

From Russell to Lean 4

Complete proof search as a deduction engine for AI-assisted formalization

Dmitry Vlasov

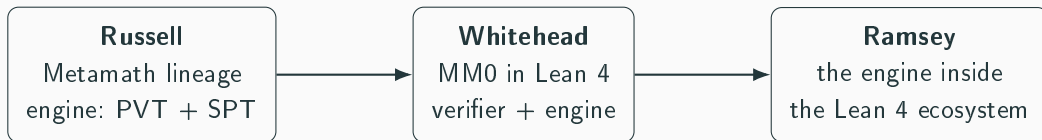
8 October 2026

ARGO seminar, Faculty of Mathematics, University of Belgrade

The thesis, first

Claim

Complete proof search at the level of substitutions is *mechanical*. Let the AI spend its tokens on conjectures, lemma statements and hints; let the engine spend cycles on mechanical deduction; and let the output always be a complete, checkable formal proof.



The rest of the talk: the argument for the claim, and the plan for the route.

Russell

Russell: what it is

- A logical framework in the **Metamath** lineage: a database of assertions (axioms, theorems) with hypotheses; a proof is a chain of *substitution* + rule application. Hilbert-style, no built-in logic — set theory, HOL, anything is a database.
- Differs from Metamath in what is separated out:
 - expression syntax is a grammar of its own (no “syntax proofs” inside the logic);
 - definitions are declared separately (but still act as axioms in deduction);
 - readable surface syntax; `set.mm` (tens of thousands of theorems) translated into it.
- An IDE, an **automated prover engine** (the subject of this talk), and facilities for ML benchmarking of proof search.
- <https://github.com/dmitry-vlasov/russell-flow>

Example: a theorem and its proof

Metamath (set.mm) style

```
axi.1 $e |- ph $.
axi   $p |- ( ps -> ph ) $=
      ( wi ax-1 ax-mp ) ABADCABEF $.
```

The proof is an RPN program that computes the statement on a stack. Nothing else exists. The verifier is a few hundred lines.

The same theorem in Russell

```
theorem axi (ph : wff, ps : wff) {
  hyp 1 : wff = |- ph ;;
  -----
  prop : wff = |- ( ps → ph ) ;;
} proof {
  step 1 : wff = ax-1 () |- ( ph → ( ps → ph ) ) ;;
  step 2 : wff = ax-mp (hyp 1, step 1) |- ( ps → ph ) ;;
}
```

Typed variables, a separate expression grammar, and a proof that names the assertion and the steps it uses.

The engine: proof variant trees and substitution proof trees

Proof Variant Tree (PVT) — the search space, kept explicitly.

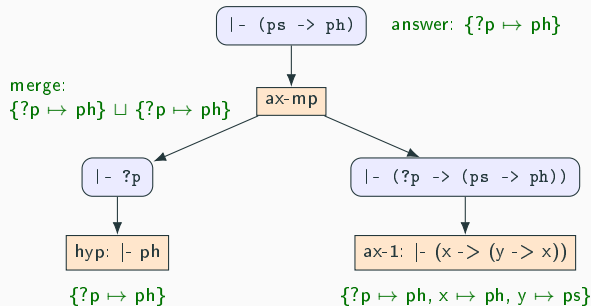
- Goal nodes and rule nodes alternate.
- A goal node is expanded by unifying it against *every* assertion in the database; the variables of the theorem being proved are *frozen*.
- Each unifying assertion gives a rule node; its substituted hypotheses become new goal nodes.
- Grows goal $\rightarrow$ premises; a forward direction (premises $\rightarrow$ goal) was added later.

Substitution Proof Trees (SPT) — the answers.

- Each node carries a set of answer substitutions for the metavariables introduced by expansion.
- A closed leaf (axiom, hypothesis) yields its unifier.
- At a rule node the children's answer sets are merged: pairwise mgu of substitutions.
- Answers propagate to the root; an answer at the root *is* a proof.

Two phases: grow variants up (PVT) $\longrightarrow$ propagate proofs down (SPT)

A small example: from $\vdash \varphi$ to $\vdash (\psi \rightarrow \varphi)$



Grow. The goal unifies with the conclusion of ax-mp (fresh metavariable $?p$); its hypotheses become goal nodes. ax-1 (renamed apart) fits the second, the hypothesis $\vdash \varphi$ the first. Direct ax-1 on the goal fails: it needs $\varphi = (y \rightarrow \psi)$, and φ is frozen.

Propagate. Closed leaves yield answers; at ax-mp the children's answer sets are merged by mgu; the result reaches the root.

Result — a complete proof:

- | | |
|--|-----------|
| 1. $\vdash \varphi$ | hyp |
| 2. $\vdash (\varphi \rightarrow (\psi \rightarrow \varphi))$ | ax-1 |
| 3. $\vdash (\psi \rightarrow \varphi)$ | ax-mp 1,2 |

Soundness and completeness (2017)

D. Vlasov, *Proof Search Algorithm in Pure Logical Framework*, Sobolev Institute of Mathematics, 2017.

Theorem (informally)

Sound: every answer at the root is a valid proof.

Complete: for every proof π of the goal from the database, the algorithm finds a proof π_0 with $\pi = \pi_0\sigma$ for some substitution σ — the found proofs are *most general*.

- For the *full* deductive system: no cut-free fragment, no Prolog-style delayed goals.
- Metavariables are instantiated only by unification: no guessing or iterating over instances. Metavariables may be left in the goal or the premises; the search solves them.
- The space is infinite; with fair (breadth-first) expansion every proof appears at a finite depth.

Closest relatives: SLD-resolution; logic programming in logical frameworks (Twelf/Elf, λ Prolog).

Differences: frozen theorem variables; answer *sets* merged at rule nodes instead of backtracking; all variants kept in one tree.

Interactive proving = clicking on PVT nodes

- The prover's state *is* the tree, and the user sees it.
- Click an open goal node $\Rightarrow$ the engine expands it against the whole database and propagates answers.
- A *tactic* is a policy for choosing which open goal nodes to expand, and in what order — the same operation a user performs by clicking.
- Automation and interaction are the *same operation* at different speeds.

Declarative *output* with an interactive *process*: readable, declarative proofs, like Mizar's or Isar's — but the engine produces the proof from the tree automatically.

	tactic script	PVT proof
stored artifact	a program	the complete proof
checkable without replay	no	yes
readable depends on	partly the tactic implementation	step by step only the assertions it cites

- **Expansion** = unify one goal against every assertion in the database (set.mm: ~50 000 assertions). Needs a term index: discrimination tree.
- **Propagation** = merging *sets* of substitutions at rule nodes. The naive cross product explodes; the sets have to be unified massively, with shared structure. This, not expansion, is the bottleneck.
- **Numbers** Reproving with oracle (only oracle node is expanded): 5000 first theorems in around 1 min. When in addition to the oracle node we add 4 more random - 2 min. At 5 additional nodes - out of memory.

A painful experiment: Mizar $\rightarrow$ Russell

Goal. Import the Mizar Mathematical Library and re-prove it in Russell.

Design. Two layers: the engine, plus a composable tactic DSL meant to reproduce Mizar's `by` ("obvious inference", Rudnicki 1987).

Outcome.

- The first MML theorems: proved. After that, every new theorem cost a large effort, and the tactics degenerated into a pile of special cases.
- The proofs the engine found were correct — and unreadable.
- A by-product: the forward (premises $\rightarrow$ goal) direction of the engine.

Conclusions.

- `by` is a decision procedure in its own right, not a combination of tactics.
- Readability is a separate objective from provability.
- Different aim from MPTP: readable re-proof, not translation for ATPs.
- Kept: the two-layer architecture, engine below, strategy above.

Metamath Zero

MM0 (Carneiro, 2019–): a sibling, not a descendant

	Metamath	Russell	MM0
expression syntax	inside the logic (“syntax proofs”)	separate grammar	separate: notation declarations + precedence parser
definitions	axioms	separate declarations, axioms in deduction	def, unfolded natively by the verifier (conservative)
proof format	text, .mm	text	binary .mmb against a human-readable spec .mm0
bound variables	\$d constraints	\$d-like	first class, with dependencies: (ph: wff x)
verifiers	many, tiny	Russell	mm0-c, mm0-rs, a Lean formalization of the semantics

Common ancestor: Metamath; same deduction: substitution + hypotheses.

What matters for a prover: **definition unfolding** and the **bound-variable discipline** — both must enter the expansion step.

Automation in MM0, and Project Whitehead

MM0 today. MM1 is the language for writing proofs, with tactics such as `refine`. No hammer-class universal search, as far as I know.

Whitehead — MM0 in Lean 4, carrying the Russell engine over.

1. a `.mm0` / `.mmb` reader;
2. an *independent* MM0 verifier in Lean, against the spec, the Lean semantics and `mm0-rs` (`mm0-c` is only a test aid: it already turned out to accept a proof the spec rejects);
3. the prover: PVT/SPT over MM0, with unfolding and the bound-variable discipline inside unification.

Why Lean: the MM0 semantics are already formalized there; verifier and correctness proof can live side by side.

Lean 4

Lean 4: a different ideology

- Foundation: dependent type theory (CIC). A proof is a *term*; checking is type checking; equality up to definitional unfolding (defeq).
- Lean 4 is a general-purpose functional language; the elaborator, the tactics and Mathlib are written in it.
- Mathlib is the de facto standard library of formalized mathematics; the community and the AI labs build on it. It is where the mathematics — and the effort — is.
- From the Metamath side: the substitution-based view is Lean's *syntactic* layer; Lean adds defeq, binders, universes, elaboration on top.

What is a proof?

Lean has two artifacts:

- the **term** — the proof in the foundational sense: complete, checkable, unreadable;
- the **tactic script** — a program that *constructs* the term: readable-ish, not itself a proof, fragile under library changes.

Is an algorithm that constructs a proof a proof? As a certificate — yes: the checker replays it deterministically. What it lacks: semantic content (which lemmas, which instances) and stability.

Russell and MM0 have one artifact: the complete proof, step by step. Lean's declarative fragments (`term mode`, `calc`, `have/show`) play a similar role.

Isabelle had this debate — apply scripts vs `Isar` — and `Isar` won. Lean has not had it yet.

Automated proving in Lean 4, as of 2026

tool	what it does	what it does not
exact?, apply?, rw?	one-level lookup against all of Mathlib (discrimination trees)	search beyond one step
simp	rewriting to normal form with a curated simp set	goal-directed search
omega, linarith, polyrith grind (core, 2025)	decision procedures for arithmetic fragments SMT-style: E-matching, congruence closure, arithmetic	anything outside the fragment general completeness
Aesop	best-first search over user-declared rule sets	completeness; rule sets are curated by hand
Duper	superposition prover inside Lean, proof-producing	dependent types natively; scale
Lean-auto LeanHammer (2025)	DTT $\rightarrow$ HOL translation to external ATPs premise selection + Aesop + Lean-auto/Zipperposition + Duper	proof structure; needs reconstruction uniform search over the whole library; a search tree; returns a script
LLM provers (DeepSeek, Kimina, Goedel, Seed, Aristotle)	generate tactic text, Lean as the checker	cheap: hundreds to thousands of samples per theorem

One-step library search exists; multi-step heuristic search exists; a *complete* database-wide search that exposes its tree does not.

What a Russell-style engine would add — Project Ramsey

Adds

- completeness with respect to the rule set (Aesop and the hammers are heuristic);
- systematic merging of answer substitutions across sibling goals — metavariables shared between goals are a known weak spot of tactic-level search;
- forward chaining from hypotheses, not only backward from the goal;
- the search tree as an *interface* — for a human or an AI — rather than a tactic that returns a term.

The caveat, stated first

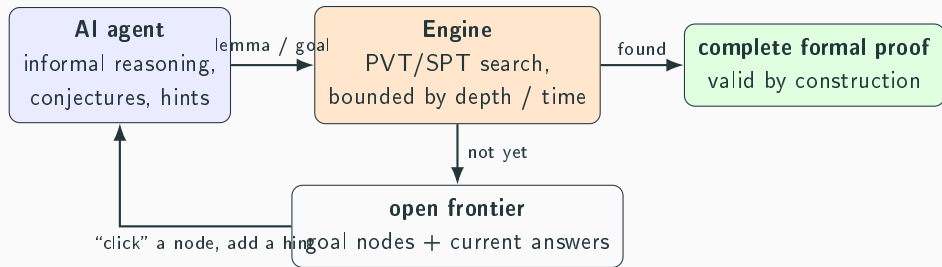
- Lean unifies up to defeq; the engine unifies syntactically.
- So: syntactic search on top; `isDefEq`, `simp`, decision procedures as *oracles* at the leaves.
- Completeness becomes completeness *relative to the oracles*.

Roadmap

1. **Russell** $\rightarrow$ **Whitehead**. Easy: the same substitution-based deduction. Hard: definition unfolding and bound-variable dependencies are native in MM0 and absent in Russell. Same ideology: the expansion step produces *all* variants — a goal with a defined symbol is also expanded through every unfolding of it, and the unifier respects MM0's dependency constraints. No laziness, no backtracking: the tree gets larger, the model stays the same. Completeness is re-proved.
2. **Formalize completeness first — for Whitehead**. The MM0 semantics are already in Lean; a verified complete prover for a Metamath-class framework is a result on its own, and the specification for the next step.
3. **Whitehead** $\rightarrow$ **Ramsey**. A Lean-tailored version of the 2017 paper (soundness, generality, completeness modulo the oracles) as the spec; then the implementation on Lean's metaprogramming API.

The target is Lean 4 because that is where formalized mathematics, and the AI effort around it, now lives.

The engine as a tool in an AI harness



- The same interface as interactive proving; a different agent is clicking.
- The AI checks its informal hypotheses fast and never writes the exact formal proof of the current goal.
- Contrast: today's LLM provers generate tactic text token by token and use Lean only as a checker.

Why this division of labour

- Complete search is exponential in depth: affordable for shallow goals, hopeless for deep ones.
- Evidence from `set.mm`: *linear* subproofs — chains without branching, replayable by matching — cover a large part of the library, and their lengths peak at 8–12 steps. Reading: between the branch points of a proof, the purely mechanical fragments are short.
- So if an agent supplies the statements at the branch points, what is left to the engine has depth ≈ 10 — within reach of indexed, unification-pruned complete search. (A hypothesis; the effective branching factor is the number to measure.)
- LLM provers have the mirror profile: good at proposing the next statement, expensive at exact long chains — one wrong token, and the whole sample is resampled.
- Hence: the AI produces the *next useful statement*; the engine produces its proof — complete within the depth bound, valid by construction, no reconstruction, no trust in the model.

Summary

- **Russell**: a Metamath-lineage framework; prover = proof variant trees + substitution proof trees; sound and complete (2017); interactive proving as clicking on nodes.
- **Mizar experiment**: by is a decision procedure; generated proofs are unreadable; keep two layers.
- **Whitehead**: the engine on MM0, in Lean 4 — verifier first, verified completeness next.
- **Ramsey**: the engine inside Lean 4; syntactic search over Lean's own oracles.
- **Purpose**: a fast, complete deduction tool in the AI loop — the model thinks, the engine deduces.

<https://github.com/dmitry-vlasov/russell-flow>

<http://semr.math.nsc.ru/v17/p988-998.pdf>

vlasov.dmitry.1976@gmail.com