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, implementation-aware 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 a reader can understand the logic of the system without reading source code.

> Reading Guide
>
> The document is intentionally ordered from basic theory to derived engineering consequences. A reader should be able to stop after the foundational sections and still obtain a coherent understanding of the QED kernel.

1. 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 with double brackets, e.g. $[\![t]\!]_\rho$ for term meaning under environment ρ.

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.

2. 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.

3. 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, implementation-aligned kernel specification.

4. Foundational Theory

4.1. 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.

4.2. 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

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

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.

4.3. 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

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

and is not inserted by user signature operations.

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

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

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.

4.4. 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: $S +\!+ [F]$ denotes stack append with F as the new innermost frame. In particular, $S +\!+ [\text{empty}]$ 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 align the scoped stack API with a judgmental presentation.

4.5. 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 implementation view: `"DefHeads"(T)` can be realized as a monotone global registry/tombstone set that is not affected 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

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

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

4.6. 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 $a(k) = n$.

In implementation terms, QED uses:

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

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

4.7. 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:

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

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.

4.8. Term Grammar

Terms are generated by:

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

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

In implementation terms, QED uses:

- `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.

4.9. 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:

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

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.

5. Elaboration and Core Typing Judgments

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

5.1. 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:

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

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

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

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

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

5.2. 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:

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

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

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

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

Lemma (Polymorphic Constant Instantiation Admissibility). If $\kappa_c : \tau_{\mathrm{gen}}$ and $\tau \preccurlyeq \tau_{\mathrm{gen}}$, 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.

5.3. 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.

6. Substitution and Alpha-Equivalence

6.1. 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.

6.2. 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.

6.3. 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:

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

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

6.4. 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 $\cong$.

7. 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.

7.1. Boundary Conversion Properties

The kernel implementation 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):

$$[\![t]\!]_{\rho,M} = [\![d]\!]_{\rho,M}$$

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:

$$\big(A_p \vdash p\big)\ \ \text{valid} \ \Leftrightarrow \big(A_d \vdash p_d\big)\ \ \text{valid}$$

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:

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

Then the lifted named rule

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

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):

$$\begin{array}{ccc}
\mathrm{DbSequent}\ /\cong & \xrightarrow{\quad R_d \quad} & \mathrm{DbSequent}\ /\cong \\
\Big\uparrow{\scriptstyle \mathrm{Thm}_{\downarrow}} & & \Big\downarrow{\scriptstyle \mathrm{Thm}_{\uparrow}} \\
\mathrm{NamedSequent}\ /\underset{\alpha}{\equiv} & \xrightarrow{\quad R \quad} & \mathrm{NamedSequent}\ /\underset{\alpha}{\equiv}
\end{array}$$

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 implemented on `DbSequent`. Named `Thm` values are boundary objects. Conversion failure is treated as boundary failure, not as logical success.

7.2. 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 are the logical counterparts of scope push/add/pop in the implementation.

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 sketch: lookup chooses the greatest index j with $c \in \mathrm{dom}(S_j)$; if $c \in \mathrm{dom}(S_n)$, maximality forces $j = n$.

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 sketch: insertion in the pushed scope only affects the temporary innermost frame; after pop, stack shape and all outer mappings are restored.

Proposition (Scope-Local Uniqueness). if c is already defined in innermost scope S_n, then $A(S, c : \tau)$ fails. Proof sketch: insertion side condition requires $c \notin \mathrm{dom}(S_n)$; violating this condition blocks the rule.

These properties specify the intended behavior of `sig_push_scope`, `sig_pop_scope`, `sig_lookup_const`, and scoped insertion APIs.

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

8. 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.

8.1. 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 modules 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.

9. 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.

9.1. 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.

10. 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 implementation-level check used by `SpecOK` must correspond to one of the six explicit conditions above; no additional silent premise is allowed.

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

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

This is a mandatory proof obligation for the specification gate design.

11. 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.

12. 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 are maintained in parallel with implementation updates.

12.1. 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:

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

12.2. 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:

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

12.3. 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:

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

12.4. 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:

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

12.5. 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:

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

12.6. 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:

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

12.7. 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:

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

12.8. Rule Schema: `DEDUCT_ANTISYM_RULE`

Input:

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

Output:

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

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:

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

12.9. Rule Schema: `INST_TYPE`

Input:

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

Output:

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

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 $\tau_i \preccurlyeq \tau_{\mathrm{gen}(\kappa_c)}$;
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:

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

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 ($\preccurlyeq$) permits polymorphic constant use without collapsing type checks;
- global admissibility envelope forbids bypassing either gate during theorem production.

12.10. 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. 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 aligns the formal argument with module boundaries and test responsibilities.

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.

13.1. 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:

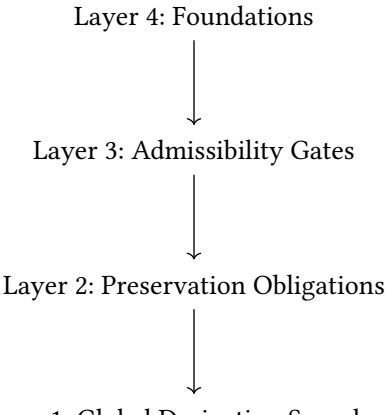
- A1: Primitive rule schemas + boundary denotation lemmas.
- A2: Definitional admissibility (`DefOK`).
- A3: Type admissibility (`TypeDefOK`).
- A4: Specification admissibility (`SpecOK`, derived over Choice + `DefOK`).

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.



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

13.2. 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 $f : \text{ind} \to \text{ind}$ such that:

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

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

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

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

No alternate theorem shape may be treated as equivalent by implementation 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.

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

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

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

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

This target is normative: any extension design failing this goal is rejected.

13.3. Type Preservation Sketch for `MK_COMB`

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

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

Then by application typing,

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

and therefore

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

So the output proposition in `MK_COMB` is well-typed as a boolean formula, which is exactly the theorem-object invariant.

14. 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.

A development task is complete only when the mathematical clause and its implementation clause are both updated.

14.1. Audit Certificates and Replay Interface

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

$$\text{ExtCert} \coloneqq (\text{gate}, \text{heads}, \text{witness\_digest})$$

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

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

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

14.2. 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).

14.3. 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 are fail-closed with 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.

14.4. Target Error Taxonomy (`LogicError`)

The target kernel-facing error type is:

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

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 normative for the final migrated API and gives a direct bridge from prose failure clauses to machine-checkable error constructors.

15. Documentation Maintenance Notes

This document remains a living formal artifact. The current revision is aligned with the implemented kernel baseline and extension-gate surface.

Near-term maintenance focus:

1. keep proof-obligation clauses and gate side conditions synchronized with regression tests,
2. preserve theorem-shape invariants for typedef/specification products as APIs evolve,
3. preserve audit-only semantics of extension certificates (no runtime bypass semantics),
4. preserve executable old-language conservativity checks alongside kernel extension work.

> Current Status
>
> This revision establishes audit-ready logical closure: reserved equality/choice discipline, two-layer judgments, alpha-quotient assumptions, boundary denotation bridges, definitional/specification conservativity gates, admissible type-extension/state-history constraints, canonical infinity anchor, typedef product contracts, extension certificates, and executable old-language replay checks.

16. 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 ($\preccurlyeq$).
- `INST` -> parallel capture-avoiding substitution; domain key well-formedness; typing preservation under term substitution.

Acceptance condition 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.

17. Appendix B: P0 Closure Checklist

Checklist for minimal audit-ready closure:

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 ≼ 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 (Abs∘Rep, Rep-range, conditional Rep∘Abs).
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.

This checklist is consumed as an implementation/regression gate (not only a pre-implementation freeze artifact).

18. Appendix C: Definition and State Soundness Audit Scenarios

Audit scenarios focused on definition-layer completeness:

1. **Ghost-Type Rejection Scenario**: attempt def $c : \text{bool} = r(\alpha)$ with $\alpha \notin \text{TVars}(\text{bool})$; expected result: rejected by `TVars(r) ⊆ 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 $c \in \text{DefHeads}(T)$ despite local lookup removal.

Passing all five scenarios is required before claiming definition/state-layer soundness closure.

19. Appendix D: Type Soundness Audit Scenarios

Audit scenarios focused on 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 $\text{fun}(\alpha, \alpha)$, then use it at $\text{fun}(\text{bool}, \text{bool})$ and $\text{fun}(\text{int}, \text{int})$; expected result: both uses are accepted via the instance relation $\tau \preccurlyeq \tau_{\text{gen}}$, without requiring per-type renamed constants.

Passing all five scenarios is required before claiming type-layer soundness closure.

20. Appendix E: Typed De Bruijn Core Audit Scenarios

Audit scenarios focused on 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 ($\text{DAbs}(\tau_1, \ldots) \neq \text{DAbs}(\tau_2, \ldots)$).
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.

Passing all four scenarios is required before claiming De Bruijn core/type-system coherence closure.

21. Appendix F: Specification and Choice Audit Scenarios

Audit scenarios focused on 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 implementation widens schema beyond $\text{Gen}(\tau)$; 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.

Passing all five scenarios is required before claiming specification-layer closure.