A Formal Mathematical Specification for QED

A Minimal LCF-Style HOL Kernel in MoonBit

Abstract

QED is an interactive theorem prover in MoonBit, designed around an LCF-style trusted kernel and a higher-order logic foundation. This document gives a formal mathematical specification of its core theory. The presentation starts from first principles: signatures, type formation, term formation, typing judgments, substitution laws, theorem objects, and primitive inference rules. The objective is to make the trust model and proof discipline explicit enough that the logic stands independently of any one implementation.

Reading Guide

The document is split into two parts. Part I is the normative logic core (definitions, rules, metatheorems). Part II is the engineering realization and conformance obligations. A reader focused on logic soundness may read Part I alone.

1. Part I: Logic Core (Normative)

Normative Scope.

- Part I is the single normative source for syntax, typing, derivability, admissibility gates, and soundness/conservativity claims.
- Part II introduces no logical axioms, primitive rules, or weakened side conditions beyond Part I.
- De Bruijn conversion and scoped signature semantics are treated as logical correctness machinery in Part I, not as optional realization detail.
- In case of disagreement, Part I is taken as authoritative and Part II is revised accordingly.

2. Notation and Meta-Level Conventions

This section fixes the notation baseline used by all later definitions and rules.

- Object-language judgments are written with $\vdash$, e.g. $\Gamma \vdash t : \tau$.
- Partial computation/evaluation at the meta level is written with $\mapsto$, e.g. $f(x) \mapsto y$.
- Named-term alpha-equivalence is written $\underset{\alpha}{\equiv}$.
- De Bruijn structural equivalence is written $\cong$.
- Semantic interpretation (denotation) is written as $\mathrm{denote}(t, \rho, M)$ for term meaning under environment/model pair (ρ, M).

Layering discipline:

1. **Object layer**: syntax, typing, theorem sequents, and primitive rules.
2. **Boundary layer**: named/De Bruijn conversion partial functions.
3. **Meta layer**: proofs about invariants, commutation, and soundness.

All claims below explicitly indicate which layer they inhabit.

Per-section dependency template (used throughout this manuscript):

- **Defines**: new syntax/judgments introduced in the section.
- **Depends on**: earlier sections required for well-formedness.
- **Used by**: later rules/theorems that rely on this section.

3. Motivation and Design Goal

The central engineering goal of QED is to separate trusted reasoning from untrusted proof search. In an LCF architecture, theorem creation is restricted to a very small set of primitive kernel operations. Everything else, including automation and tactics, is merely a theorem-producing program that calls those primitives.

This architecture has two consequences:

1. Correctness is reduced to the soundness of a small kernel.
2. Rich user tooling can evolve without expanding the trusted code base.

The mathematical role of this document is to state the object language and inference rules precisely enough to support both consequences.

4. Positioning and Related Systems

QED is positioned as a minimal, auditable HOL kernel rather than a full proof-assistant platform. The design goal is to keep the trusted core small, make boundary assumptions explicit, and allow tactic/automation layers to evolve independently.

Definition (Design Target). QED prioritizes kernel verifiability over feature breadth: every exported theorem object must be traceable to primitive kernel rules under explicit side conditions.

Compared with mainstream systems:

- HOL Light: same LCF trust model and primitive-rule discipline, but QED currently executes primitive cores on De Bruijn objects and then lifts to named boundaries.
- Coq: richer dependent type theory and broader automation ecosystem, while QED currently focuses on STT/HOL kernel minimality.
- Isabelle: mature logical framework and extensive libraries, while QED intentionally optimizes for a compact, audit-oriented kernel specification.

5. Foundational Theory

5.1. Signatures and Symbols

Let Σ_t be a type-constructor signature. Each constructor $k \in \Sigma_t$ has a natural-number arity $a(k) \in \mathbb{N}$.

Let Σ_c be a term-constant signature. Each constant $c \in \Sigma_c$ is assigned a simple type.

In the current QED kernel, Σ_c is managed by a scoped signature stack:

- each scope stores a finite partial map from constant names to types;
- lookup proceeds from the innermost scope to the outermost scope;
- same-name insertion is rejected inside one scope but allowed in a nested scope (shadowing).

For polymorphic constants, the assigned type is read as a principal type schema (universally quantified at the meta level), and concrete uses are type instances of that schema.

QED reserves two distinguished type constructors:

- bool with arity 0.
- fun with arity 2.

These are sufficient to define simply typed lambda terms with boolean propositions.

For next-stage conservative theory construction, QED additionally fixes one trusted foundation type constructor:

- ind with arity 0.

`"ind"` is the carrier used by the explicit infinity-anchor assumption; it is not introduced by user-level extension rules.

5.2. Constant Type Schemes and Instance Relation

To avoid polymorphism lockout while preserving kernel typing discipline, constant typing uses a principal-schema + instance relation.

Definition (Constant Principal Type). Each resolved constant identity carries a principal simple type schema:

$$\kappa_c : \tau_{\text{gen}}$$

where type variables in τ_{gen} are implicitly universally quantified.

Definition (Type Instance Relation). For types τ and τ_{gen}, write

$$\tau \preceq \tau_{\text{gen}}$$

iff there exists a type substitution θ such that

$$\tau = \theta(\tau_{\text{gen}})$$

and θ maps only type variables.

This relation is used by elaboration and core typing of `RConst`, including user constants and reserved logical constants.

5.3. Logical Constants and Reserved Symbols

The kernel distinguishes **logical symbols** from ordinary signature-managed constants.

Definition (Reserved Equality Symbol). The symbol $=$ is a reserved polymorphic logical constant with schematic type

$$= : \prod \alpha.\ \text{fun}(\alpha, \text{fun}(\alpha, \text{bool}))$$

and is not inserted by user signature operations.

Definition (Reserved Choice Operator). The symbol @ is a reserved polymorphic choice operator with schematic type

$$@ : \prod \alpha.\ \text{fun}(\text{fun}(\alpha, \text{bool}), \alpha)$$

and is not inserted by user signature operations.

Axiom Schema (Choice). For each type instance α and predicate $P : \text{fun}(\alpha, \text{bool})$:

$$\vdash (\exists x : \alpha . P(x)) \Rightarrow P(@(P))$$

This axiom schema is part of the trusted baseline and is used to derive specification-style constant introduction.

Constraint (No Shadowing of Reserved Symbols). For every scope stack state S, insertion is forbidden for reserved names:

$$\frac{c \in \text{Reserved}}{S \vdash \text{add}(c : \tau) \mapsto \text{fail}}$$

where currently $\text{Reserved} = \{=, @\}$.

Definition (Typed Equality Formation). Given resolved terms u, v:

$$\frac{\Gamma \vdash u : \tau,\ \Gamma \vdash v : \tau}{\Gamma \vdash (u = v) : \text{bool}}$$

This formation rule is the unique source of equality propositions used by primitive rules.

Remark (Signature Separation). Ordinary entries in Σ_c are user/system constants. Logical equality is fixed by the logic layer and cannot be rebound by scope operations.

5.4. Signature Judgments for Scoped Stacks

Definition (Scoped Signature Judgments). We use the following judgments for scoped signature operations:

- $S \vdash \text{wf}$: signature stack S is well-formed.

- $S \vdash c : \tau$: lookup for constant c succeeds with type τ.
- $S \vdash \text{push} \mapsto S'$: push succeeds and returns S'.
- $S \vdash \text{add}(c : \tau) \mapsto S'$: insertion in current scope succeeds.
- $S \vdash \text{pop} \mapsto S'$: pop succeeds and returns S'.

Notation: $S \mathbin{++} [F]$ denotes stack append with F as the new innermost frame. In particular, $S \mathbin{++} [\text{empty}]$ means push one fresh empty scope onto S.

Representative rules:

$$\frac{S \vdash \text{wf}}{S \vdash \text{push} \mapsto S \mathbin{++} [\text{empty}]}$$

$$\frac{S = [S_0, ..., S_n], c \notin \text{dom}(S_n), S' = \left[S_0, ..., S_{n[c:=\tau]}\right]}{S \vdash \text{add}(c : \tau) \mapsto S'}$$

$$\frac{S = [S_0, ..., S_n], c \in \text{dom}(S_n), S_{n(c)} = \tau}{S \vdash c : \tau}$$

$$\frac{S = [S_0, ..., S_n], c \notin \text{dom}(S_n), [S_0, ..., S_{n-1}] \vdash c : \tau}{S \vdash c : \tau}$$

$$\frac{S = [S_0, ..., S_n], n > 0, S' = [S_0, ..., S_{n-1}]}{S \vdash \text{pop} \mapsto S'}$$

Well-Formedness Side Conditions.

- no frame may bind a reserved logical symbol;
- each frame is a finite partial map;
- lookup is deterministic by innermost-first traversal.

These rules make success/failure boundaries explicit and give a judgmental presentation of scoped lookup and mutation.

5.5. Global Theory State vs Local Scope State

To avoid ambiguity between poppable lookup state and persistent logical commitments, QED distinguishes two state layers:

- global theory state T: append-only logical history (definition heads, type constructors, proved definitional theorems);
- local scope stack S: poppable name-lookup convenience layer.

Combined machine state is written (T, S).

Operational boundary:

1. `push`/`pop` mutate S only.
2. definitional extension mutates T (and may expose a binding in current S).
3. global freshness checks for definitions are performed against T, never against current-stack-only visibility.

Name history used by definitions:

$$\text{DefHeads}(T) = \{\text{all constant names ever committed by definitional extension}\}$$

Equivalent representation view: `"DefHeads"(T)` is any monotone history set unaffected by scope pop.

Monotonicity law:

$$T \mapsto T' \Rightarrow \text{DefHeads}(T) \subseteq \text{DefHeads}(T')$$

Proposition (Pop Invariance of Definitional History). If $(T_0, S) \vdash \text{pop} \mapsto (T_1, S')$, then

$$T_0 = T_1 \wedge \mathrm{DefHeads}(T_0) = \mathrm{DefHeads}(T_1)$$

trivially, because `pop` is local-scope-only and does not rollback committed theory symbols.

5.6. Type Grammar

Types are generated by the grammar

$$\tau ::= \alpha \mid k(\tau_1, ..., \tau_n)$$

where α ranges over type variables and $k \in \Sigma_t$ with $a(k) = n$.

One canonical concrete representation is:

- `TyVal(name)` for type variables.
- `TyApp(tycon, args)` for constructor application.

The representation is syntax-directed and supports recursive operations such as type substitution and constructor decomposition.

5.7. Type Constructor Extension Discipline

Non-emptiness of all well-formed types is enforced by admissible type introduction, not by unconstrained signature mutation.

Definition (Type Definition Admissibility). A new type-constructor introduction is written:

$$\mathrm{TypeDefOK}(T, k, a, \mathrm{Rep}, P, w)$$

and requires:

1. $k \notin \mathrm{TySymbols}(T)$ and a matches declared parameter arity.
2. w is a witness theorem establishing representability non-emptiness for each parameter instantiation.
3. the defining predicate P is well-typed and closed under the declared parameters.
4. all free type variables of P are exactly the declared parameters (no undeclared type-variable leakage).
5. representation head name Rep_k is globally fresh in theory history and reserved-symbol disjoint.
6. abstraction head name Abs_k is globally fresh in theory history and reserved-symbol disjoint.

Representative admissible step:

$$\mathrm{TypeDefOK}(T, k, a, \mathrm{Rep}, P, w) \Rightarrow T \mapsto T + \left\{\mathrm{typedef}\ \frac{k}{a}\right\}$$

Definition (Typedef Product Contract). For each admissible typedef step above, theory extension must also expose fresh constants:

$$\mathrm{Abs}_k : \mathrm{RepTy}_k \to k(\alpha_1, ..., \alpha_a)$$

$$\mathrm{Rep}_k : k(\alpha_1, ..., \alpha_a) \to \mathrm{RepTy}_k$$

and must provide the following theorem schemata (with the declared type parameters explicitly quantified):

1. Surjectivity of abstraction:

$$\vdash \forall n : k(\alpha_1, ..., \alpha_a).\mathrm{Abs}_{k\left(\mathrm{Rep}_{k(n)}\right)} = n$$

2. Representation range soundness:

$$\vdash \forall n : k(\alpha_1, ..., \alpha_a).P\left(\mathrm{Rep}_{k(n)}\right)$$

3. Conditional retraction on predicate range:

$$\vdash \forall r : \mathrm{RepTy}_k.P(r) \Rightarrow \mathrm{Rep}_{k\left(\mathrm{Abs}_{k(r)}\right)} = r$$

No weaker contract is admissible for kernel-level typedef extension.

Construction Invariant (No Empty-Type Escape). If base prelude types are non-empty and every later type-constructor extension satisfies "`TypeDefOK`", then every well-formed type over the resulting signature has a non-empty semantic carrier.

This invariant is the syntactic enforcement hook used by `INST_TYPE` soundness arguments.

5.8. Term Grammar

Terms are generated by:

$$\begin{aligned} t ::= {} & x : \tau && \text{variable} \\ \mid {} & c : \tau && \text{constant} \\ \mid {} & t_1 \; t_2 && \text{application} \\ \mid {} & \lambda(x : \tau).t && \text{abstraction} \end{aligned}$$

where x is a variable symbol and c is a constant symbol.

One canonical concrete representation is:

- `Var(name, tau)`
- `Const(name, tau)`
- `Comb(f, x)`
- `Abs(x, t)`

The abstraction constructor is intended to bind occurrences of the variable component of `x` in `t`.

Binding and Capture

Any substitution algorithm used by the kernel must be capture-avoiding. This is not a convenience detail: it is a semantic requirement for soundness.

5.9. Resolution Boundary (Integrated Form)

Definition (One-Shot Resolution). To avoid splitting the document into two full grammars, QED keeps one named grammar and adds a single boundary judgment:

$$\Sigma_c \vdash t \mapsto t_r$$

where t_r is a resolved term used by kernel rules. Constant occurrences are frozen at resolution time and are not re-looked-up during later theorem reuse.

Notation compatibility: this section writes resolution with $\mapsto$; the later typing section writes the same elaboration relation as $\Downarrow$.

Representative constant-resolution clause:

$$\frac{\Sigma_c \vdash c : \tau, \operatorname{resolve}(\Sigma_c, c) = \kappa_c}{\Sigma_c \vdash c : \tau \mapsto \operatorname{ConstAtom}(\kappa_c, \tau)}$$

Property (Theorem Stability Under Scope Mutation). If a theorem is constructed from resolved terms, then later `push`/`pop`/shadowing operations on Σ_c do not change that theorem's meaning or typing status.

Property (Alias Safety for Existing Theorems). Theorem aliasing is handle-level only: alias creation does not trigger re-elaboration, and therefore does not depend on the current signature stack.

6. Elaboration and Core Typing Judgments

To make scope-mutation stability and boundary safety derivable (not merely stated), QED uses a two-layer typing story.

6.1. Named Elaboration Judgment

Named terms are first elaborated against the current signature stack:

$$\Sigma_c; \Gamma \vdash t \Downarrow t_r$$

where t_r is a resolved term in which constant occurrences have fixed kernel identities.

Representative clauses:

$$\frac{(x : \tau) \in \Gamma}{\Sigma_c; \Gamma \vdash x : \tau \Downarrow \mathrm{RVar}(x, \tau)}$$

$$\frac{\Sigma_c \vdash c : \tau_{\text{gen}}, \mathrm{resolve}(\Sigma_c, c) = \kappa_c, \tau \preceq \tau_{\text{gen}}}{\Sigma_c; \Gamma \vdash c : \tau \Downarrow \mathrm{RConst}(\kappa_c, \tau)}$$

$$\frac{\Sigma_c; \Gamma \vdash f \Downarrow f_r,\ \Sigma_c; \Gamma \vdash x \Downarrow x_r}{\Sigma_c; \Gamma \vdash f\ x \Downarrow \mathrm{RComb}(f_r, x_r)}$$

$$\frac{\Sigma_c; (\Gamma, x : \tau) \vdash t \Downarrow t_r}{\Sigma_c; \Gamma \vdash \lambda(x : \tau).t \Downarrow \mathrm{RAbs}(x, \tau, t_r)}$$

Elaboration failure is explicit and does not produce a theorem object.

6.2. Core Typing over Resolved Terms

Primitive rules consume resolved terms only. Their typing judgment is:

$$\Gamma \vdash t_r : \tau$$

with rules independent of mutable signature lookup.

Representative clauses:

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash \mathrm{RVar}(x, \tau) : \tau}$$

$$\frac{\kappa_c : \tau_{\text{gen}}, \tau \preceq \tau_{\text{gen}}}{\Gamma \vdash \mathrm{RConst}(\kappa_c, \tau) : \tau}$$

$$\frac{\Gamma \vdash f_r : \mathrm{fun}(\tau_1, \tau_2),\ \Gamma \vdash x_r : \tau_1}{\Gamma \vdash \mathrm{RComb}(f_r, x_r) : \tau_2}$$

$$\frac{\Gamma, x : \tau_1 \vdash t_r : \tau_2}{\Gamma \vdash \mathrm{RAbs}(x, \tau_1, t_r) : \mathrm{fun}(\tau_1, \tau_2)}$$

Lemma (Polymorphic Constant Instantiation Admissibility). If $\kappa_c : \tau_{\text{gen}}$ and $\tau \preceq \tau_{\text{gen}}$, then `"RConst"(kappa_c, tau)` is a well-formed resolved term and may appear in any rule premise requiring type τ.

Remark (No Monomorphic Lockout). A constant introduced once at principal schema (e.g. $\mathrm{fun}(\alpha, \alpha)$) is usable at all admissible instances (e.g. $\mathrm{fun}(\mathrm{bool}, \mathrm{bool})$, $\mathrm{fun}(\mathrm{int}, \mathrm{int})$), rather than requiring one separately named constant per instance type.

6.3. Stability Theorem for Scope Mutation

Theorem (Resolved-Theorem Stability Under Scope Mutation). If $\Sigma_c; \Gamma \vdash t \Downarrow t_r$ and $\Gamma \vdash t_r : \tau$, then for any later signature stack mutation sequence μ (push/pop/shadowing on non-reserved names), the established typing fact for t_r remains valid:

$$\Gamma \vdash t_r : \tau$$

because t_r contains fixed resolved constant identities and does not re-run named lookup.

This theorem is the formal bridge behind one-shot resolution stability claims.

7. Substitution and Alpha-Equivalence

7.1. Type Substitution

Type substitution is a mapping `theta : type_variable -> hol_type` that extends structurally to types and terms.

For a type variable α,

- $\theta(\alpha)$ if defined,
- otherwise α.

For type application $k(\tau_1, ..., \tau_n)$,

- apply θ recursively to each argument.

7.2. Term Substitution

Term substitution is a finite map from variables to terms. It must satisfy two constraints:

1. Type preservation: replacement terms match the declared type of replaced variables.
2. Capture avoidance: bound variables may require renaming before substitution under abstraction.

7.3. De Bruijn Shifting (for `BETA`)

The De Bruijn core is typed. Binder-domain type labels are part of core syntax and are never erased.

Core grammar fragment:

$$d ::= \mathrm{DBound}(k, \tau) \mid \mathrm{DFree}(x, \tau) \mid \mathrm{DConst}(c, \tau) \mid \mathrm{DComb}(d_1, d_2) \mid \mathrm{DAbs}(\tau, d)$$

To make beta operationally precise, we fix two recursive operators on typed De Bruijn terms: `shift` and `subst`.

For shift, written $\mathrm{shift}(\delta, c, d)$ with increment δ and cutoff c:

- $\mathrm{shift}(\delta, c, \mathrm{DBound}(k, \tau)) = \mathrm{DBound}(k, \tau)$, when $k < c$.
- $\mathrm{shift}(\delta, c, \mathrm{DBound}(k, \tau)) = \mathrm{DBound}(k + \delta, \tau)$, when $k \geq c$.
- $\mathrm{shift}(\delta, c, \mathrm{DComb}(f, x)) = \mathrm{DComb}(\mathrm{shift}(\delta, c, f), \mathrm{shift}(\delta, c, x))$.
- $\mathrm{shift}(\delta, c, \mathrm{DAbs}(\tau, t)) = \mathrm{DAbs}(\tau, \mathrm{shift}(\delta, c + 1, t))$.

For substitution, written $\mathrm{subst}(j, s, d)$:

- $\mathrm{subst}(j, s, \mathrm{DBound}(k, \tau)) = s$, when $k = j$ and $\mathrm{type\_of}(s) = \tau$.
- $\mathrm{subst}(j, s, \mathrm{DBound}(k, \tau)) = \mathrm{DBound}(k, \tau)$, when $k \neq j$.
- $\mathrm{subst}(j, s, \mathrm{DComb}(f, x)) = \mathrm{DComb}(\mathrm{subst}(j, s, f), \mathrm{subst}(j, s, x))$.
- $\mathrm{subst}(j, s, \mathrm{DAbs}(\tau, t)) = \mathrm{DAbs}(\tau, \mathrm{subst}(j + 1, \mathrm{shift}(1, 0, s), t))$.

Then the typed De Bruijn beta contraction used by the kernel is fixed as

$$\mathrm{beta}((\mathrm{DAbs}(\tau, t))u) = \mathrm{shift}(-1, 0, \mathrm{subst}(0, \mathrm{shift}(1, 0, u), t))$$

with the side condition $\mathrm{type\_of}(u) = \tau$.

Invariant (Typed-Core Injectivity). De Bruijn structural equality is type-sensitive:

$$\mathrm{DAbs}(\tau_1, t_1) = \mathrm{DAbs}(\tau_2, t_2) \Rightarrow \tau_1 = \tau_2 \land t_1 = t_2$$

Hence abstractions that differ only by binder-domain type are distinct core terms and cannot be merged by structural matching.

Theorem (Lowering Preserves Typing). If named elaboration succeeds and named typing holds:

$$\Sigma_c; \Gamma \vdash t \Downarrow d \land \Gamma \vdash t : \tau$$

then the lowered typed De Bruijn term satisfies core typing:

$$\Gamma \vdash d : \tau$$

Proof is by induction on the structure of t, using the elaboration clauses and the typed constructors of `DBound/DFree/DConst/DComb/DAbs`.

Theorem (Lifting Preserves Typing up to Alpha). If $\mathrm{Term}_\uparrow\ d \mapsto t$ and $\Gamma \vdash d : \tau$, then:

$$\Gamma \vdash t : \tau$$

and for any second successful lift $\mathrm{Term}_\uparrow\ d \mapsto t'$, we have $t \underset{\alpha}{\equiv} t'$. Thus lift choices are presentation variants, not typing variants.

Theorem (Boundary Commutation with Capture-Avoiding Substitution). Whenever both sides are defined, lowering commutes with substitution:

$$\mathrm{Lower}(t[s/x]) \cong \mathrm{DbSubst}(x, \mathrm{Lower}(s), \mathrm{Lower}(t))$$

and therefore beta-contraction in named syntax and beta-contraction in typed De Bruijn syntax agree modulo $\underset{\alpha}{\equiv}$ after lift.

7.4. Alpha-Equivalence

Alpha-equivalence, written $t_1 \underset{\alpha}{\equiv} t_2$, identifies terms up to systematic renaming of bound variables. It is required by multiple kernel operations, including theorem transitivity-style checks where structural syntax should not distinguish alpha-variants.

The specification uses the following mandatory properties:

1. Reflexive: $t \underset{\alpha}{\equiv} t$.
2. Symmetric: $t_1 \underset{\alpha}{\equiv} t_2 \Rightarrow t_2 \underset{\alpha}{\equiv} t_1$.
3. Transitive: $t_1 \underset{\alpha}{\equiv} t_2 \wedge t_2 \underset{\alpha}{\equiv} t_3 \Rightarrow t_1 \underset{\alpha}{\equiv} t_3$.
4. Congruence for constructors (`Comb`, `Abs`) and proposition/equality contexts.

For De Bruijn forms, structural equivalence $\cong$ is used as the canonical representative-level equality. The boundary lemmas below connect $\underset{\alpha}{\equiv}$ and $\cong$.

8. Boundary Conversion and Scoped Shadowing

QED currently uses a two-layer boundary:

- external-facing terms/theorems are in named syntax;
- kernel rule cores execute on De Bruijn syntax.

Boundary conversion functions are used with the following Haskell-style signatures:

$$\mathrm{Term}_\downarrow :: \mathrm{Term} \to \mathrm{DbTerm}?$$

$$\mathrm{Term}_\uparrow :: \mathrm{DbTerm} \to \mathrm{Term}?$$

$$\mathrm{Thm}_\downarrow :: \mathrm{Thm} \to \mathrm{DbSequent}?$$

$$\mathrm{Thm}_\uparrow :: \mathrm{DbSequent} \to \mathrm{Thm}?$$

Here $\mathrm{Term}_\downarrow$ lowers named terms to De Bruijn terms, and $\mathrm{Term}_\uparrow$ reconstructs named terms from De Bruijn terms. Likewise, $\mathrm{Thm}_\downarrow$ lowers named sequents to De Bruijn sequents, and $\mathrm{Thm}_\uparrow$ lifts De Bruijn sequents back to named boundary objects. All four conversions are partial and may fail at the boundary.

For theorem objects:

$$\mathrm{Thm}_\downarrow\ A_p \vdash p = A_d \vdash p_d$$

and

$$\mathrm{Thm}_{\uparrow}\ A_d \vdash p_d = A_{p'} \vdash p'$$

defined pointwise by $\mathrm{Term}_{\downarrow}$ and $\mathrm{Term}_{\uparrow}$ on assumptions and conclusion.

8.1. Boundary Conversion Properties

The boundary metatheory relies on the following invariants.

Lemma (Alpha-Invariant Lowering).

$$\frac{t_1 \underset{\alpha}{\equiv} t_2}{\mathrm{Term}_{\downarrow}\ t_1 \cong \mathrm{Term}_{\downarrow}\ t_2}$$

Lemma (Round-Trip Stability up to Alpha).

$$\frac{\mathrm{Term}_{\downarrow}\ t \mapsto d,\ \ \mathrm{Term}_{\uparrow}\ d \mapsto t'}{t' \underset{\alpha}{\equiv} t}$$

Lemma (Lift Choice Congruence). If $\mathrm{Term}_{\uparrow}\ d \mapsto t_1$ and $\mathrm{Term}_{\uparrow}\ d \mapsto t_2$, then $t_1 \underset{\alpha}{\equiv} t_2$.

Property (Name-Insensitive Rule Matching). Any named-side premise matching required by primitive rules is interpreted modulo $\underset{\alpha}{\equiv}$. Binder spellings are presentation-level choices and cannot change rule applicability.

Lemma (Type-Sensitive Core Matching). Boundary lowering preserves binder-domain type labels. Therefore, if two named abstractions differ in binder-domain type, their lowered typed De Bruijn forms are not structurally equal:

$$\mathrm{Term}_{\downarrow}\ (\lambda(x : \tau_1).t_1) \neq \mathrm{Term}_{\downarrow}\ (\lambda(y : \tau_2).t_2)$$

whenever $\tau_1 \neq \tau_2$, even if bodies are De Bruijn-index isomorphic.

Lemma (Term Denotation Preservation Across Boundary). If $\mathrm{Term}_{\downarrow}\ t \mapsto d$, then for every valuation/ model pair (ρ, M):

$$\mathrm{denote}(t, \rho, M) = \mathrm{denote}(d, \rho, M)$$

where the right-hand side denotes De Bruijn evaluation under the environment induced by ρ.

Lemma (Sequent Denotation Preservation Across Boundary). If $\mathrm{Thm}_{\downarrow}\ A_p \vdash p \mapsto A_d \vdash p_d$, then semantic validity is preserved:

$$\left(A_p \vdash p\right)\ \ \mathrm{valid}\ \Leftrightarrow \left(A_d \vdash p_d\right)\ \ \mathrm{valid}$$

Theorem (Semantic Rule Lifting Safety). Assume a primitive core rule

$$R_d : \mathrm{DbSequent}^n \to \mathrm{DbSequent}?$$

is semantically preserving on its defined domain:

$$\mathrm{valid}(x_d) \Rightarrow \mathrm{valid}\left(R_{d(x_d)}\right)$$

Then the lifted named rule

$$R\ x = \mathrm{Thm}_{\uparrow}\ \left(R_d\ \left(\mathrm{Thm}_{\downarrow}\ x\right)\right)$$

is semantically preserving on all inputs where boundary conversions succeed. Consequently, rule lifting preserves both structure (modulo alpha) and denotation.

Property (Diagrammatic Commutation on Successful Conversions). The lifting relation is visualized by the following commutative diagram (structural and semantic commutation):

$$\begin{tikzcd}
\mathrm{DbSequent} \mathbin{/} \cong \arrow[r, "R_d"] & \mathrm{DbSequent} \mathbin{/} \cong \arrow[d, "\mathrm{Thm}_{\uparrow}"'] \\
\mathrm{NamedSequent} \mathbin{/} \underset{\alpha}{\equiv} \arrow[u, "\mathrm{Thm}_{\downarrow}"'] \arrow[r, "R"] & \mathrm{NamedSequent} \mathbin{/} \underset{\alpha}{\equiv}
\end{tikzcd}$$

Operationally, this means:

$$\frac{\mathrm{Thm}_{\downarrow}\ x \mapsto x_d,\ \ R_d\ x_d \mapsto y_d,\ \ \mathrm{Thm}_{\uparrow}\ y_d \mapsto y}{R\ x \mapsto y}$$

on all inputs where boundary conversions and core rule execution succeed, together with

$$\mathrm{valid}(x) \Rightarrow \mathrm{valid}(y)$$

under the denotation-preservation lemmas above.

Remark (Not a Strict Bijection). This correspondence is not a strict named-to-De Bruijn bijection:

- alpha-variant named terms collapse to one De Bruijn equivalence class;
- lifting from De Bruijn chooses a representative named binder presentation;
- boundary conversions are partial, so the mapping is defined only on well-formed successful cases.

Boundary Discipline

Primitive rules are interpreted over `DbSequent` core objects. Named `Thm` values are boundary presentations. Conversion failure is a boundary failure, not a logical derivation.

8.2. Scoped Shadowing Properties

Let a signature state be a stack

$$S = [S_0, S_1, ..., S_n]$$

where S_n is the innermost scope.

Notation and intent:

- $P(S)$ denotes pushing a fresh, empty scope.
- $A(S, c : \tau)$ denotes inserting a constant into the current scope.
- $Q(S)$ denotes popping the innermost scope.

These operators provide the abstract logical calculus for scope push/add/pop.

Lookup is defined by:

$$L(S, c) = S_{j(c)}$$

where j is the greatest index such that $c \in \mathrm{dom}(S_j)$.

Insertion in current scope:

$$A(S, c : \tau)$$

is allowed iff $c \notin \mathrm{dom}(S_n)$.

From these definitions:

Proposition (Shadowing Determinism). if c is defined in innermost scope S_n, then $L(S, c) = S_{n(c)}$. Proof. By definition, lookup returns $S_{j(c)}$ for the greatest index j with $c \in \mathrm{dom}(S_j)$. Since $c \in \mathrm{dom}(S_n)$ and n is maximal index of the stack, the greatest such index is $j = n$. Hence $L(S, c) = S_{n(c)}$.

Proposition (Outer Restoration by Pop). if $S' = P(S)$, $A(S', c : \tau_1) = S''$, and $Q(S'') = S$, then $L(S, c) = L(Q(S''), c)$. Proof. $P(S)$ adds one fresh empty innermost frame. $A(S', c : \tau_1)$ modifies only that fresh frame. Q then removes exactly that frame, restoring all original frames pointwise. Therefore lookup of any symbol in restored stack is identical to lookup in S, so $L(S, c) = L(Q(S''), c)$.

Proposition (Scope-Local Uniqueness). if c is already defined in innermost scope S_n, then $A(S, c : \tau)$ fails. Proof. Admissibility of $A(S, c : \tau)$ requires $c \notin \mathrm{dom}(S_n)$ by definition. Given $c \in \mathrm{dom}(S_n)$, the premise is false, so no insertion rule instance exists; hence $A(S, c : \tau)$ fails.

Theorem (Resolution Freeze under Scope Mutation). Let μ be any finite sequence of operations from `{push, add, pop}` on non-reserved names, and let $\Sigma_c; \Gamma \vdash t \Downarrow d$. If μ is applied to obtain a later stack $\Sigma_{c'}$, then:

$$\Gamma \vdash d : \tau$$

and all primitive-rule premises that mention d are unchanged by μ. Proof. d contains resolved constant identities, not deferred name lookups. Scope mutations alter only future named-resolution results; they do not rewrite existing resolved terms.

These properties specify the abstract behavior required of any faithful scoped-signature realization.

Scope-local shadowing properties above are lookup properties only. They do not authorize redefining committed definition heads recorded in `"DefHeads"(T)`.

9. Theorem Object and Trust Boundary

A theorem is represented mathematically as a sequent

$$\Gamma_p \vdash p$$

with p a boolean term and Γ_p a finite set of boolean assumptions.

9.1. Assumption Sets as Alpha-Quotients

To keep rule behavior binder-name invariant, assumptions are not raw syntax sets. They are finite sets of alpha-equivalence classes:

$$\Gamma_p \subseteq \mathrm{Term} \ / \underset{\alpha}{\equiv}$$

with all elements constrained to type bool.

Operations used by primitive rules are interpreted on equivalence classes:

- membership: $[a]_\alpha \in \Gamma_p$;
- union: $\Gamma_1 \cup \Gamma_2$ on classes;
- removal: $\Gamma_p - \{[a]_\alpha\}$.

Required laws:

1. Idempotence: $\Gamma \cup \Gamma = \Gamma$.
2. Commutativity: $\Gamma_1 \cup \Gamma_2 = \Gamma_2 \cup \Gamma_1$.
3. Alpha-compatibility of membership/removal: if $a \underset{\alpha}{\equiv} b$ then $[a]_\alpha = [b]_\alpha$ and
$$(\Gamma - \{[a]_\alpha\}) = (\Gamma - \{[b]_\alpha\})$$
4. Finiteness preservation under union/removal.

All assumption-manipulating rules (`ASSUME`, `TRANS`, `EQ_MP`, `DEDUCT_ANTISYM_RULE`) are read in this quotient semantics.

The trusted boundary condition is:

- external contexts cannot directly construct theorem values,
- theorem values are produced only by primitive kernel inference functions.

- theorem values store resolved terms; primitive rules and theorem aliases do not re-run constant lookup against the current signature stack.

This boundary is the core LCF invariant.

Kernel Integrity Condition

If external code can fabricate theorem values, the entire soundness argument collapses, regardless of how correct individual inference rules appear.

10. Definitional Extension Discipline (Conservativity Gate)

QED permits signature growth only through conservative admissible extension gates.

Rule Schema (New Constant by Definition). Given base theory T and fresh constant c:

$$\frac{c \notin \mathrm{DefHeads}(T), c \notin \mathrm{Reserved}, \Gamma \vdash r : \tau, \mathrm{closed}(r, \Gamma = \mathrm{empty}), \mathrm{acyclic}(r, c), \mathrm{TVars}(r) \subseteq \mathrm{TVars}(\tau)}{T \mapsto T + \{\mathrm{def}\ c : \tau = r\}}$$

Mandatory side conditions:

1. **Freshness**: c does not occur in global definitional history `"DefHeads"(T)` and is not reserved.
2. **Typedness**: r is well-typed at declared type τ.
3. **Closedness**: r has no free term variables (global-definition discipline).
4. **Non-circularity**: r does not mention c (directly or via definitional cycle).
5. **Type-Variable Closure**: free type variables in r are constrained by the declared head type:

$$\mathrm{TVars}(r) \subseteq \mathrm{TVars}(\tau)$$

   so no ghost type variable can appear only in the body.

Safety Note (Ghost Type Variables). Without Condition 5, an `INST_TYPE` step may change only the definition body instance while leaving the head constant unchanged, which breaks definitional conservativity.

Policy Clarification (Shadowing vs Definitional Freshness). Scoped insertion rules may allow same-name shadowing for ordinary local constants. Definitional extension is stricter: definitional heads are introduced with globally fresh names and are never shadow-reused. This resolves any apparent tension between local scope shadowing and global definition soundness.

Theorem (Conservativity of Definitional Extension). Let T' be obtained from T by one admissible definitional extension above. For every theorem statement p_0 in the old language of T:

$$T' \vdash p_0 \Rightarrow T \vdash p_0$$

Hence new definitions add abbreviatory power without increasing provability over the old signature.

10.1. Definition Admissibility Judgment

To make the definition gate reusable by later rules and metatheory, we package side conditions into one judgment:

$$\mathrm{DefOK}(T, c : \tau = r)$$

defined as the conjunction of Conditions 1–5 above.

Admissible extension step:

$$\mathrm{DefOK}(T, c : \tau = r) \Rightarrow T \mapsto T + \{\mathrm{def}\ c : \tau = r\}$$

Lemma (Definition Theorem Shape). If $\mathrm{DefOK}(T, c : \tau = r)$ and $T \mapsto T'$, then the generated definition theorem has empty assumptions:

$$\vdash c = r$$

and is well-typed at bool.

Lemma (Instantiation Coherence for Definitions). If $\mathrm{DefOK}(T, c : \tau = r)$ and θ is any admissible type substitution on $\mathrm{TVars}(\tau)$, then

$$\theta(c = r) = \theta(c) = \theta(r)$$

and no body-only type variable can be changed independently of the head.

Corollary (No Ghost-Type Instantiation Drift). Under DefOK, `INST_TYPE` cannot produce contradictory definition instances of one constant by varying a type variable that appears only in the body.

This subsection is the structural contract tying definitional extension to the primitive `INST_TYPE` rule.

11. Controlled Specification Extension Discipline

To support HOL-style derived-theory construction without unrestricted axiom injection, QED treats specification introduction as a derived discipline over Choice + `DefOK`, not as an independent primitive inference rule.

Definition (Specification Admissibility). Given theory T, fresh constant head c, and predicate $P(x)$, write:

$$\mathrm{SpecOK}(T, c : \tau, P)$$

iff all conditions below hold:

1. Freshness: $c \notin \mathrm{DefHeads}(T)$, $c \notin \mathrm{Reserved}$, and c is not already in theory symbol tables.
2. Witness theorem shape: there exists a theorem with empty assumptions ($\Gamma = \mathrm{empty}$):
$$\vdash \exists x : \tau . P(x)$$
3. Term closure: $P(x)$ has no free term variable except x.
4. Type-variable closure:
$$\mathrm{TVars}(P) \subseteq \mathrm{TVars}(\tau)$$
(no type variable appears in P that is absent from the declared type of c).
5. Strict type-schema lock:
$$\mathrm{Schema}(c) = \mathrm{Gen}(\tau)$$
for this admission step only.
6. No implicit widening: no type variable absent from τ may be generalized into $\mathrm{Schema}(c)$ by this step.

Derived Rule Schema (Specification via Choice + Definitional Admission).

$$\frac{\vdash \exists x : \tau . P(x), \mathrm{SpecOK}(T, c : \tau, P), \mathrm{DefOK}(T, c : \tau = @(\lambda x : \tau . P(x)))}{T \mapsto T + \{\mathrm{def}\ c : \tau = @(\lambda x : \tau . P(x))\}, \vdash P(c)}$$

This is the only admissible introduction path for specification constants in this stage.

Meta-Constraint (No Hidden Side Conditions). Any admission-procedure check used by `SpecOK` must correspond to one of the six explicit conditions above; no additional silent premise is allowed.

Theorem (Conservativity of Specification Extension). If T' is obtained from T by one admissible `SpecOK` step admitting fresh head c, and φ is a sentence in the old language of T, then:

$$T' \vdash \varphi \Rightarrow T \vdash \varphi$$

Constructive proof (derivation elimination for one spec head). Assume $T' = T + \{\mathrm{def}\ c : \tau = @(\lambda x : \tau . P(x))\}$ under `SpecOK`. Let D' be a finite derivation object in T' for φ. Define a recursive eliminator $\mathrm{erase\_spec}_{c(D')}$ by structural recursion:

1. leaf/axiom cases not mentioning c: unchanged;

2. primitive-rule nodes: map recursively on all premise sub-derivations, then rebuild the same rule node;
3. occurrences of the admitted theorem $\vdash P(c)$: replace by a derived sub-derivation using the witness theorem $\vdash \exists x : \tau . P(x)$ plus Choice axiom instance and the defining equation for c;
4. any theorem node whose conclusion is in the old language and does not mention c is rewritten only through Steps 1–3.

Well-foundedness: recursion is on strict sub-derivation size. Correctness invariant (proved by induction on D'):

- every rewritten node is derivable in T;
- old-language conclusions are preserved;
- no new head outside old language is introduced.

Applying the invariant to root D' yields a derivation in T of the same old-language sentence φ, establishing conservativity.

12. Global Admissibility Envelope

The kernel-level soundness argument uses a single admissibility envelope:

1. theorem values arise only from primitive rules or admissible extension gates;
2. primitive rules consume well-formed resolved sequents;
3. definitional extension steps must satisfy `"DefOK"`;
4. type-constructor extension steps must satisfy `"TypeDefOK"` (non-emptiness witness gate);
5. specification extension steps must satisfy `"SpecOK"` (derived over Choice + `DefOK`);
6. boundary conversion failures are non-derivational failures.

All later soundness obligations are stated relative to this envelope, so no rule silently bypasses definition admissibility constraints.

13. Primitive Inference Rules

QED follows a HOL Light style primitive interface:

`REFL`, `ASSUME`, `TRANS`, `MK_COMB`, `ABS`, `BETA`, `EQ_MP`, `DEDUCT_ANTISYM_RULE`, `INST_TYPE`, `INST`.

Judgmental convention in this section:

- core rule premises are checked on resolved terms/sequents;
- named forms are boundary presentations of those same rules;
- assumption-set operations are interpreted on alpha-quotient classes.

Each primitive rule must be specified by:

1. Input theorem and term constraints.
2. Side conditions (typing, freeness, alpha-matching, etc.).
3. Output sequent.
4. Failure condition classification.

For example, selected rules can be presented in antecedent style:

- `REFL`: for any term t, conclude $\vdash t = t$.
- `ASSUME`: for any boolean proposition p, conclude $p \vdash p$.

$$\frac{A_p \vdash s = t,\ B_p \vdash t = u}{A_p \cup B_p \vdash s = u}$$

$$\frac{A_p \vdash p = q,\ B_p \vdash p}{A_p \cup B_p \vdash q}$$

Detailed formal side conditions stated here are authoritative for all later rule use and metatheory.

13.1. Rule Schema: `REFL`

Input:

- a well-formed term t.

Output:

- theorem $\vdash t = t$.

Side conditions:

1. t must be typable.
2. equality constructor must be formed at the type of t.

Failure clauses:

1. malformed term input;
2. type construction failure in equality formation.

Antecedent form:

$$\frac{\Gamma \vdash t_r : T}{\vdash t_r = t_r}$$

13.2. Rule Schema: `ASSUME`

Input:

- a proposition term p.

Output:

- theorem $p \vdash p$.

Side conditions:

1. p must have type bool.
2. assumption set representation must admit p.

Failure clauses:

1. non-boolean proposition;
2. invalid assumption-set insertion.

Antecedent form:

$$\frac{\Gamma \vdash p_r : \text{bool} \ , \ \ [p_r]_\alpha \in \Gamma_p}{\Gamma_p \vdash p_r}$$

13.3. Rule Schema: `TRANS`

Input:

- theorem $A_p \vdash s = t$;
- theorem $B_p \vdash t = u$.

Output:

- theorem $A_p \cup B_p \vdash s = u$.

Side conditions:

1. both conclusions must be equalities;
2. the middle terms must match up to alpha-equivalence and type consistency;
3. under boundary lowering, core matching is performed on typed De Bruijn terms (including binder-domain labels), not on type-erased structure.

Failure clauses:

1. non-equality conclusion in either premise theorem;
2. middle-term mismatch;
3. type inconsistency in chained equality;
4. boundary/core mismatch caused by typed-core inequality.

Antecedent form:

$$\frac{A_p \vdash s = t,\ B_p \vdash t = u}{A_p \cup B_p \vdash s = u}$$

13.4. Rule Schema: `MK_COMB`

Input:

- theorem $A_p \vdash f = g$;
- theorem $B_p \vdash x = y$.

Output:

- theorem $A_p \cup B_p \vdash fx = gy$.

Side conditions:

1. both premise conclusions must be equalities;
2. f and g must have function type with argument type matching x and y;
3. codomain types of f and g must coincide.

Failure clauses:

1. non-equality premise theorem;
2. function-domain mismatch for application;
3. codomain inconsistency across the two function sides.

Antecedent form:

$$\frac{A_p \vdash f = g,\ B_p \vdash x = y}{A_p \cup B_p \vdash fx = gy}$$

13.5. Rule Schema: `ABS`

Input:

- variable term x;
- theorem $A_p \vdash s = t$.

Output:

- theorem $A_p \vdash \lambda(x : \tau).s = \lambda(x : \tau).t$.

Side conditions:

1. x must be a variable term;
2. premise conclusion must be an equality;
3. x must not occur free in assumptions A_p.

Failure clauses:

1. non-variable abstraction binder;
2. non-equality premise theorem;
3. free-variable violation in assumption set.

Antecedent form:

$$\frac{A_p \vdash s = t}{A_p \vdash \lambda(x : \tau).s = \lambda(x : \tau).t}$$

13.6. Rule Schema: `BETA`

Input:

- a typed De Bruijn beta-redex term of shape $(\mathrm{DComb}(\mathrm{DAbs}(\tau, t), u))$.

Output:

- theorem $\vdash (\mathrm{DComb}(\mathrm{DAbs}(\tau, t), u)) = \mathrm{beta}((\mathrm{DAbs}(\tau, t))u)$.

This is the De Bruijn substitution form (with the usual shift and shift-back to avoid capture). It corresponds to the named rule $\vdash ((\lambda(x : \tau).s)u) = s[u/x]$ under boundary conversion.

Side conditions:

1. the redex must be well-typed;
2. substitution is capture-avoiding;
3. the contracted De Bruijn term must be well-scoped, so the final $\mathrm{shift}(-1, 0, \ldots)$ step is defined.
4. the argument type must equal the abstraction binder-domain label: $\mathrm{type\_of}(u) = \tau$.

Lemma (Well-Scoped Beta Contraction Safety). For well-typed and well-scoped input redexes, the contraction

$$\mathrm{beta}((\mathrm{DAbs}(\tau, t))u) = \mathrm{shift}(-1, 0, \mathrm{subst}(0, \mathrm{shift}(1, 0, u), t))$$

does not create dangling indices.

Failure clauses:

1. input is not a beta-redex of the required shape;
2. type inconsistency in redex construction;
3. boundary reconstruction failure;
4. binder-domain label mismatch in typed De Bruijn core.

Antecedent form:

$$\frac{r = \mathrm{DComb}(\mathrm{DAbs}(\tau, t), u), \mathrm{welltyped}(r), \mathrm{type\_of}(u) = \tau}{\vdash r = \mathrm{beta}(\mathrm{DComb}(\mathrm{DAbs}(\tau, t), u))}$$

13.7. Rule Schema: `EQ_MP`

Input:

- theorem $A_p \vdash p = q$;
- theorem $B_p \vdash p$.

Output:

- theorem $A_p \cup B_p \vdash q$.

Side conditions:

1. first premise must conclude an equality proposition;
2. left side of equality must match the second premise conclusion up to alpha-equivalence;
3. all involved terms must be boolean propositions.

Failure clauses:

1. first premise is not an equality theorem;
2. proposition mismatch between equality lhs and premise theorem;
3. non-boolean proposition in premises.

Antecedent form:

$$\frac{A_p \vdash p = q,\ B_p \vdash p}{A_p \cup B_p \vdash q}$$

13.8. Rule Schema: `DEDUCT_ANTISYM_RULE`

Input:

- theorem $A_p \vdash p$;
- theorem $B_p \vdash q$.

Output:

- theorem $(A_p - \{q\}) \cup (B_p - \{p\}) \vdash p = q$.

Side conditions:

1. both premises must conclude propositions;
2. subtraction from assumption sets must be defined by alpha-aware proposition equality;
3. resulting assumption set must remain finite.

Failure clauses:

1. malformed assumption-set subtraction;
2. proposition mismatch in set-removal targets;
3. non-propositional premise conclusion.

Antecedent form:

$$\frac{A_p \vdash p,\ \ B_p \vdash q}{(A_p - \{q\}) \cup (B_p - \{p\}) \vdash p = q}$$

13.9. Rule Schema: `INST_TYPE`

Input:

- type substitution θ;
- theorem $A_p \vdash p$.

Output:

- theorem $\theta(A_p) \vdash \theta(p)$.

Side conditions:

1. substitution domain must contain only type variables;
2. every target type in θ must be a well-formed type admitted by the current type-extension discipline (`TypeDefOK`-closed theory state);
3. substitution application must preserve term well-typedness;
4. if the theorem being instantiated is a definitional theorem produced under `DefOK`, the instantiated head/ body pair must satisfy definitional instantiation coherence (no body-only type drift);
5. for every constant occurrence `"RConst"(kappa_c, tau_i)` in the theorem, instantiated type arguments must still satisfy $\tau_i \preceq \tau_{\text{gen}(\kappa_c)}$;
6. theorem structure must be preserved under parallel type substitution.

Failure clauses:

1. invalid substitution mapping (non-type-variable key or malformed target type);
2. inadmissible type target (violates current type-admissibility gate);
3. typing failure after substitution;
4. definitional coherence violation for definition-origin theorems;
5. constant-instance mismatch against principal schema;
6. malformed theorem structure under substitution.

Antecedent form:

$$\frac{A_p \vdash p, \text{valid\_ty\_subst}(\theta), \text{admissible\_ty\_image}(T, \theta), \text{def\_inst\_coherent}(\theta, A_p \vdash p), \text{const\_instance\_ok}(\theta, A_p \vdash p)}{\theta(A_p) \vdash \theta(p)}$$

Bridge Note. This rule is soundness-linked to three upstream contracts:

- definition admissibility (`DefOK`) prevents ghost-type-variable drift under instantiation;
- type admissibility (`TypeDefOK`) prevents empty-type semantic escape in substitution targets;
- constant principal-schema instantiation guard (`prec.eq`) permits polymorphic constant use without collapsing type checks;
- global admissibility envelope forbids bypassing either gate during theorem production.

13.10. Rule Schema: `INST`

Input:

- term substitution σ;
- theorem $A_p \vdash p$.

Output:

- theorem $\sigma(A_p) \vdash \sigma(p)$.

Definition (Parallel Substitution Snapshot). For $\sigma = \{x_1 \mapsto s_1, ..., x_n \mapsto s_n\}$, substitution is simultaneous: each s_i is read in the original pre-substitution context, and no right-hand side is rewritten by another entry of σ.

Example (Swap Case):

$$\sigma = \{x \mapsto y, y \mapsto x\}$$

must exchange x and y in one parallel step, not via sequential chaining.

Side conditions:

1. substitution domain must contain only variable terms;
2. each mapped term in σ must have the same type as its source variable;
3. substitution must be capture-avoiding and applied in parallel.

Failure clauses:

1. non-variable key in substitution map;
2. type mismatch in substitution pair;
3. variable capture or malformed substitution application.

Antecedent form:

$$\frac{A_p \vdash p,\ \text{valid}(\sigma)}{\sigma(A_p) \vdash \sigma(p)}$$

13.11. Rule-Level Constructive Preservation Capsules

To make rule soundness usable as a constructive component (not a black-box claim), each primitive rule has a canonical preservation capsule of the form:

$$\text{Preserve}_R : \text{valid}(\text{Premises}_R) \Rightarrow \text{valid}(\text{Conclusion}_R)$$

proved by direct construction from the rule side conditions.

Capsule statements:

1. $\text{Preserve}_{\text{REFL}}$: from typing of t, construct validity of $t = t$ by equality reflexivity.
2. $\text{Preserve}_{\text{ASSUME}}$: from boolean typing of p, extend environment obligation with hypothesis membership.
3. $\text{Preserve}_{\text{TRANS}}$: compose two equality-valid conclusions using middle-term alpha/core match and equality transitivity.
4. $\text{Preserve}_{\text{MK\_COMB}}$: apply congruence of application under function-domain/codomain typing side conditions.
5. $\text{Preserve}_{\text{ABS}}$: lift equality under abstraction using binder freshness in hypotheses.

6. $\text{Preserve}_{\text{BETA}}$: use typed De Bruijn substitution and well-scoped contraction lemma.
7. $\text{Preserve}_{\text{EQ\_MP}}$: rewrite proposition validity along boolean equality premise.
8. $\text{Preserve}_{\text{DEDUCT\_ANTISYM\_RULE}}$: combine two implication-style derivations via alpha-aware assumption removal.
9. $\text{Preserve}_{\text{INST\_TYPE}}$: transport validity through admissible type substitution (`TypeDefOK` + `DefOK` coherence + constant-instance guard).
10. $\text{Preserve}_{\text{INST}}$: transport validity through parallel capture-avoiding term substitution.

Theorem (Rule Capsule Closure). Let R range over the ten primitive rules above. If all side conditions of R hold and all premise sequents are valid, then Preserve_R yields validity of the rule conclusion. Proof is by case analysis on R and invocation of the corresponding capsule construction.

14. Soundness Strategy

The project-level soundness story is divided into six obligations.

1. Rule-level preservation: every primitive rule preserves semantic validity.
2. Definition-level conservativity: every constant-definition step used by the kernel satisfies DefOK and preserves old-language provability.
3. Type-level non-emptiness preservation: every type-constructor extension satisfies `"TypeDefOK"` so well-formed types remain semantically inhabited.
4. Specification-level conservativity: every specification step satisfies `"SpecOK"` and preserves old-language provability.
5. Interface safety: only primitive rules and admissibility-gated extensions can introduce theorem values.
6. Derivation closure: any finite derivation tree built from primitive rules plus admissible extensions is sound.

This decomposition is practical: it separates rule, gate, interface, and derivation obligations into reviewable proof blocks.

Dependency closure for these obligations is now explicit:

- Obligation 1 depends on: core typing, substitution lemmas, alpha-quotient assumptions, and semantic lifting lemmas.
- Obligation 2 depends on: definitional side conditions, type-variable closure, and conservativity theorem.
- Obligation 3 depends on: type-definition witness discipline and non-empty-type construction invariant.
- Obligation 4 depends on: explicit `SpecOK` derived schema over Choice + `DefOK`, freshness/closure constraints, and specification conservativity theorem.
- Obligation 5 depends on: theorem constructor encapsulation + boundary failure discipline + extension gate encapsulation.
- Obligation 6 depends on: induction on derivation depth using Obligations 1, 2, 3, 4, and 5.

14.1. One-Page Soundness Dependency Map (Reader-First)

Instead of one dense graph, we use a layered map read in page order (top -> down). In this map, each downward arrow means "derives to next layer."

Layer 4 — Foundations (top):

- F1: Signatures + reserved symbols.
- F2: Elaboration + core typing.
- F3: Substitution + alpha laws.
- F4: Explicit infinity-anchor assumption (model-class restriction).
- F5: Primitive choice-operator axiom schema.

Layer 3 — Admissibility Gates:

- A1: Primitive rule schemas + boundary denotation lemmas.
- A2: Definitional admissibility (`DefOK`).
- A3: Type admissibility (`TypeDefOK`).

- A4: Specification admissibility (`SpecOK`, derived over Choice + `DefOK`).

Layer 2 — Preservation Obligations:

- P1: Rule-level preservation.
- P2: Definition conservativity.
- P3: Type non-emptiness preservation.
- P4: Specification conservativity (for derived `SpecOK` admissions).
- P5: Interface safety.

Layer 1 — Global Result (bottom):

- G1: Global derivation soundness.

$$\begin{tikzcd}
\text{Layer 4: Foundations} \arrow[d] \\
\text{Layer 3: Admissibility Gates} \arrow[d] \\
\text{Layer 2: Preservation Obligations} \arrow[d] \\
\text{Layer 1: Global Derivation Soundness}
\end{tikzcd}$$

Review rule: every Layer 1 claim must be traceable upward through Layer 2/3 to Layer 4 foundations.

14.2. Constructive Closure: Derivation Objects and Erasure Operators

To make closure explicit inside this paper (not deferred to external prose), we fix a constructive derivation object language and prove all closure theorems by structural recursion.

Definition (Finite Derivation Objects). For theory state T, a derivation object is an inductive tree:

$$D ::= \mathrm{Leaf}(s) \mid \mathrm{Rule}(r, [D_1, ..., D_n], s) \mid \mathrm{Gate}(g, \mathrm{cert}, D, s)$$

where s is a sequent, r is one primitive rule name, and $g \in \{\mathrm{DefOK}, \mathrm{TypeDefOK}, \mathrm{SpecOK}\}$. Write

$$T \vdash \mathrm{Derives}(D, s)$$

for the judgment “D is a valid finite derivation tree of sequent s in theory T”.

Definition (Old-Language Predicate). For base theory T_0, write $\mathrm{OldLang}_{T_0}(s)$ when all symbols in s are in the language of T_0.

Definition (Single-Head Erasure Operators). For freshly admitted head h, define three recursive operators on derivation trees:

$$\mathrm{erase\_def}_h, \mathrm{erase\_spec}_h, \mathrm{erase\_typedef}_h$$

Each operator is identity on leaves/rule nodes not mentioning h, and recursively rewrites only nodes whose justification uses the extension introducing h.

Representative recursive clause (uniform shape):

$$\mathrm{erase}_{h(\mathrm{Rule}(r,[D_1,...,D_n],s))} = \mathrm{Rule}\Big(r,\ \Big[\mathrm{erase}_{h(D_1)}, ..., \mathrm{erase}_{h(D_n)}\Big], s'\Big)$$

where s' is the old-language-normalized target sequent produced by the corresponding gate-specific rewrite.

Theorem (Definitional-Head Erasure Correctness). If $T_1 = T_0 + \{\mathrm{def}\ h : \tau = r\}$ with `DefOK` and

$$T_1 \vdash \mathrm{Derives}(D, s) \wedge \mathrm{OldLang}_{T_0}(s)$$

then

$$T_0 \vdash \mathrm{Derives}\left(\mathrm{erase\_def}_{h(D)}, s\right)$$

Proof is by structural induction on D, using the definition equation for h only inside rewrite-local subtrees and then eliminating it by expansion/contraction.

Theorem (Specification-Head Erasure Correctness). If T_1 is obtained by one `SpecOK` admission of head h, and

$$T_1 \vdash \mathrm{Derives}(D, s) \wedge \mathrm{OldLang}_{T_0}(s)$$

then

$$T_0 \vdash \mathrm{Derives}\left(\mathrm{erase\_spec}_{h(D)}, s\right)$$

Proof is by structural induction on D, with the `SpecOK` witness + Choice instance used to reconstruct each occurrence of $P(h)$ without keeping h in the final old-language root.

Theorem (Type-Extension Erasure Correctness for Old Language). If $T_1 = T_0 + \left\{\mathrm{typedef}\ \frac{k}{a}\right\}$ under `TypeDefOK`, and

$$T_1 \vdash \mathrm{Derives}(D, s) \wedge \mathrm{OldLang}_{T_0}(s)$$

then

$$T_0 \vdash \mathrm{Derives}\left(\mathrm{erase\_typedef}_{k(D)}, s\right)$$

Proof is by structural induction on D, using that old-language sequents do not mention k and that typing-side rewrites are confined to extension-local nodes.

Corollary (Constructive Local Conservativity). Every single admissible extension step in this manuscript admits an explicit derivation-tree erasure operator that maps extended-theory derivations of old-language sequents back into base-theory derivations.

14.3. Semantic Assumptions

Assumption (Base Non-Empty Prelude Types). Initial built-in types (before user extensions) are interpreted by non-empty semantic carriers.

Assumption (Explicit Infinity Anchor). There exists a distinguished foundation type `"ind"` and a function $f : \mathrm{ind} \to \mathrm{ind}$ such that:

$$\mathrm{Injective}(f) \wedge \neg\ \mathrm{Surjective}(f)$$

This assumption is explicit and is part of the trusted baseline. No hidden realization shortcut may replace it.

Definition (Canonical Infinity-Anchor Theorem Identifier). The exported theorem name `"IND_INFINITY_AXIOM"` denotes exactly the sentence:

$$\vdash \exists f : \mathrm{fun}(\mathrm{ind}, \mathrm{ind}).\ \mathrm{Injective}(f) \wedge \neg\ \mathrm{Surjective}(f)$$

No alternate theorem shape may be treated as equivalent by presentation policy alone.

Theorem (Global Non-Empty Type Preservation). Given the base assumption above and admissible type-constructor extensions satisfying `"TypeDefOK"`, every well-formed type in the extended signature is interpreted by a non-empty carrier. Consequently, `INST_TYPE` ranges only over non-empty type interpretations.

Assumption (Classical HOL Model Discipline). The soundness argument is read under the standard HOL set-theoretic model discipline: typing and instantiation preserve denotation, and theorem validity is evaluated in that model class.

Assumption (Choice-Axiom Model Compatibility). The model class used for soundness must validate the reserved choice-operator schema introduced above; specification-derived constants are interpreted through that same choice-compatible model class.

Definition (Admissible Model Class). For theory state T, write $\mathrm{ModelClass}(T)$ for the class of models satisfying:

1. all Part I typing/denotation clauses;
2. the reserved Choice schema;
3. the explicit infinity-anchor sentence for `"ind"`;
4. every admissible extension theorem admitted by gates in T.

Assumption (Model-Class Non-Emptiness). For the base theory T_0, $\mathrm{ModelClass}(T_0)$ is non-empty.

Theorem (Semantic Non-Triviality Transfer). If $M \in \mathrm{ModelClass}(T)$ and a closed sentence ψ is not valid in M, then

$$\neg(T \vdash \psi)$$

Proof. By contrapositive of rule/gate soundness: if $T \vdash \psi$ then every $M \in \mathrm{ModelClass}(T)$ validates ψ. Therefore a countermodel M excludes derivability.

Corollary (Consistency Witness Form). If $\mathrm{ModelClass}(T)$ is non-empty, then there is no closed sentence χ such that both

$$T \vdash \chi \wedge T \vdash \neg\chi$$

under the same semantic negation operator in the model class.

Meta-Theorem (Global Conservativity Under Admissible Extensions). Let T' be obtained from T by a finite sequence of admissible steps from

$$\{\mathrm{DefOK}, \mathrm{TypeDefOK}, \mathrm{SpecOK}\} \quad (\text{where SpecOK is derived over Choice} + \text{DefOK})$$

Then for any sentence φ over the old language of T:

$$T' \vdash \varphi \Rightarrow T \vdash \varphi$$

Constructive proof (finite-step backward erasure). Let the extension sequence be

$$T = T_0 \mapsto T_1 \mapsto \ldots \mapsto T_n = T'$$

where each $T_i \mapsto T_{i+1}$ is one admissible gate step. Given any finite derivation D_n of φ in T_n, define recursively:

$$D_i = \mathrm{erase}_i(D_{i+1})$$

where erase_i is the gate-specific erasure operator for step $T_i \mapsto T_{i+1}$ (Def-head erasure, Spec-head erasure, or Typedef erasure).

By the three erasure correctness theorems above, each step preserves derivability of the same old-language root sentence:

$$T_{i+1} \vdash \mathrm{Derives}(D_{i+1}, \varphi) \Rightarrow T_i \vdash \mathrm{Derives}(D_i, \varphi)$$

Composing these implications for $i = n - 1, \ldots, 0$ yields

$$T \vdash \mathrm{Derives}(D_0, \varphi)$$

hence $T \vdash \varphi$. Therefore:

$$T' \vdash \varphi \Rightarrow T \vdash \varphi$$

constructively, by explicit recursion on derivation objects and finite-step composition.

14.4. Type Preservation Theorem for `MK_COMB`

Theorem (Type Preservation for `MK_COMB`). Assume premises $A_p \vdash f = g$ and $B_p \vdash x = y$, with

$$(\ \Gamma \vdash f_r : \mathrm{fun}(\tau_1, \tau_2) \wedge \Gamma \vdash g_r : \mathrm{fun}(\tau_1, \tau_2) \wedge \Gamma \vdash x_r : \tau_1 \wedge \Gamma \vdash y_r : \tau_1 \ )$$

Then by application typing,

$$(\ \Gamma \vdash f_r \ x_r : \tau_2 \wedge \Gamma \vdash g_r \ y_r : \tau_2 \ )$$

and therefore

$$\Gamma \vdash (f_r \ x_r = g_r \ y_r) : \mathrm{bool}$$

Proof. By application typing, both $f_r \ x_r$ and $g_r \ y_r$ have codomain type τ_2. Equality formation at a common type yields $(f_r \ x_r = g_r \ y_r)$ at type bool. Hence the `MK_COMB` output proposition preserves the theorem-object boolean invariant.

15. Part II: Engineering Realization (Informative + Conformance)

This part is intentionally downstream of Part I. It records one concrete realization strategy and executable conformance hooks, but does not redefine logic.

15.1. Engineering Correspondence

The formal clauses above map to implementation modules as follows.

- `src/kernel/types.mbt`: type constructors, decomposers, predicates, and type-level operators.
- `src/kernel/terms.mbt`: term constructors, decomposers, typing helper, and term-level operators.
- `src/kernel/thm.mbt`: theorem abstraction and primitive rule implementation.
- `src/kernel/sig.mbt`: scoped signature stack, constant registration, and definitional signature operations.

This mapping is informative: it supports coverage review and does not alter any Part I definition or theorem.

15.2. Audit Certificates and Replay Interface

Definition (Minimal Extension Certificate). For each successful admissible extension gate step, the kernel appends one audit certificate:

$$\mathrm{ExtCert} ::= (\mathrm{gate}, \mathrm{heads}, \mathrm{witness\_digest})$$

where:

- `"gate" in {"DefOK", "TypeDefOK", "SpecOK"}`
- `"heads"` is the finite list of newly admitted symbol heads for that step
- `"witness_digest"` is a stable theorem digest string for audit replay/indexing.

Constraint (Audit-Only Semantics). Extension certificates are observability artifacts only. They are never accepted as a runtime proof that bypasses any admissibility gate or theorem-admissibility check.

Definition (Admissible Theorem at State).

$$\mathrm{Admissible}(T, t_h)$$

means: there exists a Part I derivation object D such that

$$T \vdash \mathrm{Derives}(D, \mathrm{SequentOf}(t_h))$$

and every extension step referenced in D satisfies the corresponding Part I gate judgment.

Definition (Sentence in Base Language).

$$\mathrm{SentenceInLanguage}(T_0, t_h)$$

means: the theorem conclusion of t_h is a closed boolean sentence and every non-reserved symbol in it belongs to the language of T_0.

Definition (Executable Old-Language Replay Check). Given base state T_0, extended state T_1, and theorem t_h, define:

$$\mathrm{ConservativeReplayOK}(T_0, T_1, t_h) \coloneqq \mathrm{Admissible}(T_1, t_h) \wedge \mathrm{SentenceInLanguage}(T_0, t_h) \wedge \mathrm{Admissible}(T_0, t_h)$$

This is the executable regression proxy for the conservativity target over old-language sentences.

15.3. Rule-to-Implementation Mapping (Current)

- `REFL` -> `src/kernel/thm.mbt` (implemented; De Bruijn core + boundary lift).
- `ASSUME` -> `src/kernel/thm.mbt` (implemented; De Bruijn core + boundary lift).
- `TRANS` -> `src/kernel/thm.mbt` (implemented; De Bruijn core + boundary lift).
- `MK_COMB` -> `src/kernel/thm.mbt` (implemented; De Bruijn core + boundary lift).
- `ABS` -> `src/kernel/thm.mbt` (implemented; De Bruijn core + boundary lift).
- `BETA` -> `src/kernel/thm.mbt` (implemented; De Bruijn beta core + boundary lift).
- `EQ_MP` -> `src/kernel/thm.mbt` (implemented; De Bruijn core + boundary lift).
- `DEDUCT_ANTISYM_RULE` -> `src/kernel/thm.mbt` (implemented; De Bruijn core + boundary lift).
- `INST_TYPE` -> `src/kernel/thm.mbt` (implemented; De Bruijn substitution core + boundary lift).
- `INST` -> `src/kernel/thm.mbt` (implemented; De Bruijn substitution core + boundary lift).

15.4. Conformance Obligations (Part I -> Part II)

The following obligations define what counts as a faithful implementation of Part I:

1. **Rule fidelity**: each exported theorem-constructor path must correspond to one Part I primitive rule schema with identical side conditions.
2. **Boundary fidelity**: named/De Bruijn conversion functions must satisfy the Part I alpha/typing/denotation bridge lemmas on all successful inputs.
3. **Scope fidelity**: push/pop/shadowing behavior must preserve resolved-theorem stability exactly as stated in Part I scope theorems.
4. **Gate fidelity**: all extension admissions must be checked against Part I gate judgments (`DefOK`, `TypeDefOK`, `SpecOK`) without hidden implementation-only bypasses.
5. **Certificate non-authority**: audit certificates may index/replay checks but never serve as proof objects that bypass gate or rule checking.

Conformance failure in any item above is treated as an implementation defect, not as a change to the logic.

15.5. Conformance Transfer Theorem

Definition (Faithful Realization). A realization R is faithful to this manuscript iff all five conformance obligations above hold for every exported theorem-construction path and every extension-admission path.

Theorem (Implementation-to-Logic Transfer). If R is faithful and R accepts a theorem object with sequent s, then

$$\vdash s$$

is derivable in Part I.

Proof. By reconstruction on the acceptance trace produced by R:

1. every primitive-step acceptance maps to one Part I primitive rule instance (Rule fidelity);
2. every boundary conversion step is replaced by the corresponding Part I boundary lemma (Boundary fidelity);
3. every scope mutation in the trace is erased using Part I scope-stability theorems (Scope fidelity);
4. every extension acceptance is replaced by the matching Part I gate judgment (Gate fidelity);
5. certificate events are dropped (Certificate non-authority).

The reconstructed trace is a Part I derivation tree of s, hence $\vdash s$.

15.6. Design Delta vs HOL Light

QED and HOL Light share the LCF principle and primitive-rule trust model, but QED currently differs in two engineering choices:

1. Rule execution layer: HOL Light executes directly over named terms; QED executes rule cores over De Bruijn objects and lifts results to named boundaries.
2. Constant signature policy: HOL Light uses globally unique constant naming; QED currently uses scoped signature stacks with explicit shadowing.

These deltas are intentional and must be read as implementation-level policy choices, not changes to the object-logic proposition/equality calculus.

Error Semantics Status

Kernel gate/rule entrypoints use typed error channels (`LogicError` for theorem rules and `SigError` for theory/state admissions). The remaining option-style helpers are internal normalization/lookup utilities and are not trusted external admission interfaces.

15.7. Target Error Taxonomy (`LogicError`)

The target kernel-facing error type is:

LogicError ::=	TypeMismatch	typing mismatch
	| VariableCaptured	capture risk
	| NotAnEquality	equality shape required
	| NotBoolTerm	boolean proposition required
	| AlphaMismatch	alpha check failed
	| InvalidInstantiation	ill-formed substitution
	| VarFreeInHyp	binder free in assumptions
	| NotTrivialBetaRedex	invalid beta-redex shape
	| BoundaryFailure	named/db conversion failed

Intended alignment with rule-level failure clauses:

- `REFL`: malformed term or equality formation failure -> `BoundaryFailure` or `TypeMismatch`.
- `ASSUME`: non-boolean proposition -> `NotBoolTerm`.
- `TRANS`: non-equality premise -> `NotAnEquality`; middle mismatch -> `AlphaMismatch`; chain typing failure -> `TypeMismatch`.
- `MK_COMB`: non-equality premise -> `NotAnEquality`; function or argument typing mismatch -> `TypeMismatch`.
- `ABS`: non-variable binder -> `InvalidInstantiation`; binder free in assumptions -> `VarFreeInHyp`.
- `BETA`: non-redex input -> `NotTrivialBetaRedex`; redex typing failure -> `TypeMismatch`.
- `EQ_MP`: equality premise malformed -> `NotAnEquality`; lhs/premise mismatch -> `AlphaMismatch`; non-boolean proposition -> `NotBoolTerm`.
- `DEDUCT_ANTISYM_RULE`: set-removal target mismatch -> `AlphaMismatch`; non-propositional premise -> `NotBoolTerm`.
- `INST_TYPE` and `INST`: invalid substitution shape -> `InvalidInstantiation`; capture-risk boundary -> `VariableCaptured`; post-substitution typing failure -> `TypeMismatch`.

This mapping is a conformance target for engineering realizations and gives a direct bridge from Part I failure clauses to machine-checkable error constructors.

16. Concluding Remarks

This document remains a living formal artifact. Part I is maintained as the logic source of truth; Part II is maintained as a conformance-reporting layer against Part I.

Future refinement directions:

1. keep constructive erasure operators (`erase_def`, `erase_spec`, `erase_typedef`) synchronized with gate side conditions and regression tests,
2. preserve theorem-shape invariants for typedef/specification products as realization interfaces evolve,
3. preserve audit-only semantics of extension certificates (no runtime bypass semantics),
4. preserve executable old-language conservativity checks alongside kernel extension work.

Established Results

This revision establishes constructive closure inside the manuscript: reserved equality/choice discipline, two-layer judgments, alpha-quotient assumptions, boundary denotation bridges, explicit derivation-object erasure operators for Def/Spec/Type extensions, finite-step global conservativity composition, canonical infinity anchor, typedef product contracts, extension certificates, and executable old-language replay checks.

17. Appendix A: Primitive Rule Dependency Matrix

Each primitive rule is required to reference the following dependency blocks.

- `REFL` -> core typing of input term; reserved equality formation; theorem boundary constructor.
- `ASSUME` -> boolean typing in resolved layer; alpha-quotient insertion law.
- `TRANS` -> equality destructor/constructor typing; alpha-aware middle-term matching; typed De Bruijn core matching (binder-domain labels preserved); assumption union laws.
- `MK_COMB` -> function application typing; equality formation at codomain; assumption union laws.
- `ABS` -> binder well-formedness; free-variable side condition over assumption quotient; equality congruence under abstraction.
- `BETA` -> typed De Bruijn redex shape (`DAbs(tau, ...)`); binder/argument type agreement; well-scoped substitution lemmas; boundary lift stability.
- `EQ_MP` -> boolean equality premise typing; alpha-aware premise matching; assumption union laws.
- `DEDUCT_ANTISYM_RULE` -> quotient removal law; proposition typing for both conclusions; equality formation.
- `INST_TYPE` -> well-formed type substitution; typing preservation under type substitution; theorem-structure preservation; definitional-instantiation coherence under `DefOK`; non-empty type admissibility under `TypeDefOK`; constant principal-schema instance preservation (`prec.eq`).
- `INST` -> parallel capture-avoiding substitution; domain key well-formedness; typing preservation under term substitution.

Review criteria for this matrix:

- every side condition in each rule schema points to one of the dependency blocks above;
- no side condition remains as an unbound prose-only requirement.

18. Appendix B: Closure Checklist

Checklist for closure review:

1. Reserved logical equality symbol defined and non-shadowable.
2. Named elaboration and resolved core typing formally separated.
3. Scope-mutation stability theorem stated over resolved terms.
4. Assumption sets defined as finite alpha-quotient sets.
5. Boundary conversion includes denotation-preserving lemmas.
6. Rule lifting theorem upgraded from structural to semantic preservation.

7. Definitional extension side conditions include type-variable closure ($\mathrm{TVars}(r) \subseteq \mathrm{TVars}(\tau)$).
8. `DefOK` admissibility judgment and global admissibility envelope are stated and referenced.
9. Primitive-rule dependency matrix completed (10/10 coverage).
10. Soundness dependency graph present and cited by soundness obligations.
11. Failure clauses remain aligned with rule side conditions after closure edits.
12. Type-constructor extensions are gated by `TypeDefOK` with non-empty witness discipline.
13. Global theory history (`DefHeads`) is separated from local poppable scope and is monotone.
14. De Bruijn core matching is type-sensitive (no binder-domain type erasure during boundary lowering).
15. Constant typing uses principal schema + instance relation (`tau prec.eq tau_gen`) so polymorphic constants remain usable at admissible instances.
16. Primitive choice operator (`@`) and its axiom schema are explicitly stated in foundations.
17. `SpecOK` is documented as a derived admission rule (Choice + `DefOK`), not a standalone primitive rule.
18. Infinity-anchor theorem identifier (`IND_INFINITY_AXIOM`) is explicitly fixed.
19. Typedef admission persists the fixed three-theorem product contract (AbscomposeRep, Rep-range, conditional RepcomposeAbs).
20. Minimal extension certificates are emitted for `DefOK` / `TypeDefOK` / `SpecOK` and remain audit-only.
21. Executable old-language replay check (`ConservativeReplayOK`) is present for conservativity regressions.
22. Constructive derivation-object system (`Derives(D, s)`) is stated with explicit finite-tree recursion.
23. Per-gate erasure operators (`erase_def`, `erase_spec`, `erase_typedef`) and finite-step composition theorem are stated constructively.
24. Part II explicitly declares itself informative/downstream relative to Part I.
25. Conformance obligations (rule/boundary/scope/gate/certificate) are stated as implementation duties.
26. Conformance transfer theorem (faithful realization -> Part I derivability) is stated.
27. Engineering correspondence is marked non-authoritative for logical truth.
28. Error taxonomy mapping is marked as conformance target, not as independent logic source.

This checklist serves as a combined logic-closure and conformance/regression gate.

19. Appendix C: Definition and State Soundness Validation Scenarios

Validation scenarios for definition-layer completeness:

1. **Ghost-Type Rejection Scenario**: attempt def $c : \mathrm{bool} = r(\alpha)$ with $\alpha \notin \mathrm{TVars}(\mathrm{bool})$; expected result: rejected by `TVars(r) subset.eq TVars(tau)`.
2. **Instantiation Coherence Scenario**: from an admissible definition theorem $\vdash c = r$, apply INST_TYPE on head variables; expected result: instantiated head/body remain coherent as one definitional instance.
3. **Shadowing Separation Scenario**: local scope shadowing of ordinary constants is permitted; definitional head reuse is rejected by global freshness in DefOK.
4. **Old-Language Conservativity Scenario**: after admissible extension, any theorem not mentioning the new symbol is derivable iff it was derivable before.
5. **Pop-Then-Redefine Rejection Scenario**: define head c, pop local scope, attempt defining c again; expected result: rejected because $c \in \mathrm{DefHeads}(T)$ despite local lookup removal.

Together, these five scenarios provide structured evidence for definition/state-layer closure.

20. Appendix D: Type Soundness Validation Scenarios

Validation scenarios for type-extension completeness:

1. **Empty-Type Constructor Rejection Scenario**: propose a new type constructor without witness theorem; expected result: rejected by TypeDefOK admissibility gate.
2. **Witness-Carrying Type Definition Scenario**: introduce a type constructor with a valid non-emptiness witness; expected result: accepted and preserves global non-empty-type invariant.
3. **INST_TYPE Inhabitation Scenario**: apply INST_TYPE after admissible type extensions; expected result: substitutions range over inhabited types only.
4. **No Semantic Escape Scenario**: attempt to derive a theorem relying on empty-carrier semantics; expected result: blocked because no empty type can be introduced admissibly.

5. **Polymorphic Constant Instantiation Scenario**: define id at principal schema $\mathrm{fun}(\alpha, \alpha)$, then use it at $\mathrm{fun}(\mathrm{bool}, \mathrm{bool})$ and $\mathrm{fun}(\mathrm{int}, \mathrm{int})$; expected result: both uses are accepted via the instance relation $\tau \preceq \tau_{\mathrm{gen}}$, without requiring per-type renamed constants.

Together, these five scenarios provide structured evidence for type-layer closure.

21. Appendix E: Typed De Bruijn Core Validation Scenarios

Validation scenarios for typed-core/boundary consistency:

1. **Binder-Type Distinction Scenario**: compare $\lambda(x : \tau_1).x$ and $\lambda(x : \tau_2).x$ with $\tau_1 \neq \tau_2$; expected result: lowered typed De Bruijn abstractions are distinct ($\mathrm{DAbs}(\tau_1, ...) \neq \mathrm{DAbs}(\tau_2, ...)$).
2. **TRANS Middle-Term Guard Scenario**: chain two equalities whose middle terms are structurally similar but differ in binder-domain type labels; expected result: TRANS rejects by typed-core mismatch.
3. **BETA Binder Agreement Scenario**: attempt beta contraction with argument type not equal to abstraction binder-domain label; expected result: rejected before contraction.
4. **Boundary Lift Type Coherence Scenario**: lower and lift a theorem involving typed abstractions; expected result: reconstructed theorem preserves abstraction-domain types and cannot cross-type-identify abstractions.

Together, these four scenarios provide structured evidence for De Bruijn core/type-system coherence closure.

22. Appendix F: Specification and Choice Validation Scenarios

Validation scenarios for derived specification-admission discipline:

1. **Empty-Hypothesis Witness Requirement Scenario**: provide a witness theorem with non-empty assumptions for specification introduction; expected result: rejected by `SpecOK` witness-shape policy.
2. **Freshness Collision Scenario**: attempt to introduce specification constant c where c is reserved or already present in theory history; expected result: rejected by specification freshness constraints.
3. **Type-Schema Widening Scenario**: attempt specification admission where the admission procedure widens schema beyond $\mathrm{Gen}(\tau)$; expected result: rejected by strict type-schema lock.
4. **Derived-Path Integrity Scenario**: attempt to admit specification constant without explicit Choice + `DefOK` derivation trace; expected result: rejected because `SpecOK` is derived-only in this stage.
5. **Old-Language Conservativity Scenario**: after admissible specification extension, prove a sentence not mentioning the new symbol; expected result: derivable iff derivable before extension.

Together, these five scenarios provide structured evidence for specification-layer closure.

23. Appendix G: Part II Conformance Validation Scenarios

Validation scenarios for faithful realization transfer from Part II into Part I:

1. **Rule-Fidelity Replay Scenario**: replay each exported theorem-constructor trace as a sequence of Part I primitive rule instances; expected result: one-to-one replay succeeds without introducing extra rule premises.
2. **Boundary-Fidelity Scenario**: for accepted conversions, check alpha/typing/denotation bridge lemmas against Part I boundary requirements; expected result: accepted conversions satisfy all three bridges; rejected conversions produce no theorem.
3. **Scope-Fidelity Stability Scenario**: perform push/add/pop mutations after theorem construction and replay the same theorem object; expected result: resolved-theorem premises/conclusion remain invariant exactly as Part I stability theorem states.
4. **Gate-Fidelity Scenario**: attempt to admit extensions via any path not passing `DefOK`/`TypeDefOK`/`SpecOK`; expected result: admission rejected and no theorem object produced.
5. **Certificate Non-Authority Scenario**: supply valid-looking extension certificate metadata without matching admissibility derivation; expected result: no theorem acceptance; certificates are observability only.

Together, these five scenarios provide structured evidence for Part II conformance completeness.

24. Appendix H: Claim-to-Proof Trace Matrix

This appendix gives short review paths from high-level claims to their defining clauses and closure theorems.

Claim ID	High-level claim	Definition anchor	Proof anchor
C1	Primitive derivations are semantically preserving	Primitive rule schemas (`REFL..INST`)	Rule-Level Constructive Preservation Capsules + Rule Capsule Closure
C2	Definitional extension is conservative	`DefOK` judgment	Definitional-head erasure correctness + finite-step composition
C3	Specification extension is conservative	`SpecOK` judgment	Specification-head erasure correctness + finite-step composition
C4	Type extensions do not leak empty carriers into admissible instantiation	`TypeDefOK` + non-empty construction invariant	Type-extension erasure correctness + Global Non-Empty Type Preservation
C5	Scope mutation does not invalidate resolved theorems	scoped stack judgments + one-shot resolution	Resolution Freeze under Scope Mutation
C6	Boundary conversion does not alter denotation on successful paths	`Term`/`Thm` lowering and lifting definitions	boundary denotation lemmas + semantic rule lifting theorem
C7	Global admissibility yields global conservativity for old-language sentences	Global admissibility envelope + derivation objects	finite-step backward erasure metatheorem
C8	Faithful engineering realizations cannot exceed Part I logic	Part II conformance obligations	Conformance Transfer Theorem
C9	Non-triviality is semantically witnessed	admissible model class definition	Semantic Non-Triviality Transfer + Consistency Witness Form
C10	Audit artifacts do not change derivability	extension certificate definition	certificate non-authority obligation + conformance audit scenarios

Review rule: each row is intended to be checkable in at most three navigation steps (claim -> definition -> theorem).