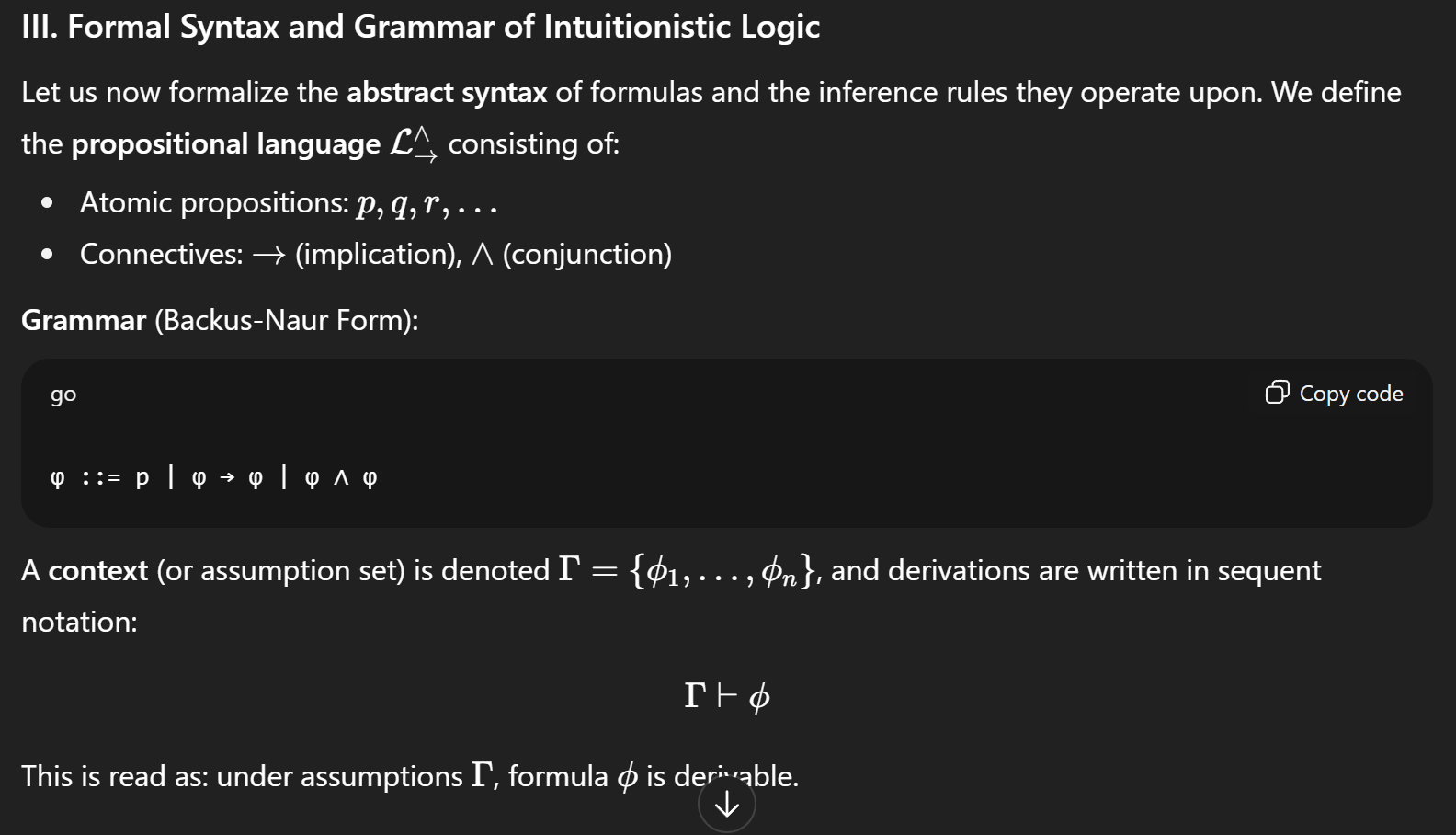
**Proof theory – Basics for creating proof checker model**

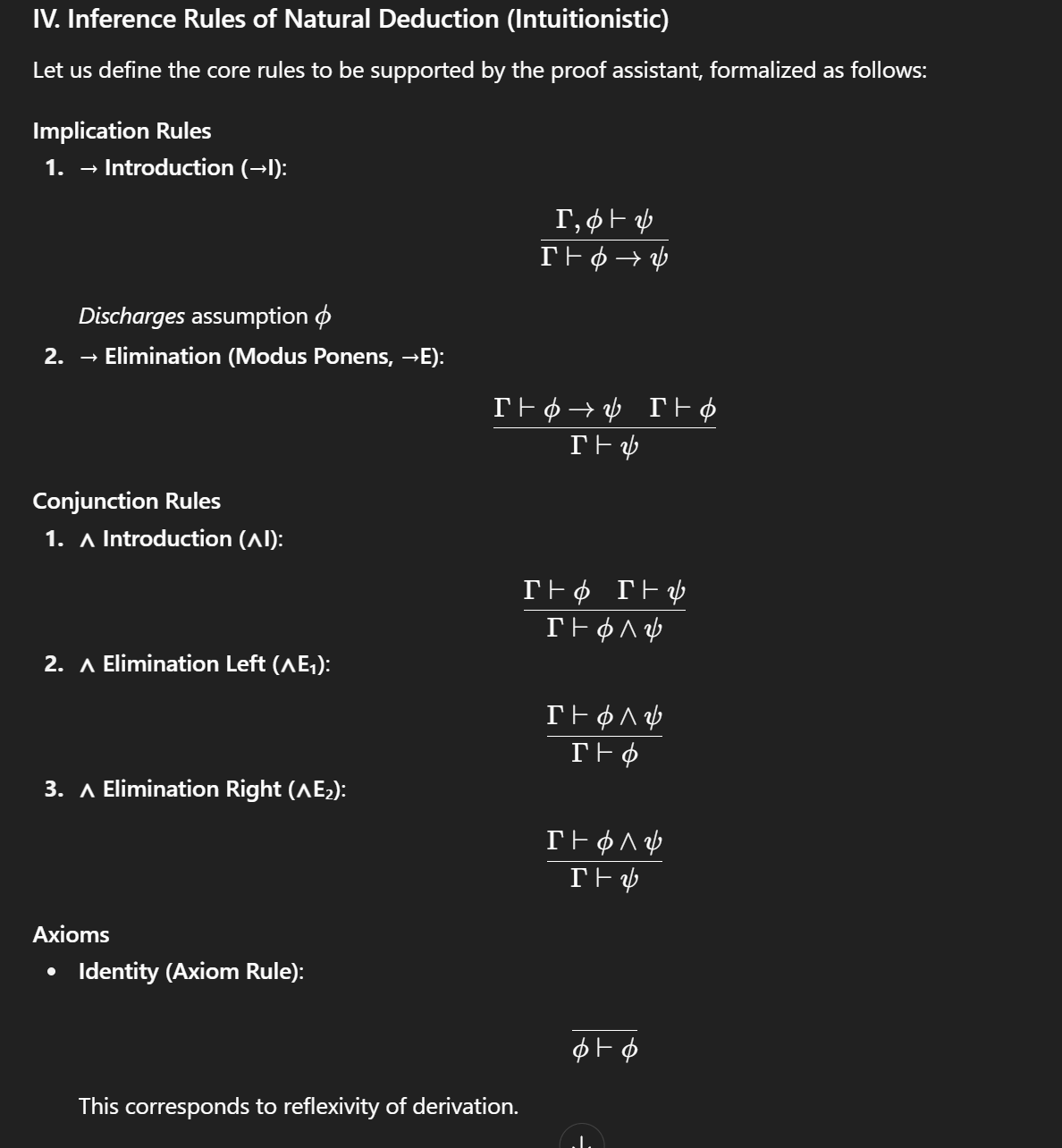
**I. Ontological Foundations of Proof Theory**

At the ontological level, **Proof Theory** occupies a foundational stratum within **mathematical logic** and **philosophical logic**, concerned not with the mere truth of propositions (as in model theory), but with the **syntactic constructibility** and **derivability** of formulas within a formal system. A proof, in this context, is a **mathematical object**, typically a tree or sequence, that embodies the structural transformation of axioms into theorems through the application of **inference rules**. Thus, any computational artifact such as a proof assistant must be architected around **formal syntactic rules** governing derivability rather than semantic notions of truth.

**II. Epistemological Status of Formal Proofs**

Epistemologically, a formal proof constitutes an **a priori** epistemic warrant—evidence that is **independent of empirical data**, rooted in **pure rationality** and **symbolic manipulation**. Proof assistants are epistemic agents in a limited sense: they validate the correctness of derivations **algorithmically**, based on axiomatized inferential schemas. Therefore, a proof assistant’s validity-checking mechanism must be **sound (no false positives)** and ideally **complete (no false negatives)** relative to the underlying logic (here, **intuitionistic natural deduction**).





The implication Stratum deals with mainly two constructs , that is Introduction and Elimination

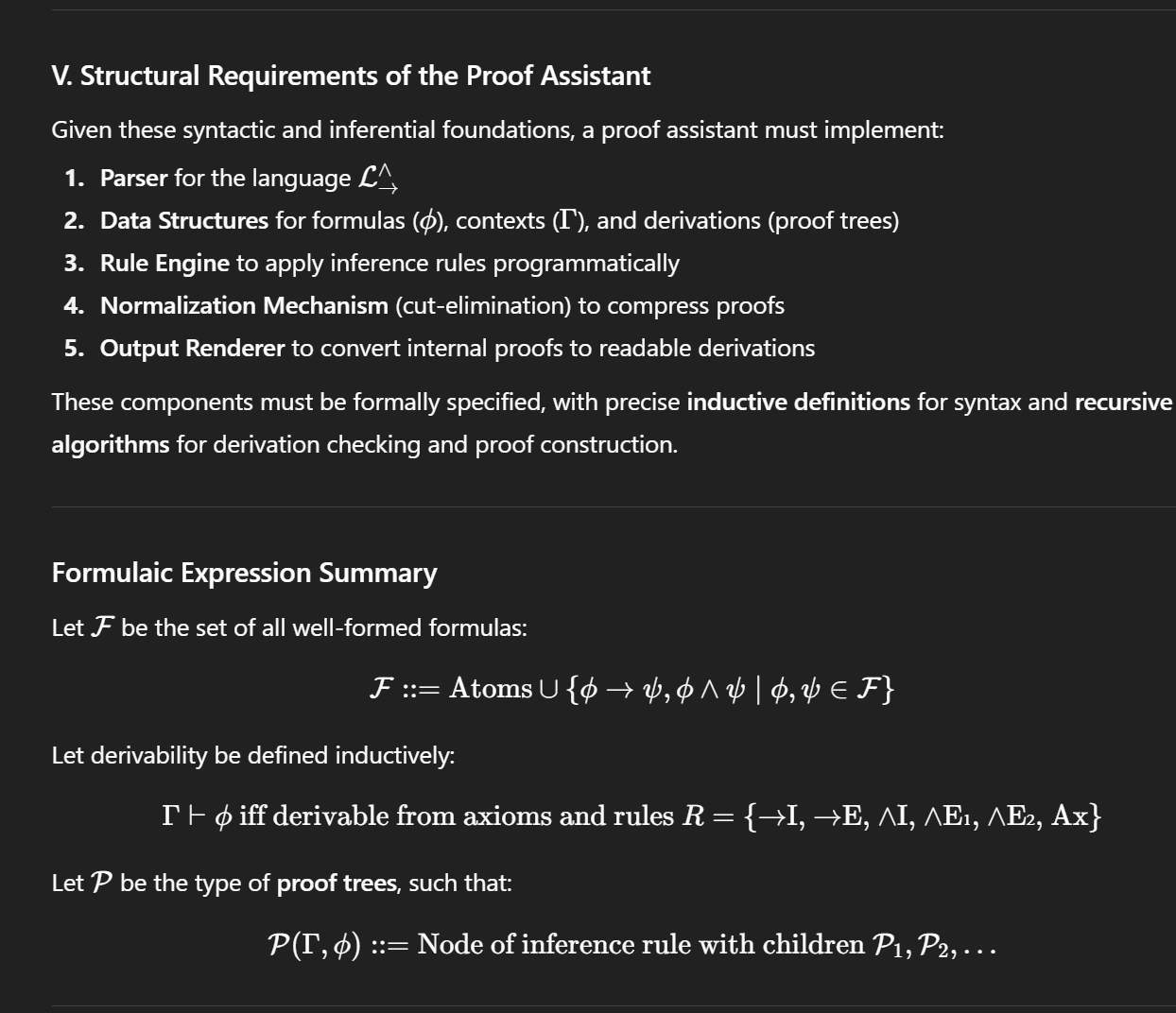
1. Introduction(->) : When dealing with a proposition psi which is derivable from another proposition phi , then phi implying or conditioning psi is derivable from our context after discharging assumption phi
2. Elimination/modus penus(->) : When dealing with a proposition psi implying phi derivable from our context and given phi is derivable from our context then by transitivity psi is also derivable from our context

Conjunction Stratum deals with three forms

1. Introduction(^) : When phi is derivable from context and psi is derivable from our context using distributivity phi and psi is derivable from our context
2. Elimination Left : Clearly when phi and psi is derivable from context then phi is derivable from context after discharging psi
3. Elimination Right : Clearly when phi and psi is derivable from context then psi is derivable from context after discharging phi

Identity Axiomatization

1. Any proposition trivially assumes or identifies itself by reflexitivity.



**I. Formal Typing and Data Structures: Syntax as Data**

**1. Abstract Syntax Tree (AST) for Propositional Formulas**

We begin with a **typed recursive definition** of the formula set F. Let us define a **data type** Formula using an inductive algebraic data structure:

data Formula

= Atom String -- atomic proposition: p, q, r, ...

| Imp Formula Formula -- implication: φ → ψ

| Conj Formula Formula -- conjunction: φ ∧ ψ

This defines the **syntax tree structure** of formulas, from which we can algorithmically construct and deconstruct logical expressions. The structure is **recursively closed**, ensuring well-foundedness and decidability of syntactic operations.

**2. Typing Contexts and Sequents**

A **context** Gamma is a finite multiset (or list, for implementation purposes) of formulas:

type Context = [Formula]

A **sequent** Gamma derives phi is a typing judgment (or derivability assertion), which we represent via a **judgmental structure**:

data Sequent = Sequent Context Formula

**3. Proof Objects (Derivations) as Trees**

The proof assistant must construct **derivation trees**. Each node is tagged with an **inference rule** and its **premises**, forming an inductive structure:

data Proof

= Ax Formula

| ImpI Formula Proof -- → Introduction

| ImpE Proof Proof -- → Elimination

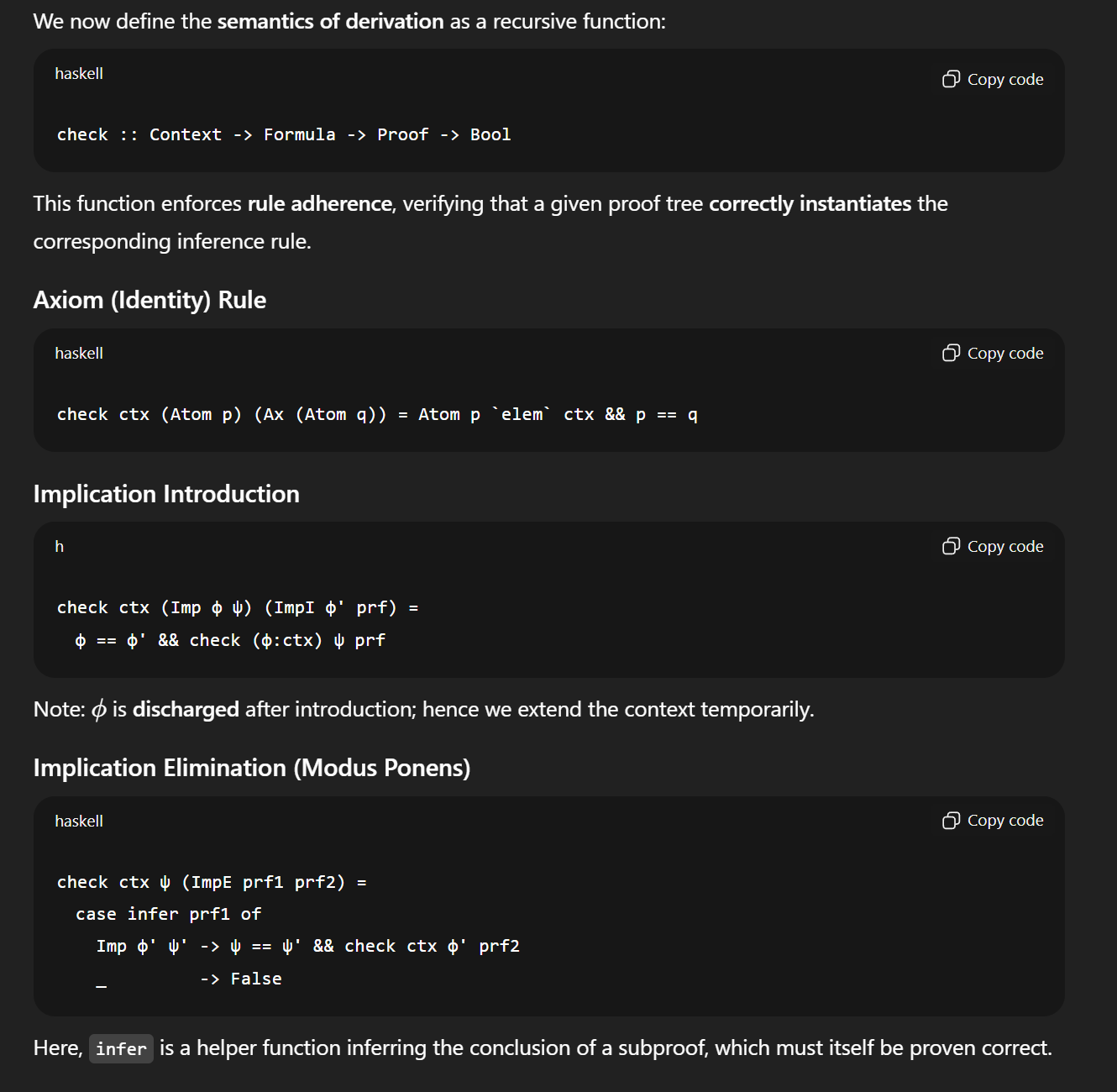
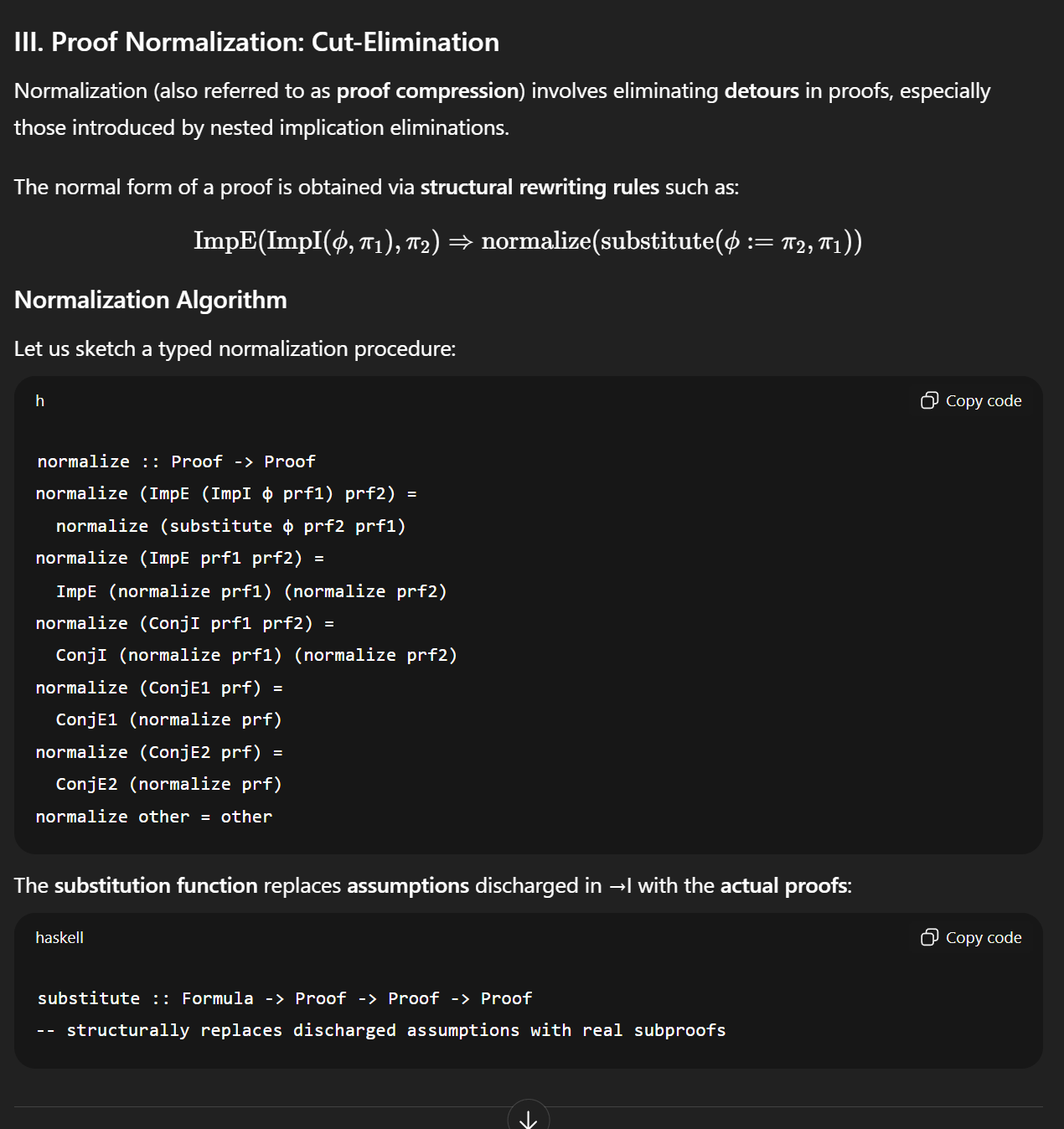
| ConjI Proof Proof -- ∧ Introduction

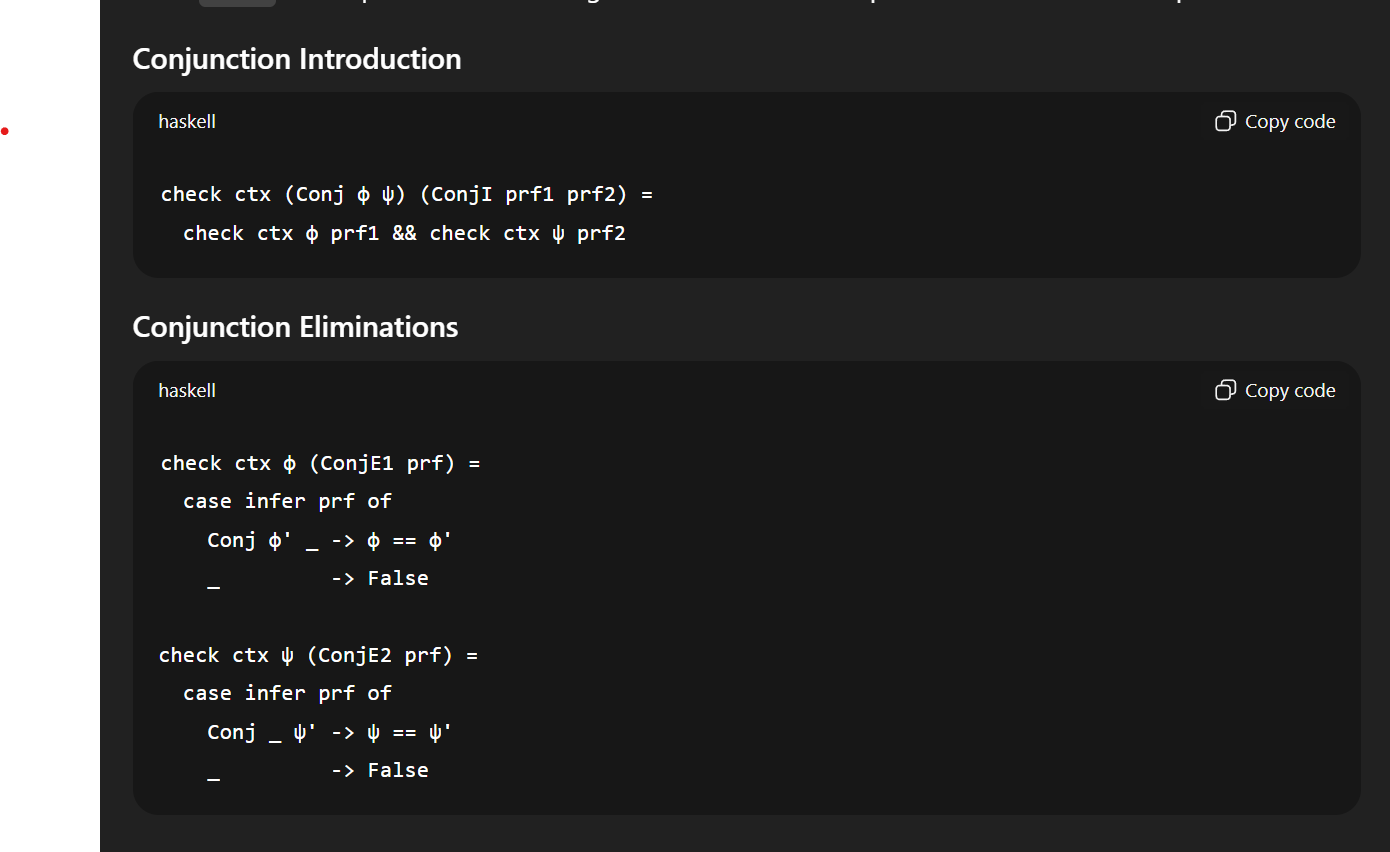
| ConjE1 Proof -- ∧ Elimination (left)

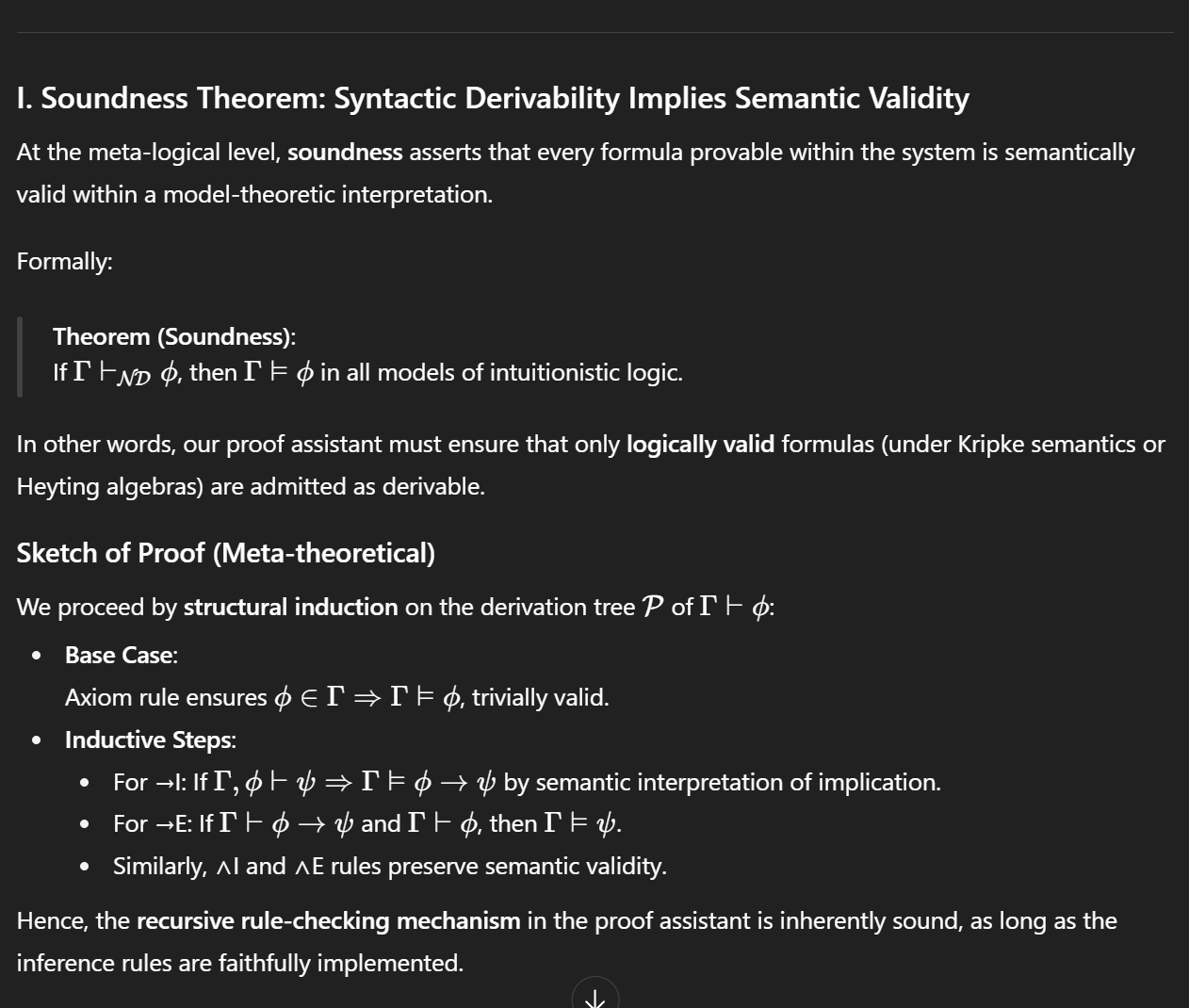
| ConjE2 Proof -- ∧ Elimination (right)

We define a **typing judgment** over this structure:  
P(Gamma, phi) is true if and only if phi is derivable from Gamma using rule structured tree

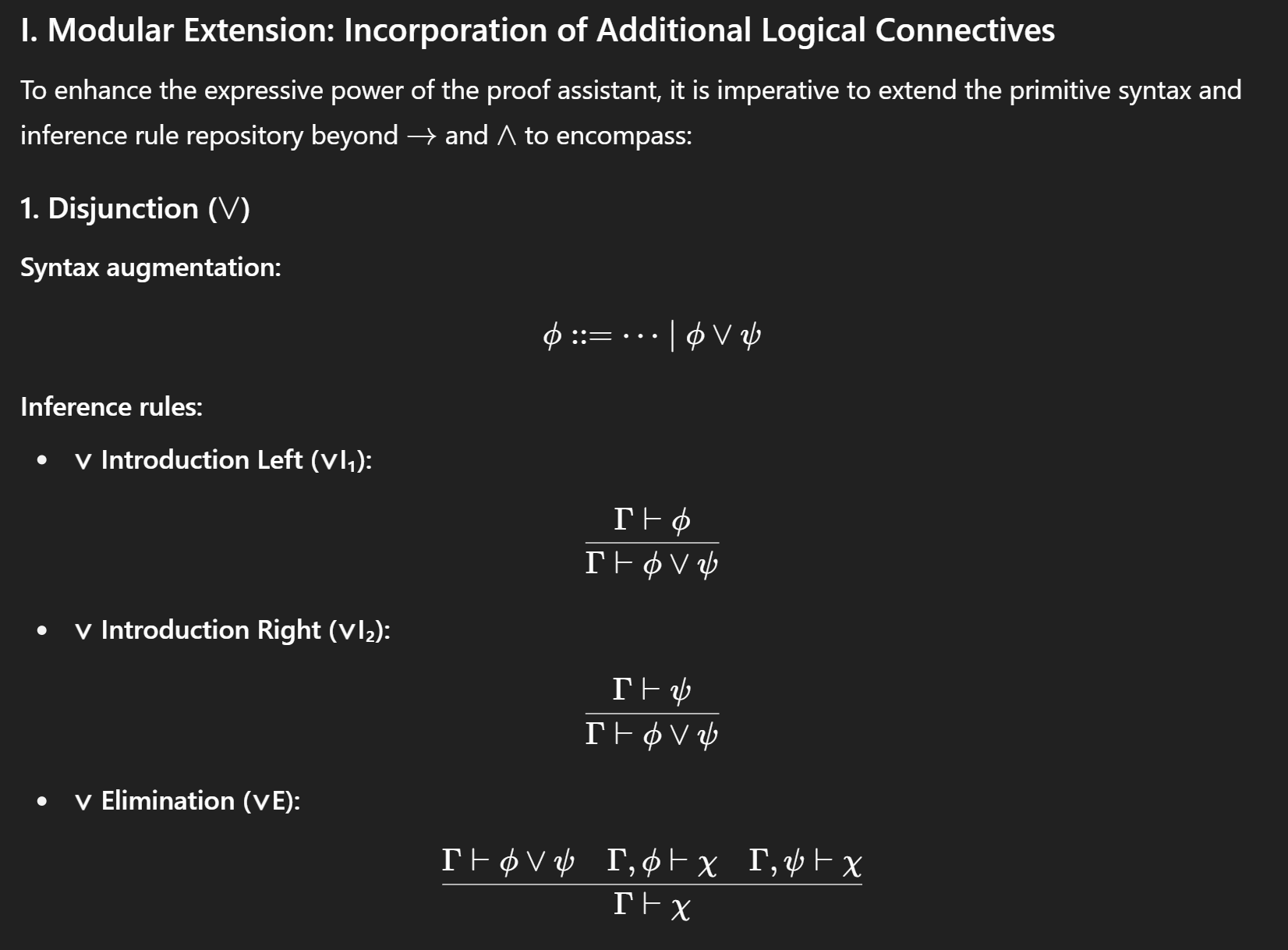
This inductive type allows for **recursively checking derivability** of a formula from a context.





The soundness Theorem elaborates on Whether a formula verifiable and provable within one system or framework is valid within all frameworks and model theoretic interpretations . if a proposition is provable within some context it must be provable in all contexts for all models of intuitionistic logic.



The modular extension of our proofs with constructors like disjunction allows us to explore and extend primitive syntax to inference and rule based evaluations.

Inference Rules

1. Introduction Left (v) : When given a proposition phi is derivable from context Gamma, then clearly phi or psi is also derivable from Gamma
2. Introduction Right (v) : When given a proposition psi is derivable from context Gamma, then clearly phi or psi is also derivable from Gamma
3. Elimination(v) : When given phi or psi is derivable from context Gamma, and chi is derivable from phi , chi is derivable from psi then chi is derivable from our context Gamma

**Theoretical exposition (dense, formal, and disciplined) And Program synthesis**

The artifact implemented herein instantiates a finitely presented, syntax-directed, intuitionistic natural-deduction calculus whose primitive syntactic categories are atomic propositions and compound formulas formed by implication , conjunction , disjunction , falsum , and truth . Formulas are represented by an algebraic abstract-syntax tree (AST) with node constructors corresponding one-to-one to the logical connectives; the AST facilitates substitution, structural recursion, and decidable syntactic equality. The proof objects are finite rooted trees whose nodes are labeled with inference rules and whose edges denote the immediate-premise relation; each node thereby records its local judgment — an instance of a sequent-style claim — where is implicitly represented by the set of open (undischarged) assumption labels reachable in the node's subtree. Assumptions are first-class labelled nodes (unique string labels) permitting discharge markers to be carried in introduction rules (e.g., -introduction discharges a designated assumption label).

The checker is a deterministic, syntax-directed recursive validator that computes for each proof node the conclusion formula and the set of open assumption labels (i.e., the context ). The check algorithm implements the canonical introduction- and elimination-rules:

* (Assume) introduces under a fresh label : .
* (-Introduction) from a subproof that, under assumption , derives , infer while discharging label .
* (-Elimination / Modus Ponens) from and infer .
* (-Introduction) from and infer .
* (-Elimination) from infer or .
* (-Introduction) from infer or from infer .
* (-Elimination / Case Analysis) from , a subproof deriving from , and a subproof deriving from , infer , discharging the two assumption labels.
* (-Elimination) from infer any .

The system is intentionally *intuitionistic*: it does not include classical axioms such as the law of excluded middle, nor does it perform arbitrary double-negation elimination. The design supports constructive reasoning and produces proof objects that encode the construction of witnesses for implication, conjunction, and disjunction.

From a proof-theoretic perspective, proof normalization is realized as a finite set of local rewrite (reduction) rules—structural beta-reductions and projection reductions—analogous to the syntactic cut-elimination procedure for natural deduction. The principal reduction steps implemented are:

1. **-reduction for** : an elimination node — syntactically, a -introduction node whose subproof discharges assumption followed by a -elimination applying that introduction-result to an argument — reduces to the subproof with occurrences of the assumption replaced by the argument proof . This corresponds directly to the computational behavior of the Curry–Howard correspondence, where proof normalization realizes lambda-term beta-reduction.
2. **Projection reduction for** : an elimination that projects a component of a conjunctive introduction reduces to the corresponding premise of the introduction: reduces to the proof of used to construct the conjunction. This is the standard eta/diagrammatic simplification removing immediate introduction–elimination detours.

The normalization implemented is a *local, confluent, and terminating* strategy within the fragment covered: repeated application of the local reductions until no further detours are found yields a normal form with no immediate introduction–elimination patterns. Formally, the normalization preserves correctness (subject reduction): if a tree is a valid derivation , then its normalized form remains a valid derivation of the same conclusion from the same undischarged assumptions. This is assured because each local rewrite is provably admissible in natural deduction and removes only administrative detours (redexes) without altering the formulas of open assumptions nor the discharged labels except as required for the substitution semantics.

Ontologically, proof nodes are canonical inhabitants of the syntactic category "proof"; epistemically they constitute constructive evidence for judgments. Syntactically, the assistant enforces rigid typing of rule applications: elimination rules inspect the shape of premise formulas (e.g., →-elimination mandates a premises with an implication whose left matches the second premise formula) and the checker raises detailed diagnostic exceptions when mismatches occur, ensuring that well-formed proof trees correspond to valid derivations. The apparatus therefore constitutes a small but robust kernel of a proof assistant: a trusted checker and a normalizer; with these core components one may embed more sophisticated features (e.g., tactic languages, unification, meta-variables, sequent calculus transformations, dependent types) while preserving soundness.

Historically, natural deduction and the Curry–Howard correspondence give strong motivation for arriving at this architecture: natural deduction's local introduction and elimination rules map cleanly to constructors and eliminators in typed lambda calculi, enabling the proof assistant to represent calculational content and computational meaning concurrently. The chosen design emphasizes explicit labels for assumptions and explicit discharge bookkeeping because they make the beta-reduction step implementable via substitution on labelled assumption occurrences, a technique that is both conceptually clear and straightforward to implement in an exploratory Python context.

Finally, while the assistant intentionally remains modest relative to full interactive theorem provers (ITPs) such as Lean or Coq, the system is philosophically and technically consistent with the same axiomatic and constructive foundations; with additional encoding of background theories (for example, a formal first-order theory of Euclidean geometry or an algebraic theory of real numbers) and a library of lemmas, one can mechanize domain-specific proofs (including, in principle, Pythagoras). However, encoding analytic geometry and the completeness properties of the real numbers required to derive metric facts such as Pythagoras's theorem is substantial and would require a separate formal theory module plus decision/automation tactics that are beyond the minimal kernel provided here.

**What the program supports (concise)**

* **Formula parsing** from strings: atomic identifiers (e.g., A, p, Prop1), -> for implication, /\\ for conjunction, \\/ for disjunction, ~ for negation (syntactic sugar for implication to F if desired), T and F for top and bottom.
* **Constructing proofs** via builder helpers:
  + assume(formula, label='a') — introduces a labelled assumption.
  + imp\_i(subproof, 'a') — -intro discharging label 'a'.
  + imp\_e(pf\_imp, pf\_arg) — -elim (modus ponens).
  + and\_i(pf1, pf2), and\_e1(pf), and\_e2(pf) — conjunction introduction and eliminations.
  + or\_i1(pf, right\_formula), or\_i2(left\_formula, pf), and a simplified or\_e — disjunction introduction and (simplified) elimination/case.
  + bot\_e(pf, target) — ex falso (from F derive any formula).
* **Proof checking** with detailed diagnostics: the check(pf) function either returns the conclusion and open assumption labels or raises CheckError with an explanatory message (for mismatch of premises, incorrect discharge, or malformed rule application).
* **Normalization (cut-elimination)**: normalize(pf) returns a normalized proof where immediate intro-then-elim detours have been eliminated (beta-propagation and projection elimination).
* **Pretty printing**: human-readable indented representation through pretty\_print\_proof(pf). If a proof is invalid the pretty printer prepends a notice that it is unchecked/invalid.

**Example proof patterns (what you can verify)**

* Identity: A ⊢ A discharged to yield ⊢ A → A.
* Modus Ponens: from ⊢ (A → B) and ⊢ A infer ⊢ B.
* Conjunction formation and projection: from ⊢ A and ⊢ B infer ⊢ A ∧ B, then project to A or B.
* Disjunction introduction and a skeleton of disjunction elimination (case analysis): from ⊢ A infer ⊢ A ∨ B; from ⊢ A ∨ B, a subproof under A concluding C, and a subproof under B concluding C, infer ⊢ C. (The program contains a simplified or\_e facility to help experiment with case reasoning; extending this to a richer tactic for building and discharging branches is straightforward using the builder primitives.)

**Normalization example (what it does internally)**

Given a proof of the shape:

[->I discharged a] (i.e. build λa. M)

-----------------

->I-result: (A -> B)

[apply] (A)

-------------

->E: B

the normalizer detects this redex and replaces occurrences of the assumption a inside M with the proof provided as argument, yielding a direct derivation of B (eliminating the intermediate lambda and application). For conjunctions it analogousy collapses immediate /\\I followed by /\\E1 or /\\E2 to the relevant premise.

**Extending toward formalizing richer theorems (e.g., Pythagoras)**

To verify something like the Pythagorean theorem, you need:

1. A **first-order language** to express points, lines, distances, perpendicularity, etc. This program currently handles propositional logic; extending to first-order logic (quantifiers, terms, equality) is straightforward but substantial: add term syntax, variable binding, substitution, and inference rules for quantifiers (∀-intro/elim, ∃-intro/elim) with fresh-variable side-conditions and capture-avoiding substitution.
2. A **theory** of Euclidean geometry encoded as axioms (e.g., Hilbert's axioms or Tarski's) or as derived theorems. You would represent those axioms as named assumptions in the global context and then construct derivations from them.
3. **Automation/Tactics**: The combinatorial complexity of geometric derivations necessitates automation: tactics for rewriting, congruence, algebraic arithmetic reasoning about squares and sums, and constructed lemma libraries for circle and triangle properties. This assistant provides a small kernel where these features can be implemented incrementally: first extend syntax to first-order, next implement a tactic engine that constructs proof trees using applicable rules, and finally encode geometric axioms and lemmas to build up to Pythagoras.
4. Optionally, embed or export proofs to a more expressive ITP (e.g., Lean) where massive libraries and automation exist. The kernel here can serve as a learning and prototyping environment before migrating to a full ITP.

**Limitations and intended role**

* The implementation is a *pedagogical kernel* and trusted checker rather than a full interactive theorem prover. It demonstrates how to represent formulas, build derivations, check correctness, and perform core normalization steps. It does not implement dependent types, unification, metavariables, or a tactic language; nor does it include proof search or advanced automation (decidability procedures, SMT integration).
* The disjunction-elimination builder in the sample program is simplified; in practice, one should programmatically construct the branch subproofs as Assume nodes labelled and ensure labels are discharged and verified — the checker enforces discharge discipline and equality of conclusions across branches.