

Lezione 6: Tipi induttivi, ricorsione e induzione

Stefano M. Nicoletti

1 Lezione 6: Tipi induttivi, ricorsione e induzione

[Registrazione video della Lezione 6](#)

1.1 Scopo della lezione

In questa lezione passiamo da domini dei quali non conosciamo la struttura a tipi dei quali specifichiamo tutti i possibili valori. Impariamo a costruire questi valori, a definire funzioni che ne seguono la struttura e a dimostrare proprietà che valgono per tutti gli elementi del tipo.

Questo passaggio introduce tre idee collegate. Un *tipo induttivo* stabilisce come sono costruiti i suoi valori. Una *definizione ricorsiva* calcola seguendo quella costruzione. Una *dimostrazione per induzione* dimostra una proprietà seguendo gli stessi casi. Vediamo inoltre come `rfl`, `unfold` e `rw` permettono di riconoscere costrutti uguali tra loro, aprire una definizione e riscrivere mediante un'uguaglianza.

1.2 Tipi induttivi

Nelle lezioni precedenti `Cosa : Type` lasciava aperta la struttura del dominio. Potevamo dichiarare oggetti, predicati e funzioni, ma Lean non sapeva quali oggetti appartenessero a `Cosa` né come fossero costruiti.

Un tipo induttivo fornisce questa informazione elencando i modi in cui possiamo costruire i suoi valori. Questi modi si chiamano *costruttori*. La dichiarazione seguente introduce un nuovo tipo e, nello stesso momento, stabilisce che un suo valore può essere costruito in tre modi.

```
inductive Semaforo where
  | rosso
  | giallo
  | verde
```

`Semaforo.rosso`, `Semaforo.giallo` e `Semaforo.verde` sono valori di tipo `Semaforo`. In altri termini: sono i costruttori che producono i valori del nuovo tipo. Poiché la dichiarazione non contiene altri costruttori, questi tre casi descrivono tutti i valori possibili di `Semaforo`.

Conoscere tutti i casi ci permette di definire operazioni che si comportano diversamente su ciascuno di essi. Definiamo un predicato mediante pattern matching:

```
def SiDeveFermare : Semaforo → Prop
| .rosso => True
| .giallo => True
| .verde => False
```

Il pattern matching specifica il risultato per ciascun costruttore. `SiDeveFermare .rosso` e `SiDeveFermare .giallo` si riducono a `True`, mentre `SiDeveFermare .verde` si riduce a `False`. La definizione copre tutti i costruttori, quindi assegna un risultato a ogni possibile semaforo.

La stessa conoscenza dei costruttori può essere usata nelle dimostrazioni. Se riceviamo un semaforo arbitrario `s`, non sappiamo inizialmente quale colore abbia. Sappiamo però che deve essere uno dei tre valori dichiarati. La tattica `cases s` sostituisce il problema originale con un problema separato per ciascun costruttore.

```
theorem lecture06_semaforo_ha_un_colore (s : Semaforo) :
  s = .rosso v s = .giallo v s = .verde := by
cases s with
| rosso =>
  apply Or.inl
  rfl
| giallo =>
  apply Or.inr
  apply Or.inl
  rfl
| verde =>
  apply Or.inr
  apply Or.inr
  rfl
```

Nel caso rosso, il goal è una disgiunzione la cui prima alternativa è vera. `Or.inl` sceglie quella alternativa e `rfl` dimostra `.rosso = .rosso`. Nel caso giallo scegliamo prima il ramo destro della disgiunzione esterna e poi il ramo sinistro di quella interna. Nel caso verde scegliamo due volte il ramo destro.

Qui `cases` è sufficiente perché dobbiamo soltanto esaminare un numero finito di possibilità. Non abbiamo bisogno di mettere in relazione un caso con uno più piccolo.

1.3 I nostri numeri naturali

Consideriamo ora un tipo con una struttura diversa. I colori del semaforo sono tre casi indipendenti. I numeri naturali, invece, sono costruiti progressivamente: partiamo da zero e otteniamo un nuovo naturale applicando il successore a uno già costruito.

Per rendere visibile questa struttura definiamo un nostro tipo di naturali, senza usare `Nat`:

```
inductive Naturale where
| zero
| succ (n : Naturale)
```

```
open Naturale
```

```
def uno : Naturale := succ zero
```

```
open Naturale
```

```
def due : Naturale := succ uno
```

```
open Naturale
```

```
def tre : Naturale := succ due
```

Il costruttore `zero` non richiede argomenti e produce direttamente un naturale. Il costruttore `succ` richiede invece un valore `n : Naturale` e produce il naturale successivo. La definizione è ricorsiva perché il tipo `Naturale` compare come argomento di uno dei propri costruttori.

Le definizioni di `uno`, `due` e `tre` sono abbreviazioni leggibili. Se le aprissimo, troveremmo rispettivamente `succ zero`, `succ (succ zero)` e `succ (succ (succ zero))`. Ogni valore finito di `Naturale` è quindi una catena di applicazioni di `succ` che termina con `zero`.

La dichiarazione afferma anche che non esistono altri modi di costruire un `Naturale`. Questa informazione sarà decisiva quando vorremo dimostrare una proprietà per tutti i naturali: ci basterà seguire i due costruttori `zero` e `succ`.

1.4 Ricorsione

Una definizione ricorsiva risolve un problema usando il risultato dello stesso problema su un dato più semplice. Per i nostri naturali, il dato più semplice contenuto in `succ n` è `n`: abbiamo rimosso un'applicazione di `succ` e ci siamo avvicinati a `zero`.

Definire una funzione ricorsiva su `Naturale` significa quindi spiegare che cosa restituisce nel caso `zero` e come usa il risultato già calcolato su `n` nel caso `succ n`. Iniziamo dalla somma. La funzione riceve due naturali, ma la sua definizione analizza il secondo argomento.

```
open Naturale
```

```
def somma (m n : Naturale) : Naturale :=  
  match n with  
  | zero => m  
  | succ n => succ (somma m n)
```

La prima riga dice che aggiungere `zero` a `m` restituisce `m`. La seconda dice che sommare `succ n` a `m` equivale ad applicare `succ` al risultato di `somma m n`. Per esempio, il calcolo di `somma uno due` segue questi passaggi:

```
somma uno (succ uno)
succ (somma uno uno)
succ (succ (somma uno zero))
succ (succ uno)
```

Ogni passaggio con un secondo argomento della forma `succ n` accorcia la catena e rimanda alla somma con `n`. Quando il secondo argomento diventa zero, raggiungiamo il caso base.

```
open Naturale

def doppio (n : Naturale) : Naturale :=
  match n with
  | zero => zero
  | succ n => succ (succ (doppio n))
```

La definizione di `doppio` segue la stessa struttura. Il doppio di zero è zero. Nel caso `succ n`, usiamo il doppio di `n` e aggiungiamo due applicazioni di `succ`. Anche qui la chiamata ricorsiva elimina un `succ`, quindi continuando raggiungeremo il caso zero.

`match n with` rende esplicito quale argomento guida la computazione. In somma, se conosciamo la forma del secondo argomento, Lean può scegliere una riga della definizione. Se quel posto è occupato da una variabile della quale non conosciamo la forma, la computazione resta in attesa.

1.5 Computazione, ‘`rfl`’ e ‘`unfold`’

Due espressioni sono *definizionalmente uguali* quando Lean può calcolarle fino a ottenere la stessa espressione. Non occorre applicare un teorema già dimostrato: è sufficiente aprire le definizioni e seguire le loro regole di computazione.

La tattica `rfl` dimostra un’uguaglianza quando, dopo questa computazione, i due lati coincidono. Nel primo esempio Lean apre `uno`, `due` e `somma`, quindi riconosce che entrambi i lati descrivono lo stesso valore.

```
open Naturale

example : somma uno uno = due := by
  rfl
```

```
open Naturale

theorem lecture06_somma_zero_destra (m : Naturale) :
  somma m zero = m := by
  rfl
```

```
open Naturale

theorem lecture06_somma_succ_destra (m n : Naturale) :
  somma m (succ n) = succ (somma m n) := by
```

```
rfl
```

```
open Naturale
```

```
theorem lecture06_doppio_succ (n : Naturale) :  
  doppio (succ n) = succ (succ (doppio n)) := by  
  rfl
```

Vediamo i tre teoremi successivi: in `somma m zero` e `somma m (succ n)`, il secondo argomento mostra già quale ramo del match scegliere. Analogamente, `doppio (succ n)` mostra subito il ramo successore.

È importante distinguere queste uguaglianze da risultati come `somma zero n = n`. In quest'ultimo, il secondo argomento è la variabile `n`. Lean non può scegliere un ramo finché non conosce se `n` sia zero oppure `succ k`.

`unfold` rende esplicitamente visibile una definizione nel goal. È utile quando vogliamo osservare i passaggi che `rfl` può normalmente eseguire senza mostrarli:

```
open Naturale
```

```
example : doppio uno = due := by  
  unfold doppio  
  unfold uno due  
  rfl
```

Dopo `unfold doppio`, il lato sinistro espone il pattern matching usato dalla funzione. Aprendo anche `uno` e `due`, Lean può scegliere il ramo `succ` e ridurre entrambi i lati alla stessa catena di costruttori. `unfold` rende dunque visibile una definizione già disponibile, espandendola.

1.6 Riscrivere con 'rw'

Non tutte le uguaglianze utili derivano dalla computazione. Talvolta abbiamo un'uguaglianza come assunzione o come teorema già dimostrato e vogliamo usarla per trasformare il goal.

Se abbiamo `h : a = b`, `rw [h]` cerca occorrenze di `a` nel goal e le sostituisce con `b`. `rw [← h]` usa la stessa uguaglianza nella direzione opposta, sostituendo `b` con `a`. La freccia cambia soltanto il verso nel quale la usiamo.

```
open Naturale
```

```
theorem lecture06_rw_avanti  
  (a b : Naturale)  
  (h : a = b) :  
  somma a zero = b := by  
  rw [h]  
  rfl
```

Prima della riscrittura, il primo goal è $\text{somma } a \text{ zero} = b$. Con `rw [h]`, l'occorrenza di a viene sostituita da b e il goal diventa $\text{somma } b \text{ zero} = b$. La definizione di somma riduce il lato sinistro a b , quindi `rfl` conclude la dimostrazione.

Nel verso opposto, il goal contiene b a sinistra e a a destra. Usando `rw [← h]`, Lean sostituisce b con a e ottiene $\text{somma } a \text{ zero} = a$.

open Naturale

```
theorem lecture06_rw_indietro
  (a b : Naturale)
  (h : a = b) :
  somma b zero = a := by
  rw [← h]
  rfl
```

La direzione va scelta osservando quale sostituzione porta il goal verso una forma più semplice o già nota.

1.7 Induzione

$\text{somma } m \text{ zero} = m$ si riduce subito perché il secondo argomento è esplicitamente zero. Lean sceglie il primo ramo della definizione di somma , e i due lati diventano identici.

Nell'enunciato $\text{somma } \text{zero } n = n$, invece, il secondo argomento è la variabile n . Il teorema non ci assegna un valore concreto per n : deve valere qualunque naturale venga scelto. Lean non può quindi ridurre direttamente $\text{somma } \text{zero } n$, perché non sa ancora se usare il ramo zero oppure il ramo succ .

Controllare zero, uno, due e altri esempi può suggerire che l'enunciato sia corretto, ma non dimostra che valga per ogni catena finita di successori. Abbiamo bisogno di un argomento che copra tutti i valori generati dai costruttori di **Naturale**.

Una dimostrazione per induzione segue precisamente questi costruttori. Nel *caso base* dimostriamo la proprietà per zero. Nel *passo induttivo* consideriamo un naturale della forma $\text{succ } n$: assumiamo che la proprietà valga per n e dimostriamo che vale anche per $\text{succ } n$. L'assunzione ricevuta nel passo si chiama *ipotesi induttiva*.

Questi due casi sono sufficienti perché ogni naturale è ottenuto partendo da zero e applicando succ un numero finito di volte. L'induzione dunque fornisce il caso iniziale e un modo per trasferire il risultato da ciascun naturale al successivo.

Nel proof script, `induction n with` genera un ramo per ciascun costruttore. Il ramo zero deve dimostrare $\text{somma } \text{zero } \text{zero} = \text{zero}$, che segue direttamente dalla definizione. Il ramo $\text{succ } n \text{ ih}$ introduce sia il naturale precedente n sia l'ipotesi induttiva $\text{ih} : \text{somma } \text{zero } n = n$.

Nel passo usiamo `rewrite [ih]`. Come `rw [ih]`, il comando sostituisce $\text{somma } \text{zero } n$ con n . Qui usiamo la forma estesa `rewrite` perché lascia visibile il goal successivo, che chiudiamo esplicitamente con `rfl`. `rw` può invece chiudere automaticamente un goal che, dopo la riscrittura, ha la forma $t = t$.

`open Naturale`

```
theorem lecture06_zero_somma (n : Naturale) :  
  somma zero n = n := by  
  induction n with  
  | zero =>  
    rfl  
  | succ n ih =>  
    unfold somma  
    rewrite [ih]  
    rfl
```

Valutiamo infine la scelta della variabile d'induzione. La definizione di somma analizza il secondo argomento. Nel goal `somma zero n = n`, quel posto è occupato da `n`, ed è proprio la sua forma sconosciuta a bloccare la computazione. Facciamo quindi induzione su `n`.

Nel caso successore, `unfold somma` espone la riga `somma zero (succ n) = succ (somma zero n)`. Il goal diventa `succ (somma zero n) = succ n`. L'ipotesi induttiva riguarda esattamente la sottoespressione rimasta, `somma zero n`, e la riscrittura la sostituisce con `n`. Otteniamo così `succ n = succ n`, concluso da `rfl`.

1.8 Fonti

- Lean developers, *Theorem Proving in Lean 4*, «Inductive Types»: https://lean-lang.org/theorem_proving_in_lean4/Inductive-Types/.
- Lean developers, *Theorem Proving in Lean 4*, «Induction and Recursion»: https://lean-lang.org/theorem_proving_in_lean4/Induction-and-Recursion/.
- Lean developers, *Functional Programming in Lean*, «Datatypes and Patterns»: https://lean-lang.org/functional_programming_in_lean/Getting-to-Know-Lean/Datatypes-and-Patterns/.
- Lean developers, *The Lean Language Reference*, «Recursive Definitions»: <https://lean-lang.org/doc/reference/latest/Definitions/Recursive-Definitions/>.
- Lean community, *Natural Number Game*: <https://adam.math.hhu.de/>.