Verified Chain-of-Thought for Finite-Domain Constraint Reasoning: A Complete Implementation with Machine-Checked Certificates

Laurence Avent
laurence@arbitersec.com

20 July 2026

(vcot v0.2; companion to *Avent's Razor* [1])

Abstract

We describe vcot, a complete, running implementation of the Verified Chain-of-Thought (VCoT) architecture proposed in the companion paper *Avent's Razor* [1]: untrusted proposers — large language models or a deterministic solver — construct reasoning steps in a strongly-typed step language for finite-domain integer constraint problems; a 16-rule kernel computes every conclusion (proposers can never state one); completed derivations are re-checked from their serialized bytes by two independent checkers (Python, and a zero-dependency parallel Rust verifier held to exact verdict agreement by differential fuzzing over 2,197 certificates); and each accepted chain is compiled to a Lean 4 file in which *every step is re-proved as an independent lemma from exactly its cited premises* by the Lean kernel.

All results reported here are machine-checked instances, tested properties, or direct measurements; the paper tags each claim with its evidence class, and Section 7 lists what has *not* been established. Highlights: a 25-variable Einstein riddle, 4×4 and 9×9 Sudoku, cryptarithm, pigeonhole and N-queens certificates accepted by Lean 4 (the 9×9 Sudoku derivation comprises 1,348 independently re-proved steps); certified Hall-set (alldifferent) and table-constraint propagation that closes these benchmarks with *zero* search nodes; a measured, monotone relationship between exactly-computed constraint capacity κ and search speedup on 300 random instances; and an unedited session in which the language model Claude (Fable 5), acting as the untrusted proposer, solved the full Einstein riddle through the formal gate in 29 calls — 200 accepted gate actions, zero kernel-level rejections — its chain then accepted by both checkers and by Lean in 7.1 s. The trusted base of the final guarantee is the Lean kernel plus the printed axiom list; everything else — engine, gate, language models, and both in-loop checkers — is redundant or untrusted by construction.

Contents

1 Introduction

Chain-of-thought prompting improves the problem-solving behaviour of large language models [13], but the chains themselves are prose: nothing prevents a fluent, convincing derivation whose steps do not follow from one another, and the reasoning a model reports is not reliably the reasoning that produced its answer [14]. The companion framework paper [1] argues for an alternative regime — *verified chain-of-thought* (VCoT) — in which reasoning steps are typed objects checked by formal machinery, so that enforced constraints both prune search and make its outcome certifiable.

This paper is the implementation half of that program: a complete realization of the VCoT loop with no mock components, for a deliberately restricted but non-trivial reasoning domain — finite-domain integer constraint satisfaction — together with everything we can honestly

report about it. The system, `vcot` v0.2, comprises roughly 4,200 lines of Python, 2,000 lines of dependency-free Rust, and a compiler into Lean 4 [9].

1.1 What "verified" means here, exactly

For a problem P — variables $x_1, \ldots, x_n$ with finite integer domains, plus constraint axioms $A_0, \ldots, A_m$ — an accepted derivation with conclusion C compiles to a Lean 4 theorem of the form

$$\forall\, x_1 \ldots x_n \in \mathbb{Z}.\ \big(A_0 \wedge \cdots \wedge A_m\big) \longrightarrow C,$$

in a file containing no `sorry` and no `axiom` declarations, in which every derivation step is additionally re-proved as its own lemma from exactly the premises it cited. Acceptance by the Lean frontend (parser, elaborator, kernel; exit code 0) is the guarantee. For refutations, $C = \bot$: the axioms are unsatisfiable. For forced solutions, C pins every variable to a single value and a companion theorem proves $\exists\, x_1 \ldots x_n.\ A_0 \wedge \cdots \wedge A_m$; together these give machine-checked existence *and* uniqueness.

Two things are explicitly *outside* the guarantee. First, whether the axioms A_i faithfully formalize the informal problem is not machine-checkable in principle; every certificate therefore prints its axioms with human-readable labels for audit (Section 4.4). Second, the statement-level fidelity of the certificate compiler (about 370 lines) is trusted in the small sense discussed in Section 4.4: a compiler bug can only mis-state a theorem visibly or make Lean reject; it cannot make Lean accept an invalid proof of the stated theorem.

1.2 Evidence classes

To keep this paper free of overclaims, every result is tagged with one of four evidence classes:

Tag	Meaning
[L]	a concrete artifact checked by the Lean 4 kernel in this environment
[T]	a property exercised by an automated test suite (fuzzing, mutation, differential)
[M]	a quantity measured in this environment, reproducible from the repository
[D]	a design argument; reviewed code, not a machine-checked theorem

Nothing in this paper is a mathematical theorem about the system as a whole: in particular, the soundness of the 16 rule *schemas* has not been formalized in a proof assistant (only every emitted *instance* is Lean-checked), and no claim is made beyond the finite-domain constraint setting. Section 7 collects these boundaries.

1.3 Contributions

1. **A typed step language and 16-rule kernel** for finite-domain constraint reasoning, including certificate-level Hall-set (alldifferent) rules and generalized disequalities (Section 3). Conclusions are computed by the kernel, never accepted from a proposer. **[D]**, instances **[L]**
2. **Proof objects and a proposer gate**: derivations are serializable trees with scoped case analysis; untrusted proposers interact through a strict JSON protocol with typed rejections, batched steps, and model-driven case analysis (Section 3.4). **[D]**, **[T]**
3. **Certificate compilation to Lean 4**: one lemma per step from exactly its cited premises; structural steps compile to structural Lean; Hall lemmas compile to explicit case trees with `omega` leaves for predictable cost (Section 4.3). **[L]**

4. **Redundant, diverse re-checking**: an independent Python checker and a zero-dependency, memory-safe, parallel Rust verifier with compile-time LCF discipline, held to exact agreement by differential fuzzing (2,197/2,197 files; Section 4.2). **[T]**, **[M]**
5. **A proof-emitting engine** whose every propagation event is a kernel step, including bounded Hall-set detection, table constraints, and a union form of constructive disjunction; several classic benchmarks close under certified propagation with zero search (Section 5). **[M]**, instances **[L]**
6. **Measurements with stated baselines**: node-count comparisons on eight benchmark problems; a 300-instance random-CSP sweep relating exactly-computed constraint capacity κ to speedup; Lean oracle latency; Rust verifier throughput and scaling (Section 6). **[M]**
7. **Unedited language-model-in-the-loop sessions**: Claude Fable 5 solving the full Einstein riddle through the gate (200 accepted gate actions, zero kernel-level rejections, final chain Lean-accepted in 7.1 s), and a smaller Claude Haiku 4.5 session whose typed rejections and recoveries illustrate the gate's steering effect (Section 6.6). **[M]**, chains **[L]**

All artifacts — proof objects, Lean certificates, transcripts, measurement scripts — ship with the repository, and Appendix E gives the exact commands that regenerate every number in this paper.

2 Background and Related Work

LCF kernels and proof-carrying artifacts. The architectural pattern is old and deliberately so. Edinburgh LCF [2] introduced the discipline in which an abstract theorem type can only be constructed by a small set of inference-rule functions, so anything of theorem type is correct by construction. Proof- carrying code [3] moved the burden from trusting a producer to checking a certificate. `vcot` combines both: proposers are untrusted producers, derivations are the certificates, and the certificate is re-checked from bytes — ultimately by a proof assistant's kernel rather than by any code written for this project. The Rust verifier realizes the LCF discipline at compile time (its theorem type has a private constructor; Section 4.2); the Python components approximate it by convention.

Certified solving in SAT and CP. Modern SAT solvers emit DRAT certificates checked by independent tools [4]; pseudo-Boolean proof logging in the VeriPB line has extended certification to constraint-programming reasoning, including alldifferent propagation [5] and symmetry/dominance arguments [6]. Our Hall-set rules (Section 3.2) are the same mathematical content as classic alldifferent filtering [7, 8], packaged as explicit inference rules whose instances a proof assistant re-proves. We claim no advance over VeriPB-style proof logging as certification technology; the differences are of emphasis — an LLM-facing typed step protocol, a proof-assistant final oracle rather than a bespoke checker, and an end-to-end loop in which a language model authors the certificate interactively.

LLMs and formal provers. Systems such as LeanDojo [10], DeepSeek-Prover [11] and AlphaProof [12] couple language models to Lean for mathematical theorem proving: the model proposes tactics or whole proofs and the proof assistant is the referee. `vcot` sits in the same family with a different granularity and domain: steps are single constraint-propagation inferences in a fixed 16-rule system, which makes each proposal cheap to check in-loop (microseconds in the kernel, Section 4.2) and makes the final Lean artifact structurally isomorphic to the model's

accepted reasoning, one lemma per step. The restriction to finite-domain constraints is what buys this tractability; we do not compete with the systems above on open-ended mathematics.

Chain-of-thought and faithfulness. Chain-of-thought prompting [13] improves task performance, but stated reasoning can be unfaithful to the underlying computation [14]. `vcot` does not solve faithfulness in general; it changes the object of trust. The rendered chain *is* the machine-checked derivation — every sentence is backed by a step id in a certificate — so the question "did the model reason validly?" is replaced by "what do the axioms and the checked conclusion say?". Whether some other, unverified computation guided the model's *choice* of steps is deliberately out of scope: soundness does not depend on it.

Constraint programming. The engine is a textbook propagation-and-search FD solver [15], with AC-3-style worklists [16]; its only unusual property is that every deduction it makes is emitted as a kernel step, so solving and certificate construction are the same act (Section 5).

The companion framework. Avent's Razor [1] frames intelligence as constrained search: constraint capacity $\kappa = \log_2(|\mathcal{P}|/|\mathcal{P}_\mathcal{C}|)$ measures the search space a constraint system deletes, exponential gain is an idealized ceiling stated there under explicit assumptions (its strong form is a labeled conjecture), and the architectural prescription is the system this paper implements. This paper neither re-derives nor depends on the framework; Section 8 reports which of its expectations the measurements do and do not support on this domain.

3 The Step Language and Rule Kernel

3.1 Problems and propositions

A *problem* is a list of variables with finite integer domains, $x_i \mapsto D_0(x_i) \subseteq_{\text{fin}} \mathbb{Z}$, and a list of constraint axioms. The kernel sees one uniform axiom list: domain facts first ($A_i \equiv x_i \in D_0(x_i)$), then the declared constraints. Propositions are generated by:

$$\begin{aligned} \varphi \quad ::= \quad & x \in S \mid x \neq y \mid x \neq y + k \mid x = y + k \mid \textstyle\sum_i c_i x_i \leq c \\ \mid \quad & \varphi_1 \vee \cdots \vee \varphi_r \mid \varphi_1 \wedge \cdots \wedge \varphi_r \mid \bot \end{aligned}$$

with S a finite set literal and k, c, c_i integer literals. The semantics is the evident one over integer assignments; it is executable (used for witness checking and for the model-theoretic test oracle of Section 6.2) but plays no role in certificate acceptance. Well-formedness (declared variables, distinct variables in binary atoms, non-zero coefficients, non-empty connective arities) is checked before any rule fires; ill-formed propositions are the type errors of the language. Alldifferent is a modeling-level notion — a clique of pairwise $\neq$-axioms plus an *engine hint* naming the group (Section 5); the kernel has no global-constraint concept.

3.2 The sixteen rules

A derivation step names a rule, cites premises by id, and supplies rule-specific parameters. The kernel *computes* the conclusion from the premises; a proposed conclusion is never accepted, which forecloses the failure mode in which a fluent proposer launders an unsupported claim through a plausible-looking step ("conclusion laundering"). The rules are summarized below; Appendix A gives the full premise signatures and side conditions.

Rule	Reading	Conclusion computed as
AXIOM	cite axiom i	A_i
INTERSECT	two facts about x	$x \in S_1 \cap S_2$
NE_PRUNE	$x{=}v$, $y \in S$, $x \neq y$	$y \in S \setminus \{v\}$
NEOFF_PRUNE	$x{=}v$, $y \in S$, $x \neq y + k$	$y \in S \setminus \{v - k\}$ (or $\{v + k\}$)
SHIFT	$y \in S$, $x = y + k$	$x \in S + k$
FLIP	$x = y + k$	$y = x - k$
LIN_PRUNE	$\sum_i c_i x_i \leq c$, all domains	target domain filtered by interval bound
HALL_CONFLICT	m pairwise-distinct vars, $\lvert\bigcup S_i\rvert < m$	$\bot$
HALL_PRUNE	tight set ($\lvert\bigcup S_i\rvert = m$), outsider y	$y \in S_y \setminus \bigcup S_i$
EMPTY	$x \in \emptyset$	$\bot$
SPLIT	$x \in S$	$\bigvee_{v \in S} x \in \{v\}$
WEAKEN	$x \in S$, $S \subseteq S'$	$x \in S'$
EX_FALSO	$\bot$, well-formed target φ	φ
AND_INTRO / AND_ELIM	pairing / projection	$\bigwedge \varphi_i$ / φ_j
OR_ELIM	$\bigvee \varphi_i$; branches $\varphi_i \vdash \psi$	ψ

Every side condition is a finite, exact integer or set computation; there are no oracles and no floating point. The two Hall rules carry the pigeonhole and naked-subset content of alldifferent propagation [7, 8]: their premises are the m domain facts, the $\binom{m}{2}$ pairwise $\neq$-facts (in fixed lexicographic order), and for HALL_PRUNE additionally the target's domain fact and its m $\neq$-facts against the set. Making the $\neq$-clique an explicit premise list keeps the kernel free of any group concept at the price of wide steps ($m{+}1 + \binom{m}{2} + m$ premises at $m{=}4$), which we consider a good trade for auditability.

Soundness status, stated precisely: each rule is sound with respect to the integer semantics by inspection of a few lines of arithmetic **[D]**; this has *not* been proved schema-level in a proof assistant; every rule *instance* occurring in every certificate in this paper has been re-proved by the Lean kernel **[L]**; and 400-instance model-theoretic fuzzing plus 1,500-mutant adversarial testing found no semantically false accepted conclusion **[T]** (Section 6.2).

3.3 Proof objects and scoping

A derivation is a tree: a list of steps, where an OR_ELIM step carries one sub-branch per disjunct, each with a local hypothesis id bound to that disjunct and a nested step list. Scoping is lexical: a premise reference resolves to an earlier step in the same or an enclosing branch; branch-local ids go out of scope when the branch closes; ids are globally unique. The whole object serializes to JSON (Appendix B); step conclusions are deliberately *not* serialized — checkers recompute them — so a certificate cannot even express a claimed-but- underived conclusion.

Two properties worth stating because the test suite attacks them directly **[T]**: branch hypotheses are assigned by the checker from the disjunct being eliminated (never read from the file), and OR_ELIM requires all branches to conclude the *same* proposition, which becomes the step's conclusion. Hand-crafted attacks — citing branch-local ids from outside, duplicating ids, pointing the final conclusion into a branch, dropping branches, truncating a branch so its conclusion changes — are all rejected (Section 6.2).

3.4 The proposer gate

Untrusted proposers interact with a gate that accepts one JSON object per turn: a batch of steps, or a case-analysis command. The gate parses strictly, resolves premises in scope, calls the kernel, and returns either the new step id with the *computed* conclusion, or a typed rejection (machine-readable code plus message) that is fed back to the proposer. Protocol v2 adds:

- **Batches with local references**: `"#k"` in a premise list cites the id accepted for the k-th step of the same batch; a reference to a rejected item fails cleanly. Rejections do not abort the batch — later items that depended on a rejected item fail on their own, and independent items proceed.
- **Model-driven case analysis**: `begin_cases` opens an OR_ELIM on a cited Or-fact and enters case 1, returning the hypothesis id; `next_case` closes the current case and opens the next; `end_cases` completes the elimination (all case conclusions must agree); `abort_cases` discards the whole construction — a sound recovery hatch, since nothing inside an open case is visible outside it.

The accepted prefix of a session is a valid derivation at every moment: a rejected proposal changes nothing. The full protocol is reproduced in Appendix C, and Section 6.6 reports two unedited sessions with production language models.

4 Verification: Two Checkers and a Proof Assistant

Acceptance of a derivation never depends on trusting the process that produced it. Three independent decision points consume the serialized bytes.

4.1 The independent Python checker

`checker.py` re-parses the problem and every step from JSON, re-validates identifiers, scoping, branch structure and arities, and recomputes every conclusion through the kernel. It shares the proposition and kernel modules with the rest of the Python package (a deliberate, documented trust overlap — diversity is supplied by the Rust checker and finality by Lean), but no state with the engine, builder or gate: its input is bytes.

4.2 The Rust verifier: LCF at compile time, zero dependencies

`vcot-check` (∼2,000 lines) re-implements the proposition language, kernel and checker in Rust with three properties chosen for a trusted component:

- **Zero dependencies.** The crate has an empty dependency list, including a hand-written strict JSON parser: a verifier should be auditable end-to-end without a supply chain. **[D]**
- **Compile-time LCF discipline.** The theorem type `Thm` has a private field and no public constructor; the only functions that return `Thm` are the sixteen rule functions (plus the assumption rule used by the checker for case hypotheses, whose discharge OR_ELIM enforces). The Python implementation follows this discipline by convention; in Rust the compiler enforces it. **[D]**
- **Memory safety and safe parallelism.** `#![forbid(unsafe_code)]`; certificates are checked concurrently on scoped threads with no shared mutable state beyond a work queue. **[D]**

Differential agreement. The two checkers are held to *exact* agreement by a differential harness: a corpus of valid certificates (all benchmark problems plus 250 random-CSP proofs) and adversarial mutants (premise rewiring, rule renaming, value tampering, id games, branch surgery, domain edits — twelve mutants per base) is fed to both. On 2,197 files the verdicts agreed 2,197/2,197, and on all 1,805 mutually accepted files the recomputed conclusions were identical as canonical JSON **[T][M]**. Known, documented divergence corners that the corpus does not exercise: non-ASCII identifiers (Python's `str.isidentifier` is Unicode-aware, the Rust mirror is ASCII), integers outside $i64$ (Python has bigints), and floats in ignored positions

(parsed then discarded on both sides, but by different mechanisms). Agreement is evidence of implementation correctness, not a proof; neither checker is load-bearing for the final guarantee.

Throughput. On a kernel-bound corpus (120 large certificates, 98,880 steps, parsing included on both sides), the measured rates on the 2-core evaluation machine were: Python 27,099 steps/s; Rust 365,789 steps/s single-thread (13.5×); Rust 707,003 steps/s on two threads (26.1×, near-linear file-level scaling) **[M]**. An implementation note in the interest of honesty: the first Rust build was 3× *slower* than Python due to an accidentally quadratic string scan in the hand-written JSON parser; the benchmark caught it. Micro-optimization claims should be assumed fragile at this maturity.

4.3 Compilation to Lean 4

The certificate compiler maps a checked derivation to one self-contained Lean 4 file (no `mathlib`; core tactics only):

- **One lemma per step.** Every non-structural step becomes a top-level `private theorem` whose binders are exactly the variables mentioned, whose hypotheses are exactly the cited premises, and whose statement is the kernel-computed conclusion, discharged by `omega` (linear integer arithmetic). Lean re-proves the step *from its stated premises alone*; a kernel bug upstream cannot survive this stage for the affected step.
- **Structural steps compile to structural Lean.** AXIOM becomes hypothesis reference; EMPTY is the identity (an empty domain fact *is* `False` under the encoding); EX_FALSO is `.elim`; AND_INTRO/AND_ELIM are anonymous-constructor pairing and projection; OR_ELIM becomes a `cases` tree mirroring the branch structure, with case-arm hypothesis names equal to the derivation's hypothesis ids.
- **Hall lemmas as explicit case trees.** A probe file whose Hall examples included an m=4 instance with a 9-value target took 51 s under single whole-lemma `omega` calls, dominated by the larger joint disjunctions; the compiler instead emits a nested `cases` tree over the m small domain facts with an `omega` leaf per combination, giving predictable per-lemma cost (Section 6.5). **[M]**
- **Statement-level guardrails.** The oracle driver refuses any certificate containing `sorry` or `axiom` declarations before invoking Lean, and tampering with a compiled certificate's conclusion is rejected by Lean (a standing test) **[T]**.

The main theorem binds all problem variables as `Int`, takes every axiom (domain facts and constraints, in kernel order, named `A0...`) as hypotheses, and concludes the derivation's final proposition by chaining the step lemmas; satisfiable problems additionally get the existence theorem via an explicit witness. Long certificates produce deeply nested `have` terms; the driver raises the elaborator's recursion budget and the process stack (512 MiB), which was required for the 1,348-step Sudoku certificate **[M]**.

4.4 The trusted base, smallest first

1. **Lean 4 kernel** (external; version 4.10.0 here). Nothing in the repository can override it.
2. **The axiom list.** Lean proves *axioms* ⇒ *conclusion*; whether the axioms mean what the informal puzzle means is the permanent formalization gap (the misspecified-constraints failure mode). Certificates print every axiom with its human-readable label precisely so this audit is a reading exercise.
3. **The certificate compiler** (∼370 lines), trusted only to *state* theorems faithfully. Misstatement is visible in the printed certificate; proof breakage is fail-closed (Lean rejects).

4. **The in-loop checkers**, redundant and diverse (Python ∼1,160 lines across propositions/kernel/checker; Rust ∼2,000 lines, disjoint implementation). Not load-bearing: a bug here can only admit a step that Lean later rejects, or reject a valid step (a completeness, not soundness, failure).
5. **Everything else is untrusted**: engine, gate, proposers, renderer. The engine additionally cross-asserts its set arithmetic against the kernel on every emitted step and aborts on divergence.

5 The Proof-Emitting Engine (Untrusted)

The engine is a conventional finite-domain propagation-and-search solver [15, 16] with one organizing principle: *every deduction is one (or a few) kernel steps*. Solving and certificate construction are the same computation. One search routine is parameterized by an emitter that either records steps through the proof builder or records nothing; the measured search (Section 6.3) and the emitted proof therefore traverse identical trees by construction.

Propagators as rules. Unary intersection, $\neq$/offset-disequality pruning on ground values, offset-equality range transport (SHIFT/FLIP + INTERSECT), and linear bounds filtering map one-to-one onto kernel rules. Two families deserve detail:

- **Bounded Hall sets.** Within each declared alldifferent group (an engine *hint*; the engine verifies the pairwise $\neq$-axioms exist before using it, and checkers ignore hints entirely), the engine searches subsets of size 2..4 of non-singleton variables whose domains fit within subset-size many values: strictly fewer values yields a HALL_CONFLICT certificate of $\bot$; exactly that many yields HALL_PRUNE steps against every outside member. This is naked-subset propagation with certificates, *not* full generalized arc consistency [7]; the bound is a documented completeness limitation, never a soundness one.
- **Constructive disjunction, union form.** For an Or-axiom (including table constraints: disjunctions of conjunctions of domain literals), the engine classifies disjuncts as refuted or live against current domains. All-refuted yields an OR_ELIM concluding $\bot$ in which each branch refutes its own disjunct (e.g. SHIFT $\to$ INTERSECT $\to$ empty $\to \bot$). Otherwise, for each variable that *every* live disjunct confines, the union of the confinements is entailed: the engine emits an OR_ELIM per narrowed variable in which live branches derive their confinement and WEAKEN to the union, and dead branches go through $\bot$/EX_FALSO. This subsumes the usual one-live-disjunct propagation and gives table constraints per-column filtering with certificates.

Search and forced-solution proofs. Search splits the smallest open domain (SPLIT + OR_ELIM); dead branches end in $\bot$. Verdicts are decided by a proof-free first pass (identical traversal, counting solutions up to two). For UNSAT, the second pass emits a refutation. For a unique solution, it emits a *forced* derivation of the full conjunction $\bigwedge_i x_i \in \{v_i\}$: sibling branches are refuted and lifted to the goal with EX_FALSO, so the certificate proves that every satisfying assignment equals v — uniqueness — while the witness theorem proves existence (Section 4.3). Problems with multiple solutions are reported with a checked witness only; no forced derivation is claimed. Termination is immediate (domains are finite and strictly shrink); completeness for the engine-supported constraint forms holds because splitting is exhaustive and every supported form is decidable on ground assignments **[D]**. The engine does not accept constraint forms it cannot propagate (it fails fast at load), so there is no silent incompleteness within an accepted problem.

Effect of the certified global rules. Version 0.2's rules changed the search profile of the benchmarks qualitatively **[M]**: the Einstein riddle, 4×4 and 9×9 Sudoku, and the pigeonhole refutation all close under pure certified propagation — zero search nodes — where v0.1 needed case splits (four for the riddle) or produced longer proofs (the pigeonhole refutation fell from 51 steps over a 9-node search tree to a direct 12-step Hall certificate). Section 6.3 quantifies this against a stated baseline.

6 Experiments

6.1 Environment and reproduction

All numbers were produced on one modest machine — an isolated Linux sandbox with 2 CPU cores and 7 GiB RAM — with CPython 3.11.15, rustc 1.95.0, and Lean 4.10.0 (core toolchain from the `leanprovercommunity/lean4` image; no `mathlib`). Every table regenerates from the repository with the commands in Appendix E. Single-machine, single-run measurements: we report them as observed and make no statistical claims beyond what the sample sizes support.

6.2 Soundness testing

The automated suite (42 tests) includes, beyond per-rule unit tests and the scoping attacks of Section 3.3:

- **Model-theoretic fuzz [T]**: 400 random CSPs (3–5 variables, domains ≤ 5, mixed constraint kinds including offset disequalities, tables and alldifferent cliques) are solved, checked, and compared against brute-force enumeration. Required invariants: the engine verdict (UNSAT/unique/multiple) equals ground truth; every emitted proof is checker-accepted; every accepted conclusion holds in *every* model of its axioms; forced conclusions pin exactly the true unique solution. No violations.
- **Adversarial mutation [T]**: 1,500 mutants of valid certificates (value tampering, premise rewiring, rule renaming, id/conclusion redirection, branch surgery, problem-header edits). Required invariant: the checker either rejects the mutant, or accepts it with a conclusion still entailed by the (possibly mutated) axioms — checked model-theoretically. No violations. Note that many mutants are benign or no-ops; the harness verifies the accept path is genuinely exercised.
- **Differential fuzz [T]**: Section 4.2; 2,197/2,197 verdict agreement between the Python and Rust checkers, identical conclusions on all 1,805 mutual acceptances.
- **Oracle guardrails [T]**: Lean rejects a certificate whose conclusion is tampered; the driver refuses `sorry`/`axiom` smuggling. `mypy -strict` passes over the Python package; the Rust crate forbids unsafe code.

6.3 Benchmark certificates and search effort

Table 1 lists the eight benchmark problems. The baseline is stated precisely because node-count ratios are meaningless without it: *the identical search skeleton* (same variable order, same smallest- domain-first splitting) *with propagation disabled*, checking each constraint once all its variables are ground — i.e. standard chronological backtracking with ground checks, not pure enumeration. A node is one tried variable assignment.

Rather than quote degenerate ratios where the engine needs zero nodes, we state the observation plainly: on five of eight benchmarks certified propagation alone decides the instance, while the same search without propagation explores between 27 and more than $2{\cdot}10^6$ nodes. The small UNSAT instances (pigeonhole, queens_3) show the certificate compression separately: the Hall rule turns pigeonhole's 9-node case tree into a direct 12-step certificate.

Table 1: Benchmarks **[M]**. "Steps" is the kernel-step count of the emitted derivation (forced conjunction for unique instances, refutation for UNSAT). Every derivation is accepted by both checkers and by Lean 4 **[L]**. Baseline capped at $2{\cdot}10^6$ nodes ($5{\cdot}10^4$ for 9×9 Sudoku); "≥" marks a cap hit.

problem	verdict	space $\lvert\mathcal{P}\rvert$	engine nodes	steps	baseline nodes
mini_houses (6 vars)	unique	$7.3{\cdot}10^2$	0	32	27
pigeonhole $4 \to 3$	UNSAT	$8.1{\cdot}10^1$	0	12	48
queens_3	UNSAT	$2.7{\cdot}10^1$	3	30	18
zebra / Einstein (25 vars)	unique	$3.0{\cdot}10^{17}$	0	250	2,865
SEND+MORE=MONEY	unique	$6.5{\cdot}10^8$	10	153	≥ 2,000,001
queens_8, rows 1–2 fixed	unique	$1.7{\cdot}10^7$	9	210	296
sudoku 4×4	unique	$4.3{\cdot}10^9$	0	117	312
sudoku 9×9 (easy)	unique	$2.0{\cdot}10^{77}$	0	1,348	≥ 50,001

Table 2: Random-CSP sweep **[M]**: speedup vs. exactly-computed κ (300 instances).

κ (bits)	n	geo-mean speedup	median	max
$[0, 2)$	10	1.27×	1.24×	2×
$[2, 4)$	31	2.41×	1.80×	11×
$[4, 6)$	70	8.53×	9.50×	48×
$[6, 8)$	99	19.58×	18.00×	233×
$[8, 10)$	67	43.77×	40.00×	594×
≥ 10	23	56.24×	44.00×	1,216×

6.4 Speedup versus measured constraint capacity

For 300 random CSPs (4–6 variables, domains ≤ 5) small enough to enumerate, we computed the companion paper's constraint capacity *exactly*: $\kappa = \log_2(|\mathcal{P}|/\max(1, |\mathcal{P}_\mathcal{C}|))$ with $\mathcal{P}$ the assignment space and $\mathcal{P}_\mathcal{C}$ the satisfying set, and measured speedup as baseline nodes over engine nodes (same conventions as above; all 300 instances completed under the baseline cap). Table 2 buckets the results.

The relationship is monotone and steep. Two honest qualifications. First, the measured factors are far below the theory's headline 2^κ scaling ($\kappa \geq 10$ bits would suggest $\geq 1024\times$; we observe a geometric mean of 56×). This is expected given the baseline: ground checking already exploits constraints, so the ratio lower-bounds the gain over genuinely unconstrained enumeration rather than estimating it; and instances this small truncate the achievable gap. Second, the sweep is one domain, one generator, one baseline. We report the trend as consistent with the framework's qualitative prediction on this domain and claim nothing quantitative or cross-domain (Section 8).

6.5 Lean oracle latency

Table 3 gives per-certificate cost. Per-step cost is tens of milliseconds except where `omega` works hardest: wide-domain linear lemmas (the cryptarithm's 201 ms/step) and, before we switched Hall lemmas to explicit case trees, multi-way joint disjunctions (a single m=4 probe lemma cost tens of seconds). Run-to-run variation of a few percent (and ∼10% on the long Sudoku run) was observed and not controlled for. An engineering target of a sub-50 ms verification call is met at step granularity on most certificates, not at file granularity; batching per file amortizes startup but serializes latency.

Table 3: Lean 4 checking cost per certificate **[M]**. Wall time includes process startup (≈0.3 s) and, for satisfiable instances, the existence theorem. All rows ACCEPTED **[L]**.

certificate	steps	wall time	ms/step
mini_houses	32	1.24 s	39
pigeonhole_4_3	12	1.37 s	114
queens_3	30	0.85 s	28
zebra	250	11.00 s	44
send_more_money	153	30.82 s	201
queens_8 (2 fixed)	210	12.17 s	58
sudoku 4×4	117	3.22 s	28
sudoku 9×9	1,348	150.14 s	111

6.6 Language models as proposers (unedited sessions)

Both sessions ran through the `claude` CLI with a fresh context per call: the prompt carries the axioms, a compact view of accepted facts, the protocol specification, and the gate's recent feedback; the model returns one JSON object. Transcripts ship verbatim with the repository; nothing was retried or edited.

Claude Fable 5 on the full Einstein riddle (protocol v2). The model solved the 25-variable puzzle end-to-end through the gate **[M]**: 29 calls (mean 27.4 s per call; 795 s wall), 200 accepted gate actions — 188 steps and 12 case-analysis commands — and **zero kernel-level rejections**; the only 3 rejected turns were output-formatting failures (NO_JSON: prose or fencing around the object), each recovered on the following call. The model drove four two-way case analyses on the "next to" axioms itself, using exactly the constructive- disjunction discipline: in dead cases SHIFT $\to$ INTERSECT $\to x \in \emptyset \to \bot \to$ EX_FALSO to the common target; in live cases WEAKEN to align. Its final deduction was $\mathit{fish} \in \{4\}$ — the German owns the fish. The finished proof object (193 kernel steps: 188 proposed, 4 OR_ELIM steps created by `end_cases`, 1 final assembly) was accepted by the Python checker, the Rust checker, and the Lean kernel in 7.12 s **[L]**. Appendix D reproduces the horses/Dunhill case block verbatim.

Claude Haiku 4.5 on a 6-variable puzzle (protocol v1). A smaller, earlier session illustrates the gate under a weaker proposer **[M]**: 28 proposals, 17 accepted, 11 rejected — 6 UNKNOWN_ID (citing raw axiom indices as premises instead of first admitting the axiom as a step) and 5 BAD_SHAPE (non-string ids in premise lists). Every rejection carried a machine-readable code and message; the model corrected each class of error in subsequent proposals and solved the puzzle. Its 18-step chain was checker-accepted and Lean-accepted in 0.48 s **[L]**.

What these sessions do and do not show. They demonstrate the intended mechanics at two capability levels: typed rejections steer a mid-tier model to protocol compliance without ever admitting an invalid step, and a frontier model can operate the full protocol — including case analysis — essentially without friction, producing a machine-checked solution to a classic reasoning benchmark. They are single sessions (n=1 each), on puzzles that are certainly in distribution for these models; they support no quantitative claim about model capability, and the soundness of the resulting chains never depended on model quality in the first place.

7 Limitations and Threats to Validity

We list everything we know that bounds the results.

Scope of the logic. The system decides nothing outside finite-domain integer constraints with the connectives of Section 3.1. It is not a general theorem prover, and no path from this fragment to open-ended mathematics is claimed. The *architecture* — untrusted proposer, typed gate, conclusion-computing kernel, byte-level re-checking, proof-assistant oracle — is domain-general; every empirical statement in this paper is domain-specific.

Schema-level soundness is not formalized. Each rule's soundness is a short arithmetic argument reviewed by humans **[D]** and stress-tested empirically **[T]**, and every emitted rule *instance* in this paper was re-proved by Lean **[L]**. But we have not proved in a proof assistant that the rule schemas are sound, nor that the certificate compiler always states the intended theorem. A hostile derivation could not exploit this (Lean re-checks instances), but a compiler bug could in principle produce a theorem *weaker* than intended while still true; the printed-axioms audit is the mitigation.

The formalization gap. Lean proves *axioms* $\Rightarrow$ *conclusion*. Whether the axioms formalize the intended problem is checked by no machine here. This is not a removable limitation of this system but a permanent property of formal verification; we surface it (labels on every axiom) rather than solve it.

Engine completeness bounds. Hall-set detection is a bounded naked-subset search (size ≤ 4), not full GAC alldifferent [7]; Or-propagation covers the disjunct forms listed in Section 5; unsupported constraint forms are rejected at load. All such bounds cost only completeness — proofs never overstate.

Baseline- and scale-dependence of speedups. The node-count baseline (identical search, propagation off, ground checking) is one defensible choice among several; a weaker baseline (pure enumeration) would inflate every ratio, a stronger one (e.g. forward checking) would deflate them. The κ sweep uses instances small enough for exact enumeration; both the truncation of achievable gaps at this scale and the generator's idiosyncrasies limit external validity. The measured trend under-shoots the framework's idealized 2^{κ} ceiling by more than an order of magnitude at $\kappa \geq 10$ bits, for the reasons given in Section 6.4; we treat the strong quantitative form as *not* validated by these data — the companion paper states it as a conjecture with a specified experiment.

Single environment, small samples. One 2-core machine, single runs, no confidence intervals. The LLM sessions are n=1 per model, on famous puzzles almost surely present in training data; they demonstrate mechanics, not capability statistics. Latency figures include provider-side variance we do not control.

Dual-role disclosure. The assistant that implemented the system, designed the prompts, and drafted this paper runs on the same model family (Claude Fable 5) evaluated as a proposer in Section 6.6. Soundness is insulated by construction — the gate admits nothing invalid regardless of who proposes — but the *smoothness* of the Fable 5 session (zero kernel rejections) plausibly benefits from protocol and prompt being co-designed by the same family, and readers should discount it accordingly. The Haiku session, with its rejection-and-recovery pattern, is the more representative picture of an out-of-family proposer.

Checker redundancy is evidence, not proof. Exact differential agreement over 2,197 files is strong evidence the two checkers decide the same relation, with documented blind corners (Section 4.2). The guarantee never rests on either.

Numeric range. The Rust checker's integers are $i64$ (with $i128$ intermediates for linear arithmetic); Python's are unbounded. Certificates using integers beyond $i64$ would be rejected by Rust and possibly accepted by Python — a documented, untriggered divergence.

Not implemented. Thermodynamic sampling layers and any learning of constraints remain unimplemented here; nothing in this paper bears on them.

8 Relation to Avent's Razor

The companion framework makes one architectural prescription and one quantitative conjecture. The prescription — enforce validity with typed, mechanically checked constraints so that invalid reasoning states are unrepresentable, and let untrusted generators search inside that envelope — is what this system implements, and the observed division of labour matched it: proposers ranged from a deterministic engine to two language models of different strengths, and the certificate quality was invariant to the proposer, as the design intends. The typed-rejection channel also functioned as the framework's "thought constraint": under protocol v1 a mid-tier model was steered from malformed to well-formed reasoning by error codes alone (Section 6.6).

The quantitative conjecture — search gain scaling like 2^{κ} up to polynomial factors — receives qualified support at best: the *direction* and monotonicity are clearly present (Table 2), the *magnitude* is well below the idealized scaling against our (already partially constrained) baseline, and we did not attempt the careful baseline ladder that a real test of the theorem would need. We flag this as the most useful next empirical question rather than claim it settled either way.

9 Future Work

In rough order of value per effort: (i) matching-based full-GAC alldifferent certificates (the Régin-style analogue of what VeriPB proof logging achieves for CP [5]); (ii) schema-level formalization: prove the sixteen rules sound in Lean once, making instance re-proof a redundancy rather than the sole formal layer; (iii) richer domains with economic stakes — scheduling, register allocation, configuration — where the certificate is the deliverable; (iv) a baseline ladder (enumeration, ground checking, forward checking, full propagation) to measure the κ relationship against each rung; (v) multi-session LLM evaluation with out-of-family models and unseen puzzle generators to turn the n=1 demonstrations into statistics; (vi) a Rust port of the engine and gate for end-to-end throughput.

10 Conclusion

The claim of this paper is deliberately narrow and, we believe, fully supported: for finite-domain constraint reasoning, the verified chain-of-thought loop can be built *whole* — typed steps, a conclusion-computing kernel, adversarially tested independent checkers in two languages, and a proof-assistant oracle that re-proves every step of every accepted chain — on commodity hardware, with language models operating the loop through a strict protocol, and with every artifact shipping alongside the claims it certifies. Within this domain the companion framework's central bet held up: making invalid steps unrepresentable did not prevent capable generators from reasoning; it prevented them from being wrong silently. Everything beyond this domain remains exactly that — beyond it.

Acknowledgments

The implementation, the experiments, and the drafting of this paper were carried out by the author working with Claude (Anthropic). The assistant runs on the same model family (Claude Fable 5) that is evaluated as an untrusted proposer in Section 6.6; this dual role is disclosed and discussed as a threat to validity in Section 7. The author thanks the Lean community for tooling that makes kernel-checked certificates practical on commodity hardware.

References

[1] L. Avent. Avent's razor: intelligence as constrained search. Companion manuscript, July 2026.

[2] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*. LNCS 78, Springer, 1979.

[3] G. C. Necula. Proof-carrying code. In *POPL*, 1997.

[4] N. Wetzler, M. J. H. Heule, and W. A. Hunt Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *SAT*, 2014.

[5] J. Elffers, S. Gocht, C. McCreesh, and J. Nordström. Justifying all differences using pseudo-Boolean reasoning. In *AAAI*, 2020.

[6] B. Bogaerts, S. Gocht, C. McCreesh, and J. Nordström. Certified symmetry and dominance breaking for combinatorial optimisation. In *AAAI*, 2022.

[7] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *AAAI*, 1994.

[8] P. Hall. On representatives of subsets. *Journal of the London Mathematical Society*, 10(1):26–30, 1935.

[9] L. de Moura and S. Ullrich. The Lean 4 theorem prover and programming language. In *CADE*, 2021.

[10] K. Yang, A. Swope, A. Gu, R. Chalamala, P. Song, S. Yu, S. Godil, R. Prenger, and A. Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. In *NeurIPS*, 2023.

[11] H. Xin et al. DeepSeek-Prover: Advancing theorem proving in LLMs through large-scale synthetic data. arXiv:2405.14333, 2024.

[12] AlphaProof and AlphaGeometry teams, Google DeepMind. AI achieves silver-medal standard solving International Mathematical Olympiad problems. Technical blog post, July 2024.

[13] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *NeurIPS*, 2022.

[14] T. Lanham et al. Measuring faithfulness in chain-of-thought reasoning. arXiv:2307.13702, 2023.

[15] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[16] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

A Rule Reference

Premise signatures and side conditions, as implemented identically in the Python kernel (`kernel.py`, 461 lines) and the Rust kernel (`rust/src/kernel.rs`). "In-fact" means a proposition $x \in S$; premise order is mandatory.

- AXIOM(i): $0 \le i < |A|$. Concludes A_i verbatim.
- INTERSECT($p_1{:}x \in S_1$, $p_2{:}x \in S_2$): same variable. Concludes $x \in S_1 \cap S_2$.
- NE_PRUNE($p_1{:}x \in \{v\}$, $p_2{:}y \in S$, $p_3{:}\, a \neq b$): $\{a,b\} = \{x,y\}$, $x \neq y$. Concludes $y \in S \setminus \{v\}$.
- NEOFF_PRUNE($p_1{:}x \in \{v\}$, $p_2{:}y \in S$, p_3) with $p_3 \in \{x \neq y+k,\ y \neq x+k\}$. Concludes $y \in S \setminus \{v-k\}$ resp. $y \in S \setminus \{v+k\}$.
- SHIFT($p_1{:}y \in S$, $p_2{:}\, x = y+k$): p_1 must concern the right-hand variable. Concludes $x \in \{s+k : s \in S\}$.
- FLIP($p{:}\, x = y+k$): concludes $y = x-k$.
- LIN_PRUNE($p_0{:} \sum_i c_i x_i \le c$, $p_1..p_r{:}x_i \in S_i$ in term order; *target* x_t): all S_i non-empty. Concludes $x_t \in \{v \in S_t : c_t v \le c - \sum_{i \neq t} \min(c_i S_i)\}$ (exact integer arithmetic).
- HALL_CONFLICT(m; $x_1 \in S_1..x_m \in S_m$; $\binom{m}{2}$ pairwise $\neq$ facts, lexicographic): variables distinct, S_i non-empty, $|\bigcup_i S_i| < m$. Concludes $\bot$.
- HALL_PRUNE(m; … as above plus $y \in S_y$ and $x_i \neq y$ for each i): $y \notin \{x_i\}$, $|\bigcup_i S_i| = m$. Concludes $y \in S_y \setminus \bigcup_i S_i$.
- EMPTY($p{:}x \in \emptyset$): concludes $\bot$.
- SPLIT($p{:}x \in S$), $S \neq \emptyset$: concludes $\bigvee_{v \in S,\ \text{ascending}} x \in \{v\}$.
- WEAKEN($p{:}x \in S$; S'): $S \subseteq S'$. Concludes $x \in S'$.
- EX_FALSO($p{:}\bot$; φ): φ well-formed over the declared variables. Concludes φ.
- AND_INTRO($p_1..p_r$), $r \ge 1$: concludes $\bigwedge_i \varphi_i$. AND_ELIM($p{:} \bigwedge \varphi_i$; j): concludes φ_j.
- OR_ELIM($p{:} \bigvee_{i=1}^{r} \varphi_i$; r branches): branch i assumes exactly φ_i (bound by the checker, not read from the certificate) and its last step's conclusion is the branch conclusion (the hypothesis itself if the branch is empty); all branch conclusions must be structurally equal, and that proposition is concluded.

B Certificate Format

A certificate is one JSON document:

```
{ "format": "vcot-proof-v1",
  "problem": { "name": "...",
               "vars": [["x", [1,2,3]], ...],
               "constraints": [ {"t":"ne","a":"x","b":"y"}, ... ],
               "labels": ["..."], "groups": [["x","y","z"]] },
  "steps": [ {"id":"s1","rule":"AXIOM","axiom":0},
             {"id":"s9","rule":"OR_ELIM","prem":["s5"],
              "branches":[{"hyp":"h1","steps":[...]}, ...]}, ... ],
  "conclusion": "s42" }
```

Proposition tags: `in`, `ne`, `neoff`, `eqoff`, `lin`, `or`, `and`, `false`. Step conclusions are not representable in the format; checkers recompute them. `groups` is an engine hint ignored by all checkers. Identifiers match `[a-z][a-zA-Z0-9_]*` and are globally unique, including branch hypothesis ids.

C Gate Protocol v2

One JSON object per proposer turn: `{"steps":[...]}` (batch; `"#k"` cites the id accepted for batch item k; per-item verdicts; rejections do not abort the batch) or one command: `begin_cases` (open OR_ELIM on a cited Or-fact; enters case 1 and returns its hypothesis id), `next_case`, `end_cases`, `abort_cases`. Acceptances return the computed conclusion; rejections return a machine-readable code from the kernel/scoping vocabulary (EXPECTED_IN_FACT, NOT_SINGLETON, VAR_MISMATCH, UNKNOWN_ID, ARITY, BRANCH_CONCL_MISMATCH, NO_SUCH_AXIOM BAD_SHAPE, ...). The accepted prefix is a valid derivation after every turn.

D Transcript Excerpt (Verbatim)

Claude Fable 5, Einstein riddle session, calls 9–13: the horses/Dunhill "next to" axiom, previously admitted as fact `s67`, with `s94`: *dunhill* $\in \{1\}$ already derived. Model output JSON, reproduced exactly (one `note` field abbreviated with `[..]`); gate replies paraphrased in comments. Unicode in notes is shown as escaped by the JSON serializer.

```
call 9  -> {"cmd": "begin_cases", "prem": "s67"}
        <- case 1/2 opened, hypothesis h118: horses = dunhill + 1
call 10 -> {"steps": [{"rule": "SHIFT", "prem": ["s94", "h118"],
             "note": "dunhill=1, horses=dunhill+1 => horses in {2}; [..]"}]}
        <- ACCEPTED s119 : horses in {2}
call 11 -> {"cmd": "next_case"}
        <- case 2/2 opened, hypothesis h120: horses = dunhill - 1
call 12 -> {"steps": [
             {"rule": "SHIFT", "prem": ["s94", "h120"],
              "note": "horses = dunhill - 1, dunhill in {1} => horses in {0}"},
             {"rule": "AXIOM", "axiom": 23},
             {"rule": "INTERSECT", "prem": ["#1", "#2"],
              "note": "{0} \u2229 {1..5} = \u2205"},
             {"rule": "EMPTY", "prem": ["#3"]},
             {"rule": "EX_FALSO", "prem": ["#4"],
              "target": {"t": "in", "var": "horses", "vals": [2]},
              "note": "align with case 1 conclusion horses \u2208 {2}"}]}
        <- ACCEPTED s121..s125 (last: horses in {2})
call 13 -> {"cmd": "end_cases"}
        <- CASES CLOSED: s126 : horses in {2}   (unconditional)
```

The corresponding step lemmas in the Lean certificate (`fable_zebra.lean`, accepted by Lean 4.10.0), verbatim except that the disjunction symbol is rendered here in math mode:

```
private theorem step_s119 (dunhill horses : Int)
  (q1 : dunhill = 1) (q2 : horses = dunhill + 1) : horses = 2 := by omega
private theorem step_s121 (dunhill horses : Int)
  (q1 : dunhill = 1) (q2 : horses = dunhill - 1) : horses = 0 := by omega
private theorem step_s123 (horses : Int) (q1 : horses = 0)
  (q2 : horses = 1 ∨ horses = 2 ∨ horses = 3 ∨ horses = 4 ∨ horses = 5)
  : False := by omega
```

E Reproduction Guide

From the repository root (vcot v0.2):

```
# everything: tests, mypy --strict, Rust build, both checkers on all
# artifacts, differential fuzz, Lean round-trips, tamper check
bash scripts/verify_all.sh

# Tables 1-3 (benchmarks, kappa sweep, Lean latency)
```

```
python3 scripts/measure.py            # writes artifacts/measurements.md

# Rust/Python differential agreement and throughput
python3 scripts/differential.py       # writes artifacts/differential.json
python3 scripts/bench_throughput.py   # writes artifacts/throughput.json

# one problem end-to-end (engine -> checker -> Lean certificate)
PYTHONPATH=src python3 -m vcot solve zebra --lean --render

# check any certificate independently, then compile it to Lean
PYTHONPATH=src python3 -m vcot check artifacts/zebra.proof.json
./rust/target/release/vcot-check --jobs 2 artifacts/*.proof.json
PYTHONPATH=src python3 -m vcot lean artifacts/zebra.proof.json

# language-model sessions (requires the claude CLI)
python3 demos/llm_gate_demo.py claude-haiku-4-5
python3 demos/fable_zebra_demo.py claude-fable-5
```

Environment used throughout: Linux sandbox, 2 cores, 7 GiB RAM; CPython 3.11.15; rustc 1.95.0 (edition 2021, zero dependencies, `cargo build -release` needs no network); Lean 4.10.0 core (any Lean $4 \geq 4.9$ with core `omega` should work; set `LEAN_BIN` if not on `PATH`).