ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Alma Mater Studiorum · Università di Bologna

Dipartimento di Informatica — Scienza e Ingegneria

Corso di Laurea Magistrale in Informatica

Implementazione e visualizzazione di alberi di deduzione naturale in Lean a supporto della didattica della logica

Relatore:
Chiar.mo Prof.
Claudio Sacerdoti Coen

Presentata da:
Riccardo Baviera

Correlatore:
Dott. Nicolò Pizzo

Sessione Luglio 2026

Anno Accademico 2025/2026

Abstract

Durante i laboratori del corso di logica per l'informatica del primo anno, viene utilizzato Lean 4, un linguaggio di programmazione per la dimostrazione assistita per permettere agli studenti di approcciarsi alla scrittura di prove formali. Tuttavia, durante le esercitazioni a lezione, si adotta anche un approccio grafico basato sugli alberi di deduzione naturale. Questo lavoro di tesi propone lo sviluppo di un ponte tra le due modalità di visualizzazione delle prove logiche. A questo scopo, è stato progettato e realizzato in Lean un widget in grado di tradurre automaticamente le dimostrazioni in Lean in alberi di deduzione naturale e visualizzarle.

Indice

Introduzione

In questo lavoro di tesi ho implementato un widget interattivo per *Lean 4* in grado di tradurre automaticamente le dimostrazioni formali in alberi di deduzione naturale visualizzabili in tempo reale all'interno dell'editor. Il widget è composto da un backend in Lean che visita ricorsivamente il λ-termine rappresentante la prova e lo converte in una struttura ad albero di deduzione naturale, e da un frontend React che riceve tale struttura via RPC e ne esegue il rendering grafico dinamico nell'infoview di Lean. Lo scopo didattico è mostrare agli studenti del corso di *Logica per l'informatica* la corrispondenza diretta tra le dimostrazioni scritte in Lean e gli alberi di deduzione naturale usati a lezione.

Didattica della logica

Durante il primo semestre del primo anno, gli studenti del corso di laurea in Informatica seguono l'insegnamento di *Logica per l'informatica*, incentrato sulla teoria assiomatica degli insiemi e sulla logica proposizionale e del primo ordine. Le lezioni frontali in aula introducono i principi fondanti di questi modelli, supportati dall'uso degli alberi di deduzione naturale per la formalizzazione logica. Il percorso è integrato da esercitazioni pratiche in laboratorio, dove gli studenti utilizzano un assistente alla dimostrazione (*Lean*) per familiarizzare con le dimostrazioni formali applicate alla teoria degli insiemi.

Questa separazione tra i due ambienti — la lavagna con gli alberi da un

lato, l'editor Lean dall'altro — genera spesso un disorientamento negli studenti: la prova scritta in Lean e l'albero tracciato a mano sembrano oggetti concettualmente distinti, mentre sono due rappresentazioni dello stesso ragionamento. L'esigenza didattica da cui nasce questo progetto è precisamente quella di rendere visibile e interattiva questa corrispondenza.

Introduzione alla deduzione naturale

La deduzione naturale è un sistema formale utilizzato per strutturare argomenti e dimostrazioni attraverso formule logiche e regole di inferenza. Questo approccio riproduce il ragionamento matematico intuitivo in modo rigoroso, facilitando la derivazione di conclusioni corrette a partire da un insieme di premesse.

La struttura logica di queste dimostrazioni viene presentata principalmente in due formati:

- **Rappresentazione lineare**: la dimostrazione si sviluppa come una sequenza cronologica di passaggi, in cui ogni riga rappresenta un'inferenza basata sulle asserzioni precedenti. Viene tipicamente redatta in linguaggio naturale o in un linguaggio controllato (formale), necessario per l'elaborazione da parte di software di verifica.

- **Alberi di derivazione** (o di deduzione): la dimostrazione si sviluppa graficamente in modo gerarchico. I nodi dell'albero rappresentano le formule logiche, mentre le linee di frazione orizzontali indicano l'applicazione delle regole di inferenza. Questo metodo offre una visione d'insieme immediata delle dipendenze logiche, facilitando l'individuazione di errori o passaggi mancanti.

Tipicamente, una rappresentazione lineare non è perfettamente equivalente ad un albero di derivazione, perché ogni inferenza è non atomica, cioè rappresenta 0 o più passaggi di deduzione naturale.

Isomorfismo di Curry-Howard

L'isomorfismo di Curry-Howard stabilisce una relazione diretta tra dimostrazioni matematiche e programmi informatici, permettendo di esprimere le une in funzione degli altri. Ci fornisce uno schema con cui possiamo formalizzare prove matematiche tramite programmi informatici:

- Una proposizione A è un tipo `A: Prop`
- Una dimostrazione di A è un termine di tipo `A`, quindi `t: A`
- Un'implicazione $A \to B$ è un tipo funzione, rappresentato da una funzione `f` che trasforma un termine di tipo `A` in un termine di tipo `B`, quindi `f: A → B`
- Una congiunzione $A \wedge B$ è un tipo prodotto (coppia) dei due tipi `A` e `B`, quindi `And A B: Prop` (rappresentato anche come coppia `⟨ t_A, t_B ⟩`)
- Una disgiunzione $A \vee B$ è un tipo somma dei due tipi `A` e `B`, quindi `Or A B: Prop`
- I valori di verità $\top$ e $\bot$ sono dati rispettivamente dal tipo unitario (contenente un solo valore banale) e dal tipo vuoto (contenente nessun valore), quindi `True: Prop` e `False: Prop`

Con queste definizioni siamo in grado di rappresentare completamente la logica proposizionale intuizionista. Per estendere l'isomorfismo alla logica del prim'ordine, introduciamo i **tipi dipendenti** (ovvero tipi che dipendono da termini), i quali permettono di formalizzare i quantificatori:

- Il quantificatore universale $\forall x : X, P(x)$ corrisponde a un **tipo funzione dipendente** (o tipo Π). È una funzione che accetta come argomento un qualsiasi elemento `x` di tipo `X` e restituisce una dimostrazione di `P(x)`. In Lean si esprime come `∀ x : X, P x` o `(x : X) → P x`.

- Il quantificatore esistenziale $\exists x : X, P(x)$ corrisponde a un **tipo coppia dipendente** (o tipo Σ). È una struttura che contiene due informazioni: un elemento specifico `x` di tipo `X` (chiamato *testimone*) e una dimostrazione che il predicato `P(x)` è soddisfatto da quel testimone. In Lean si esprime come `∃ x : X, P x`.

Dimostrazione assistita da elaboratore

La dimostrazione assistita da elaboratore è un approccio alla verifica formale in cui un software specializzato, detto *assistente alla dimostrazione* (o *proof assistant*), guida e controlla la correttezza di ogni passo di una dimostrazione matematica. A differenza della verifica manuale, in cui gli errori possono sfuggire anche a un lettore esperto, l'assistente garantisce meccanicamente che ciascuna inferenza sia valida rispetto alle regole del sistema logico adottato. Questo paradigma è particolarmente rilevante in informatica, dove la correttezza formale di algoritmi e sistemi software riveste un'importanza critica.

Il flusso di lavoro tipico prevede che il dimostratore formalizzi le definizioni, gli enunciati e i passi dimostrativi in un linguaggio controllato accettato dall'assistente, il quale verifica in tempo reale la validità delle inferenze e segnala eventuali lacune o incongruenze logiche. L'interazione tra utente e strumento rende il processo interattivo: l'elaboratore non si limita a controllare passivamente, ma fornisce retroazione immediata, suggerendo sottobiettivi da dimostrare e indicando quali ipotesi siano già disponibili nel contesto corrente.

Lean. *Lean* è un assistente alla dimostrazione open source sviluppato originariamente da Leonardo de Moura presso Microsoft Research, attualmente alla sua quarta versione principale (*Lean 4*). Il sistema si fonda su una variante della teoria dei tipi dipendenti e adotta *Mathlib* come principale libreria matematica, che raccoglie migliaia di risultati formalizzati in analisi, algebra,

teoria dei numeri e logica. In ambito didattico, Lean consente agli studenti di sperimentare concretamente il rigore della dimostrazione formale.

Obiettivo del progetto

L'obiettivo del progetto è sviluppare uno strumento che renda esplicita e interattiva l'equivalenza tra le dimostrazioni in forma lineare (scritte in Lean) e quelle strutturate come alberi di derivazione. Questo formalismo è cruciale in ambito didattico, poiché gli studenti faticano spesso a percepire l'intercambiabilità e il medesimo potere espressivo di queste due rappresentazioni. Per rispondere a questa esigenza, è stato sviluppato un widget per Lean che analizza il λ-termine interno associato al teorema corrente e lo traduce in un albero di deduzione naturale. La visualizzazione e la gestione dell'interattività dell'interfaccia sono delegate a una componente React che, ricevuta la struttura dell'albero in formato JSON, ne esegue il rendering grafico dinamico.

Struttura della tesi

Il resto dell'elaborato è organizzato come segue:

- Il Capitolo 1 introduce la deduzione naturale dal punto di vista teorico: la logica proposizionale, la struttura degli alberi di derivazione e le regole di inferenza per ciascun connettivo logico, sia nel frammento proposizionale sia nell'estensione alla logica del primo ordine. Questo capitolo fornisce il vocabolario formale a cui il resto della tesi farà costantemente riferimento.

- Il Capitolo 2 descrive l'architettura software del progetto: come backend e frontend comunicano tramite il protocollo RPC del Lean Language Server, quali tecnologie sono coinvolte (le monadi `MetaM` e `RequestM` di Lean, il meccanismo dei widget, React nell'infoview)

e come è organizzato il tipo di dato `NDTree` che rappresenta l'albero di deduzione naturale.

- Il Capitolo 3 entra nel dettaglio dell'implementazione, distinguendo la parte di backend, che implementa la traduzione del λ-termine della prova in un `NDTree` tramite visita ricorsiva e pattern matching sui costruttori logici di Lean, dalla parte di frontend, ossia il componente React che riceve la struttura JSON e ne cura il rendering grafico, incluso il calcolo dinamico del layout dell'albero e la gestione dell'interattività.
- Le Conclusioni riassumono i risultati della validazione condotta sul corpus di esercizi degli anni accademici precedenti, quantificano il lavoro svolto e discutono le direzioni di sviluppo future.

Capitolo 1

Deduzione naturale

1.1 Logica proposizionale

La logica proposizionale è un linguaggio formale che studia il ragionamento basato su proposizioni, ovvero asserzioni che possono assumere un valori di verità.

Formalmente, si utilizzano le variabili proposizionali $(A, B, C, \dots)$ insieme ai connettivi logici $(\neg, \wedge, \vee, \rightarrow)$ e alle costanti per i valori di verità assoluti $(\top, \bot)$ per comporre formule ben formate in grado di rappresentare i ragionamenti logici.

Per esempio, si consideri la seguente argomentazione in linguaggio naturale:

> *Se c'è il sole, allora mi metto la crema solare e, se mi metto la crema solare, allora non mi scotto. Quindi, se c'è il sole, allora non mi scotto.*

Per formalizzare l'argomento, si mappa ogni proposizione atomica su una variabile logica:

- S: *c'è il sole*
- C: *mi metto la crema solare*

- B: *mi scotto* (da cui $\neg B$: *non mi scotto*)

L'intera struttura del ragionamento viene quindi espressa tramite la formula:

$$((S \to C) \land (C \to \neg B)) \to (S \to \neg B)$$

1.2 Alberi di deduzione naturale

Un albero di deduzione naturale è una struttura formale che rappresenta la derivazione sintattica $\Gamma \vdash \phi$, tale per cui:

- Ogni nodo è etichettato con una formula.
- I nodi interni (non foglie) sono associati all'applicazione di una regola di inferenza.
- Le foglie corrispondono a formule scaricate $[\psi]$ (ipotesi locali non più attive) o a formule non scaricate γ (ipotesi globali correnti, $\gamma \in \Gamma$).
- La radice dell'albero è etichettata con la conclusione ϕ.

Una dimostrazione è valida se l'albero di deduzione è corretto: tutte le foglie non scaricate appartengono all'insieme delle premesse Γ e ogni transizione rispetta le regole del calcolo. In tal caso, si afferma che $\Gamma \vdash \phi$ (l'insieme Γ deriva sintatticamente ϕ).

$$\dfrac{\dfrac{\vdots}{\phi'}\ \textit{rule'} \qquad [\psi] \qquad \gamma}{\phi}\ \textit{rule}$$

Regole di inferenza

Le regole di inferenza stabiliscono le transizioni lecite tra i nodi dell'albero. Esse si dividono in regole di *introduzione* di un connettivo (che specificano come derivare una formula contenente il connettivo nella conclusione della regola) e regole di *eliminazione* di un connettivo (che descrivono come utilizzare una formula che ha una premessa che usa tale connettivo).

Di seguito vengono presentate le regole strutturate per ciascun connettivo logico:

$\wedge$-introduzione

Intuitivamente, se si vuole dimostrare la validità della congiunzione $A \wedge B$, è necessario derivare autonomamente e indipendentemente sia l'asserzione A sia l'asserzione B.

$$\frac{A \qquad B}{A \wedge B} \wedge_i$$

$\wedge$-eliminazione

Intuitivamente, assumendo come valida la congiunzione $A \wedge B$, è logicamente lecito dedurre singolarmente la validità di ciascun membro (A oppure B).

$$\frac{A \wedge B}{A} \wedge_{e1} \qquad \frac{A \wedge B}{B} \wedge_{e2}$$

$\vee$-introduzione

Intuitivamente, per dimostrare la disgiunzione $A \vee B$, è sufficiente verificare la validità di uno solo dei due disgiunti (A oppure B).

$$\frac{A}{A \vee B} \vee_{i1} \qquad \frac{B}{A \vee B} \vee_{i2}$$

$\vee$-eliminazione

Intuitivamente, disponendo della disgiunzione $A \vee B$ e volendo derivare una conclusione C, occorre dimostrare che C consegue sia assumendo l'ipotesi locale $[A]$, sia assumendo l'ipotesi locale $[B]$.

$$\frac{A \vee B \qquad \begin{array}{c}[A]\\ \vdots \\ C\end{array} \qquad \begin{array}{c}[B]\\ \vdots \\ C\end{array}}{C} \vee_e$$

$\rightarrow$-introduzione

Intuitivamente, per dimostrare l'implicazione $A \rightarrow B$, si assume ipoteticamente l'antefatto A e si procede alla derivazione del conseguente B. Ottenuto B, l'ipotesi $[A]$ viene scaricata.

$$\dfrac{\begin{array}{c}[A] \\ \vdots \\ B\end{array}}{A \rightarrow B} \rightarrow_i$$

$\rightarrow$-eliminazione (o *modus ponens*)

Intuitivamente, se si dispone della validità dell'implicazione $A \rightarrow B$ e si verifica l'antefatto A, si deduce direttamente il conseguente B.

$$\dfrac{A \rightarrow B \qquad A}{B} \rightarrow_e$$

Con questo nucleo di regole si definisce la *logica minimale*. Aggiungendo la regola di eliminazione del falso ($\bot_e$) si estende il sistema alla *logica intuizionistica*, in cui si applica il principio del *ex falso quodlibet*: dalla falsità logica si può derivare qualsiasi formula ben formata.

$\bot$-eliminazione

Intuitivamente, se il sistema logico permette di derivare una contraddizione assoluta ($\bot$), qualsiasi enunciato arbitrario diventa formalmente valido all'interno della derivazione.

$$\dfrac{\bot}{A} \bot_e$$

Poiché l'operatore di negazione $\neg A$ è formalmente definito come l'abbreviazione sintattica di $A \rightarrow \bot$, le regole dell'implicazione e dell'eliminazione del falso sono sufficienti a governarne il comportamento. Tali regole non estendono il potere espressivo della logica intuizionistica, ma vengono esplicitate di seguito per ragioni di chiarezza descrittiva.

¬-introduzione

Intuitivamente, per dimostrare $\neg A$, si assume ipoteticamente l'asserto positivo A e si mostra come esso conduca a una contraddizione ($\bot$). Validata l'assurdità, l'ipotesi $[A]$ viene scaricata.

$$\dfrac{\begin{array}{c}[A]\\ \vdots \\ \bot\end{array}}{\neg A}\ \neg_i$$

¬-eliminazione

Intuitivamente, se si assume la validità simultanea di un enunciato A e della sua negazione $\neg A$, si genera una contraddizione logica immediata ($\bot$).

$$\dfrac{\neg A \qquad A}{\bot}\ \neg_e$$

Infine, estendendo il sistema con il principio del terzo escluso ($A \vee \neg A$), si ottiene la *logica classica*. Questa estensione si riflette sintatticamente nell'introduzione della regola di riduzione all'assurdo (*reductio ad absurdum*, o *raa*).

reductio ad absurdum

Intuitivamente, nella logica classica, per dimostrare l'enunciato A è sufficiente ipotizzare la sua negazione $[\neg A]$ e ricavare da essa una contraddizione ($\bot$), rigettando così l'ipotesi di partenza.

$$\dfrac{\begin{array}{c}[\neg A]\\ \vdots \\ \bot\end{array}}{A}\ raa$$

Si definiscono così tre livelli gerarchici di potere deduttivo crescente: logica minimale $\subset$ logica intuizionistica $\subset$ logica classica.

1.3 Logica del primo ordine

L'estensione alla logica del primo ordine richiede l'arricchimento del linguaggio tramite variabili individuali, termini e predicati, unitamente all'in-

troduzione dei quantificatori universale ($\forall$) ed esistenziale ($\exists$).

Le regole associate ai quantificatori impongono rigorosi vincoli (condizioni sulle variabili libere) per preservare la correttezza del sistema:

- Nella regola $\forall_i$, la variabile d'appoggio y deve essere sufficientemente *fresca*, ovvero non deve comparire libera in alcuna delle ipotesi attive (foglie non scaricate) della sotto-derivazione di $P[y/x]$.
- Nella regola $\exists_e$, la variabile di supporto y non deve comparire libera nella conclusione A, né nelle restanti ipotesi attive della derivazione, garantendo che la deduzione sia indipendente dallo specifico testimone scelto.

$\forall$-introduzione

Intuitivamente, se si dimostra che una proprietà P vale per un individuo arbitrario y sul quale non è attiva alcuna assunzione preliminare, è lecito concludere che la proprietà valga per ogni elemento.

$$\begin{array}{c} y \notin \mathrm{FV}(\mathrm{Foglie}(\vdots)) \\ \\ \vdots \\ \dfrac{P[y/x]}{\forall x.P}\,\forall_i \end{array}$$

$\forall$-eliminazione

Intuitivamente, assumendo che una proprietà P sia universalmente valida per ogni x, si può istanziare la medesima relazione su un qualunque termine specifico t appartenente al dominio.

$$\frac{\forall x.P}{P[t/x]}\,\forall_e$$

∃-introduzione

Intuitivamente, se si dimostra che una proprietà P è soddisfatta da uno specifico termine testimone t, è legittimo asserire l'esistenza di almeno un elemento del dominio che verifichi tale relazione ($\exists x.P$).

$$\frac{P[t/x]}{\exists x.P}\,\exists_i$$

∃-eliminazione

Intuitivamente, se si assume l'esistenza di un elemento soddisfacente P ($\exists x.P$), si può procedere nella derivazione introducendo l'ipotesi locale istanziata $[P[y/x]]$. La variabile y funge da parametro generico e non deve essere soggetto ad ulteriori vincoli oltre $P[y/x]$.

$$y \notin \mathrm{FV}(\mathrm{Foglie}(\vdots) \cup \mathrm{FV}(A)$$

$$\frac{\exists x.P \qquad \begin{array}{c}[P[y/x]]\\ \vdots \\ A\end{array}}{A}\,\exists_e$$

Capitolo 2

Architettura software del progetto

2.1 Panoramica

Il progetto è strutturato come un *widget* per l'infoview di Lean 4, ossia un pannello laterale interattivo che si aggiorna in tempo reale mentre il cursore dell'utente si muove all'interno del file sorgente. L'architettura del progetto è strutturata in due componenti principali che comunicano tramite il protocollo RPC del *Lean Language Server*:

- **Backend (Lean)**: recupera il λ-termine della prova corrente, lo visita ricorsivamente e costruisce un albero di deduzione naturale (`NDTree`), che viene poi serializzato in JSON.
- **Frontend (React/JavaScript)**: riceve il JSON via RPC, lo deserializza e ne esegue il rendering grafico dinamico nell'infoview, gestendo l'interattività (click sulle formule, visualizzazione delle ipotesi).

Il flusso complessivo è il seguente: il cursore dell'utente nel file `.lean` scatena una chiamata RPC verso il server Lean; il metodo `getTreeAsJson` recupera il λ-termine, costruisce l'`NDTree` e restituisce il JSON; il componente React lo riceve, lo analizza e aggiorna la visualizzazione.

2.2 Tecnologie e infrastruttura

2.2.1 Lean 4 e la metaprogrammazione

Lean 4 espone un ricco insieme di API di metaprogrammazione che permettono di ispezionare e manipolare le proprie strutture interne durante l'elaborazione di un file sorgente. Le due principali monadi utilizzate in questo progetto sono:

- **`MetaM`**: la monade principale per l'elaborazione di espressioni, tipi e metavariabili. Fornisce accesso al contesto locale (*local context*), all'assegnamento delle metavariabili (*metavariable context*) e alle operazioni di unificazione e inferenza di tipo.
- **`RequestM`**: la monade per la gestione delle richieste RPC del Language Server. Permette di leggere lo stato corrente del documento, di localizzare l'albero informativo (*info tree*) che corrisponde alla posizione del cursore e di eseguire computazioni `MetaM` nel contesto giusto.

Il meccanismo dei **widget** (`@[widget_module]`) permette di associare a un modulo Lean un file JavaScript che viene caricato ed eseguito dall'infoview di VS Code. I metodi RPC (`@[server_rpc_method]`) consentono al codice JavaScript di invocare funzioni Lean e ottenerne il risultato in modo asincrono, mantenendo la sincronizzazione con lo stato dell'editor.

2.2.2 Il protocollo RPC e il Language Server

Il *Lean Language Server* implementa il protocollo LSP (*Language Server Protocol*), esteso da Lean con un meccanismo proprietario di chiamate RPC tipizzate. Ogni metodo RPC è definito lato Lean come una funzione con parametri e risultato derivanti da `ToJson`/`FromJson`; lato JavaScript, il runtime dell'infoview espone l'hook `useRpcSession` che restituisce un oggetto `rs` su cui è possibile invocare `rs.call(methodName, params)`.

Questo canale trasporta il JSON dell'albero come semplice stringa: la scelta è stata dettata dalla necessità di trasmettere formule matematiche contenenti caratteri Unicode (∀, ∃, → , ∧, ∨ , …) che richiedono una gestione attenta della serializzazione.

2.2.3 React nell'infoview

L'infoview di Lean per VS Code espone un ambiente React nel quale è possibile definire componenti personalizzati. Il widget riceve come *props* la posizione corrente del cursore (`pos`), che viene usata come chiave per rieseguire la chiamata RPC e aggiornare la visualizzazione ogni volta che il cursore si sposta su un punto diverso del documento.

2.3 Struttura dei moduli

Il progetto è organizzato nei seguenti file:

- **`DeductionTreeWidget.lean`**: contiene il tipo di dato `NDTree`, le funzioni di conversione da `Expr` a `NDTree` (`toNDTreeM` e `getHypoteses`), la serializzazione JSON, il metodo RPC `getTreeAsJson` e la dichiarazione del widget.
- **`NDTreeJsonViewer.js`**: il componente React che riceve il JSON e lo visualizza, con gestione dello stato delle ipotesi selezionate e dell'albero delle ipotesi espandibile.

2.4 Il tipo di dato `NDTree`

Il cuore della rappresentazione è il tipo induttivo `NDTree`, definito in Lean come segue:

```
1 inductive NDTree where
2   | leaf      : List (Name × NDTree) → Name → Formula
```

```
3                   → isDischarged → NDTree
4     | node      : List (Name × NDTree) → Formula → Rule
5                   → List NDTree → NDTree
6     | openNode  : List (Name × NDTree) → Formula
7                   → isCurrentGoal → NDTree
8     | unhandled : NDTree
```

I quattro costruttori modellano tutti i possibili nodi dell'albero:

- **`leaf`**: una foglia dell'albero, corrispondente a un'ipotesi (assunzione). Porta con sé la lista delle ipotesi attive nel contesto al momento della sua creazione, il proprio nome, la formula e un booleano che indica se l'ipotesi è stata *scaricata* (cioè è un'assunzione locale, da visualizzare tra parentesi quadre).
- **`node`**: un nodo interno, corrispondente all'applicazione di una regola di inferenza. Porta la lista delle ipotesi attive, la formula conclusione, il nome della regola applicata ($\to$`I`, $\vee$`E`, $\wedge$`I`, …) e la lista dei sotto-alberi (premesse).
- **`openNode`**: un nodo aperto, corrispondente a un *goal* non ancora dimostrato (una metavariabile non risolta). Porta la lista delle ipotesi attive, la formula del goal e un booleano che indica se questo è il goal corrente (quello su cui si trova il cursore).
- **`unhandled`**: un nodo sentinella usato nei casi in cui la struttura del λ-termine non corrisponde a nessun pattern riconosciuto, segnalando una limitazione del sistema di traduzione.

La lista di ipotesi presente nei nodi è formata da una lista di coppie `Name` $\times$ `NDTree`, che permette di rappresentare le ipotesi tramite il loro nome e la prova associata, a sua volta rappresentata dall'albero di deduzione naturale.

I tipi alias `Formula`, `Rule`, `isDischarged`, `isCurrentGoal` sono tutti abbreviazioni per i tipi primitivi `String` e `Bool`, definiti per

chiarezza semantica. Il tipo `NDTree` deriva automaticamente le istanze `FromJson`, `ToJson`, `RpcEncodable` e `BEq` tramite i meccanismi di *deriving* di Lean 4. L'istanza `ToJson` è stata comunque re-implementata per generare una struttura ad hoc per gli alberi di deduzione naturale.

2.5 Pipeline di esecuzione

Quando il cursore si trova all'interno di un blocco `by` di una dimostrazione, il metodo RPC `getTreeAsJson` esegue i seguenti passi:

1. **Localizzazione del nodo informativo**: tramite la funzione `Elab.InfoTree.goalsAt?`, viene recuperato il nodo dell'albero informativo di Lean corrispondente alla posizione del cursore, con il relativo contesto di elaborazione e lo stato delle metavariabili.

2. **Selezione della metavariabile principale**: tra le metavariabili nell'assegnamento estensionale (`eAssignment`), viene selezionata la più recente (in base all'indice numerico del nome), che corrisponde alla radice dell'albero di prova attualmente costruito.

3. **Raccolta delle ipotesi radice**: tramite `getHypoteses`, vengono estratte dal contesto locale le ipotesi *globali* (quelle presenti nel contesto della metavariabile radice), che saranno usate per distinguere le foglie scaricate da quelle non scaricate nei nodi interni.

4. **Visita ricorsiva**: la funzione `toNDTreeM` visita il λ-termine in profondità (*depth-first*), costruendo l'`NDTree` nodo per nodo secondo i pattern descritti nel Capitolo 3.

5. **Serializzazione**: l'`NDTree` viene convertito in JSON tramite il metodo `NDTree.toJson` e trasmesso al frontend come stringa.

Capitolo 3

Implementazione

3.1 Backend: traduzione del λ-termine in `NDTree`

3.1.1 Rappresentazione interna delle prove in Lean

In Lean 4, ogni dimostrazione completata è internamente un termine del *Calcolo delle Costruzioni Induttive* (*Calculus of Inductive Constructions*, CIC), un λ-calcolo tipato con tipi dipendenti. Il tipo `Expr` di Lean cattura questa struttura con i seguenti costruttori principali:

- `.fvar id`: variabile, che nel contesto di una prova corrisponde a un'ipotesi (un nome nel contesto locale).
- `.lam n t b bi`: astrazione lambda $\lambda n : t, b$, che sotto l'isomorfismo di Curry-Howard corrisponde a una regola di introduzione ($\to_i$ o $\forall_i$).
- `.app f a`: applicazione della funzione f all'argomento a (per permettere applicazioni con più argomenti, f deve essere un `.app f a` a sua volta. Dipendentemente dal tipo di f cambia la regola codificata.
- `.const name us`: costante globale con nome `name` e una lista di livelli `us`, rappresentante a quale livello di universo appartengono gli argomenti di una eventuale costante polimorfa (come `List` α); i costrut-

tori logici come `And.intro`, `Or.inl`, `Exists.elim` sono costanti di questo tipo.

- `.mvar id`: metavariabile, ossia un goal ancora aperto (non dimostrato).
- `.letE n t v b`: espressione `let`, usata internamente da alcune tattiche (come `have`) per definire frammenti di prove con nomi, per renderli riutilizzabili come ipotesi.

L'intera funzione di traduzione `toNDTreeM` si basa su un *pattern matching* su questa struttura, riconoscendo le costanti logiche Lean e associandole alle corrispondenti regole di deduzione naturale.

3.1.2 Gestione delle metavariabili delegate

Un problema non banale che si incontra nell'ispezione dei λ-termini durante una dimostrazione *in progress* è la presenza di *delayed assignments*: metavariabili che non sono state risolte con un termine diretto, bensì con un'altra metavariabile in un contesto locale arricchito. Lean le gestisce tramite `dAssignment` nel *metavariable context*, distinguendole dalle assegnazioni dirette in `eAssignment`.

Per gestire correttamente questi casi, è stata implementata la funzione `withAggressiveInstantiateMVars`:

```
partial def withAggressiveInstantiateMVars
    (e: Expr) (k: Expr → MetaM α) : MetaM α := do
  let e ← instantiateMVars e
  match e with
  | .app _ _ ⇒ do
    match e.withApp fun e a ⇒ (e, a) with
    | (.mvar mid, args) ⇒
      match (← getMCtx).dAssignment.find? mid with
      | some i ⇒
        if i.fvars.size == args.size then
          -- Ricostruisce il contesto locale aggiungendo
```

```
12              -- i parametri della delayed assignment come
                let-binding,
13              -- poi continua la visita sulla metavariabile
                interna.
14              let mut lctx := ← getLCtx
15              for fv in i.fvars, arg in args do
16                let fvid := fv.fvarId!
17                lctx := LocalContext.mkLetDecl lctx fvid
18                  (← i.mvarIdPending.withContext
                    fvid.getUserName)
19                  (← i.mvarIdPending.withContext fvid.getType)
20                  arg
21              withLCtx' lctx do
22                withAggressiveInstantiateMVars
23                  (.mvar i.mvarIdPending) k
24            else k e
25          | none ⇒ k e
26      | _ ⇒ k e
27    | _ ⇒ k e
```

La funzione intercetta il pattern `(.mvar mid) args ...`: quando `mid` è una metavariabile con una *delayed assignment* che ha esattamente tanti parametri liberi quanti gli argomenti ricevuti, ricostruisce il contesto locale legando ogni parametro al corrispondente argomento tramite *let-binding*, e poi prosegue la visita sulla metavariabile interna (`mvarIdPending`). Questo meccanismo è necessario, ad esempio, quando le tattiche `rcases` o `obtain` introducono variabili in modo non diretto.

3.1.3 Raccolta delle ipotesi: `getHypoteses`

Prima di costruire ogni nodo dell'albero, la funzione `getHypoteses` percorre il contesto locale corrente e raccoglie le ipotesi attive, trasformandole in nodi `NDTree`:

```
1 partial def getHypoteses (rootHyps : List Hyp := []) (isLHS :
  Bool := false) : MetaM (List Hyp) := do
2   let list ← (← getLCtx).decls.foldlM (fun s i ⇒ do
```

```
    match s, i with
    | [], _ ⇒ return [.none]
    | _, none ⇒ return s --slot vuoto nel contesto
    | _, some d ⇒
      let t := (← inferType (.fvar d.fvarId))
      if (← inferType t).isSort1 then return s -- filtra le
      proposizioni di tipo Type (non Prop)
      if rootHyps.contains (d.userName, (NDTree.leaf []
      d.userName s!"{← ppExpr t}" false)) then
        return s -- se l'ipotesi è già presente nel contesto
        della radice, non lo aggiungo
      match d.value? (allowNondep := true) with
      | none ⇒ do
      -- se l'ipotesi non ha un valore associato, significa
      che è solo una foglia
        let leaf :NDTree := .leaf [] d.userName s!"{← ppExpr
        t}" (!isLHS && !(← rootHyps.foldlM (fun res h ⇒
          match res, h with
          | true, _ ⇒ return true
          | false, (name, .leaf _ _ f _) ⇒ return  name ==
          d.userName && f == s!"{← ppExpr t}"
          | _, _ ⇒ return false
        ) false ))
        return s ++ [.some (d.userName, leaf)]
      | some v ⇒ do
      -- se ha un valore associato, l'ipotesi è formata da un
      sottoalbero generato da have o let
        return s ++ [.some (d.userName, (← v.toNDTreeM
        (withHyp := false) (rootHyps := rootHyps)))])
    []

return list.filterMap id
```

Alcuni aspetti importanti di questa funzione:

- Le dichiarazioni di tipo *non* proposizionale (ossia con tipo diverso da `Prop`) vengono filtrate: non sono ipotesi logiche ma instanziazioni di

tipo nel caso di termini polimorfi o termini argomento, e non devono comparire nell'albero.

- Le ipotesi già presenti nel contesto *radice* (quelle globali, raccolte prima di iniziare la visita) vengono escluse dai nodi interni, per evitare duplicazioni durante il rendering delle ipotesi.
- Se una dichiarazione ha un valore (`value? = some v`), si tratta di un'ipotesi introdotta da `have` o `let`: in tal caso viene convertita ricorsivamente, producendo un sotto-albero anziché una semplice foglia.

3.1.4 Visita del λ-termine: `toNDTreeM`

La funzione `toNDTreeM` è il cuore del backend. Il suo schema generale è:

1. Applicare `withAggressiveInstantiateMVars` per risolvere le metavariabili delegate.
2. Decomporre l'espressione in testa e argomenti con `e.withApp`.
3. Filtrare dagli argomenti quelli di tipo `Type` o superiore (non proposizionale): essi corrispondono ai parametri di tipo impliciti delle costanti polimorfiche e non devono essere rappresentati nell'albero.
4. Raccogliere le ipotesi attive tramite `getHypoteses`.
5. Fare pattern matching sulla coppia `(testa, argomenti)` per determinare la regola di deduzione naturale da applicare.

Di seguito vengono illustrati i casi principali.

Astrazione lambda (`.lam`) — regole $\to_i$ e $\forall_i$.

```
| (.lam n t b bi), _ ⇒ do
  withLocalDecl n bi t fun fv ⇒ do
    let b := b.instantiate1 fv
    -- Se il tipo t è una proposizione →I, altrimenti ∀I
    let ruleName :=
      if (← inferType t).isProp then "→I" else "∀I"
    let child ← b.toNDTreeM
      (withHyp := withHyp) (rootHyps := rootHyps)
      (currentGoal := currentGoal)
    return .node hyps
      s!"{← ppExpr (← inferType fn)}" ruleName [child]
```

Un'astrazione lambda `fun n : t ⇒ b` corrisponde alla regola di introduzione dell'implicazione ($\to_i$) quando `t` è di tipo `Prop`, o alla regola di introduzione del quantificatore universale ($\forall_i$) quando `t` è di tipo `Type`. Il corpo `b` viene istanziato sulla variabile libera fresca `fv` tramite `withLocalDecl` (che la aggiunge al contesto locale) e `instantiate1` (che sostituisce il de Bruijn index 0 con la nuova variabile libera).

Costante `And.intro` — regola $\wedge_i$.

```
| (.const `And.intro _), arg0::arg1::_ ⇒
  return .node hyps formula "∧I"
    [ ← arg0.toNDTreeM ... ,
      ← arg1.toNDTreeM ... ]
```

Il costruttore `And.intro : A → B → A ∧ B` porta due argomenti proposizionali: la dimostrazione di `A` e quella di `B`. Il pattern matching si assicura che gli argomenti di tipo (`A` e `B` stessi, che sarebbero i primi due argomenti nel termine polimorfico) siano già stati filtrati dal passo 3 della pipeline.

Costante `Or.casesOn` — regola $\vee_e$. L'eliminazione della disgiunzione è il caso più articolato, perché la continuazione è rappresentata da due funzioni lambda (una per il caso sinistro, una per quello destro):

```
| (.const `Or.casesOn _),
    orArg::(.lam nl tl bl bil)::(.lam nr tr br bir)::_ ⇒ do
  let childL ← 
    withLocalDecl nl bil tl fun fvl ⇒ do
      let bl' := bl.instantiate1 fvl
      bl'.toNDTreeM ...
  let childR ← 
    withLocalDecl nr bir tr fun fvr ⇒ do
      let br' := br.instantiate1 fvr
      br'.toNDTreeM ...
  return .node hyps formula "vE"
    [ ← orArg.toNDTreeM ... , childL, childR ]
```

Viene aperto un contesto locale separato per ciascun ramo, aggiungendo l'ipotesi locale corrispondente (`nl : tl` per il caso `Or.inl`, `nr : tr` per il caso `Or.inr`). Sono gestiti anche i casi in cui uno dei rami non è una lambda (che non corrisponde naturalmente a un passaggio di deduzione naturale).

Metavariabile aperta (`.mvar`) — goal non dimostrato.

```
| (.mvar mmmid), _ ⇒ do
  mmmid.withContext do
    match currentGoal with
    | none ⇒
      return .openNode
        (← getHypoteses (rootHyps := rootHyps))
        formula false
    | some mvar ⇒
      return .openNode
        (← getHypoteses (rootHyps := rootHyps))
        formula (mmmid == mvar)
```

Una metavariabile non assegnata corrisponde a un goal ancora aperto nella dimostrazione. Il contesto della metavariabile viene attivato tramite `mmmid.withContext`, e il booleano `isCurrentGoal` viene impostato a `true` solo se la metavariabile coincide con quella passata co-

me `currentGoal` (il goal attivo al cursore), permettendo al frontend di evidenziare il goal corrente.

Variabile libera con argomenti (`.fvar args`) — $\to_e$ e $\forall_e$ iterati. Quando un'ipotesi locale viene applicata a uno o più argomenti, si tratta di una sequenza di eliminazioni. Il codice le costruisce iterativamente:

```
| (.fvar _), _::_ ⇒ do
  let baseNode := (← fn.toNDTreeM ...)
  let (_, finalNode) ← args'.foldlM (
    fun (accApp, accNode) nextArg ⇒ do
      let newApp := .app accApp nextArg
      match (← inferType (← inferType nextArg)) with
      | (.sort 0) ⇒   -- argomento proposizionale → →E
        let newNode := .node hyps
          s!"{← ppExpr (← inferType newApp)}"
          "→E"
          [accNode, (← nextArg.toNDTreeM ...)]
        return (newApp, newNode)
      | (.sort (.succ 0)) ⇒  -- arg. di tipo → ∀E
        let newNode := .node hyps
          s!"{← ppExpr (← inferType newApp)}"
          "∀E" [accNode]
        return (newApp, newNode)
      | _ ⇒ ...
  ) (fn, baseNode)
  return finalNode
```

Per ogni argomento, si distingue se esso è una proposizione (`Sort 0`, ossia `Prop`) — nel qual caso si genera un nodo $\to_e$ con due figli — oppure un termine di tipo (`Sort 1`) — nel qual caso si genera un nodo $\forall_e$ con un solo figlio, dato che l'istanza del tipo non è un sotto-albero logicamente significativo.

Caso residuale. Per le costanti non riconosciute esplicitamente, viene generato un nodo generico con il nome della costante come etichetta della regola e tutti gli argomenti come figli:

```
| fn, args ⇒ do
  let children ← args.mapM fun arg ⇒ arg.toNDTreeM ...
  match fn with
  | .const (.str _ n) _ ⇒
      return .node hyps formula s!"{n.toName}" children
  | _ ⇒
      return .node hyps formula s!"{← ppExpr fn}" children
```

Questo caso gestisce, per esempio, lemmi dimostrati al di fuori del contesto corrente (teoremi già dimostrati usati come foglie dell'albero) o costrutti non ancora coperti dalla traduzione.

3.1.5 Serializzazione JSON

La serializzazione avviene tramite il metodo `NDTree.toJson`, che produce manualmente una stringa JSON anziché utilizzare la derivazione automatica di `ToJson`. Questa scelta è motivata da due ragioni:

1. Il meccanismo di *deriving* produce un JSON con chiavi non controllabili, mentre il frontend React si aspetta una struttura precisa (con i campi `type`, `formula`, `rule`, `children`, `hypotheses`, ecc.).

2. I nomi delle variabili Lean possono contenere caratteri Unicode e caratteri di controllo (in particolare i nomi *hygienici* generati automaticamente dal sistema di macro contengono sequenze come `_@` e `_hyg`): lato JavaScript, questi vengono rimossi con una pulizia preventiva della stringa (`replace(/[\x00-\x1F\x7F-\x9F]/g, '')`) prima del parsing JSON.

I nomi *igienici* vengono inoltre soppressi lato Lean tramite la funzione `Name.isHygName`, che li riconosce ed elimina il nome, cosicché lato JavaScript il nome non venga visualizzato, rendendo le ipotesi generate automaticamente da Lean più leggibili.

3.2 Frontend: rendering dell'albero in React

3.2.1 Struttura del componente

Il componente principale `NDTreeViewer` gestisce il ciclo di vita dell'interazione:

1. All'avvio e ad ogni cambio di `props.pos`, effettua la chiamata RPC `getTreeAsJson` e salva il risultato nello stato React `result`.
2. Lo stato `sharedHypotheses` mantiene la lista di ipotesi dell'ultimo nodo cliccato dall'utente, distribuita ai componenti figli tramite il contesto React `HypothesesContext`.
3. Lo stato `selectedHypTree` mantiene l'eventuale sotto-albero dell'ipotesi selezionata (per le ipotesi che sono esse stesse prove strutturate, non semplici foglie).

3.2.2 Il componente `NDTree`

Il componente `NDTree` è un dispatcher che, in base al campo `type` del nodo JSON, istanzia il componente corretto:

```
1 const NDTree = ({ tree, disableClick, uniqueId }) ⇒
  {
2   switch (tree?.type) {
3     case "leaf":     return <Leaf     {...tree}
      ... />;
4     case "node":     return <Node     {...tree}
      ... />;
5     case "openNode": return <OpenNode {...tree}
      ... />;
6     default:         return <Unhandled
      uniqueId={uniqueId}/>;
```

```
  }
};
```

Ogni nodo riceve un `uniqueId` generato dalla concatenazione degli indici di percorso (es. `"node010"` indica il primo figlio del secondo figlio della radice): questo identificatore viene usato per interrogare il DOM e calcolare le dimensioni delle barre orizzontali.

3.2.3 Calcolo dinamico delle barre orizzontali

Una delle sfide principali del rendering è il calcolo della larghezza della barra orizzontale che separa premesse e conclusione in un nodo interno. La barra deve essere larga quanto il massimo tra la larghezza della formula di conclusione e la larghezza complessiva occupata dalle formule che sono le radici dei sottoalberi delle premesse:

```
useLayoutEffect(() => {
  const firstFormula = document
    .querySelector(`#${uniqueId}0 > .formula`)
    .getBoundingClientRect();
  const lastFormula = document
    .querySelector(`#${uniqueId}${children.length-1}
    > .formula`)
    .getBoundingClientRect();
  const parentFormula = document
    .querySelector(`#${uniqueId} > .formula`)
    .getBoundingClientRect();

  const lineWidth = Math.max(
    lastFormula.right - firstFormula.left,
    parentFormula.width,
  );
  const marginLeft =
```

```
      firstFormula.left -
      (parentFormula.left + parentFormula.width/2 -
      lineWidth/2);

    setLineInfo({ width: lineWidth, marginLeft: ...
    });
}, []);
```

Si usa `useLayoutEffect` (anziché `useEffect`) per garantire che il calcolo avvenga *dopo* che il DOM è stato aggiornato ma *prima* che il browser dipinga il frame, evitando un flickering visivo. Le coordinate vengono lette con `getBoundingClientRect()` sulle formule del primo e dell'ultimo figlio e sulla formula di conclusione, e la larghezza e il margine sinistro della barra vengono calcolati di conseguenza.

3.2.4 Gestione delle ipotesi e interattività

Quando l'utente clicca su un nodo (foglia, nodo interno o goal aperto), il componente aggiorna il contesto `HypothesesContext` con la lista delle ipotesi di quel nodo. Il componente `HypothesesDisplay` mostra queste ipotesi come *badge* cliccabili sopra l'albero principale.

Per le ipotesi che sono esse stesse sotto-prove strutturate (introdotte da **`have`**), cliccando sul badge corrispondente viene espanso un secondo albero — il sotto-albero dell'ipotesi — mostrato in un riquadro separato sopra l'albero principale. Questo meccanismo permette di navigare le prove che fanno uso di lemmi intermedi senza perdere il contesto dell'albero principale.

Il goal corrente (l'open node con `isCurrentGoal = true`) viene evidenziato automaticamente al caricamento dell'albero tramite `useEffect`:

```
useEffect(() => {
  if (isCurrentGoal) {
    handleClick();  // seleziona automaticamente il
    goal corrente
```

```
  }}, []);
```

3.2.5 Trattamento dei caratteri non stampabili

Prima di passare la stringa JSON al parser, il frontend applica una pulizia dei caratteri di controllo:

```
const validJSON =
  res.treeJson.replace(/[\x00-\x1F\x7F-\x9F]/g, "");
const tree = JSON.parse(validJSON);
```

Questa operazione è necessaria perché alcune formule prodotte da `ppExpr` possono contenere caratteri di escape interni ai nomi delle variabili Lean (in particolare nei nomi generati automaticamente dalle macro), che renderebbero il JSON non valido secondo lo standard RFC 8259.

Conclusioni

Validazione

Il sistema è stato validato manualmente su un corpus di esercizi proposti negli anni accademici precedenti del corso di *Logica per l'informatica*. Il corpus comprende 28 dimostrazioni distinte

Il widget sarà introdotto durante le esercitazioni di laboratorio del corrente anno accademico, costituendo la prima validazione su studenti reali. L'esperienza sul campo permetterà di raccogliere feedback su casi non gestiti e priorità di miglioramento.

Quantificazione del lavoro

Lo sviluppo del progetto si è esteso per circa due mesi, con un impegno complessivo stimato in 200 ore di lavoro effettivo.

Dal punto di vista della quantità di codice prodotto, il progetto si compone di:

- **474 righe** di codice Lean (`DeductionTreeWidget.lean`), che costituiscono il backend del sistema.
- **650 righe** di codice JavaScript/React (`NDTreeJsonViewer.js`), che costituiscono il frontend.

Le principali tecnologie e conoscenze acquisite nel corso del progetto includono:

- **Metaprogrammazione in Lean 4**: il progetto ha richiesto uno studio approfondito delle API interne di Lean, in particolare delle monadi `MetaM` e `RequestM`, del funzionamento dell'*info tree*, del *metavariable context* e delle operazioni di ispezione del contesto locale. Non essendo queste API documentate in modo esaustivo, buona parte dell'apprendimento è avvenuta per esplorazione diretta del codice sorgente di Lean e di Mathlib.

- **Teoria dei tipi dipendenti**: la necessità di distinguere tra argomenti proposizionali e argomenti di tipo, e di gestire correttamente le variabili legate e libere nel λ-calcolo, ha reso necessario un approfondimento della semantica del Calcolo delle Costruzioni e dell'isomorfismo di Curry-Howard a un livello più tecnico di quanto fosse richiesto dalla sola conoscenza del corso.

- **Architettura LSP e widget**: lo studio del protocollo di comunicazione tra il Language Server e l'infoview di VS Code, incluso il meccanismo dei widget e delle chiamate RPC, è stato completamente nuovo e non coperto dalla documentazione ufficiale di Lean, richiedendo l'analisi di esempi esistenti nel repository ufficiale.

- **React e rendering DOM**: la gestione del layout degli alberi — in particolare il calcolo dinamico delle larghezze delle barre orizzontali tramite `getBoundingClientRect` e `useLayoutEffect` — ha richiesto una comprensione approfondita del ciclo di vita dei componenti React e delle tempistiche di aggiornamento del DOM.

L'esperienza ha rappresentato un percorso formativo su più livelli simultaneamente: logico (teoria dei tipi, isomorfismo di Curry-Howard), tecnico (metaprogrammazione, protocolli LSP) e ingegneristico (progettazione di un'interfaccia interattiva). La difficoltà principale non è stata la complessità algoritmica del singolo passo, bensì l'integrazione di componenti molto eterogenei — Lean, il Language Server, JavaScript, React — che comunicano attraverso canali non completamente documentati.

Lavori futuri

Il progetto costituisce una prima versione funzionante del widget, ma lascia aperte diverse direzioni di sviluppo:

- **Copertura di ulteriori pattern**: alcuni costrutti Lean non sono ancora tradotti in regole riconoscibili. Un'estensione sistematica della funzione `toNDTreeM` potrebbe coprire la grande maggioranza dei casi che si incontrano negli esercizi del corso.
- **Integrazione come package ufficiale**: il widget potrebbe essere distribuito come package Lean separato (ad esempio su `lake`), rendendo più semplice l'adozione da parte di altri corsi o istituzioni che usano Lean a scopo didattico.

Appendice A

Codice sorgente principale

É possibile trovare il codice in versione integrale all'indirizzo `https://github.com/BaviTriesCoding/NDTreeViewer`. Il codice sotto riportato può essere visionato nel commit `154f4cf`.

DeductionTreeWidget.lean

```
import DeductionTreeWidget.ExprInfo
import Lean
import Lean.Server.Rpc.Basic
open Lean Meta Server Widget exprInfo
-- ═══════════════════════════════════════════════════════════════
-- Natural Deduction Tree Type
-- ═══════════════════════════════════════════════════════════════

abbrev Formula := String -- Quello che c'è scritto nelle foglie
abbrev Rule := String -- ¬i, ∧e1, ∧e2, ∧i, vi1, vi2, ve, →i,
→e, etc.
abbrev isDischarged := Bool --Il booleano è per capire se è una
foglia aperta o scaricata
abbrev isCurrentGoal := Bool
abbrev Proof := String -- Nei nodi unhandled è la prova non
riconosciuta

```

```
15 inductive NDTree where
16   | leaf      : List (Name × NDTree) → Name → Formula →
     isDischarged → NDTree
17   | node      : List (Name × NDTree) → Formula → Rule → List
     NDTree → NDTree
18   | openNode  : List (Name × NDTree) → Formula → isCurrentGoal
     → NDTree
19   | unhandled : NDTree
20   deriving FromJson, ToJson, Server.RpcEncodable, BEq
21
22 def NDTree.isLeaf: NDTree → Bool
23   | .leaf _ _ _ _ ⇒ true
24   | _ ⇒ false
25
26 def NDTree.isNode : NDTree →  Bool
27   | .node _ _ _ _ ⇒ true
28   | _ ⇒ false
29
30 def NDTree.isOpenNode : NDTree → Bool
31   | .openNode _ _ _ ⇒ true
32   | _ ⇒ false
33
34 def NDTree.isUnhandled : NDTree → Bool
35   | .unhandled ⇒ true
36   | _ ⇒ false
37
38
39 abbrev Hyp := (Name × NDTree)
40
41 def Lean.Name.isHygName (n : Name) : Bool := if
   (n.toString.contains "_@" || n.toString.contains "_hyg") then
   true else false
42
43 partial def NDTree.toJson : NDTree → String
44   | .leaf hypotheses name f isDischarged ⇒
45     "{
46       \"type\":\"leaf\",
```

```
    \"hypotheses\": [" ++ ", ".intercalate (hypotheses.map
    fun ⟨name, value⟩ ⇒
    "{" ++
      (if !name.isHygName then "\"name\":\"" ++ name.toString
      ++ "\"," else "")
      ++ "\"value\":" ++ value.toJson ++ "
    }") ++ "],"
    ++
    (if !name.isHygName then "\"name\":\"" ++ name.toString
    ++ "\"," else "")
    ++
    "\"formula\":\""++ f ++"\",
    \"isDischarged\":" ++ s!"{repr isDischarged}" ++ "
  }"
| .node hypotheses f rule children ⇒
  "{
    \"type\":\"node\",
    \"hypotheses\": [" ++ ", ".intercalate (hypotheses.map
    fun ⟨name, value⟩ ⇒
    "{" ++
      (if !name.isHygName then "\"name\":\"" ++ name.toString
      ++ "\"," else "")
      ++ "\"value\":" ++ value.toJson ++ "
    }") ++ "],
    \"formula\":\"" ++ f ++ "\",
    \"rule\": \"" ++ rule ++ "\",
    \"children\": [" ++ ", ".intercalate (children.map
    toJson) ++ "]
  }"
| .openNode hypotheses f isCurrentGoal ⇒
  "{
    \"type\":\"openNode\",
    \"hypotheses\": [" ++ ", ".intercalate (hypotheses.map
    fun ⟨name, value⟩ ⇒
    "{" ++
      (if !name.isHygName then "\"name\":\"" ++ name.toString
      ++ "\"," else "")
      ++ "\"value\":" ++ value.toJson ++ "
```

```
      }") ++ "],
      \"formula\":\"" ++ f ++ "\",
      \"isCurrentGoal\":" ++ s!"{repr isCurrentGoal}" ++ "
    }"
  | .unhandled  ⇒
    "{
      \"type\":\"unhandled\"
    }"


def isTreeDischarged (t : NDTree) : Bool :=
  match t with
  | .leaf hyps name formula _ ⇒
    hyps.foldl (fun res (name', value) ⇒
      match value, res with
      | _, true ⇒ true
      | .leaf _ _ formula' _ , false ⇒
        name == name' && formula == formula'
      | _, _ ⇒ false
    ) false
  | _ ⇒ false


def Lean.Expr.isSort1 (e: Expr) : Bool :=
  match e with
  | Expr.sort 1 ⇒ true
  | _ ⇒ false

def reprLCtx (hyps : List Hyp) : MetaM String := do
  return ", ".intercalate (hyps.map fun hyp ⇒
    match hyp with
    | (n, .leaf _ _ f _) ⇒ s!"{n} : {f}"
    | (n, .node _ f _ _) ⇒ s!"{n} : {f}"
    | _ ⇒ ""
  )

mutual
```

```
114 partial def getHypoteses (rootHyps : List Hyp := []) (isLHS :
    Bool := false) : MetaM (List Hyp) := do
115   let list ← (← getLCtx).decls.foldlM (fun s i ⇒ do
116     match s, i with
117     | [], _ ⇒ return [.none]
118     | _, none ⇒ return s
119     | _, some d ⇒
120         let t := (← inferType (.fvar d.fvarId))
121         if (← inferType t).isSort1 then return s
122         if rootHyps.contains (d.userName, (NDTree.leaf []
            d.userName s!"{← ppExpr t}" false)) then
123           return s
124         match d.value? (allowNondep := true) with
125         | none ⇒ do
126           let leaf :NDTree := .leaf [] d.userName s!"{← ppExpr
            t}" (!isLHS && !(← rootHyps.foldlM (fun res h ⇒
127             match res, h with
128             | true, _ ⇒ return true
129             | false, (name, .leaf _ _ f _) ⇒ return  name ==
              d.userName && f == s!"{← ppExpr t}"
130             | _, _ ⇒ return false
131           ) false ))
132           return s ++ [.some (d.userName, leaf)]
133         | some v ⇒ do
134           return s ++ [.some (d.userName, (← v.toNDTreeM
            (withHyp := false) (rootHyps := rootHyps)))])
135     []
136
137   return list.filterMap id
138
139
140 partial def Lean.Expr.toNDTreeM (e : Expr) (withHyp := true)
    (rootHyps : List Hyp := []) (currentGoal : Option MVarId :=
    none) : MetaM NDTree := do
141   withAggressiveInstantiateMVars e fun e ⇒ do
142   let (fn, args') := e.withApp fun e a ⇒ (e, a.toList)
143   let args ← args'.filterMapM (fun arg ⇒ do
144     return match (← inferType (← inferType arg)) with
```

```
145     | .sort (.succ _) ⇒ none
146     | _ ⇒ some arg)
147   let hyps ← if withHyp then getHypoteses (rootHyps :=
      rootHyps) else pure []
148   let formula := s!"{← ppExpr (← inferType e)}"
149   match fn, args with
150
151   -- ↓ casi normalmente impossibili ↓
152
153   | (.forallE _ _ _ _ ), _  ⇒ throwError ".forallE unexpected
      in a proof"
154
155   | (.bvar _), _            ⇒ throwError ".bvar unexpected in
      a proof"
156
157   | (.sort _), _            ⇒ throwError ".sort unexpected in
      a proof"
158
159   | (.lit _), _             ⇒ throwError ".lit unexpected in a
      proof"
160
161   -- ↓ foglie ↓
162
163   | (.const `True.intro _), _ ⇒ do
164     return .node hyps formula "⊤I" []
165
166   | (.const (.str _ n) _), [] ⇒ return .node hyps formula n []
167
168
169   | (.fvar id), [] ⇒ do
170       let decl ← Meta.getFVarLocalDecl (.fvar id)
171       match decl.value? with
172       | none ⇒ return .leaf hyps decl.userName (toString (←
          ppExpr decl.type)) ( isTreeDischarged (.leaf hyps
          decl.userName (toString (← ppExpr decl.type)) false))
173       | some v ⇒ return ← v.toNDTreeM (withHyp := withHyp)
          (rootHyps := rootHyps) (currentGoal := currentGoal)
174
```

```
175
176 | (.const `sorryAx _), _ ⇒ return .leaf hyps `sorryAx
    "sorry" false
177
178 -- ↓ nodi aperti ↓
179
180 | (.mvar mmmid), _ ⇒ do
181   mmmid.withContext do
182     match currentGoal with
183     | none ⇒ return .openNode (← if withHyp then do
        getHypoteses (rootHyps := rootHyps) else pure []) formula
        false
184     | some mvar ⇒ return .openNode (← if withHyp then do
        getHypoteses (rootHyps := rootHyps) else pure []) formula
        (mmmid == mvar)
185 -- ↓ nodi ↓
186
187 | (.letE n t v b _), _ ⇒ do
188   withLetDecl n t v fun fv ⇒
189     let b' := b.instantiate1 fv
190
191       return ← b'.toNDTreeM (withHyp := withHyp) (rootHyps :=
        rootHyps) (currentGoal := currentGoal)
192
193 | (.lam n t b bi), _ ⇒ do
194   withLocalDecl n bi t fun fv ⇒ do
195     let b := b.instantiate1 fv
196     let ruleName := if (← inferType t).isProp then "→I" else
      "∀I"
197     let child ← b.toNDTreeM (withHyp := withHyp) (rootHyps :=
      rootHyps) (currentGoal := currentGoal)
198     return .node hyps s!"{← ppExpr (← inferType fn)}"
      ruleName [child]
199
200 | (.fvar _), _::_ ⇒ do
201   let baseNode := (← fn.toNDTreeM (withHyp := withHyp)
    (rootHyps := rootHyps) (currentGoal := currentGoal))
202   let (_, finalNode) ← args'.foldlM (
```

```
203       fun (accApp, accNode) nextArg ⇒ do
204       let newApp  := .app accApp nextArg
205       match (← inferType (← inferType nextArg)) with
206       | (.sort 0) ⇒
207         let newNode := .node hyps s!"{← ppExpr (← inferType
          newApp)}" "→E" [accNode, (← nextArg.toNDTreeM (withHyp
          := withHyp) (rootHyps := rootHyps) (currentGoal :=
          currentGoal))]
208         return (newApp, newNode)
209       | (.sort (.succ 0)) ⇒
210         let newNode := .node hyps s!"{← ppExpr (← inferType
          newApp)}" "∀E" [accNode]
211         return (newApp, newNode)
212       | _ ⇒
213         let newNode := (← nextArg.toNDTreeM (withHyp :=
          withHyp) (rootHyps := rootHyps) (currentGoal :=
          currentGoal))
214         return (newApp, newNode)
215     ) (fn, baseNode)
216     return finalNode
217
218   | (.const `And.intro _), arg0::arg1::_ ⇒
219     return .node hyps formula "∧I" [← arg0.toNDTreeM (withHyp
      := withHyp) (rootHyps := rootHyps) (currentGoal :=
      currentGoal), ← arg1.toNDTreeM (withHyp := withHyp)
      (rootHyps := rootHyps) (currentGoal := currentGoal)]
220
221
222   | (.const `And.casesOn _), andArg::(.lam n t (.lam n' t' b
  bi') bi)::_ ⇒ do
223     withLocalDecl n bi t fun fv ⇒ do
224       let b' := b.instantiate1 fv
225       withLocalDecl n' bi' t' fun fv' ⇒ do
226         let b'' := b'.instantiate1 fv'
```

```
227         return .node hyps formula "∧E" [← andArg.toNDTreeM
            (withHyp := withHyp) (rootHyps := rootHyps)
            (currentGoal := currentGoal), ← b''.toNDTreeM (withHyp
            := withHyp) (rootHyps := rootHyps) (currentGoal :=
            currentGoal)]
228
229   | (.const `And.casesOn _), andArg::(.lam n t b bi)::_ ⇒ do
230     withLocalDecl n bi t fun fv ⇒ do
231       let b' := b.instantiate1 fv
232       return .node hyps formula "∧E" [← andArg.toNDTreeM
            (withHyp := withHyp) (rootHyps := rootHyps) (currentGoal
            := currentGoal), ← b'.toNDTreeM (withHyp := withHyp)
            (rootHyps := rootHyps) (currentGoal := currentGoal)]
233
234   | (.const `And.casesOn _), andArg::arg0::_ ⇒ do
235     return .node hyps formula "∧E" [← andArg.toNDTreeM (withHyp
      := withHyp) (rootHyps := rootHyps) (currentGoal :=
      currentGoal), ← arg0.toNDTreeM (withHyp := withHyp)
      (rootHyps := rootHyps) (currentGoal := currentGoal)]
236   | (.const `And.left _), arg::_ ⇒
237     return .node hyps formula "∧E₁" [← arg.toNDTreeM (withHyp
      := withHyp) (rootHyps := rootHyps) (currentGoal :=
      currentGoal)]
238
239   | (.const `And.right _), arg::_ ⇒
240     return .node hyps formula "∧E₂" [← arg.toNDTreeM (withHyp
      := withHyp) (rootHyps := rootHyps) (currentGoal :=
      currentGoal)]
241
242   | (.const `Or.inl _), arg::_ ⇒
243     return .node hyps formula "∨I₁" [← arg.toNDTreeM (withHyp
      := withHyp) (rootHyps := rootHyps) (currentGoal :=
      currentGoal)]
244
245   | (.const `Or.inr _), arg::_ ⇒
246     return .node hyps formula "∨I₂" [← arg.toNDTreeM (withHyp
      := withHyp) (rootHyps := rootHyps) (currentGoal :=
      currentGoal)]
```

```
247
248 | (.const `Or.casesOn _), orArg::(.lam nl tl bl bil)::(.lam
    nr tr br bir)::_ ⇒ do
249   let childL ← withLocalDecl nl bil tl fun fvl ⇒ do
250     let bl' := bl.instantiate1 fvl
251     bl'.toNDTreeM (withHyp := withHyp) (rootHyps := rootHyps)
      (currentGoal := currentGoal)
252   let childR ← withLocalDecl nr bir tr fun fvr ⇒ do
253     let br' := br.instantiate1 fvr
254     br'.toNDTreeM (withHyp := withHyp) (rootHyps := rootHyps)
      (currentGoal := currentGoal)
255   return .node hyps formula "vE" [← orArg.toNDTreeM (withHyp
    := withHyp) (rootHyps := rootHyps) (currentGoal :=
    currentGoal), childL, childR]
256
257 | (.const `Or.casesOn _), orArg::arg0::(.lam nr tr br bir)::_
    ⇒ do
258   let childR ← withLocalDecl nr bir tr fun fvr ⇒ do
259     let br' := br.instantiate1 fvr
260     br'.toNDTreeM (withHyp := withHyp) (rootHyps := rootHyps)
      (currentGoal := currentGoal)
261   return .node hyps formula "vE" [← orArg.toNDTreeM (withHyp
    := withHyp) (rootHyps := rootHyps) (currentGoal :=
    currentGoal), ← arg0.toNDTreeM (withHyp := withHyp)
    (rootHyps := rootHyps) (currentGoal := currentGoal),
    childR]
262
263 | (.const `Or.casesOn _), orArg::(.lam nl tl bl bil)::arg1::_
    ⇒ do
264   let childL ← withLocalDecl nl bil tl fun fvl ⇒ do
265     let bl' := bl.instantiate1 fvl
266     bl'.toNDTreeM (withHyp := withHyp) (rootHyps := rootHyps)
      (currentGoal := currentGoal)
267   return .node hyps formula "vE" [← orArg.toNDTreeM (withHyp
    := withHyp) (rootHyps := rootHyps) (currentGoal :=
    currentGoal), childL, ← arg1.toNDTreeM (withHyp := withHyp)
    (rootHyps := rootHyps) (currentGoal := currentGoal)]
268
```

```
269   | (.const `Not.intro _), arg::_ ⇒
270     return .node hyps formula "¬I" [← arg.toNDTreeM (withHyp :=
      withHyp) (rootHyps := rootHyps) (currentGoal :=
      currentGoal)]
271
272   | (.const `absurd _), arg0::arg1::_ ⇒
273     return .node hyps formula "¬E" [← arg0.toNDTreeM (withHyp
      := withHyp) (rootHyps := rootHyps) (currentGoal :=
      currentGoal), ← arg1.toNDTreeM (withHyp := withHyp)
      (rootHyps := rootHyps) (currentGoal := currentGoal)]
274
275   | (.const `False.elim _), arg::_ ⇒
276     return .node hyps formula "⊥E" [← arg.toNDTreeM (withHyp :=
      withHyp) (rootHyps := rootHyps) (currentGoal :=
      currentGoal)]
277
278   | (.const `Exists.intro _), arg::_ ⇒
279     return .node hyps formula "∃I" [← arg.toNDTreeM (withHyp :=
      withHyp) (rootHyps := rootHyps) (currentGoal :=
      currentGoal)]
280
281   | (.const `Exists.elim _), arg0::arg1::_ ⇒
282     return .node hyps formula "∃E" [← arg0.toNDTreeM (withHyp
      := withHyp) (rootHyps := rootHyps) (currentGoal :=
      currentGoal), ← arg1.toNDTreeM (withHyp := withHyp)
      (rootHyps := rootHyps) (currentGoal := currentGoal)]
283
284   | (.const `Exists.casesOn _), exArg::(.lam n t (.lam n' t' b
  bi') bi)::_ ⇒ do
285     withLocalDecl n bi t fun fv ⇒ do
286       let b' := b.instantiate1 fv
287       withLocalDecl n' bi' t' fun fv' ⇒ do
288         let b'' := b'.instantiate1 fv'
289         return .node hyps formula "∃E" [
290           ← exArg.toNDTreeM (withHyp := withHyp) (rootHyps :=
            rootHyps) (currentGoal := currentGoal),
291           ← b''.toNDTreeM (withHyp := withHyp) (rootHyps :=
            rootHyps) (currentGoal := currentGoal)]
```

```
292
293   | (.const `Exists.casesOn _), exArg::(.lam n t b bi)::_ ⇒ do
294     withLocalDecl n bi t fun fv ⇒ do
295       let b' := b.instantiate1 fv
296       return .node hyps formula "∃E" [
297         ← exArg.toNDTreeM (withHyp := withHyp) (rootHyps :=
          rootHyps) (currentGoal := currentGoal),
298         ← b'.toNDTreeM (withHyp := withHyp) (rootHyps :=
          rootHyps) (currentGoal := currentGoal)]
299
300   | (.const `Exists.casesOn _), arg0::arg1::_ ⇒
301     return .node hyps formula "∃E" [← arg0.toNDTreeM (withHyp
      := withHyp) (rootHyps := rootHyps) (currentGoal :=
      currentGoal), ← arg1.toNDTreeM (withHyp := withHyp)
      (rootHyps := rootHyps) (currentGoal := currentGoal)]
302
303   | (.const `Iff.intro _), arg0::arg1::_ ⇒ do
304     return .node hyps formula "↔I" [← arg0.toNDTreeM (withHyp
      := withHyp) (rootHyps := rootHyps) (currentGoal :=
      currentGoal), ← arg1.toNDTreeM (withHyp := withHyp)
      (rootHyps := rootHyps) (currentGoal := currentGoal)]
305
306   | (.const `Iff.mp _), arg::_ ⇒ do
307     return .node hyps formula "↔E₁" [← arg.toNDTreeM (withHyp
      := withHyp) (rootHyps := rootHyps) (currentGoal :=
      currentGoal)]
308
309   | (.const `Iff.mpr _), arg::_ ⇒ do
310     return .node hyps formula "↔E₂" [← arg.toNDTreeM (withHyp
      := withHyp) (rootHyps := rootHyps) (currentGoal :=
      currentGoal)]
311
312   | (.const `matita.iff_e _), arg::_ ⇒ do
313     return .node hyps formula "↔E" [← arg.toNDTreeM (withHyp :=
      withHyp) (rootHyps := rootHyps) (currentGoal :=
      currentGoal)]
314
315   | (.const `False.casesOn _), _ ⇒ do
```

```
316       let children ← args.mapM fun arg ⇒  arg.toNDTreeM (withHyp
          := withHyp) (rootHyps := rootHyps) (currentGoal :=
          currentGoal)
317       return .node hyps formula "⊥E" (children)
318
319     | (.const `True.casesOn _), _ ⇒ do
320       let children ← args.mapM fun arg ⇒  arg.toNDTreeM (withHyp
          := withHyp) (rootHyps := rootHyps) (currentGoal :=
          currentGoal)
321       return .node hyps formula "⊤E" (children)
322
323     | fn, args ⇒ do
324         let children ← args.mapM fun arg ⇒  arg.toNDTreeM
          (withHyp := withHyp) (rootHyps := rootHyps) (currentGoal
          := currentGoal)
325         match fn with
326         | .const (.str _ n) _ ⇒ return .node hyps formula
          s!"{n.toName}" (children)
327         | _ ⇒ return .node hyps formula s!"{← ppExpr fn}"
          (children)
328
329 end
330
331 -- ═══════════════════════════════════════════════════════════
332 -- RPC METHOD: GET TREE AS JSON
333 -- ═══════════════════════════════════════════════════════════
334
335 structure DebugLogParams where
336   msg : String
337   deriving FromJson, ToJson
338
339 structure DebugLogResult where
340   msg : String
341   deriving FromJson, ToJson, Server.RpcEncodable
342
343 open Lean Server RequestM in
344 @[server_rpc_method]
```

```
345 def debugLog (p : DebugLogParams) : RequestM (RequestTask
    DebugLogResult) := do
346   dbg_trace "[DEBUG] {p.msg}"
347   asTask do return {msg := s!"Logged: {p.msg}"}
348
349
350 structure DeductionAtCursorParams where
351   pos : Lsp.Position
352   deriving FromJson, ToJson
353
354 structure TreeAsJsonResult where
355   thmName : String
356   thmType : String
357   treeJson : String
358   deriving FromJson, ToJson, Server.RpcEncodable
359
360 open RequestM in
361 @[server_rpc_method]
362 def getTreeAsJson (params : DeductionAtCursorParams) :
363     RequestM (RequestTask TreeAsJsonResult) :=
364   withWaitFindSnapAtPos params.pos fun snap ⇒ do
365     runTermElabM snap do
366     let doc ← readDoc
367     let txt := doc.meta.text
368     let pos := txt.lspPosToUtf8Pos params.pos
369     let l := Elab.InfoTree.goalsAt? txt snap.infoTree pos
370     let [item] := l | throwError "Impossibile trovare il nodo a
    questa posizione"
371     let .some name := item.ctxInfo.parentDecl? | throwError
    "Impossibile trovare la dichiarazione"
372     let metavarctx := item.tacticInfo.mctxAfter
373     let mvar := item.tacticInfo.goalsAfter.head?
374     withMCtx metavarctx do
375       let younger : Name -> Name -> Bool
376       | .num _ 0, .num _ _ ⇒ false
377       | .num _ n, .num _ m ⇒ n < m
378       | _, _ ⇒ false
```

```
379       let mmmid := metavarctx.eAssignment.foldl (fun m d _ ⇒
          if younger m.name d.name then m else d)  (MVarId.mk (.num
          (.anonymous) 0))
380       let proofTerm := Expr.mvar mmmid
381       mmmid.withContext do
382         if proofTerm.isSort1 then throwError s!"Il termine
          trovato non è una prova: {← exprInfo proofTerm} : {←
          ppExpr (← inferType proofTerm)}"
383         let rootHyps ← getHypoteses (isLHS := true)
384         let tree ← (proofTerm.toNDTreeM (rootHyps := rootHyps)
          (currentGoal := mvar))
385         return { thmName := toString name, thmType := s!"{←
          reprLCtx rootHyps} ⊢ {← ppExpr (← inferType
          proofTerm)}", treeJson := tree.toJson }
386
387   -- ═══════════════════════════════════════════════════════════
388   -- WIDGET JAVASCRIPT
389   -- ═══════════════════════════════════════════════════════════
390
391   @[widget_module]
392   def NDTreeJsonViewerWidget : Widget.Module where
393     javascript := include_str "NDTreeJsonViewer.js"
```

NDTreeJsonViewer.js

```
import * as React from "react";
const { useState, useEffect, useContext, createContext, useLayoutEffect } =
  React;
import { useRpcSession, EditorContext, RpcContext } from "@leanprover/infoview";
// ──────────────────────────────────────────────────────────────────────────
// Style constants (invariati)
// ──────────────────────────────────────────────────────────────────────────
const versionInfo = `Loaded: ${new Date().toLocaleTimeString()}`;
const WHITE = "var(--vscode-foreground)";
const LIGHTER_GRAY = "var(--vscode-tab-inactiveForeground)";
const ACCENT = "var(--vscode-symbolIcon-eventForeground)";
const PRIMARY = "var(--vscode-textLink-activeForeground)";
const ERROR_RED = "var(--vscode-errorForeground)";

const EMBEDDED_STYLES = `
.rule:hover {
  background: var(--vscode-editor-background);
  color: ${WHITE} !important;
  font-size: .75rem !important;
  max-width: max-content !important;
  overflow: visible !important;
  text-overflow: clip !important;
  z-index: 100;
  border-radius: .25rem;
  outline: solid .05rem ${WHITE};
  position-anchor: --anchor;
  right: anchor(left) !important;
}`;

// ──────────────────────────────────────────────────────────────────────────
// HypothesesContext
// ──────────────────────────────────────────────────────────────────────────

```

```
34 const HypothesesContext = createContext({
35   sharedHypotheses: [],
36   setHypotheses: () => {},
37   selectedHypTree: null,
38   setSelectedHypTree: () => {},
39 });
40
41 const resultContext = createContext({
42   result: null,
43   setResult: () => {},
44 });
45
46 // ──────────────────────────────────────────────────────────────
47 // Natural Deduction Tree Node Renderer
48 // ──────────────────────────────────────────────────────────────
49
50 const Leaf = ({
51   hypotheses,
52   name,
53   formula,
54   isDischarged,
55   disableClick,
56   uniqueId,
57 }) => {
58   const { sharedHypotheses, setHypotheses, setSelectedHypTree }
     =
59     useContext(HypothesesContext);
60   const [selected, setSelected] = useState(false);
61
62   useEffect(() => {
63     if (selected && sharedHypotheses !== hypotheses) {
64       setSelected(false);
65     }
66   }, [sharedHypotheses]);
67
68   const handleClick = () => {
69     if (disableClick) return;
70     setSelected((prev) => {
```

```
      const newSelected = !prev;
      setHypotheses(newSelected ? (hypotheses ?? []) : null);
      setSelectedHypTree(null);
      return newSelected;
    });
  };

  return React.createElement(
    "div",
    {
      id: uniqueId,
      style: {
        display: "inline-flex",
        flexDirection: "column",
        alignItems: "center",
      },
    },
    React.createElement(
      "span",
      {
        className: "formula",
        style: {
          color: "#0ecb00",
          background: selected
            ? `color-mix(in srgb, ${ACCENT},
            rgba(255,255,255,0.1))`
            : "transparent",
          cursor: disableClick ? "default" : "pointer",
          whiteSpace: "nowrap",
          textAlign: "center",
          lineHeight: "1rem",
          padding: "0 0.5rem",
        },
        onClick: handleClick,
      },
      `${isDischarged ? "[" : ""}${name ? name + " : " :
      ""}${formula}${isDischarged ? "]" : ""}`,
    ),
```

```
  );
};

const Node = ({
  hypotheses,
  formula,
  rule,
  children,
  disableClick,
  uniqueId,
}) => {
  const {
    sharedHypotheses,
    setHypotheses,
    selectedHypTree,
    setSelectedHypTree,
  } = useContext(HypothesesContext);
  const { result } = useContext(resultContext);

  const [selected, setSelected] = useState(false);
  const [lineInfo, setLineInfo] = useState({
    width: 0,
    marginLeft: 0,
  });

  const rs = useRpcSession();

  useEffect(() => {
    if (selected && sharedHypotheses !== hypotheses)
    setSelected(false);
  }, [sharedHypotheses]);

  useLayoutEffect(() => {
    const firstFormula = document
      ?.querySelector(`#${uniqueId}0 > .formula`)
      ?.getBoundingClientRect();
    const lastFormula = document
```

```
      ?.querySelector(`#${uniqueId}${children.length - 1} > .formula`)
      ?.getBoundingClientRect();
    const parentFormula = document
      ?.querySelector(`#${uniqueId} > .formula`)
      ?.getBoundingClientRect();
    const lineWidth = Math.max(
      ((lastFormula?.right - firstFormula?.left) || 0),
      parentFormula?.width,
    );
    const marginLeft =
      firstFormula?.left -
      (parentFormula?.left + parentFormula?.width / 2 -
      lineWidth / 2) || 0;

    setLineInfo({
      width: lineWidth,
      marginLeft:
        parentFormula?.width >= ((lastFormula?.right -
        firstFormula?.left) || 0)
          ? 0
          : 2 * marginLeft,
    });
  }, []);

  const handleClick = () => {
    if (disableClick) return;
    setSelected((prev) => {
      const newSelected = !prev;
      setHypotheses(newSelected ? (hypotheses ?? []) : null);
      setSelectedHypTree(null);
      return newSelected;
    });
  };

  return React.createElement(
    "div",
    {
```

```
      id: uniqueId,
      style: {
        display: "inline-flex",
        flexDirection: "column",
        alignItems: "center",
        gap: "0.5rem",
        position: "relative",
      },
    },
    // ── Riga delle premesse ──────────────────────────────
    React.createElement(
      "div",
      {
        style: {
          display: "flex",
          flexDirection: "row",
          alignItems: "flex-end",
          gap: "3rem",
        },
      },
      children.map((child, i) ⇒
        React.createElement(NDTree, {
          key: i,
          tree: child,
          disableClick,
          uniqueId: uniqueId + i,
        }),
      ),
    ),

    // ── Barra + etichetta regola ─────────────────────────
    React.createElement(
      "div",
      {
        className: "line",
        style: {
```

```
        position: "relative",
        height: "0.125rem",
        borderRadius: "10rem",
        background: LIGHTER_GRAY,
        width: `${lineInfo.width}px`,
        marginLeft: `${lineInfo.marginLeft}px`,
        display: "flex",
        alignItems: "center",
      },
    },
    React.createElement(
      "span",
      {
        className: "rule",
        style: {
          position: "absolute",
          left: "100%",
          color: LIGHTER_GRAY,
          userSelect: "none",
          padding: "0.25rem",
          lineHeight: 1,
          fontSize: "0.5rem",
          maxWidth: "2.5rem",
          overflow: "hidden",
          whiteSpace: "nowrap",
          textOverflow: "ellipsis",
        },
      },
      rule,
    ),
  ),

  // ── Conclusione (cliccabile) ──────────────────────────
  React.createElement(
    "span",
    {
      className: "formula",
```

```
        style: {
          color: WHITE,
          background: selected
            ? `color-mix(in srgb, ${ACCENT},
            rgba(255,255,255,0.1))`
            : "transparent",
          cursor: disableClick ? "default" : "pointer",
          whiteSpace: "nowrap",
          textAlign: "center",
          lineHeight: "1rem",
          padding: "0 0.5rem",
        },
        onClick: handleClick,
      },
      formula,
    ),
  );
};

const OpenNode = ({
  hypotheses,
  formula,
  isCurrentGoal,
  disableClick,
  uniqueId,
}) => {
  const { sharedHypotheses, setHypotheses, setSelectedHypTree }
  =
    useContext(HypothesesContext);
  const [selected, setSelected] = useState(false);

  useEffect(() => {
    if (isCurrentGoal) {
      handleClick();
    }
  }, []);

  useEffect(() => {
```

```
    if (selected && sharedHypotheses !== hypotheses) {
      setSelected(false);
    }
  }, [sharedHypotheses]);

  const handleClick = () => {
    if (disableClick) return;
    setSelected((prev) => {
      const newSelected = !prev;
      setHypotheses(newSelected ? (hypotheses ?? []) : null);
      setSelectedHypTree(null);
      return newSelected;
    });
  };

  return React.createElement(
    "div",
    {
      id: uniqueId,
      style: {
        display: "inline-flex",
        flexDirection: "column",
        alignItems: "center",
      },
    },
    React.createElement(
      "span",
      {
        className: "formula",
        style: {
          color: WHITE,
          background: selected
            ? `color-mix(in srgb, ${ACCENT},
            rgba(255,255,255,0.1))`
            : "transparent",
          cursor: disableClick ? "default" : "pointer",
          whiteSpace: "nowrap",
          textAlign: "center",
```

```
          lineHeight: "1rem",
          padding: "0 0.5rem",
          outline: "dashed 0.0625rem",
          borderRadius: "0.25rem",
        },
        onClick: handleClick,
      },
      formula,
    ),
  );
};

const Unhandled = ({ uniqueId }) => {
  return React.createElement(
    "div",
    {
      id: uniqueId,
      style: {
        display: "inline-flex",
        flexDirection: "column",
        alignItems: "center",
      },
    },
    React.createElement(
      "span",
      {
        className: "formula",
        style: {
          color: ERROR_RED,
          whiteSpace: "nowrap",
          textAlign: "center",
          lineHeight: "1rem",
          padding: "0 0.5rem",
        },
      },
      "Unhandled",
    ),
  );
```

```
};

const NDTree = ({ tree, disableClick, uniqueId }) => {
  switch (tree?.type) {
    case "leaf":
      return React.createElement(Leaf, {
        ...tree,
        disableClick: disableClick,
        uniqueId: uniqueId,
      });
    case "node":
      return React.createElement(Node, {
        ...tree,
        disableClick: disableClick,
        uniqueId: uniqueId,
      });
    case "openNode":
      return React.createElement(OpenNode, {
        ...tree,
        disableClick: disableClick,
        uniqueId: uniqueId,
      });
    default:
      return React.createElement(Unhandled, { uniqueId:
      uniqueId });
  }
};

// ─────────────────────────────────────────────────────────
// HypothesesDisplay — mostra le ipotesi affiancate
// ─────────────────────────────────────────────────────────

const HypothesesDisplay = () => {
  const { sharedHypotheses, selectedHypTree, setSelectedHypTree
  } =
    useContext(HypothesesContext);

  const handleHypClick = (hyp, i) => {
```

```
    if (hyp.value.type === "leaf") return;
    if (selectedHypTree?.id === i) {
      // deselect
      setSelectedHypTree(null);
    } else {
      setSelectedHypTree({ id: i, value: hyp.value });
    }
  };

  return React.createElement(
    "div",
    {
      style: {
        display: "flex",
        flexDirection: "row",
        flexWrap: "wrap",
        alignItems: "center",
        gap: "0.5rem",
      },
    },
    sharedHypotheses.map((hyp, i) =>
      React.createElement(
        "span",
        {
          key: i,
          onClick: () => handleHypClick(hyp, i),
          style: {
            border: `0.0125rem solid ${selectedHypTree?.id ===
            i ? PRIMARY : LIGHTER_GRAY}`,
            borderRadius: "0.125rem",
            color: WHITE,
            cursor: hyp.value.type !== "leaf" ? "pointer" :
            "default",
            userSelect: "none",
            display: "flex",
            alignItems: "center",
            justifyContent: "center",
            whiteSpace: "nowrap",
```

```
            textAlign: "center",
            lineHeight: "1rem",
            padding: "0.25rem 0.5rem",
          },
        },
        React.createElement(
          "span",
          {
            style: {
              color: hyp.value.type === "leaf" ? "#0ecb00" :
              WHITE,
            },
          },
          `${hyp.value?.isDischarged ? "[" : ""}${hyp.name ?
          hyp.name + " : " :
          ""}${hyp.value.formula}${hyp.value?.isDischarged ?
          "]" : ""}`,
        ),
      ),
    ),
  );
};

// ─────────────────────────────────────────────────────────
// Main Widget
// ─────────────────────────────────────────────────────────

export default function NDTreeViewer(props) {
  const rs = useRpcSession();

  const [result, setResult] = useState(null);
  const [error, setError] = useState(null);
  const [sharedHypotheses, setHypotheses] = useState(null);
  const [selectedHypTree, setSelectedHypTree] = useState(null);

  // Inject embedded styles
  useEffect(() => {
    const styleEl = document.createElement("style");
```

```
468     styleEl.textContent = EMBEDDED_STYLES;
469     document.head.appendChild(styleEl);
470     return () => styleEl.remove();
471   }, []);
472
473   useEffect(() => {
474     const pos = props.pos;
475     if (!pos || !rs) return;
476     setError(null);
477     rs.call("getTreeAsJson", { pos })
478       .then((res) => {
479         const validJSON =
          res.treeJson.replace(/[\x00-\x1F\x7F-\x9F]/g, "");
480         const tree = JSON.parse(validJSON);
481         setResult({
482           name: res.thmName,
483           type: res.thmType,
484           tree,
485           treeJson: res.treeJson,
486         });
487         setHypotheses(null); // reset al cambio di teorema
488         setSelectedHypTree(null);
489       })
490       .catch((e) => setError(e.message ?? "Errore RPC
          sconosciuto"));
491   }, [props.pos, rs]);
492
493   useEffect(() => {
494     setSelectedHypTree(null);
495   }, [sharedHypotheses]);
496
497   if (!rs)
498     return React.createElement(
499       "div",
500       { style: { color: ERROR_RED } },
501       `⚠ RpcContext non disponibile (${versionInfo})`,
502     );
503
```

```
504   if (error)
505     return React.createElement(
506       "div",
507       { style: { color: ERROR_RED } },
508       `⚠ ${error} (${versionInfo})`,
509     );
510
511   if (!result)
512     return React.createElement(
513       "div",
514       { style: { color: LIGHTER_GRAY } },
515       `Sposta il cursore su un teorema per visualizzare
          l'NDTree... (${versionInfo})`,
516     );
517
518   return React.createElement(
519     // Outer container: flex column, all children start from
        left
520     "div",
521     {
522       style: {
523         fontFamily: "monospace",
524         display: "flex",
525         flexDirection: "column",
526         alignItems: "stretch",
527         gap: "0.5rem",
528         padding: "0.25rem",
529       },
530     },
531
532     // ── Header: nome e tipo del teorema ──
533     React.createElement(
534       "div",
535       {
536         style: {
537           display: "flex",
538           flexDirection: "column",
539           alignItems: "start",
```

```
          gap: "0.5rem",
        },
      },
      React.createElement(
        "p",
        {
          style: { color: ACCENT, flexGrow: 0, margin: 0 },
        },
        result.name + ":",
      ),
      React.createElement(
        "p",
        {
          style: { color: PRIMARY, opacity: "0.75", flexGrow:
          1, margin: 0 },
        },
        result.type,
      ),
    ),
  ),

  React.createElement(
    HypothesesContext.Provider,
    {
      value: {
        sharedHypotheses,
        setHypotheses,
        selectedHypTree,
        setSelectedHypTree,
      },
    },
    React.createElement(
      resultContext.Provider,
      {
        value: {
          result,
          setResult,
        },
      },
```

```
577
578         selectedHypTree &&
579           React.createElement(
580             "div",
581             {
582               style: {
583                 overflowX: "auto",
584                 outline: `0.0625 solid ${LIGHTER_GRAY}`,
585                 display: "flex",
586                 padding: "1rem",
587               },
588             },
589             React.createElement(
590               "div",
591               {
592                 style: {
593                   display: "flex",
594                   alignItems: "flex-start",
595                   justifyContent: "center",
596                   margin: "0 auto",
597                 },
598               },
599               React.createElement(NDTree, {
600                 key:
                    `hyp-${JSON.stringify(selectedHypTree).length}`,
601                 tree: selectedHypTree?.value,
602                 disableClick: true,
603                 uniqueId: "hypNode",
604               }),
605             ),
606           ),
607
608         // Hypothesis display — fixed above, never scrolls with
            the tree
609         sharedHypotheses?.length > 0 &&
610           React.createElement(HypothesesDisplay, null),
611
612         React.createElement(
```

```
          "div",
          {
            style: {
              overflowX: "auto",
              outline: `0.0625rem solid ${LIGHTER_GRAY}`,
              display: "flex",
              padding: "1rem",
            },
          },
          React.createElement(
            "div",
            {
              style: {
                display: "flex",
                alignItems: "flex-start",
                justifyContent: "center",
                margin: "0 auto",
              },
            },
            React.createElement(NDTree, {
              key:
              `tree-${result.treeJson.length}-${result.name}`,
              tree: result.tree,
              disableClick: false,
              uniqueId: "node",
            }),
          ),
        ),
      ),
    ),
  );
}
```

Appendice B

Glossario

AST (*Abstract Syntax Tree*) Struttura dati ad albero che rappresenta la struttura sintattica astratta di un programma o di un'espressione, indipendentemente dai dettagli lessicali.

CIC (*Calculus of Inductive Constructions*) Sistema formale di tipi dipendenti su cui si fondano proof assistant come Lean e Rocq (ex. Coq). Estende il λ-calcolo con tipi induttivi e universi gerarchici.

Deduzione naturale Sistema di regole di inferenza che formalizza il ragionamento logico in modo vicino all'intuizione matematica, strutturato attorno a regole di introduzione ed eliminazione per ciascun connettivo.

Info tree Struttura dati prodotta dall'elaboratore di Lean durante la compilazione di un file, che mappa ogni posizione nel testo sorgente alle informazioni di tipo e contesto corrispondenti.

Isomorfismo di Curry-Howard Corrispondenza biunivoca tra dimostrazioni logiche e programmi tipati: le proposizioni corrispondono a tipi, le dimostrazioni a termini, le regole di inferenza a costrutti del λ-calcolo.

JSON (*JavaScript Object Notation*) Formato testuale leggero per lo scambio di dati strutturati, basato su una sintassi derivata da JavaScript.

LSP (*Language Server Protocol*) Protocollo standard per la comunicazione tra editor di testo e server linguistici, che fornisce funzionalità come completamento automatico, navigazione al tipo e visualizzazione degli errori.

MetaM Monade di Lean 4 per la metaprogrammazione, che fornisce accesso al contesto locale, alle metavariabili e alle operazioni di inferenza di tipo.

Metavariabile In Lean, un goal non ancora dimostrato, rappresentato internamente come una variabile il cui valore (la prova) deve essere ancora fornito.

Proof assistant Software per la dimostrazione assistita da elaboratore, che verifica meccanicamente la correttezza di ogni passo di una dimostrazione formale.

React Libreria JavaScript per la costruzione di interfacce utente dichiarative basate su componenti.

Tipo dipendente Tipo che dipende da un termine: ad esempio, il tipo "vettore di lunghezza n" dipende dal valore n. I tipi dipendenti sono alla base della formalizzazione dei quantificatori logici.

Widget In Lean 4, un modulo JavaScript che può essere caricato nell'infoview di VS Code per visualizzare informazioni personalizzate associate a una posizione nel file sorgente.

λ-termine Termine del λ-calcolo; nel contesto di Lean, il termine interno che rappresenta una dimostrazione completata o parziale.

Bibliografia

[1] Leonardo de Moura, Sebastian Ullrich. *The Lean 4 Theorem Prover and Programming Language*. CADE 2021.

[2] Dag Prawitz. *Natural Deduction: A Proof-Theoretical Study*. Almqvist & Wiksell, 1965.

[3] Philip Wadler. *Propositions as Types*. Communications of the ACM, 2015.

[4] Morten Heine Sørensen, Paweł Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Elsevier, 2006.

[5] Arthur Paulino *et al. Metaprogramming in Lean 4*. Disponibile su: `https://leanprover-community.github.io/lean4-metaprogramming-book/`

[6] Microsoft. *Language Server Protocol Specification*. Disponibile su: `https://microsoft.github.io/language-server-protocol/`

[7] Samuel Buss. *The* `bussproofs` *package for LaTeX*. Disponibile su: `https://ctan.org/pkg/bussproofs`