva `ih` sostituisce quella sottoespressione, dopo di che i due lati coincidono.

La scelta della variabile d'induzione segue dunque la struttura della computazione. Cerchiamo la variabile che occupa l'argomento analizzato dalla funzione e osserviamo se, nel caso successore, l'assunzione induttiva potrà riscrivere la chiamata ricorsiva ottenuta.

1.4 Associatività, commutatività e 'simp'

L'associatività afferma che il modo di raggruppare tre addendi non modifica il risultato. Nel nostro enunciato la somma esterna a sinistra ha c come secondo argomento. A destra, c compare come secondo argomento di `somma b c`, che a sua volta determina la computazione della somma esterna. Scegliamo quindi c come variabile d'induzione.

`open Naturale`

```
theorem lecture07_somma_assoc (a b c : Naturale) :
  somma (somma a b) c = somma a (somma b c) := by
  induction c with
  | zero =>
    rfl
  | succ c ih =>
    rw [lecture07_somma_succ_destra]
    rw [lecture07_somma_succ_destra]
    rw [lecture07_somma_succ_destra]
    rw [ih]
```

Nel caso `zero`, le somme che hanno zero come secondo argomento si riducono e `rfl` conclude. Nel caso `succ c` eseguiamo tre riscritture con `lecture07_somma_succ_destra`. La prima apre la somma esterna a sinistra. La seconda apre `somma b (succ c)` a destra. La terza apre la nuova somma esterna prodotta sul lato destro. A quel punto resta esattamente l'uguaglianza fornita dall'ipotesi induttiva, racchiusa sotto `succ`, e `rw [ih]` conclude.

La commutatività afferma invece che scambiare i due addendi non modifica il risultato. Non segue soltanto dalla computazione, perché `somma` non tratta simmetricamente i due argomenti: analizza il secondo e lascia invariato il primo. Facciamo induzione su b . Nel caso base il lato sinistro si riduce per definizione, mentre sul lato destro usiamo `lecture07_zero_somma`. Nel passo successore, `lecture07_somma_succ_destra` tratta il successore a destra e `lecture07_succ_somma` tratta quello che compare a sinistra dopo lo scambio.

`open Naturale`

```
theorem lecture07_somma_comm (a b : Naturale) :
  somma a b = somma b a := by
  induction b with
  | zero =>
```

```

      rw [lecture07_zero_somma]
      rfl
| succ b ih =>
  rw [lecture07_somma_succ_destra]
  rw [lecture07_succ_somma]
  rw [ih]

```

Questa dimostrazione mostra perché costruiamo una raccolta di lemmi piccoli. La definizione fornisce direttamente le equazioni sul secondo argomento. I teoremi `zero_somma` e `succ_somma` forniscono le forme corrispondenti sul primo argomento. La commutatività può quindi combinare i due punti di vista.

Quando una dimostrazione richiede molte riscritture elementari, possiamo usare il semplificatore. `simp` applica equazioni orientate verso forme più semplici e svolge anche computazioni che Lean riconosce direttamente. Non cerca una dimostrazione in modo arbitrario: usa un insieme di regole chiamato *simp-set*.

Confrontiamo tre modi di usare lo stesso risultato. `rw` richiede una precisa riscrittura scelta da noi. `simp only [lemmi]` usa soltanto i lemmi elencati, oltre alle riduzioni essenziali del semplificatore. `simp [lemma]` usa le regole standard e aggiunge lemma per quel comando.

open Naturale

```

theorem lecture07_zero_somma_con_rw (n : Naturale) : somma zero n = n := by
  rw [lecture07_zero_somma]

```

Nel primo teorema indichiamo esattamente la riscrittura. Nel secondo chiediamo al semplificatore di lavorare con una sola regola dichiarata. Nel terzo aggiungiamo il lemma al `simp-set` già disponibile per quella singola chiamata. Il risultato è lo stesso, ma cambia il grado di controllo e di automazione.

open Naturale

```

theorem lecture07_zero_somma_con_simp_only (n : Naturale) : somma zero n = n
  := by
  simp only [lecture07_zero_somma]

```

open Naturale

```

theorem lecture07_zero_somma_con_simp (n : Naturale) : somma zero n = n := by
  simp [lecture07_zero_somma]

```

open Naturale

section SemplificazioneLocale

attribute [local simp] lecture07_zero_somma lecture07_succ_somma

```

theorem lecture07_succ_somma_con_simp (m n : Naturale) :

```

```
somma (succ m) n = succ (somma m n) := by
simp
```

end SemplificazioneLocale

L'attributo `local simp` vale soltanto all'interno della sezione. In questo modo possiamo sperimentare con un insieme di regole senza modificare tutte le dimostrazioni successive del modulo. Nel teorema interno alla sezione basta scrivere `simp`: il semplificatore trova automaticamente `lecture07_succ_somma` nel `simp-set` locale.

open Naturale

```
@[simp] theorem lecture07_doppio_zero :
  doppio zero = zero := by
rfl
```

`@[simp]` registra una regola per il resto del modulo. Lo usiamo per una regola canonica come `doppio zero = zero`. Associatività e commutatività non sono regole adatte al `simp-set`: cambiano soltanto il raggruppamento o l'ordine degli argomenti.

La scelta delle regole è importante. Una buona regola di semplificazione porta in modo prevedibile verso una forma più breve o canonica. Orientare automaticamente l'associatività richiederebbe scegliere sempre un particolare raggruppamento. La commutatività potrebbe invece continuare a scambiare gli argomenti senza produrre una forma evidentemente più semplice. Per questi risultati preferiamo una riscrittura esplicita nel punto in cui serve.

1.5 Una dimostrazione conclusiva

Combiniamo induzione e riscritture per dimostrare che il doppio di una somma è la somma dei doppi. La sottoespressione che resta bloccata è `somma m n`, perciò facciamo induzione su `n`.

open Naturale

```
theorem lecture07_doppio_somma (m n : Naturale) :
  doppio (somma m n) = somma (doppio m) (doppio n) := by
induction n with
| zero =>
  rfl
| succ n ih =>
  rw [lecture07_somma_succ_destra]
  rw [lecture07_doppio_succ]
  rw [lecture07_doppio_succ]
  rw [lecture07_somma_succ_destra]
  rw [lecture07_somma_succ_destra]
  rw [ih]
```

Il caso base segue dalle definizioni di `somma` e `doppio`. Nel passo successore, la prima riscrittura porta il successore fuori da `somma m (succ n)`. Le due riscritture con

lecture07_doppio_succ descrivono il doppio di un successore sui due lati. Le due successive riscritture della somma espongono la stessa struttura di successori. Infine `ih` sostituisce il doppio della somma più piccola con la somma dei doppi.

Dopo avere verificato ogni passaggio, possiamo esprimere la stessa dimostrazione con un insieme controllato di regole di semplificazione:

```
open Naturale
```

```
theorem lecture07_doppio_somma_breve (m n : Naturale) :  
  doppio (somma m n) = somma (doppio m) (doppio n) := by  
  induction n with  
  | zero =>  
    rfl  
  | succ n ih =>  
    simp only [lecture07_somma_succ_destra, lecture07_doppio_succ, ih]
```

La versione abbreviata non esprime un argomento matematico diverso. Usa le stesse equazioni, ma lascia che `simp only` scelga dove applicarle. Manteniamo la lista esplicita perché, in questa fase, vogliamo sapere quali risultati rendono valida la dimostrazione.

1.6 Dall'induzione alla verifica del software

Un programma riceve un input e produce un output. Verificarlo significa formulare con precisione una proprietà attesa e dimostrarla per ogni input. Per applicare questa idea a una nuova struttura dati, iniziamo definendo induttivamente il tipo delle liste.

```
inductive Lista  $\alpha$  ( : Type) where  
  | vuota  
  | cons (testa :  $\alpha$ ) (coda : Lista  $\alpha$ )
```

Una lista è quindi vuota oppure è costruita con un elemento iniziale e una lista più piccola. Nella prossima lezione useremo questa struttura per definire un algoritmo e verificarne alcune proprietà.

1.7 Fonti

- Jeremy Avigad, Leonardo de Moura, Soonho Kong e Sebastian Ullrich, *Theorem Proving in Lean 4*, «Inductive Types»: https://lean-lang.org/theorem_proving_in_lean4/Inductive-Types/.
- Jeremy Avigad, Leonardo de Moura, Soonho Kong e Sebastian Ullrich, *Theorem Proving in Lean 4*, «Induction and Recursion»: https://lean-lang.org/theorem_proving_in_lean4/Induction-and-Recursion/.
- Jeremy Avigad, Leonardo de Moura, Soonho Kong e Sebastian Ullrich, *Theorem Proving in Lean 4*, «Tactics»: https://lean-lang.org/theorem_proving_in_lean4/Tactics/.

- David Thrane Christiansen, *Functional Programming in Lean*, «Datatypes and Patterns»: https://lean-lang.org/functional_programming_in_lean/Getting-to-Know-Lean/Datatypes-and-Patterns/.
- Lean project, *The Lean Language Reference*, «Recursive Definitions»: <https://lean-lang.org/doc/reference/latest/Definitions/Recursive-Definitions/>.
- Kevin Buzzard e Jon Eugster, *Natural Number Game*: <https://adam.math.hhu.de/>.
- Vrije Universiteit Amsterdam, *Logical Verification*: <https://research.vu.nl/en/courses/logical-verification-5/>.
- Marieke Huisman e Anton J. Wijs, *Concise Guide to Software Verification: From Model Checking to Annotation Checking*: <https://research.utwente.nl/en/publications/concise-guide-to-software-verification-from-model-checking-to-ann/>.
- Hana Chockler, Joseph Y. Halpern e Orna Kupferman, *What Causes a System to Satisfy a Specification?*: <https://arxiv.org/abs/cs/0312036>.