Lezione 1: Che cos’è un interactive theorem prover?

Stefano M. Nicoletti

1 Lezione 1: Che cos’è un interactive theorem prover?

1.1 Scopo della lezione

Questa lezione introduce l’idea generale di interactive theorem prover e poi presenta Lean come uno specifico sistema di questo tipo. L’obiettivo non è ancora imparare molte tattiche o formalizzare esempi complessi. L’obiettivo è capire che cosa significa costruire una dimostrazione in modo interattivo, che cosa vede l’utente durante la costruzione, e perché un sistema come Lean può essere utile per analizzare argomenti, programmi, dimostrazioni matematiche e modelli scientifici.

La lezione ha tre parti:

- una parte teorica: che cos’è un theorem prover?
- una parte di dimostrazione in classe;
- una parte di esercizi.

1.2 Che cos’è un interactive theorem prover?

Un theorem prover è un sistema che aiuta a costruire e verificare dimostrazioni formali. Alcuni theorem prover sono prevalentemente automatici: l’utente formula un problema e il sistema prova a trovare una dimostrazione. Un interactive theorem prover, invece, mette al centro l’interazione tra utente e sistema (Fonte: Theorem Proving in Lean 4, Propositions and Proofs).

In un sistema interattivo l’utente scrive definizioni, enunciati e passaggi della dimostrazione. Il sistema risponde mostrando lo stato corrente della dimostrazione (o *proof state*) e accettando solo passaggi che rispettano le regole formali del sistema. La dimostrazione completa viene accettata e ritenuta corretta perché può essere verificata meccanicamente (si parla di *machine-checkable proofs*).

Un proof assistant da solo non sostituisce il giudizio umano. Non decide da solo quale sia la formalizzazione giusta di un problema, né stabilisce se le premesse di un argomento siano empiricamente o storicamente fondate. Il suo contributo è circoscritto: una volta fissati enunciati, definizioni e assunzioni, controlla se una certa costruzione formale dimostra davvero ciò che dichiara di dimostrare.

1.3 Lean

Lean è uno specifico interactive theorem prover. È anche un linguaggio di programmazione. Questa doppia natura è centrale: Lean permette di scrivere oggetti computazionali, strutture matematiche, enunciati logici e dimostrazioni in un unico ambiente formale (Fonte: de Moura and Ullrich, Lean 4).

In Lean possiamo:

- definire oggetti;
- formulare enunciati;
- costruire dimostrazioni;
- seguire lo sviluppo di una dimostrazione in modo interattivo;
- verificare formalmente il risultato prodotto.

Per questa prima lezione basta tenere ferma un'idea: Lean non controlla un testo informale, ma un oggetto formale. Se la dimostrazione viene accettata, il sistema ha verificato che l'oggetto costruito ha il tipo richiesto dall'enunciato. Ma cosa significa questo?

1.4 Oggetti e tipi

In Lean ogni oggetto, o *termine*, ha un tipo. Il tipo ci dice che genere di oggetto stiamo trattando e orienta ciò che ha senso fare con esso (Fonte: Theorem Proving in Lean 4, Dependent Type Theory).

Per esempio, oggetti che rappresentano numeri naturali (di tipo *Nat*) possono entrare in ragionamenti computazionali: possiamo sommare, moltiplicare, definire funzioni ricorsive o dimostrare proprietà aritmetiche. Oggetti che rappresentano proposizioni (di tipo *Prop*), invece, entrano in ragionamenti su verità e falsità.

Il tipo di un oggetto, dunque, informa il modo in cui Lean può trattarlo. Se abbiamo un oggetto di tipo numerico, Lean si aspetta operazioni e proprietà adatte a numeri. Il rapporto sistematico tra tipi, termini, proposizioni e dimostrazioni sarà affrontato nelle prossime lezioni.

1.5 Dimostrazioni interattive

In una dimostrazione interattiva possiamo osservare lo stato corrente della dimostrazione. Questo stato è spesso chiamato *proof state*.

Il proof state contiene almeno due informazioni fondamentali:

- il *goal*, cioè ciò che resta da dimostrare in quel punto;
- le *ipotesi e assunzioni* locali disponibili in quel punto.

All'inizio il goal coincide con l'obiettivo dell'enunciato. Durante la dimostrazione può cambiare: può essere trasformato in un goal più semplice, può essere diviso in più goal, o può essere chiuso/risolto quando viene fornito esattamente ciò che serve. Una dimostrazione è completa quando non restano goal aperti.

Le ipotesi sono le informazioni che possiamo usare in un dato momento della dimostrazione. Alcune vengono direttamente dall'enunciato. Altre possono essere introdotte durante la dimostrazione. Alcune valgono solo localmente, per esempio dentro un ramo di una dimostrazione per casi.

1.6 Tattiche

Una *tattica* è un comando che trasforma il proof state. Non è la dimostrazione finale nel senso più fondamentale, ma è un modo pratico per costruirla interattivamente.

Una tattica può, per esempio:

- introdurre una nuova assunzione;
- applicare una regola disponibile;
- dividere un problema in casi;
- produrre un risultato intermedio;
- chiudere un goal quando è disponibile un oggetto del tipo richiesto.

La prospettiva didattica della prima lezione è questa: una dimostrazione interattiva è una sequenza controllata di trasformazioni del proof state. L'utente propone un passaggio; Lean aggiorna il goal e il contesto, oppure rifiuta il passaggio se non è formalmente valido.

1.7 Perché usare un theorem prover?

1.7.1 Argomentazione

Un theorem prover costringe a distinguere premesse, passaggi inferenziali e conclusione. Questo è utile anche quando l'interesse principale non è matematico. In un argomento ordinario spesso molte premesse restano implicite, oppure il passaggio dalle premesse alla conclusione viene lasciato alla competenza del lettore.

Una formalizzazione rende visibile la struttura:

- quali assunzioni vengono usate;
- quale conclusione si vuole ottenere;
- quale passaggio inferenziale collega premesse e conclusione.

Naturalmente questo non risolve tutti i problemi dell'argomentazione. Lean non decide se una premessa sia accettabile, se una fonte sia affidabile, o se la formalizzazione catturi davvero l'argomento informale. Verifica però la parte formale dei passaggi inferenziali.

1.7.2 Software correctness

Un programma può compilare, passare molti test e violare comunque la specifica del comportamento atteso. I test mostrano che alcuni casi funzionano. Una prova formale può invece stabilire che una proprietà desiderata vale per tutte le esecuzioni possibili di un dato programma.

La verifica formale è rilevante quando vogliamo garanzie più forti dei soli test: compilatori, protocolli, sistemi critici, algoritmi e infrastrutture software possono richiedere proprietà esplicite e verificabili (Fonte: VU Amsterdam, Logical Verification).

1.7.3 Matematica

In matematica una dimostrazione formalizzata rende verificabile il rapporto tra definizioni, lemmi e conclusione. Ogni passaggio deve essere giustificato rispetto a regole esplicite. Questo non elimina il valore della scrittura matematica informale, ma aggiunge un livello di controllo meccanico.

Lean è particolarmente importante anche per la sua biblioteca matematica, `mathlib`, costruita in modo collaborativo. Una volta formalizzato, un risultato può essere riusato da altri come parte di una libreria di matematica verificata (Fonte: The mathlib Community, The Lean Mathematical Library).

1.7.4 Fisica

In fisica un theorem prover può aiutare a rendere esplicite le ipotesi di un modello, a controllare passaggi matematici o logici, e a chiarire quali assunzioni fanno parte di una derivazione. Non decide se un modello fisico sia empiricamente adeguato. Può però aiutare a separare la questione empirica dalla questione formale: dato un certo modello e date certe ipotesi, che cosa segue?

Physlib è un progetto comunitario per formalizzare risultati di fisica in Lean 4. È un buon esempio del modo in cui la formalizzazione può diventare un lavoro collaborativo anche fuori dalla matematica pura (Fonte: Physlib).

1.8 Che cosa verifica Lean?

Lean verifica la correttezza di una costruzione formale rispetto a un enunciato formalizzato. Lean non decide, tuttavia:

- se una premessa empirica è vera;
- se una fonte storica è affidabile;
- se un modello fisico è adeguato a un fenomeno;
- se una formalizzazione cattura tutto ciò che volevamo dire;
- se una dimostrazione è pedagogicamente chiara.

Lean controlla che, date certe definizioni e assunzioni, la conclusione formale segua secondo le regole del sistema. La verifica meccanica non elimina il giudizio umano, confina però molti errori o sviste dalla lettura informale al controllo esplicito di un oggetto formale.

1.9 Come fidarsi di Lean?

La fiducia in Lean dipende dal modo in cui il sistema è costruito. L'idea centrale è ridurre la parte del sistema di cui dobbiamo fidarci.

Lean può offrire interfacce complesse, tattiche sofisticate e librerie enormi. Tuttavia, il risultato finale deve essere controllato da un kernel relativamente piccolo. Il *kernel* è la parte del sistema che verifica se la dimostrazione costruita è corretta. Per questo appartiene alla *trusted computing base*: è la parte di cui dobbiamo fidarci perché la verifica abbia valore (Fonte: de Moura and Ullrich, Lean 4).

Il principio del kernel minimale è pragmatico: se il sistema contiene molti strumenti complessi, vogliamo che gli errori di questi strumenti non bastino da soli ad accettare una dimostrazione sbagliata.

La fiducia in Lean è poi sostenuta anche grazie al carattere aperto e comunitario del sistema. Kernel, librerie e strumenti possono essere discussi pubblicamente, ispezionati, corretti e confrontati con implementazioni o checker indipendenti. Progetti come *Lean4Lean* mostrano un'ulteriore direzione: usare strumenti formali per verificare parti del metalinguaggio e del controllo formale stesso (Fonte: Carneiro, Lean4Lean).

1.10 Argomento e buon argomento

Cominciamo la nostra avventura in Lean dalla formalizzazione di alcuni argomenti. Ma prima di tutto: un argomento collega premesse e conclusione attraverso un'inferenza (Fonte: Stanford Encyclopedia of Philosophy, Argument and Argumentation).

- Le premesse sono ciò da cui partiamo.
- La conclusione è ciò che vogliamo sostenere.
- L'inferenza è il passaggio dalle premesse alla conclusione.

Un buon argomento richiede almeno tre dimensioni.

Dimensione	Domanda
Accettabilità	Le premesse sono vere (credibili/difendibili)?
Validità	La conclusione segue dalle premesse?
Soundness/correttezza	Le premesse sono accettabili/vere e l'inferenza è valida?

La validità riguarda la forma del passaggio inferenziale. Un argomento può avere premesse plausibili ma inferenza non valida. La soundness, o correttezza, richiede entrambe le cose: premesse vere e inferenza valida.

Lean subentra soprattutto sulla dimensione formale della validità. Può controllare che una conclusione segua in maniera valida da certe premesse formalizzate. Non può, da solo, stabilire che quelle premesse siano accettabili nel contesto filosofico, storico, scientifico o empirico in cui le stiamo usando.

1.11 Linguaggio ordinario e forma logica

Prima di formalizzare bisogna riconoscere alcune forme ricorrenti nel linguaggio ordinario. In questa lezione ci limitiamo a tre connettivi fondamentali (Fonte: MIT OpenCourseWare, Logic I).

Linguaggio ordinario	Forma logica
Abbiamo entrambe le informazioni	E, cioè ∧
Abbiamo almeno una tra due alternative	O, cioè ∨
Se vale una condizione o ipotesi, allora segue una conseguenza	IMPLICA, cioè →

1.12 Installare Lean

Per lavorare con Lean useremo VS Code con l'estensione Lean 4. Lo standard consigliato dalla pagina ufficiale di installazione è (Fonte: Lean official install page):

- installare VS Code;
- installare la Lean 4 VS Code extension;
- seguire la guida di setup aperta dall'estensione;
- aprire la cartella del progetto in VS Code.

La pagina ufficiale rimane il riferimento operativo per il setup aggiornato.

1.13 Dimostrazione in classe

Il file usato per il live coding della lezione è `Classroom.lean`, che trovate sul sito di riferimento. Lo scopo del file è vedere la costruzione della dimostrazione mentre cambia il proof state. Gli esempi procedono in modo graduale: prima controlliamo tipi ed espressioni, poi costruiamo prove elementari, poi usiamo ∧, →, ¬ e ∨.

All'inizio vediamo alcuni comandi di orientamento:

```
#check Prop
#check Nat
#check Float

#eval 3 + 1
#eval 3.5 + 2
```

`#check` chiede a Lean il tipo di un'espressione. `#eval` chiede invece di valutare un'espressione computazionale. Qui non stiamo ancora dimostrando teoremi: stiamo osservando che Lean lavora sempre con oggetti tipati.

Il primo esercizio di dimostrazione è minimale:

```
-- Linguaggio naturale: se assumiamo che piove, allora possiamo concludere che
    piove.
example (Piove : Prop) (hPiove : Piove) : Piove := by
  sorry
```

`sorry` è un segnaposto: dice a Lean di accettare provvisoriamente un buco nella dimostrazione. La versione completa è:

```
-- Linguaggio naturale: se assumiamo che piove, allora possiamo concludere che
    piove.
example (Piove : Prop) (hPiove : Piove) : Piove := by
  exact hPiove
```

La tattica `exact` chiude il goal fornendo esattamente un termine del tipo richiesto. Il goal è `Piove`; nel contesto abbiamo `hPiove : Piove`; quindi `exact hPiove` è sufficiente.

Il passo successivo introduce un'implicazione:

```
-- Linguaggio naturale: se piove, allora piove.
example (Piove : Prop) : Piove → Piove := by
  intro h --intro nomeInformativo, qualsiasi nome ci aiuti a ricordare l'
   ipotesi
  exact h
```

La tattica `intro` serve quando il goal è un'implicazione. Per dimostrare `Piove → Piove`, assumiamo temporaneamente `Piove` e chiamiamo questa ipotesi `h`. A quel punto il goal diventa `Piove`, che è già disponibile come `h`. Dunque possiamo chiudere la dimostrazione con `exact`.

Ora passiamo a una congiunzione:

```
-- Linguaggio naturale: se piove allora prendo l'ombrello, e piove, allora
   questo implica che
-- prendo l'ombrello.
example (Piove PrendoOmbrello : Prop) :
    ((Piove → PrendoOmbrello) ∧ Piove) → PrendoOmbrello := by
  intro ipotesi
  have hPioveOmbrello := ipotesi.left
  have hPiove := ipotesi.right
  have hPrendoOmbrello := hPioveOmbrello hPiove
  exact hPrendoOmbrello
```

Qui `ipotesi` è una congiunzione. La parte sinistra è la regola `Piove → PrendoOmbrello`; la parte destra è la dimostrazione di `Piove`. La tattica `have` introduce un risultato intermedio nel contesto. Prima separiamo la parte sinistra della congiunzione con `.left`, ponendola in `hPioveOmbrello`, poi facciamo lo stesso con la parte destra della congiunzione e `hPiove`. Applichiamo dunque `hPioveOmbrello` a `hPiove` e otteniamo `PrendoOmbrello`.

Lo stesso argomento può essere scritto in forma più breve:

```
-- Dimostrazione più breve
example (Piove PrendoOmbrello : Prop) :
    ((Piove → PrendoOmbrello) ∧ Piove) → PrendoOmbrello := by
  intro h
  exact h.left h.right
```

Qui non diamo nomi intermedi. `h.left` è la funzione `Piove → PrendoOmbrello`; `h.right` è la dimostrazione di `Piove`; dunque `h.left h.right` è una dimostrazione di `PrendoOmbrello`.

Possiamo anche dichiarare le assunzioni direttamente nel preambolo dell'esempio:

```
-- Dichiaro le assunzioni in modo esplicito nel preambolo dell'esempio
-- Linguaggio naturale: assumiamo che, se piove, allora prendo l'ombrello;
-- assumiamo anche che piove; quindi prendo l'ombrello.
example (Piove PrendoOmbrello : Prop)
    (hPioveOmbrello : Piove → PrendoOmbrello) (hPiove : Piove) :
    PrendoOmbrello := by
  exact hPioveOmbrello hPiove
```

In questo caso non serve estrarre nulla da una congiunzione: le due assunzioni sono già separate nel contesto.

Il passo successivo è dare un nome allo schema di ragionamento perché sia riutilizzabile in altre dimostrazioni:

```
-- Codifichiamo, verifichiamo e usiamo lo schema di ragionamento valido
-- Linguaggio naturale: se P è vero e da P segue Q, allora Q è vero.
theorem lecture01_modus_ponens
    {P Q : Prop}
    (hP : P) (hPQ : P → Q) :
    Q := by
  exact hPQ hP
```

Questo è il modus ponens. Le parentesi graffe in `{P Q : Prop}` indicano argomenti impliciti: Lean spesso riesce a inferirli dal tipo delle ipotesi. Una volta dimostrato il teorema, possiamo usarlo applicandolo a un caso concreto:

```
example (Piove PrendoOmbrello : Prop)
    (hPioveOmbrello : Piove → PrendoOmbrello) (hPiove : Piove) :
    PrendoOmbrello := by
  exact lecture01_modus_ponens hPiove hPioveOmbrello
```

Qui `exact` chiude il goal con l'applicazione del teorema generale al caso concreto.

Poi formalizziamo una catena argomentativa:

```
-- Linguaggio naturale: se abbiamo un'assunzione, se dall'assunzione segue
-- una conseguenza, e se dalla conseguenza segue una conclusione, allora
   abbiamo
-- la conclusione.
theorem lecture01_informal_argument_schema
    (Assunzione Conseguenza Conclusione : Prop) :
    (Assunzione ∧ (Assunzione → Conseguenza)) ∧
      (Conseguenza → Conclusione) →
    Conclusione := by
  intro hArgomento
  have hPrimoPasso := hArgomento.left
  have hAssunzione := hPrimoPasso.left
  have hPasso1 := hPrimoPasso.right
  have hPasso2 := hArgomento.right
  have hConseguenza := hPasso1 hAssunzione
```

```
  have hConclusione := hPasso2 hConseguenza
  exact hConclusione
```

Qui la struttura è annidata. Prima separiamo il primo blocco dal secondo, poi estraiamo l'assunzione e la prima implicazione, poi costruiamo la conseguenza, infine applichiamo il secondo passaggio alla conseguenza.

La negazione in Lean si legge come implicazione verso `False`. L'esempio seguente dice che, se bere vino implica essere ubriachi e non siamo ubriachi, allora non abbiamo bevuto vino:

```
-- Linguaggio naturale: se bevo vino allora mi ubriaco; ma non sono ubriaco;
-- dunque non ho bevuto vino.
theorem lecture01_wine_example
    (BevoVino Ubriaco : Prop) :
    ((BevoVino → Ubriaco) ∧ ¬Ubriaco) → ¬BevoVino := by
  intro hArgomento
  have hRegola := hArgomento.left
  have hNonUbriaco := hArgomento.right
  intro hBevoVino
  have hUbriaco := hRegola hBevoVino
  exact hNonUbriaco hUbriaco
```

Il goal finale `¬BevoVino` significa `BevoVino → False`. Per questo usiamo `intro hBevoVino`: assumiamo temporaneamente `BevoVino` e cerchiamo una contraddizione. La regola produce `Ubriaco`; l'ipotesi `hNonUbriaco` trasforma `Ubriaco` in `False`.

Un altro passaggio importante è la disgiunzione. Se sappiamo `Piove ∨ Nevica`, non sappiamo quale dei due lati sia vero. Perciò dobbiamo mostrare che la conclusione segue in entrambi i casi:

```
-- Linguaggio naturale: se piove oppure nevica, e in ciascuno dei due casi
-- prendo l'ombrello, allora prendo l'ombrello.
example (Piove Nevica PrendoOmbrello : Prop) :
    ((Piove → PrendoOmbrello) ∧
      (Nevica → PrendoOmbrello)) ∧
    (Piove ∨ Nevica) →
    PrendoOmbrello := by
  intro hArgomento
  have hRegole := hArgomento.left
  have hPioveONevica := hArgomento.right
  have hPioveOmbrello := hRegole.left
  have hNevicaOmbrello := hRegole.right
  -- Non sappiamo quale lato della ∨`` è valido, quindi consideriamo entrambi
   i casi.
  cases hPioveONevica with
  -- Primo caso: l'ipotesi `Piove ∨ Nevica` è vera perché vale `Piove`,
   dimostriamo `PrendoOmbrello`.
  | inl hPiove =>
      exact hPioveOmbrello hPiove
  -- Secondo caso: l'ipotesi `Piove ∨ Nevica` è vera perché vale `Nevica`,
   dimostriamo `PrendoOmbrello`.
  | inr hNevica =>
```

```
      exact hNevicaOmbrello hNevica
```

La tattica `cases` apre un ramo per ogni modo in cui una disgiunzione può essere vera. Nel ramo `inl` abbiamo una dimostrazione di `Piove`; nel ramo `inr` abbiamo una dimostrazione di `Nevica`. In entrambi i rami dobbiamo produrre lo stesso risultato: `PrendoOmbrello`.

La stessa idea può essere scritta con `Or.elim`. Qui compare anche la tattica `apply`: invece di fornire subito la dimostrazione completa, applichiamo un principio generale che trasforma il goal in sottogoal più specifici.

```
-- La stessa dimostrazione, usando direttamente `Or.elim`.
example (Piove Nevica PrendoOmbrello : Prop) :
    ((Piove → PrendoOmbrello) ∧
      (Nevica → PrendoOmbrello)) ∧
    (Piove ∨ Nevica) →
    PrendoOmbrello := by
  intro hArgomento
  have hRegole := hArgomento.left
  have hPioveONevica := hArgomento.right
  have hPioveOmbrello := hRegole.left
  have hNevicaOmbrello := hRegole.right
  apply Or.elim hPioveONevica
  -- Se vale il lato sinistro della ∨``, cioè `Piove`,
  -- usiamo la regola `Piove → PrendoOmbrello`.
  · intro hPiove
    exact hPioveOmbrello hPiove
  -- Se vale il lato destro della ∨``, cioè `Nevica`,
  -- usiamo la regola `Nevica → PrendoOmbrello`.
  · intro hNevica
    exact hNevicaOmbrello hNevica
```

`Or.elim` dice esplicitamente: per usare una dimostrazione di `A ∨ B`, basta dare una dimostrazione della conclusione C a partire da `A` e una dimostrazione della stessa conclusione C a partire da `B`.

L'ultimo esempio combina congiunzioni, disgiunzioni e una catena di implicazioni:

```
-- Linguaggio naturale: se la fonte è autentica oppure i dati sono coerenti,
-- e ciascuno dei due casi supporta la tesi, e se una tesi supportata rende
-- plausibile la conclusione, allora la conclusione è plausibile.
theorem lecture01_history_of_science_example
    (FonteAutentica DatiCoerenti TesiSupportata ConclusionePlausibile : Prop)
   :
    ((FonteAutentica ∨ DatiCoerenti) ∧
      ((FonteAutentica → TesiSupportata) ∧
        (DatiCoerenti → TesiSupportata))) ∧
    (TesiSupportata → ConclusionePlausibile) →
    ConclusionePlausibile := by
  intro hArgomento
  have hPrimoPasso := hArgomento.left
  have hFonteODati := hPrimoPasso.left
  have hRegoleSupporto := hPrimoPasso.right
```

```
  have hFonteSupporta := hRegoleSupporto.left
  have hDatiSupportano := hRegoleSupporto.right
  have hSupportoConclude := hArgomento.right
  have hTesi :=
    Or.elim hFonteODati hFonteSupporta hDatiSupportano
  have hConclusione := hSupportoConclude hTesi
  exact hConclusione
```

La disgiunzione `FonteAutentica ∨ DatiCoerenti` viene eliminata con `Or.elim`. Le due funzioni `hFonteSupporta` e `hDatiSupportano` mostrano che, qualunque sia il lato valido della disgiunzione, otteniamo `TesiSupportata`. A quel punto applichiamo `hSupportoConclude` e otteniamo `ConclusionePlausibile`.

Le tattiche principali viste in questa sezione sono:

- `intro`: introduce un'ipotesi quando il goal è un'implicazione;
- `exact`: chiude il goal fornendo un termine del tipo richiesto;
- `have`: introduce un risultato intermedio con un nome;
- `cases`: divide una disgiunzione nei suoi casi;
- `apply`: applica una regola o un principio generale e genera i sottogoal necessari.

1.14 Esercizi e soluzioni

Trovate file aggiuntivi esercizi in più e soluzioni (`Exercises.lean` e `Solutions.lean`) sul sito di riferimento.

1.15 Fonti e letture

- Lean official install page: collegamento.
- Jeremy Avigad, Leonardo de Moura, Soonho Kong, Sebastian Ullrich, *Theorem Proving in Lean 4*, capitolo "Propositions and Proofs": collegamento.
- Jeremy Avigad, Leonardo de Moura, Soonho Kong, Sebastian Ullrich, *Theorem Proving in Lean 4*, capitolo "Dependent Type Theory": collegamento.
- Leonardo de Moura and Sebastian Ullrich, "The Lean 4 Theorem Prover and Programming Language": collegamento.
- Mario Carneiro, "Lean4Lean: Towards a Verified Typechecker for Lean, in Lean": collegamento.
- The mathlib Community, "The Lean Mathematical Library": collegamento.
- Physlib, physics in Lean 4: collegamento.
- Stanford Encyclopedia of Philosophy, "Argument and Argumentation": collegamento.
- MIT OpenCourseWare, Logic I: collegamento.
- VU Amsterdam, Logical Verification: collegamento.