Internal Architecture and Type–Checking Algorithms of Contemporary Proof Assistants

1 Layered Architecture Overview

Contemporary proof assistants adopt a stratified design that separates a *tiny, trusted kernel* from progressively less–trusted outer layers. Fig. **??** summarises the canonical stack used by Coq, Lean 4, Agda, Isabelle/HOL and HOL Light[**?**, **?**, **?**, **?**, **?**].

User Interfaces (VS Code, Proof General, CoqIDE, etc.)
Elaborator / Front-End
Tactic Engine & Automation
Kernel (Type Checker + Inference Rules)
Logical Foundations (CIC, HOL, MLTT, . . .)

Figure 1: Standard proof-assistant layer cake.

2 Kernels in Practice

2.1 Size and Trusted Computing Base

Each system's kernel implements only the primitive inference rules of its object logic, plus definitional equality. Coq's kernel ($\approx$6 kLoC OCaml) checks elaborated *proof terms* for well-typedness in the Calculus of Inductive Constructions (CIC)[**?**]. Lean 4's C++ kernel is slightly larger ($\approx$9.5 kLoC) but mirrors the same minimalist philosophy[**?**]. HOL Light, in contrast, fits its entire kernel into $\approx$2.3 kLoC of OCaml by exploiting HOL's small rule set[**?**].

2.2 De Bruijn Criterion

All surveyed assistants satisfy the *de Bruijn criterion*: every externally supplied proof is reduced to kernel-checked primitives, guaranteeing that only the kernel must be trusted[**?**]. Isabelle achieves this by embedding object logics inside a generic meta-logic implemented in Standard ML[**?**].

3 Type-Checking Algorithms

3.1 Bidirectional Checking

Coq, Lean 4 and Agda implement *bidirectional* algorithms that alternate between *checking* ($\Gamma \vdash t : T$) and *inference* ($\Gamma \vdash t \Rightarrow T$) modes to localise where conversion ($\equiv$) needs to be solved[**?**, **?**]. Lean 4 pushes most reductions eagerly (strong head -normalisation) to minimise expensive definalisation at the leaves[**?**, **?**].

3.2 Conversion and Normalisation

All kernels rely on decidable conversion: two terms are definitionally equal iff their normal forms are syntactically identical under (and sometimes , ,) reduction. Coq uses a weak-head reduction with on-the-fly unfolding of transparent constants; Agda employs *proof-irrelevance annotations* to erase compile-time proofs before equality checking, thereby accelerating normalisation[?].

3.3 Universe Management

Lean 4 and Coq feature cumulative universe hierarchies. Lean's algorithm stores constraints in a union–find structure and relies on Tarjan's algorithm for $\leq$–closure[?]. Agda instead treats universe levels as first-class terms subject to unification, simplifying metaprogramming at the cost of heavier constraints[?].

4 Term Representation

Names. Coq, Lean and Agda all compile surface names to *de Bruijn indices*, ensuring -equivalent terms share a binary encoding[?, ?]. Isabelle and HOL Light keep explicit identifiers because their HOL core lacks binding-sensitive rules[?, ?].

Hash-Consing. Lean 4 hashes every node and maintains pointer equality to allow $O(1)$ conversion checks on already-normalised sub-terms[?]. HOL Light uses a related pointer-tagging trick for quick syntactic comparisons[?].

5 Elaboration Front-Ends

The elaborator translates user syntax (with holes, overloading, implicit arguments) into fully explicit kernel terms. Coq's `Vernac` language feeds an OCaml elaborator that performs first-order unification followed by metavariable resolution[?]. Lean 4's elaborator is written partly in Lean itself and exploits reflective tactics; it is intentionally *not* in the trusted base[?]. Agda's interactive mode allows partially written programs; its elaborator inserts '¿-holes and later solves them by constraint propagation[?].

6 Automation and Tactics

While tactics are outside the trusted core, their output is merely a proof term later verified by the kernel, preserving soundness[?]. Isabelle/Isar uses a declarative proof language whose statements are compiled into kernel inferences[?].

7 Persistence and Compilation

Coq compiles verified libraries into `.vo` objects that cache the normalised term and universe constraints to accelerate later replay; Lean 4 analogously stores `.olean` files[?]. HOL Light relies on OCaml's marshalled values, whereas Isabelle uses poly/ML heaps[?, ?].

8 Verified Kernels

Research projects like *Candle*, a CakeML-verified re-implementation of HOL Light, show that entire kernels can be machine-checked down to machine code[?]. MetaCoq and Lean4Lean pursue analogous self-verification for CIC and Lean respectively[?, ?, ?].

9 Conclusion (omitted)

References

[1] C. Paulin-Mohring. *Introduction to the Coq Proof Assistant* (Course notes, 2025). [1]

[2] Coq development team. *Core Language Reference Manual*, version 8.18.0. [2]

[3] T. Nipkow, L. C. Paulson, et al. *Isabelle System Manual.* [3][4]

[4] J. Harrison. *HOL Light Tutorial.* [5]

[5] X. Tang. A Comprehensive Survey of the Lean 4 Theorem Prover. *arXiv 2501.18639*, 2025. [6]

[6] M. Carneiro. Lean4Lean: a Lean 4 kernel in Lean. GitHub repository, 2025. [22]

[7] Q. Pu. A Note on the Agda Codebase—Type Checking & Reflection. HackMD, 2023. [8]

[8] Agda team. Quick Guide to Editing, Type Checking and Compiling Agda Code, 2022. [11]

[9] M. Lennon-Bertrand. Complete Bidirectional Typing for the Calculus of Inductive Constructions. *ITP 2021.* [9]

[10] R. Pollack. Type Checking in Pure Type Systems. Technical Report, 1994. [12]

[11] A. Kumar et al. Candle: A Verified Implementation of HOL Light. *ITP 2022.* [10]

[12] A. Anand et al. MetaCoq project: certifying Coq's metatheory. Project page, 2018. [9]