SEMANTEQ: FAITHFUL BIDIRECTIONAL EQUIVALENCE OF FORMAL STATEMENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Statement autoformalization translates mathematics from natural language into a proof assistant such as Lean 4, and models now do it at scale. As provers get stronger, the burden of correctness shifts from the proof to the statement, that is, to whether the formal statement faithfully translates its source. Given a trusted reference formalization, the natural test is logical equivalence, and it proves too much. $P \leftrightarrow Q$ holds trivially for any two statements P and Q that the library proves, because a prover can prove each separately and pass through $P \leftrightarrow \mathrm{True} \leftrightarrow Q$. The stronger the prover, the more unrelated pairs pass. Earlier methods restrict the prover and check that each proof names the other statement. We introduce `SemantEq`, a more principled relation that instead requires each statement to turn into the other only through a chain of allowed steps, such as logical rearrangement and rewriting with the library's own equations. Our programs judge a proof by the proof term Lean checks, not by its tactic script. `SemantEq-Eval`, a search without a model, reaches 100% precision and 65.6% recall on the ProofNetVerif test split under our adjudicated labels, against 95.8% and 61.8% for `BEq+`, and an 8.9-fold increase in its CPU time adds at most three pairs. A language model finds more proofs, but under two of four prompts it proves both directions of about two in five of the non-equivalent pairs. `SemantEq-ProofAudit` audits each model-written proof under a policy derived from the relation and accepts none of the 104 non-equivalent pairs, and it raises recall on 407 equivalent pairs from 65.4% to 81.1% under the best prompt. Separating the generation of equivalence proofs from their acceptance lets a stronger prover raise recall without changing what an acceptance means. We release code and data at: https://anonymous.4open.science/r/SemantEq-2FBF/

1 INTRODUCTION

Formal theorem proving has improved quickly on benchmarks such as miniF2F, ProofNet, and PutnamBench (Zheng et al., 2022; Azerbayev et al., 2023; Tsoukalas et al., 2024), in part because the Lean kernel checks every proof and gives models a reliable signal (de Moura & Ullrich, 2021). Models now also formalize at scale, from large problem sets (Ying et al., 2024; Lin et al., 2025; Wu et al., 2026) to whole textbooks and research papers (Rammal et al., 2026; Zhang et al., 2026b), and the Formal Conjectures project collects Lean statements of open problems (The Formal Conjectures Authors, 2025). The kernel, however, checks a proof only against its formal statement, not whether the statement says what the source says. Audits of benchmarks find statements that drop hypotheses, change domains, or become vacuous (Ospanov et al., 2025; Ammanamanchi et al., 2026), and an expert review of an agent's formalization found it weakest at choosing definitions, not at filling in proofs (Ilin & Nugent, 2026). As provers get stronger, the burden of correctness shifts from the proof to the statement.

At this scale no one can read every statement, so an automatic evaluator decides which formalizations count as correct. It is also often the reward for training the next formalizer (Wu et al., 2026; Yu et al., 2026) or the filter that keeps its outputs (Peng et al., 2026). A formalizer trained against an evaluator that accepts unfaithful statements therefore learns to produce them, and one trained against an evaluator that rejects valid rewordings is held back.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

The ideal evaluator would compare a formal statement with its informal source directly, and learned judges try to (Lu et al., 2025; Peng et al., 2026; Rammal et al., 2026), but their verdicts come with no proof that anyone can check. We work in the setting where a trusted formal reference exists, as it does when large projects formalize the statements of their main results before the proofs (van Doorn et al., 2023; Tao, 2023; Commelin & Topaz, 2024) and when benchmarks ship references checked by people. The task is still hard, because one informal statement has many correct formalizations, which differ in variable names, hypothesis order, library definitions, or the form of the conclusion. Text similarity to the reference, such as BLEU (Papineni et al., 2002), therefore cannot tell a rewording from a different claim (Azerbayev et al., 2023).

A natural test is to prove that the candidate and the reference imply each other, but the test proves too much. In Lean, `Nat.Prime 5` and `Irrational (Real.sqrt 2)` are logically equivalent, because Mathlib proves both. Each implication can then ignore its antecedent, the statement it assumes, and prove its conclusion outright. A candidate that states something true but different from the reference passes the same way, and a stronger prover finds such proofs for more pairs. The existing evaluators `BEq` and `BEq+` restrict how the proof is found and check the proof text for the antecedent's name (Liu et al., 2025b; Poiroux et al., 2025). A restricted search can still close a goal from library facts alone, however, and a name in the text does not show that the antecedent did any work (Section 2). The question is therefore which proofs to accept.

`SemantEq` answers that question before any prover runs. It relates two statements when each turns into the other by a chain of four kinds of step, namely logical rearrangement, rewriting with an equation or biconditional that the library states, a change of variables, and structural weakening (Figure 1). Our programs read the proof term that Lean checks, not the tactic script that produced it, to see which steps a proof takes and whether it depends on the other statement. Because the rule is fixed apart from the prover, it separates the generation of equivalence proofs from their acceptance. Any prover can search for chains, and a stronger prover can find more of them without changing what an acceptance means.

We study two ways to find chains. `SemantEq-Eval` is a deterministic Lean metaprogram that searches for a chain in each direction and audits the proof term it finds, with no model and no GPU, in a median of two seconds per pair (mean 30.5 s, on the 407 equivalent pairs of Section 3.3) on a laptop CPU. On the ProofNetVerif test split, under our adjudicated labels, it accepts no non-equivalent pair, for a precision of 100% against 95.8% for `BEq+`, and its recall is 65.6% against 61.8%. The published labels give the same ranking. On 100 non-equivalent pairs assembled to test each audit, it accepts none, while `BEq+` accepts 54 (Section 3). More search barely helps. Across our grid an 8.9-fold increase in CPU time adds at most three pairs, because many of the missed pairs need a step that the search is not built to take, such as building a new object to plug into the other statement (Section 3.3).

The second way lets a language model write the proofs. A compiling proof establishes the implication but not, by itself, that the two statements say the same thing, since under prompts that ask only for a compiling proof the model proves both directions of about two in five of the non-equivalent pairs. `SemantEq-ProofAudit` reads each proof term and accepts a direction only when it is built from the four kinds of defined step, apart from exceptions it records, and it names the rule that a rejected proof breaks. It accepts none of the 104 non-equivalent pairs and certifies 330 of the 407 equivalent pairs under the best of four prompts, against 266 for the search (Sections 4.1 and 4.2).

Our contributions are as follows.

- We define `SemantEq`, a relation that fixes the allowed steps by their effect, and prove that it is an equivalence relation whose related pairs are logically equivalent (Section 2.1).
- We build `SemantEq-Eval`, a search without a model that improves on `BEq+` in precision and recall, and find where more search stops helping (Sections 2.2 and 3).
- We introduce `SemantEq-ProofAudit`, which audits proofs from any prover and separates equivalent from non-equivalent pairs where compilation does not (Section 4).

`SemantEq` checks agreement with a formal reference, not with the informal source, so a candidate that copies a mistake in the reference still passes. A rejection means only that no proof passed the search and its audits, not that the two statements differ.

2 BETTER SEMANTIC EQUIVALENCE FOR MATHEMATICAL STATEMENTS

A good notion of mathematical equivalence should accept different formulations of the same claim. Definitional equality, which Lean checks by computing and unfolding definitions, is too strict. It identifies $1+1=2$ with $2=2$ but not $x+y=z$ with $y+x=z$. Unrestricted provable equivalence is too loose, as the introduction shows. `BEq` and `BEq+` aim for a test between the two by restricting the proofs in both directions.

`BEq` (Liu et al., 2025b) defines equivalence as converting each statement into the other by restricted transformations, which reduce to definitional equality when none are allowed. To check one direction, it assumes the other statement, the antecedent, as a theorem closed with `sorry`, Lean's placeholder for a missing proof. It then looks for a proof of the goal, first with `exact?`, which searches the library and the hypotheses for a lemma that closes it, and then with a language model limited to eleven tactics, Lean's proof commands. `BEq+` (Poiroux et al., 2025) replaces the model with fixed strategies that use the antecedent through `apply`, `convert`, or `have` and finish with tactics such as `simp_all_arith!`, so it is deterministic and runs on a CPU. Both look for the antecedent's name in the proof, although `BEq+` does so only for its first `exact?` attempt and inserts the name itself in its other strategies.

Appendix E shows two ways this fails on pairs that ProofNetVerif labels not equivalent. First, a closing tactic can prove the goal from the library alone. In the first pair each statement has its own direct proof, so accepting both directions says nothing about whether they state the same claim. The `BEq+` authors report related false acceptances and a fix that is now their default (Poiroux et al., 2025, Appendix A.6.1). Second, a name in a script does not show that the antecedent did any work. In one direction of the second pair the antecedent never appears in the proof term, and in the other it appears but is not needed. An evaluator therefore has to check how a proof uses the antecedent and the library. `SemantEq` does this by defining the allowed steps first, as `BEq` does, and by examining the proof term, the proof object that Lean's kernel checks.

2.1 SEMANTEQ: AN EQUIVALENCE RELATION DEFINED BY ALLOWED STEPS

Our key idea is to fix a set of allowed steps, each simple enough for a person or a program to audit, and to call two statements equivalent when a chain of such steps leads from each to the other. We first fix notation and then define four kinds of step by their effect on a statement.

Setting. Fix the library $\mathcal{L}$, for us Lean with Mathlib. A *statement* is a proposition over $\mathcal{L}$ with no free variables, taken up to renaming of bound variables and definitional unfolding, and $\mathcal{S}$ is the class of statements. The *atoms* of a formula φ, written $\mathrm{atoms}(\varphi)$, are its equations and applications of predicates. We write $S = \mathcal{C}[\varphi]$ to single out one occurrence of a formula or term φ in S, and $\mathcal{C}[\varphi']$ for S with φ' in its place. The *context* Γ of the occurrence holds the variables quantified over it and the hypotheses it stands under. A term or formula *over* Γ uses only the variables of Γ and the symbols of $\mathcal{L}$, and $\mathcal{L}, \Gamma \vdash \psi$ means that $\mathcal{L}$ proves ψ from the hypotheses in Γ. An occurrence is *positive* when the path from the root to it passes only through $\wedge$, $\vee$, $\to$, and the bodies of quantifiers, and enters the left side of an implication an even number of times. The *prefix* of a statement is its leading string of universal quantifiers and hypotheses, and $S = \Pi\ \varphi$ cuts S after an initial segment Π of its prefix. A *bridge* is a theorem $b = \forall \bar{y}.\ \psi_1 \to \cdots \to \psi_m \to (u \asymp v)$ of $\mathcal{L}$, with $\asymp \in \{=, \leftrightarrow\}$, that is not a law of pure logic. We take $\mathcal{B}$ to be the set of all bridges, so no bridge is chosen by hand. An *instance* $\sigma(b)$ is given by an assignment σ of terms to $\bar{y}$, with sides $\sigma(u)$ and $\sigma(v)$ and hypotheses $\sigma(\psi_i)$. Every domain D below is a type, and a term or formula that a step introduces contains, as written, no proof other than a hypothesis of its context. Appendix C.1 gives the notation in full.

Definition 1 (Faithful derivation)**.** A statement S ***faithfully derives*** a statement T with respect to the library $\mathcal{L}$, written $S \rightsquigarrow_{\mathcal{L}} T$, when one of the following holds.

- **Logical rearrangement.** $S = \mathcal{C}[\varphi]$ and $T = \mathcal{C}[\varphi']$, where $\mathrm{atoms}(\varphi) = \mathrm{atoms}(\varphi')$, $\varphi \leftrightarrow \varphi'$ holds under every interpretation of the symbols in these atoms, and every atom matters on both sides, in that for each atom some interpretation makes the truth value of φ, and likewise of φ', change with that atom alone. Equations are atoms like any other here.

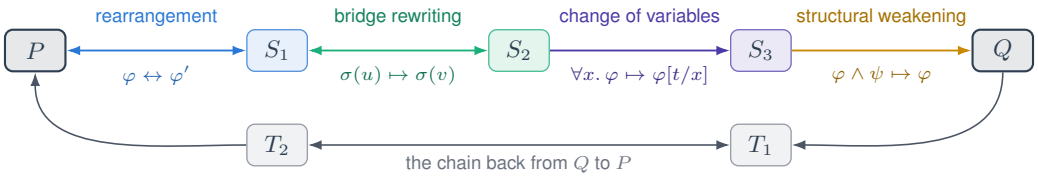

Figure 1: Chains witnessing $P \simeq_{Sem,\,\mathcal{L}} Q$. Rearrangement and bridge rewriting can be reversed, but a change of variables or a weakening need not be, so a separate chain leads back from Q to P.

- **Bridge rewriting.** $S = \mathcal{C}[\sigma(u)]$ and $T = \mathcal{C}[\sigma(v)]$, or the same with u and v exchanged, for some bridge $b \in \mathcal{B}$ and some instance $\sigma(b)$ whose assignment σ consists of terms over Γ and whose hypotheses satisfy $\mathcal{L}, \Gamma \vdash \sigma(\psi_i)$ for each $i \leq m$.
- **Change of variables.** Either $T = \forall y{:}\mathsf{D}.\ S$ for a new variable y ranging over a domain D, or $S = \Pi\ \forall x{:}\mathsf{D}.\ \varphi$ and $T = \Pi\ \varphi[t/x]$ for a term t of D built from the variables of Π and the symbols of $\mathcal{L}$. If D is a kind of structure on an underlying set, such as topologies on a set, then t must be a structure that S already carries on that set, a variable of Π or a structure occurring in S.
- **Structural weakening.** A one-directional move of pure logic, at a positive occurrence for the first three and in the prefix for the last two. Dropping a conjunct, $\mathcal{C}[\varphi \wedge \psi] \rightsquigarrow_{\mathcal{L}} \mathcal{C}[\varphi]$; adding a disjunct, $\mathcal{C}[\varphi] \rightsquigarrow_{\mathcal{L}} \mathcal{C}[\varphi \vee \psi]$ for a formula ψ over Γ; witnessing an existential, $\mathcal{C}[\varphi[t/x]] \rightsquigarrow_{\mathcal{L}} \mathcal{C}[\exists x{:}\mathsf{D}.\ \varphi]$ for a term t of D over Γ; adding a hypothesis, $\Pi\ \varphi \rightsquigarrow_{\mathcal{L}} \Pi\ (\psi \to \varphi)$ for a formula ψ over the variables of Π; and removing a hypothesis the others imply, $\Pi\ (\psi \to \varphi) \rightsquigarrow_{\mathcal{L}} \Pi\ \varphi$ when ψ follows from the hypotheses of Π in every interpretation.

Every step preserves implication. Rearrangement and bridge rewriting also preserve equivalence in context, but a change of variables or a weakening can give a special case or a weaker statement, so the relation below asks for a chain in each direction (Figure 1). Moreover, the relation depends on the library, through the bridges it contains and the facts that meet their hypotheses.

Definition 2 (`SemantEq`)**.** Statements P and Q are ***`SemantEq`*** over $\mathcal{L}$, pronounced "semantic", written $P \simeq_{Sem,\,\mathcal{L}} Q$, if there are statements $S_1, \ldots, S_k \in \mathcal{S}$ with $k \geq 1$, $S_1 = P$, $S_k = Q$, and $S_i \rightsquigarrow_{\mathcal{L}} S_{i+1}$ for each $i < k$, and likewise a chain from Q to P. In other words, $P \simeq_{Sem,\,\mathcal{L}} Q$ exactly when $P \rightsquigarrow^*_{\mathcal{L}} Q$ and $Q \rightsquigarrow^*_{\mathcal{L}} P$, where $\rightsquigarrow^*_{\mathcal{L}}$ is the reflexive transitive closure of $\rightsquigarrow_{\mathcal{L}}$.

Proposition 1. *The relation $\simeq_{Sem,\,\mathcal{L}}$ is an equivalence relation. Moreover, for all statements P, Q,*

$$P \simeq_{Sem,\,\mathcal{L}} Q \quad \Longrightarrow \quad \mathcal{L} \vdash (P \leftrightarrow Q).$$

Proof. Each permitted step is implication-sound. $S \rightsquigarrow_{\mathcal{L}} T$ gives $\mathcal{L} \vdash S \to T$. Rearrangement and bridge rewriting replace a subexpression by one that is equivalent to it in its context, in the second case because the hypotheses of the bridge instance are provable there. Adding an unused variable and specializing a universal quantifier are valid implications, and so is each structural weakening at a positive occurrence or in the prefix. Composing the steps of a chain gives $\mathcal{L} \vdash P \to Q$, and chains in both directions give $\mathcal{L} \vdash P \leftrightarrow Q$. Reflexivity is the chain of one statement, symmetry is built into Definition 2, and transitivity concatenates chains. Appendix C.2 gives each case in detail. □

Proposition 2 (Strictly finer)**.** *Rearrangement, change of variables and weakening alone, with the symbols of $\mathcal{L}$ treated as opaque, relate strictly fewer statements than logical equivalence in $\mathcal{L}$ does. The statements $\forall x{:}\mathbb{N}.\ 0 \leq x$ and $(\forall x{:}\mathbb{N}.\ 0 \leq x) \wedge (\forall x{:}\mathbb{N}.\ x = x)$ are logically equivalent in $\mathcal{L}$, and no chain of these steps derives the second from the first.*

Proof. We use a property of every chain, proved as Lemma 1 in Appendix C.2: if $S \rightsquigarrow^*_{\mathcal{L}} T$, then $S \to T$ follows by logic alone from the conclusions of the bridge instances used, each taken under the hypotheses of the context where it was applied, with equality read as an unknown relation of which only the replacement of equals by equals is assumed, so no chain uses that $t = t$ holds. A chain without bridges from the first statement to the second would therefore make the first imply the second under every interpretation of $\mathbb{N}$, $\leq$ and equality. The interpretation that makes $\mathbb{N}$ a nonempty

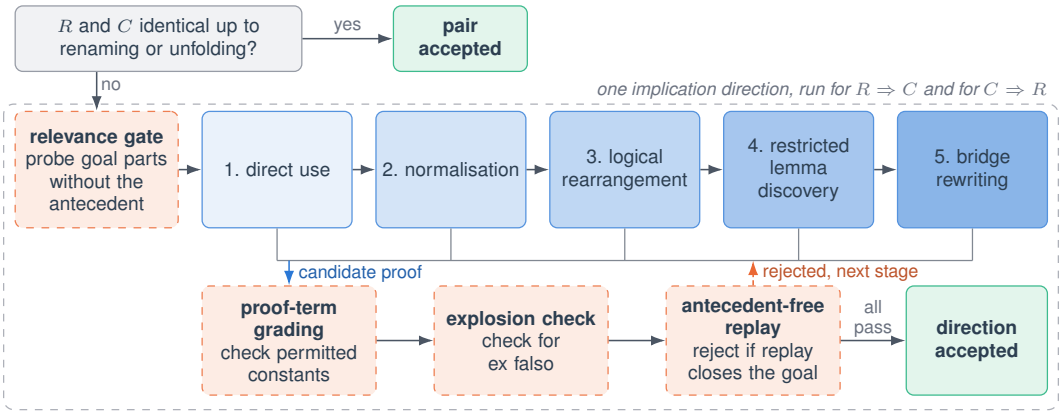

Figure 2: One implication direction. The relevance gate runs before the search. Each proof that a stage finds must pass the three audits below it, or the search moves on to the next stage.

set, $\leq$ true everywhere and equality the empty relation, with every other sort and symbol interpreted arbitrarily, makes the first statement true and the second false. □

Proposition 1 says `SemantEq` implies logical equivalence, and Proposition 2 shows that logic alone never identifies two statements by their truth values, so without bridges even a provable statement such as $4 = 4$ cannot be added as a new conjunct. The rest of the relation depends on the identifications the library states. `SemantEq-Eval` meets their hypotheses only with local hypotheses, its fixed rewrite set and sixteen numeral facts (Appendix F).

2.2 SemantEq-Eval: A PROOF SEARCH WITHOUT A MODEL

Given a reference R and a candidate C, `SemantEq-Eval` accepts the pair at once if the two are identical up to renaming and definitional unfolding. Otherwise it attempts $R \Rightarrow C$ and $C \Rightarrow R$, and it accepts the pair only if both directions succeed.

Each direction runs the five stages of Figure 2 in order, from cheap to expensive, and stops at the first proof that passes the audits. The first stage applies the antecedent to the goal directly. The second rewrites both statements into a normal form with a fixed set of logical identities and cast lemmas and tries again, and the third takes both apart into their logical components and matches them piece by piece. No library fact outside the fixed set enters these three stages. Rearrangement alone settles a large share of real pairs, since formalizers most often differ in presentation rather than in substance (Appendix D). The last two stages collect a small pool of library equations and biconditionals whose sides match parts of the two statements, and rewrite with them in either direction, one bridge at a time.

Audits. The stages call Lean's simplifier, rewriter, and backward search. We choose the lemmas that these procedures may use, but we do not control how they build the proof term. A proof they find may hide work in an auxiliary constant, close the goal by a route other than the intended rewrite, or use the antecedent in a way that contributes nothing. Four audits guard against such proofs (Figure 2), and passing them is the acceptance criterion of the program, not a consequence of Proposition 1. **Relevance** runs before the search. It refuses the direction if the stage procedures prove, without the antecedent, a part of the goal that the antecedent does not assert. The other three audits inspect every proof that the search finds. **Proof-term grading** expands auxiliary constants and admits only the antecedent, logical rules, bridges, and a few further classes (Appendix F). The **explosion check** fails a proof that derives `False` from its context and concludes the goal from it, but it allows a refutation inside a negation, as in proving $\neg Q$ by assuming Q. **Antecedent-free replay** reruns every accepting transformation with the antecedent withheld, and it discards the acceptance if the goal still closes. Removing any one audit admits the errors it targets (§3.2).

Table 1: Equivalence judgements on the ProofNetVerif test split, scored on the 1,303 of 1,452 pairs that elaborate on our toolchain, under the published labels and under our adjudicated labels.

Labels	System	TP	FP	FN	TN	Precision	Recall	F1
Published	Ours	242	0	229	832	**100.0**	**51.4**	**67.9**
	BEq+‡	230	8	241	824	96.6	48.8	64.9
Adjudicated	Ours	242	0	127	934	**100.0**	**65.6**	**79.2**
	BEq+‡	228	10	141	924	95.8	61.8	75.1

‡ Ported to our toolchain with compatibility fixes and no change of method.

3 EVALUATION

We evaluate `SemantEq-Eval` as a judge of statement equivalence. Our main benchmark is ProofNetVerif (Poiroux et al., 2025), 3,752 pairs of a reference and a candidate formalization in a validation and a test split, each labelled by people as faithful or not to the informal statement. We score the 1,303 of the 1,452 test pairs that elaborate on our toolchain. Our criterion, equivalence with the reference, is different, and recent audits have found errors in the statements of formal benchmarks (Ospanov et al., 2025; Ammanamanchi et al., 2026), so we review the test pairs labelled equivalent. The review changes 102 labels, all from equivalent to non-equivalent[1], and we report both label sets. For 88 of the relabelled pairs the review was prompted because no configuration of `SemantEq-Eval` accepts them and `SemantEq-ProofAudit` certifies them in at most one direction (§4). The other 14 came from a language-model screen and manual rulings, and a person made every final decision (Appendix H). We compare with `BEq+` ported to our toolchain (Appendix K).

3.1 HEAD TO HEAD ON PROOFNETVERIF

Table 1 compares the two systems on the same pairs. Under the adjudicated labels `SemantEq-Eval` accepts 14 more equivalent pairs than `BEq+`, for a precision of 100 against 95.8 and a recall of 65.6 against 61.8. The published labels give the same ranking, and there the `BEq+` numbers are close to the 98.0 and 48.3 that its authors report on the whole benchmark under an older Lean version. `SemantEq-Eval` makes no false acceptance under either label set, and `BEq+` makes 8 and 10 (Appendix E shows two). On the validation split, which we do not score, we know of one false acceptance by `SemantEq-Eval` (Appendix F).

3.2 AUDIT ABLATION

To test whether each audit of §2.2 does the work it was built for, and whether the precision gap comes from the design rather than from the benchmark's particular pairs, we build an *ablation set* of 100 non-equivalent pairs (Appendix H). Fifty are ProofNetVerif test pairs, including all ten that `BEq+` accepts. Fifty are constructed in four classes. A real statement is given one extra claim, the most common defect among the relabelled benchmark pairs, or it is paired with a neighbouring theorem about the same objects, with an unrelated true fact, or with a statement whose hypotheses contradict each other. Every added claim and every partner statement is easy to prove from the library, so a rejection comes from the restricted steps and the audits rather than from a lack of proving power. The set does not estimate how often such pairs occur in practice. We remove each audit in turn, and all four at once, and run `BEq+` on the same pairs (Table 2).

Each audit catches its own family of pairs. Removing relevance admits 19 pairs, 15 of them added claims. Removing proof-term grading admits 8, 6 of them unrelated true facts. Removing the explosion check admits 3, 2 of them benchmark pairs, and removing the antecedent-free replay admits 2. Removing all four admits 35, more than the 32 of the single removals together, so removing several audits exposes failures that no single removal shows. The search alone still rejects the other 65, including 20 of the 54 pairs that `BEq+` accepts, so its restricted steps and the audits both contribute.

[1] We release the adjudicated relabelling of the test split with our code.

Table 2: Contribution of each audit of §2.2 on the ablation set of 100 non-equivalent pairs, fifty from the ProofNetVerif test split and fifty constructed. Entries count erroneous acceptances.

	ProofNetVerif test split (50)	Added claim (18)	Neighbouring theorem (14)	Unrelated true fact (14)	Vacuous hypotheses (4)	Total
Ours, all checks	0	0	0	0	0	**0**
without relevance	0	15	4	0	0	19
without proof-term grading	0	0	2	6	0	8
without explosion check	2	0	0	0	1	3
without antecedent-free replay	0	0	1	1	0	2
without all four	2	17	7	7	2	35
BEq+ (our toolchain)	10	18	9	13	4	54

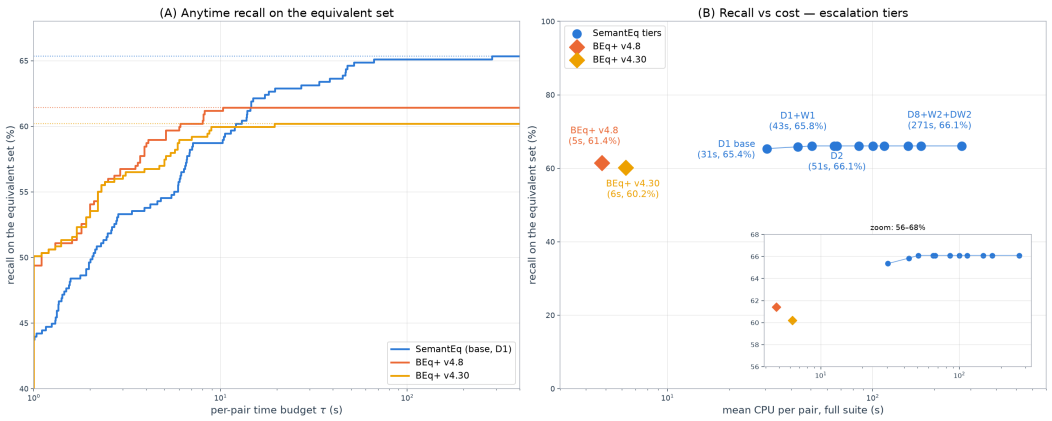

Figure 3: (A) Recall attainable when each pair is allowed at most τ seconds, computed from the uncapped runs on the *equivalent set*. BEq+ leads below roughly 15 s per pair, and the curves level off at 65.4% for SemantEq-Eval and at 61.4% and 60.2% for BEq+ on Lean v4.8 and on our toolchain. (B) Recall against mean cost per pair for each search configuration (Appendix I).

3.3 Search strength and saturation

We next ask how much more search helps on the *equivalent set*. It holds the 407 pairs labelled equivalent after the review, 369 ProofNetVerif test pairs and 38 of the 40 pairs of a curated recall suite whose references come from ProofNet# (Azerbayev et al., 2023; Poiroux et al., 2025), PutnamBench (Tsoukalas et al., 2024), and Compfiles (Renshaw et al., 2026) (Appendix H), so its recall is not comparable with Table 1. We raise the depth of chained rewrites, the number of bridges admitted per step, and the number of intermediate statements kept at each depth far past their useful range (Table 5, Appendix I). The base configuration accepts 65.4% of the pairs. Doubling the bridge pool (D1+W1) adds two pairs and a second chained rewrite (D2) adds one, for 66.1%. Eight stronger configurations add none, while the mean time per pair grows from 50.7 to 271.3 seconds. The median stays at 2.0 seconds in every configuration, because a larger budget only adds time to the pairs that fail. Figure 3 shows the same plateau against a time budget per pair, with BEq+ the cheaper judge below about 15 seconds per pair and SemantEq-Eval ahead above it.

The saturation is a property of this search algorithm. The search has no general way to build a new object or prove a new bridge, and it meets a bridge's hypotheses only with local hypotheses, its fixed rewrite set and the numeral facts of Appendix F. Many of the pairs that the search misses need such a step, for example applying the antecedent at a term or a structure that the goal only implies. Others need an identification that Mathlib states in one direction only, which gives no chain until the library states it both ways. The bottleneck is therefore the set of operations, not the search budget, so Section 4 changes the prover and keeps the audit fixed.

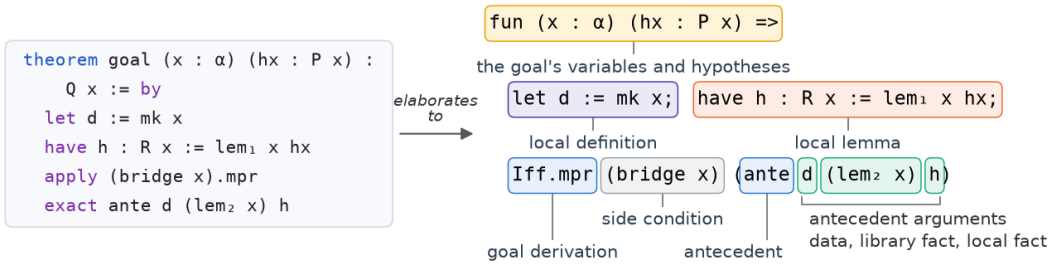

Figure 4: The auditor gives each position in a proof term a role. The same library fact can be admitted as the justification of a rewrite and refused when it proves the goal directly.

4 Decoupling Proof Generation from Proof Acceptance

We test a language model as the prover on the *certification set*, the full 511 pairs labelled equivalent before the adjudication, namely 471 pairs of the ProofNetVerif test split and the 40 recall pairs. After the adjudication the set holds 407 equivalent and 104 non-equivalent pairs. For each pair and each of four prompt styles, which range from rewriting only to expedient shortcuts (Appendix B), GPT 5.6 Sol in Codex writes a proof of each direction in one session per pair. A proof that compiles establishes the implication, not that the statements say the same thing. Under the expedient and the unrestricted prompts the model's proofs compile in both directions for 43 and 45 of the 104 non-equivalent pairs, so a judge that only checks compilation would accept them. We therefore keep acceptance separate from generation. Every returned proof is checked by `SemantEq-ProofAudit`, and a pair is accepted only when both directions are certified.

4.1 SemantEq-ProofAudit: Auditing Supplied Proofs

`SemantEq-ProofAudit` is built from the audits of `SemantEq-Eval`. Lean checks that a supplied proof is valid, and `SemantEq-ProofAudit` then tests it against an audit policy based on Definition 1. It reads the proof term that Lean builds from the tactic script and gives each position one of three roles (Figure 4): the goal derivation, an argument handed to the antecedent, or the justification of a rewrite. A constant is judged by its shape and by the role of its position. On the goal derivation only the antecedent, logical rules, and bridges are admitted, a bridge must convert the goal rather than state it, and the antecedent must do work in every part of the proof, such as each branch of a case split. Each rejection names the check that failed, so `SemantEq-ProofAudit` can serve as a standalone judge or give feedback to a prover in a loop (Appendix J).

Model proofs also contain material that the search never produces, such as local lemmas and objects built along the way, and three rules cover it.

- **Local lemmas.** A lemma proved from the antecedent is audited with its proof expanded. A lemma proved without it is judged by its shape like a library fact, and each direction of its proof is then checked as an implication proof of its own, so a direction that proves its conclusion without using its hypothesis fails. No proof may contain an unproved hole.
- **Instantiation of the antecedent.** The arguments of each application of the antecedent must come from the types, objects, and structures of the goal. A replay then withholds the antecedent but keeps the proof's own constructions, and it refuses the proof if the goal closes with those constructions but not from the library alone.
- **Library facts at a hypothesis of the antecedent.** Beyond the definition, the auditor tolerates a library fact that meets a hypothesis of the antecedent and records every such use, because a change of domain often leaves a fact that the library states in one direction only. A stricter policy refuses such uses when they draw on no hypothesis of the goal (Appendix G).

The auditor has no relevance audit and tolerates library facts in the justification of a rewrite, because it sees only the final proof term. The search has no instantiation check, and it builds no data objects, but instance synthesis once supplied a structure that the goal does not carry (Appendix F).

Table 3: Certificates on the certification set, the 407 equivalent pairs and the 104 that our review relabelled non-equivalent. A pair is accepted only when both directions are certified, and precision and recall are over that acceptance. The reference rows apply the same count to `SemantEq-Eval` and `BEq+`.

	Equivalent (407)			Non-equivalent (104)				
	both	one	neither	both	one	neither	Precision	Recall
`BEq+`	245	29	133	2	26	76	99.2	60.2
`SemantEq-Eval` (default)	266	58	83	0	34	70	100.0	65.4
`SemantEq-ProofAudit`, by prompt style								
Expedient	309	40	58	0	49	55	100.0	75.9
Unrestricted	322	53	32	0	57	47	100.0	79.1
Faithful	**330**	50	27	0	67	37	100.0	**81.1**
Rewrite	329	51	27	0	70	34	100.0	80.8

4.2 Results

`SemantEq-ProofAudit` certifies none of the 104 non-equivalent pairs in both directions under any style (Table 3). It certifies many of them in one direction, as expected when one statement is a special case of the other, and the certified direction then shows which statement is at least as strong. The reasons for rejection often show where they differ. A recurring pattern applies the antecedent at an object the goal never names, or binds it and closes the goal with a library theorem (Appendix J).

On the 407 equivalent pairs the faithful style certifies 330, 64 more than the 266 of `SemantEq-Eval`, and `BEq+` certifies 245. The model's 330 share 255 pairs with the search's 266, add 75 that `SemantEq-Eval` never closes, and miss 11 that it does, so together the two certify 341 of the 407 pairs (83.8%). The expedient style certifies 309 and leaves 58 pairs with no certified direction, against 27 for the faithful style, so the auditor refuses shortcuts that the other styles avoid. Under the expedient and unrestricted prompts the model wrote proofs that compile in both directions for about two in five of them, and the audit rejected every one. Under the stricter audit policy of Appendix G the faithful count falls to 323, which keeps 57 of the 64 extra pairs. Of the 77 equivalent pairs that the faithful style misses, 57 have a compiling proof that the auditor refused in some direction, and 20 lack an elaborating proof in some direction. The model never saw a verdict, so these counts measure one session per pair.

5 Conclusion

We present `SemantEq`, which defines an equivalence relation meant to capture when two formal statements are mathematically equivalent in meaning. Earlier proof-based evaluators accept a pair when a chosen proof procedure succeeds, whereas `SemantEq` defines the allowed steps by their effect and judges the proof itself. A search without a model improves on `BEq+` in precision and recall, and an 8.9-fold increase in its CPU time adds at most three pairs. A language model checked by `SemantEq-ProofAudit` certifies more equivalent pairs, 330 of 407, while accepting no non-equivalent pair. Separating the generation of equivalence proofs from their acceptance lets more expressive provers raise recall without changing what an acceptance means.

Limitations and future work Proposition 2 covers chains without bridges, and a bridge whose hypotheses a library proof meets can connect any two statements that the library proves equivalent. `SemantEq-Eval` restricts the facts that meet bridge hypotheses, and `SemantEq-ProofAudit` accepted no non-equivalent pair, so acceptance is evidence of faithful reformulation rather than a proof of it. The relabelled test set reflects our own adjudication, and we tested one prover model in one session per pair and prompt style. The method also needs a trusted reference. Evaluating a formal statement against its informal source when no reference exists is the natural next step.

AI USE STATEMENT

In this work, we used generative AI tools for the following tasks that require disclosure. They helped implement `SemantEq-Eval` and `SemantEq-ProofAudit`, and they helped build and clean the evaluation data, including a first screen of the ProofNetVerif test labels by Claude Fable 5 and the assembly of our test sets. In the experiments, GPT 5.6 Sol in OpenAI Codex wrote the equivalence proofs that `SemantEq-ProofAudit` checks, and these proofs are the object of study (§4.2). We also used Claude Fable 5.1 in Claude Code, and `gpt-6-astra` in OpenAI Codex to check the proof of Proposition 1 and Proposition 2, and to find counterexamples to an earlier version of Definition 1. We have not used generative AI tools to generate synthetic data sets or to translate, and qualitative data analysis is not applicable to this work. Additionally, we used these tools to spell-check parts of the paper and find errors, to find related work, and to check and merge the bibliography and check the consistency of the numbers. We have reviewed all AI-assisted work. Three authors verified and tested the code of both programs for correctness. The authors also checked every label change and every item of our test sets, and every final label is a human decision. The authors checked the final definition and proof themselves. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We release the code of `SemantEq-Eval` and `SemantEq-ProofAudit` together with the exact Lean and Mathlib versions they run on. The repository gives instructions to reproduce every table in the paper and to use `SemantEq` as a general tool on new pairs of statements. Appendix K lists the toolchain, the two changes we made to run `BEq+` on it, the settings of the model campaign, and the hardware we used. Appendix H describes the datasets and the audit of the ProofNetVerif test labels, and Appendix B gives the four prompts in full.

REFERENCES

Tudor Achim, Alex Best, Alberto Bietti, Kevin Der, Mathïs Fédérico, Sergei Gukov, Daniel Halpern-Leistner, Kirsten Henningsgard, Yury Kudryashov, Alexander Meiburg, et al. Aristotle: IMO-level automated theorem proving. *arXiv preprint arXiv:2510.01346*, 2025.

Pawan Sasanka Ammanamanchi, Siddharth Bhat, and Stella Biderman. Faults in our formal benchmarking: Dataset defects and evaluation failures in Lean theorem proving. In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*, volume 306 of *Proceedings of Machine Learning Research*, 2026. arXiv:2606.29493.

Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward W. Ayers, Dragomir Radev, and Jeremy Avigad. ProofNet: Autoformalizing and formally proving undergraduate-level mathematics. *arXiv preprint arXiv:2302.12433*, 2023.

Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. doi: 10.1017/CBO9781139172752.

Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. Hammering towards QED. *Journal of Formalized Reasoning*, 9(1):101–148, 2016. doi: 10.6092/issn.1972-5787/4593.

Luoxin Chen, Jinming Gu, Liankai Huang, Wenhao Huang, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Kaijing Ma, et al. Seed-Prover: Deep and broad reasoning for automated theorem proving. *arXiv preprint arXiv:2507.23726*, 2025.

Joshua Clune, Yicheng Qian, Alexander Bentkamp, and Jeremy Avigad. Duper: A proof-producing superposition theorem prover for dependent type theory. In *15th International Conference on Interactive Theorem Proving (ITP)*, volume 309 of *LIPIcs*, pp. 10:1–10:20, 2024. doi: 10.4230/LIPIcs.ITP.2024.10.

Johan Commelin and Adam Topaz. Abstraction boundaries and spec driven development in pure mathematics. *Bulletin of the American Mathematical Society*, 61(2):241–255, 2024. doi: 10.1090/bull/1831.

Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28*, volume 12699 of *Lecture Notes in Computer Science*, pp. 625–635. Springer, 2021. doi: 10.1007/978-3-030-79876-5_37.

Guoxiong Gao, Yutong Wang, Jiedong Jiang, Qi Gao, Zihan Qin, Tianyi Xu, and Bin Dong. Herald: A natural language annotated Lean 4 dataset. In *International Conference on Learning Representations (ICLR)*, 2025.

Hojae Han, Jongyoon Kim, Sanghyeok Park, Dongwook Cheon, Yeachan Park, Myeong Jae Jeon, Sunjong Choe, Soonho Kong, Wonseok Hur, Seung-won Hwang, and Donghoon Hyeon. ShadowBench: Toward reliable automatic evaluation of semantic alignment in autoformalization. In *Proceedings of the 2026 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2026. arXiv:2608.29270.

Thomas Hubert, Rishi Mehta, Laurent Sartran, et al. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, 651(8106):607–613, 2025. doi: 10.1038/s41586-025-09833-y.

Vasily Ilin and Brian Nugent. Sorries are not the hard part: An expert-review case study of a semi-autonomous formalization. *arXiv preprint arXiv:2606.13925*, 2026.

Albert Qiaochu Jiang, Sean Welleck, Jin Peng Zhou, Timothée Lacroix, Jiacheng Liu, Wenda Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *International Conference on Learning Representations (ICLR)*, 2023.

Guillaume Lample, Timothée Lacroix, Marie-Anne Lachaux, Aurelien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. HyperTree proof search for neural theorem proving. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 35, pp. 26337–26349, 2022.

Zenan Li, Yifan Wu, Zhaoyu Li, Xinming Wei, Fan Yang, Xian Zhang, and Xiaoxing Ma. Autoformalize mathematical statements by symbolic equivalence and semantic consistency. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 37, pp. 53598–53625, 2024.

Jannis Limperg and Asta Halkjær From. Aesop: White-box best-first proof search for Lean. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pp. 253–266, 2023. doi: 10.1145/3573105.3575671.

Yong Lin, Shange Tang, Bohan Lyu, Jiayun Wu, Hongzhou Lin, Kaiyu Yang, Jia Li, Mengzhou Xia, Danqi Chen, Sanjeev Arora, and Chi Jin. Goedel-Prover: A frontier model for open-source automated theorem proving. In *Conference on Language Modeling (COLM)*, 2025.

Yong Lin, Shange Tang, Bohan Lyu, Ziran Yang, Jui-Hui Chung, Haoyu Zhao, Lai Jiang, Yihan Geng, Jiawei Ge, Jingruo Sun, et al. Goedel-Prover-V2: Scaling formal theorem proving with scaffolded data synthesis and self-correction. In *International Conference on Learning Representations (ICLR)*, 2026.

Junqi Liu, Xiaohan Lin, Jonas Bayer, Yaël Dillies, Weijie Jiang, Xiaodan Liang, Roman Soletskyi, Haiming Wang, Yunzhou Xie, Beibei Xiong, et al. CombiBench: Benchmarking LLM capability for combinatorial mathematics. *arXiv preprint arXiv:2505.03171*, 2025a.

Qi Liu, Xinhao Zheng, Xudong Lu, Qinxiang Cao, and Junchi Yan. Rethinking and improving autoformalization: Towards a faithful metric and a dependency retrieval-based approach. In *The Thirteenth International Conference on Learning Representations (ICLR)*, pp. 86205–86233, 2025b. URL https://openreview.net/forum?id=hUb2At2DsQ.

Xiaoyang Liu, Kangjie Bao, Jiashuo Zhang, Yunqi Liu, Yu Chen, Yuntian Liu, Yang Jiao, and Tao Luo. ATLAS: Autoformalizing theorems through lifting, augmentation, and synthesis of data. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 38, pp. 56290–56326, 2025c.

Yuntian Liu, Tao Zhu, Xiaoyang Liu, Yu Chen, Zhaoxuan Liu, Qingfeng Guo, Jiashuo Zhang, Kangjie Bao, and Tao Luo. Generalized tree edit distance (GTED): A faithful evaluation metric for statement autoformalization. In *ICML 2025 Workshop on AI for Math*, 2025d. arXiv:2507.07399.

Jianqiao Lu, Yingjia Wan, Zhengying Liu, Yinya Huang, Jing Xiong, Chengwu Liu, Jianhao Shen, Hui Jin, Jipeng Zhang, Haiming Wang, Zhicheng Yang, Jing Tang, and Zhijiang Guo. Process-driven autoformalization in Lean 4. *arXiv preprint arXiv:2406.01940*, 2024.

Jianqiao Lu, Yingjia Wan, Yinya Huang, Jing Xiong, Zhengying Liu, and Zhijiang Guo. FormalAlign: Automated alignment evaluation for autoformalization. In *International Conference on Learning Representations (ICLR)*, 2025.

Logan Murphy, Kaiyu Yang, Jialiang Sun, Zhaoyu Li, Anima Anandkumar, and Xujie Si. Autoformalizing Euclidean geometry. In *International Conference on Machine Learning (ICML)*, volume 235 of *Proceedings of Machine Learning Research*, pp. 36847–36893, 2024.

Azim Ospanov, Farzan Farnia, and Roozbeh Yousefzadeh. miniF2F-Lean revisited: Reviewing limitations and charting a path forward. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 38, pp. 79964–80002, 2025. arXiv:2511.03108.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. BLEU: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 311–318, 2002. doi: 10.3115/1073083.1073135.

Zhongyuan Peng, Yifan Yao, Kaijing Ma, Shuyue Guo, Yizhe Li, Yichi Zhang, Chenchen Zhang, Yifan Zhang, Zhouliang Yu, Luming Li, Minghao Liu, Yihang Xia, et al. CriticLean: Critic-guided reinforcement learning for mathematical formalization. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 3049–3088, 2026.

Auguste Poiroux, Gail Weiss, Viktor Kunčak, and Antoine Bosselut. Reliable evaluation and benchmarks for statement autoformalization. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 17947–17969. Association for Computational Linguistics, November 2025. doi: 10.18653/v1/2025.emnlp-main.907. URL https://aclanthology.org/2025.emnlp-main.907/.

Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393*, 2020.

Yicheng Qian, Joshua Clune, Clark Barrett, and Jeremy Avigad. Lean-auto: An interface between Lean 4 and automated theorem provers. In *Computer Aided Verification (CAV)*, Lecture Notes in Computer Science, pp. 175–196, 2025. doi: 10.1007/978-3-031-98682-6_10.

Ahmad Rammal, Niket Patel, Fabian Gloeckle, Amaury Hayat, Julia Kempe, Remi Munos, Charles Arnal, and Vivien Cabannes. Formalizing mathematics at scale. *arXiv preprint arXiv:2605.29955*, 2026.

Z. Z. Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, et al. DeepSeek-Prover-V2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. *arXiv preprint arXiv:2504.21801*, 2025.

David Renshaw et al. Compfiles: Catalog of math problems formalized in Lean. https://github.com/dwrensha/compfiles, 2026. GitHub repository.

Daniel Selsam and Leonardo de Moura. Congruence closure in intensional type theory. In *International Joint Conference on Automated Reasoning (IJCAR)*, volume 9706 of *Lecture Notes in Computer Science*, pp. 99–115. Springer, 2016. doi: 10.1007/978-3-319-40229-1_8.

Terence Tao. Formalizing the proof of PFR in Lean4 using Blueprint: a short tour. Blog post, https://terrytao.wordpress.com/2023/11/18/formalizing-the-proof-of-pfr-in-lean4-using-blueprint-a-short-tour/, 2023.

The Formal Conjectures Authors. The Formal Conjectures repository. `https://github.com/google-deepmind/formal-conjectures`, 2025.

The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pp. 367–381, 2020. doi: 10.1145/3372885.3373824.

George Tsoukalas, Jasper Lee, John Jennings, Jimmy Xin, Michelle Ding, Michael Jennings, Amitayush Thakur, and Swarat Chaudhuri. PutnamBench: Evaluating neural theorem-provers on the Putnam mathematical competition. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2024.

Floris van Doorn, Patrick Massot, and Oliver Nash. Formalising the h-principle and sphere eversion. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pp. 121–134, 2023. doi: 10.1145/3573105.3575688.

Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, et al. Kimina-Prover Preview: Towards large formal reasoning models with reinforcement learning. *arXiv preprint arXiv:2504.11354*, 2025.

Haocheng Wang, Baiyu Huang, Yingjia Wan, Xiao Zhu, Xiaoyang Liu, Yinya Huang, and Zhijiang Guo. FormalRx: Rectify and eXamine semantic failures in autoformalization. In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*, volume 306 of *Proceedings of Machine Learning Research*, 2026. arXiv:2607.04655.

Qingxiang Wang, Cezary Kaliszyk, and Josef Urban. First experiments with neural translation of informal to formal mathematics. In *Intelligent Computer Mathematics (CICM)*, volume 11006 of *Lecture Notes in Computer Science*, pp. 255–270. Springer, 2018. doi: 10.1007/978-3-319-96812-4_22.

Ke Weng, Lun Du, Sirui Li, Wangyue Lu, Haozhe Sun, Hengyu Liu, and Tiancheng Zhang. Autoformalization in the era of large language models: A survey. *arXiv preprint arXiv:2505.23486*, 2025.

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–29, 2021. doi: 10.1145/3434304.

Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

Yutong Wu, Di Huang, Ruosi Wan, Yue Peng, Shijie Shang, Chenrui Cao, Lei Qi, Rui Zhang, Xishan Zhang, Zidong Du, Jie Yan, and Xing Hu. StepFun-Formalizer: Unlocking the autoformalization potential of LLMs through knowledge-reasoning fusion. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 33980–33988, 2026. doi: 10.1609/aaai.v40i40.40691. arXiv:2508.04440.

Huajian Xin, Daya Guo, Zhihong Shao, Zhizhou Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. DeepSeek-Prover: Advancing theorem proving in LLMs through large-scale synthetic data. *arXiv preprint arXiv:2405.14333*, 2024.

Huajian Xin, Z. Z. Ren, Junxiao Song, Zhihong Shao, Wanjia Zhao, Haocheng Wang, Bo Liu, Liyue Zhang, Xuan Lu, Qiushi Du, et al. DeepSeek-Prover-V1.5: Harnessing proof assistant feedback for reinforcement learning and Monte-Carlo tree search. In *International Conference on Learning Representations (ICLR)*, 2025. arXiv:2408.08152.

Kaiyu Yang, Gabriel Poesia, Jingxuan He, Wenda Li, Kristin E. Lauter, Swarat Chaudhuri, and Dawn Song. Position: Formal mathematical reasoning—a new frontier in AI. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, volume 267 of *Proceedings of Machine Learning Research*, pp. 82384–82398. PMLR, 2025.

Huaiyuan Ying, Zijian Wu, Yihan Geng, Jiayu Wang, Dahua Lin, and Kai Chen. Lean Workbook: A large-scale Lean problem set formalized from natural language math problems. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, volume 37, pp. 105848–105863, 2024.

Roozbeh Yousefzadeh and Xuenan Cao. Advocate for complete benchmarks for formal reasoning with formal/informal statements and formal/informal proofs. *arXiv preprint arXiv:2507.04719*, 2025.

Xuejun Yu, Jianyuan Zhong, Zijin Feng, Pengyi Zhai, Roozbeh Mohit, et al. Mathesis: Towards formal theorem proving from natural languages. In *International Conference on Learning Representations (ICLR)*, 2026. arXiv:2506.07047.

Zhouliang Yu, Ruotian Peng, Keyi Ding, Yizhe Li, Zhongyuan Peng, Minghao Liu, et al. FormalMATH: Benchmarking formal mathematical reasoning of large language models. *arXiv preprint arXiv:2505.02735*, 2025.

Ke Zhang, Patricio Gallardo Candela, Sudhir Murthy, Yi Xie, Zhi Wang, and Maziar Raissi. Beyond compilation: Evaluating faithful natural-language-to-Lean statement formalization. *arXiv preprint arXiv:2606.31002*, 2026a.

Lan Zhang, Xin Quan, and André Freitas. Consistent autoformalization for constructing mathematical libraries. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 4020–4033, 2024.

Yuanhe Zhang, Yuekai Sun, Taiji Suzuki, Jason D. Lee, and Fanghui Liu. LeanMarathon: Toward reliable AI co-mathematicians through long-horizon Lean autoformalization. *arXiv preprint arXiv:2606.05400*, 2026b.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. miniF2F: A cross-system benchmark for formal Olympiad-level mathematics. In *International Conference on Learning Representations (ICLR)*, 2022.

Thomas Zhu, Joshua Clune, Jeremy Avigad, Albert Q. Jiang, and Sean Welleck. Premise selection for a Lean hammer. In *International Conference on Learning Representations (ICLR)*, 2026.

A Related work

The main text compares SemantEq only with BEq and BEq+. Here we cover the wider literature.

A.1 Autoformalization systems and benchmarks

Most benchmarks for formal mathematics give a model a formal statement and ask for a proof, e.g., miniF2F (Zheng et al., 2022), ProofNet (Azerbayev et al., 2023), PutnamBench (Tsoukalas et al., 2024), FormalMATH (Yu et al., 2025), and CombiBench (Liu et al., 2025a). Provers evaluated on these benchmarks have improved quickly (Polu & Sutskever, 2020; Lample et al., 2022; Xin et al., 2024; 2025; Ren et al., 2025; Lin et al., 2025; 2026; Wang et al., 2025; Hubert et al., 2025; Chen et al., 2025; Achim et al., 2025). When the proof is in Lean, the Lean kernel checks it (de Moura & Ullrich, 2021), but the kernel does not check that the formal statement says what the informal problem says. Early work on statement autoformalization used neural machine translation (Wang et al., 2018), and large language models made few-shot translation practical, for statements (Wu et al., 2022) and for proofs (Jiang et al., 2023). Later work trains dedicated formalizers (Ying et al., 2024; Gao et al., 2025; Lu et al., 2024; Liu et al., 2025c; Yu et al., 2026; Wu et al., 2026) or improves prompted models with retrieval and prover feedback (Zhang et al., 2024). Some of these systems use a language-model judge of faithfulness to train or to filter their output (Peng et al., 2026; Yu et al., 2026). The quality of the judge then limits the quality of the formalizer. Weng et al. (2025) and Yang et al. (2025) survey the area.

A.2 Surface metrics and source-conditioned evaluation

The first automatic metrics compared a predicted statement with a reference as text. BLEU (Papineni et al., 2002) penalizes valid rephrasings, and Azerbayev et al. (2023) found that it did not track expert judgments of accuracy. The rate at which predictions type-check shows only that they are well formed (Poiroux et al., 2025). A second group of methods uses a model to compare the formal statement with the informal source. FormalAlign trains a model to score their alignment (Lu et al., 2025), and other pipelines translate the formal statement back into natural language and compare the result with the source (Ying et al., 2024; Gao et al., 2025). Li et al. (2024) combine such a consistency score with symbolic equivalence among sampled candidates. Beyond Compilation combines compilation, judgments from several models, and human calibration, and it reports a large gap between statements that compile and statements judged faithful (Zhang et al., 2026a). FormalRx predicts an alignment verdict together with an error category, its location, and a correction (Wang et al., 2026). The judges also need checking, since Yousefzadeh & Cao (2025) report a large disagreement between a back-translation judge and manual review on miniF2F.

Methods in this group can read the informal source, which SemantEq does not. Their verdicts come from a model or a score, however, and not from a proof. SemantEq returns a binary judgment of agreement with a formal reference, and each acceptance is backed by a derivation that Lean checks under explicit restrictions.

A.3 Proof-based reference equivalence

The closest work proves the candidate and the reference equivalent inside the proof assistant. BEq restricts a prover model to a small set of tactics and requires the proof of each direction to use the other statement (Liu et al., 2025b). BEq+ replaces the model with a fixed list of symbolic strategies, and the same work introduces ProofNetVerif (Poiroux et al., 2025). Li et al. (2024) and Murphy et al. (2024) also prove equivalences, among sampled candidates and among geometry statements respectively. In these methods acceptance depends on what a fixed proof procedure finds and on the text of the proof. For BEq+, Section 2 shows that closing tactics may prove the goal from library facts alone, which its authors also report, and that a BEq+ script can name the antecedent without using it. SemantEq keeps the bidirectional design of BEq, and like BEq it fixes a set of allowed transformations, but it defines them by their effect on the statement and checks them on the proof term. Poiroux et al. (2025) restrict `exact?` because library search can make "unrelated true theorems" count as equivalent, which is the problem our introduction starts from, and they report their restricted check, BEq_{L}, at a precision of 100.0 and a recall of 30.9 on ProofNetVerif. A false acceptance found by Simon Sorg led to a guard that is now their default (their Appendix

A.6.1), and our ablation set includes that pair. `SemantEq` builds on this approach by separating the generation of a proof from an audit of the elaborated proof term, which examines the role of each library fact, the use of the antecedent in each part of the proof, and the arguments at which it is instantiated. `SemantEq-Eval` searches for chains of permitted steps directly. `SemantEq-ProofAudit` audits a proof written by any prover under rules built on the same kinds of step (Appendix G), and it records every use of its tolerances. A stronger prover can therefore be used without accepting every proof that compiles.

A.4 Structural transformation metrics

GTED also compares two statements through transformations (Liu et al., 2025d). It standardizes both Lean statements, builds their operator trees, and computes a generalized tree edit distance in which some transformations, such as renaming a bound variable, cost less than others. GTED and `SemantEq` share the aim of accepting structural rephrasings, but they differ in what they return. GTED returns a similarity score and produces no proof. `SemantEq` returns a binary judgment, and it accepts a pair only when it has a chain of permitted steps in both directions that Lean checks.

A.5 Shadow statements

ShadowBench evaluates a candidate against auxiliary statements, called shadows, that a language model drafts and experts review for each problem (Han et al., 2026). Its SA-Pass metric gives full credit only when the candidate compiles, implies each shadow, and is implied by the conjunction of the shadows. The method does not depend on one reference formalization, but each new problem needs new shadows. `SemantEq` instead applies one set of rules to every problem and compares the candidate with a single reference. The two methods could be combined, e.g., by using `SemantEq` to compare a candidate with a shadow.

A.6 Quality of formal references

Any reference-based metric assumes that the reference is a faithful formalization, and recent audits show that the assumption often fails. Ospanov et al. (2025) found many discrepancies between the formal and informal statements of miniF2F and released a corrected version. Ammanamanchi et al. (2026) document missing hypotheses, altered domains, and vacuous statements across several Lean benchmarks, and the Formal Conjectures project relies on human review to catch misformalized statements (The Formal Conjectures Authors, 2025). A pair that our programs accept therefore agrees with its reference, and the acceptance does not show that either statement matches the informal source.

A.7 Rewriting, equality saturation, and hammers

`SemantEq-Eval` uses the same parts as ordinary Lean automation, namely the simplifier, rewriting with library equations, and library search with `exact?` (de Moura & Ullrich, 2021; The mathlib Community, 2020). Rewriting modulo a set of equations is a classical topic (Baader & Nipkow, 1998), and congruence closure has been extended to the type theory of Lean (Selsam & de Moura, 2016). Equality saturation, as implemented in egg, stores many equal expressions in one shared data structure and applies rewrite rules in no fixed order (Willsey et al., 2021). It could make the search faster, but it does not settle which library equations should count as bridges. Stronger automation is also available, e.g., Aesop (Limperg & From, 2023), Duper (Clune et al., 2024), Lean-auto and LeanHammer (Qian et al., 2025; Zhu et al., 2026), and Sledgehammer in Isabelle (Blanchette et al., 2016). The steps of such tools cannot easily be restricted one by one, so `SemantEq` does not constrain their search. `SemantEq-ProofAudit` checks the proofs they produce instead.

B The four prompt styles

Every proof certified by `SemantEq-ProofAudit` was requested with one of four prompts. The prompts share a task description, reproduced once below, and differ only in their final "Proof style" section, reproduced for each style. Nothing else varied between styles.

Common task description

````
# Task

You are contributing to a corpus of Lean 4 equivalence arguments between
alternative formalizations of the same mathematical problem.

You are given two Lean 4 files, `ref.lean` and `cand.lean`. Each contains a
preamble (imports, `open` lines, possibly local definitions or structures)
and one theorem statement whose proof is `sorry`. The two files formalize
the same informal mathematical statement, possibly in different ways.

Your job: show that the two statements are semantically equivalent by
writing TWO implication proofs, one per direction, as two self-contained
Lean files:

- `ref_imp_cand.lean` — the ref statement is assumed, the cand statement is
  proved from it;
- `cand_imp_ref.lean` — the cand statement is assumed, the ref statement is
  proved from it.

## File layout (follow exactly)

Each output file has this shape:

```
import Mathlib
<the union of the two input files' `open` / `open scoped` lines>

-- ===== unified context (both sides' local defs, merged) =====
<local definitions/structures from BOTH preambles;
 if neither side has any, keep this header comment and note that it is empty>

-- ===== assumed statement (antecedent) =====
theorem <original_name>_assumed <binders> : <statement> := by
  sorry

-- ===== goal statement, PROVED from the antecedent =====
theorem <goal_name> <binders> : <statement> := by
  <your proof>
```

Rules:

- Copy both theorem statements VERBATIM — binders, implicit and instance
  arguments, unicode notation, hypothesis names. Do not strengthen, weaken,
  reorder, or otherwise "clean up" either statement.
- Rename only the theorem identifiers: append `_assumed` to the assumed
  side's name; the proved side keeps its original name (append `_goal` if
  the two names collide).
- Unified context: include each side's local definitions/structures under
  their original names. If both sides define the same name with identical
  bodies, include it once. If the same name is defined differently on the
  two sides, rename one of them apart and adjust that side's statement
  references consistently — nothing else about the statement may change.
- The assumed theorem's proof must be exactly `sorry`, and it must be the
  only `sorry` in the file. The goal theorem's proof may use the assumed
  theorem by name and must compile.
- Each file must compile standalone against current Mathlib with
  `import Mathlib` (Lean 4 syntax).
- If, after a genuine attempt, you cannot prove a direction, leave that
  goal's proof as `sorry` with a one-line comment `-- not proved: <reason>`.
````

```
## Deliverables

Write the two files into the pair's folder, next to `ref.lean` and
`cand.lean`, named exactly `ref_imp_cand.lean` and `cand_imp_ref.lean`.
Create or overwrite them; do not modify `ref.lean` or `cand.lean`.

If a Lean toolchain with Mathlib is available in your workspace, check that
each file actually compiles (e.g. `lake env lean <file>`) and repair it
until it does; otherwise take extra care that the syntax is valid current
Lean 4 / Mathlib.
```

Prompt style, expedient

```
# Proof style

Treat each implication as an exercise in efficiency: the shortest
compiling proof wins, and how it gets there is unimportant.

- The ambient library is fair game. When the goal — or any subgoal — is a
  known result or within reach of automation, prove it directly; there is
  no requirement to route the argument through the assumed statement, and
  the assumed statement may go unused wherever it is not needed.
- Prefer powerful closing tactics, decision procedures, and computation
  over step-by-step transformation.
- Any logically valid route is acceptable, including indirect ones: argue
  by contradiction or by eliminating cases whenever that is quicker.
- Optimize for brevity, not for making the proof reflect why the two
  statements correspond.
```

Prompt style, unrestricted

```
# Proof style

No constraints beyond correctness. Use whatever proof technique gets
each implication to compile.
```

Prompt style, faithful

```
# Proof style

Write proofs that genuinely TRANSPORT the assumed statement into the
goal — each proof should be evidence that the two statements say the same
thing, not merely that the goal is true. Concretely:

- Derive every part of the goal from the assumed statement. The assumed
  theorem must carry the mathematical work everywhere; if deleting the
  `_assumed` theorem would leave your proof easily repairable, the proof
  misses the point.
- Use outside library lemmas only as re-expression bridges: equation- or
  iff-shaped facts (and definitional unfolding) that convert between the
  two statements' spellings of the same hypothesis or conclusion. Do not
  use library results that supply the mathematical content itself, and do
  not prove the goal — or any conjunct of it — from the library or from
  scratch, even where that would be easy.
- Feed the assumed statement from the goal's own hypotheses (bridged as
  needed), not from facts obtained elsewhere.
- Argue directly: no proof by contradiction that manufactures `False`, no
  discharging the goal from an impossible case, and no case split whose
  branches bypass the assumption.
- Keep the file minimal: auxiliary `have` steps only for bridges that are
```

```
  actually used.
```

Prompt style, rewrite

```
# Proof style

Write proofs that genuinely TRANSPORT the assumed statement into the
goal — each proof should be evidence that the two statements say the same
thing, not merely that the goal is true. Concretely:

- Derive every part of the goal from the assumed statement. The assumed
  theorem must carry the mathematical work everywhere; if deleting the
  `_assumed` theorem would leave your proof easily repairable, the proof
  misses the point.
- Use outside library lemmas only as re-expression bridges: equation- or
  iff-shaped facts (and definitional unfolding) that convert between the
  two statements' spellings of the same hypothesis or conclusion. Do not
  use library results that supply the mathematical content itself, and do
  not prove the goal — or any conjunct of it — from the library or from
  scratch, even where that would be easy.
- Feed the assumed statement from the goal's own hypotheses (bridged as
  needed), not from facts obtained elsewhere.
- Argue directly: no proof by contradiction that manufactures `False`, no
  discharging the goal from an impossible case, and no case split whose
  branches bypass the assumption.
- Keep the file minimal: auxiliary `have` steps only for bridges that are
  actually used.

In addition, where possible, shape each proof as a pure rewrite transport:

- `intro` the goal's binders, then align the goal with the assumed
  statement through a chain of `rw [...]` / `simp only [...]` steps using
  equation/iff lemmas — rewriting the goal, or a `have` of the instantiated
  assumption — and finish with `exact` / `apply` of the assumed statement.
- Structural glue is fine: `intro`, `exact`, `apply`, `refine`,
  `constructor`, `obtain`/`rcases` on the assumption, `And.intro`,
  `Iff.intro`/`.mp`/`.mpr`.
- Avoid everything else: `calc`, computation and decision tactics
  (`norm_num`, `omega`, `decide`, `nlinarith`, `field_simp`, `ring`),
  `aesop`, bare `simp` with the default simp set, induction, and
  constructing witnesses or data beyond what the assumption provides.
- If a direction genuinely cannot be closed in this shape, fall back to
  the transport style above for the smallest possible step and mark that
  line with `-- non-rewrite step`.
```

C Notation and proof for the relation

C.1 Notation

Fix the library $\mathcal{L}$, here Lean with Mathlib. The class $\mathcal{S}$ consists of propositions over $\mathcal{L}$ with no free variables, identified up to renaming of bound variables and definitional unfolding. The atoms of a formula φ, written $\mathrm{atoms}(\varphi)$, are its atomic subformulas, namely equations and predicate applications.

Occurrences and contexts. Write $S = \mathcal{C}[\varphi]$ when one occurrence of a formula or term φ in S is selected. The expression $\mathcal{C}[\,\cdot\,]$ is the rest of the statement with a hole at that occurrence, and $\mathcal{C}[\varphi']$ replaces it by φ'. The context Γ contains the variables quantified over the occurrence and the hypotheses under which it stands. It contains nothing quantified or assumed inside the occurrence. A term or formula over Γ uses only its variables and the symbols of $\mathcal{L}$. The judgement $\mathcal{L}, \Gamma \vdash \psi$ means that ψ is provable from the hypotheses of Γ and the theorems of $\mathcal{L}$. Substitutions are well

typed and avoid variable capture. The domain of a change of variables or of a witnessed existential is a type rather than a proposition, and a term or formula that a step introduces contains, as written, no proof other than a hypothesis of its context.

Positive occurrences. An occurrence is positive if the path from the root passes only through $\wedge$, $\vee$, $\to$, and quantifier bodies and enters the left side of an implication an even number of times. The path never enters a biconditional, an equation, or a function or predicate argument. Negation is read as $\varphi \to \mathit{False}$ when determining polarity. The restriction applies to the first three structural weakening rules. It does not restrict replacement by an equal or equivalent expression under logical rearrangement or bridge rewriting.

Bridges. A bridge is a theorem $b \in \mathcal{B} \subseteq \mathcal{L}$ of the form

$$b = \forall \bar{y}.\ \psi_1 \to \cdots \to \psi_m \to (u \asymp v), \qquad \asymp \in \{=, \leftrightarrow\},$$

whose statement is not a schema of pure logic. For the abstract relation, $\mathcal{B}$ contains all Mathlib theorems whose conclusion after their hypotheses is an equation or a biconditional, apart from pure logical schemas. Logical identities such as commutativity of conjunction count as rearrangements. No bridge is chosen by hand. An instance $\sigma(b)$ assigns terms to $\bar{y}$. Its sides are $\sigma(u)$ and $\sigma(v)$, and its hypotheses are $\sigma(\psi_1), \ldots, \sigma(\psi_m)$. For a rewrite at context Γ, the assigned terms must be over Γ, and every hypothesis must be provable in $\mathcal{L}, \Gamma$. The available bridges and the facts used to discharge their hypotheses therefore depend on the library.

Prefixes. The prefix of a statement is its leading sequence of universal quantifiers and hypotheses. Write $S = \Pi\ \varphi$ when Π is an initial segment of that sequence and φ is the remainder. For example, a prefix of the form $\forall x_1.\ \psi_1 \to \forall x_2.\ \cdots$ can be cut after any quantifier or hypothesis. A term over the variables of Π may use those variables and library symbols. In the existential weakening rule, the witness t must have the quantified type D.

The first-order reading. Proposition 2 and Lemma 1 concern the steps read in a separate many-sorted first-order syntax. Sorts may be empty, the symbols of $\mathcal{L}$ are opaque, and statements are taken as written, without Lean's definitional identifications or any computation that depends on proofs. An interpretation gives each sort a set, each function symbol a function and each predicate symbol a relation, and a formula with free variables is evaluated under an assignment of values to them, so an atom has a truth value at each value of its variables. Equality is a relation like any other, with only the replacement of equals by equals. These results describe this first-order core of the steps, not every chain that Lean's full language admits.

C.2 Proofs of Proposition 1 and Lemma 1

Proof. Every permitted step preserves implication. For logical rearrangement, the selected formulas are equivalent under every interpretation, so in particular in the library's own logic. For bridge rewriting, the assigned terms are available in the local context Γ, and the bridge hypotheses are provable there. The instantiated bridge therefore gives equality or equivalence in Γ. Replacing the selected expression preserves equivalence of the whole statement in both cases.

For a change of variables, S implies $\forall y{:}\mathsf{D}.\ S$ because y is fresh. Universal elimination also gives $(\forall x{:}\mathsf{D}.\ \varphi) \to \varphi[t/x]$ for a term $t{:}\mathsf{D}$ available under the prefix. The restriction on structure arguments narrows the permitted instantiations without affecting soundness.

For structural weakening, dropping a conjunct, adding a disjunct, and introducing an existential with a witness $t{:}\mathsf{D}$ each give a valid local implication. Conjunction, disjunction, and quantifier bodies preserve implication. The right side of an implication preserves its direction, and the left side reverses it. An even number of entries into left sides therefore preserves the direction from the selected occurrence to the root. The restriction on the path excludes positions where this argument does not apply. In the prefix, adding a hypothesis preserves implication. Removing a hypothesis preserves implication when the remaining prefix hypotheses imply it by logic alone.

Hence $S \rightsquigarrow_{\mathcal{L}} T$ implies $\mathcal{L} \vdash (S \to T)$. Composing the implications along a finite chain from P to Q gives $\mathcal{L} \vdash (P \to Q)$. Chains in both directions give $\mathcal{L} \vdash (P \leftrightarrow Q)$.

A chain consisting of one statement proves reflexivity. The requirement of chains in both directions gives symmetry. For transitivity, concatenate a chain from P to Q with one from Q to R, and concatenate the reverse chains from R to Q and from Q to P. The resulting chains establish $P \simeq_{Sem,\,\mathcal{L}} R$. □

The proof of Proposition 2 rests on the following property of chains.

Lemma 1 (What a chain uses)**.** In the first-order reading of Appendix C.1, if $S \rightsquigarrow^{*}_{\mathcal{L}} T$, then $S \to T$ follows by the rules of logic alone from the conclusions of the bridge instances used along the chain, each taken under the hypotheses of the context where it was applied, where throughout equality is read as an unknown relation of which only the replacement of equals by equals is assumed, so that every equation occurring as a formula is an atom of unknown truth. Hence if $P \simeq_{Sem,\,\mathcal{L}} Q$, then $P \leftrightarrow Q$ follows in the same way from the instances used along the two chains.

Proof. By induction on the chain, one case for each kind of step. A rearrangement holds under every interpretation, so in particular under this reading. Each weakening move is an implication of logic, and removing a hypothesis is allowed only when it follows from the others under every interpretation. A change of variables is universal generalisation or instantiation. Bridge rewriting replaces $\sigma(u)$ by $\sigma(v)$ inside a formula. The hypotheses of the instance are discharged by the library, which is why the lemma assumes the instance's conclusion under the hypotheses of its context rather than the bridge itself; from that assumption the replacement follows by logic, for a biconditional by replacement of equivalents and for an equation by replacement of equals inside an atom, whatever the atom's head symbol denotes, and under a binder by the same replacement inside it, since the conclusion is assumed for every value of the variables it binds. Composing the steps gives $S \to T$, and for $P \simeq_{Sem,\,\mathcal{L}} Q$ the chain back gives $Q \to P$. □

D WORKED CHAINS FOR THE DEFINITION

Admitted rearrangements. The logical rearrangement action of Definition 1 admits every equivalence that holds under every interpretation, preserves atoms and leaves no atom idle. This covers commutativity, associativity and distribution of the connectives, De Morgan's laws, implication as disjunction, double negation, currying, the commutation of like quantifiers, the distribution of $\forall$ over $\wedge$ and of $\exists$ over $\vee$, prenex moves valid over possibly empty domains, and the unfolding of a biconditional into two implications, each admitted only when an instance satisfies both conditions on atoms. It excludes whatever introduces or removes an atom, such as a true conjunct with a new atom or a case distinction, any equivalence that relies on a part being always true or always false, the laws of equality, since equality is an atom like any other here and its symmetry or the substitution of equals are steps only when the library states them as bridges, and the few quantifier laws that hold only over nonempty domains, such as $(\forall x.\, \varphi) \leftrightarrow \varphi$ for x not in φ.

A pair settled by rearrangement alone. Formalisers differ most often in presentation rather than in substance, and a recurring pattern is the same components re-ordered:

Lean 4

```
-- reference (Munkres 29.10)
theorem exercise_29_10 {X : Type*}
  [TopologicalSpace X] [T2Space X] (x : X)
  (hx : ∃ U : Set X, x ∈ U ∧ IsOpen U ∧ (∃ K : Set X, U ⊂ K ∧ IsCompact K))
  (U : Set X) (hU : IsOpen U) (hxU : x ∈ U) :
  ∃ (V : Set X), IsOpen V ∧ x ∈ V ∧ IsCompact (closure V) ∧ closure V ⊆ U
-- candidate: same context, conclusion conjuncts permuted
theorem exercise_29_10_alt {X : Type*}
  [TopologicalSpace X] [T2Space X] (x : X)
  (hx : ∃ U : Set X, x ∈ U ∧ IsOpen U ∧ (∃ K : Set X, U ⊂ K ∧ IsCompact K))
  (U : Set X) (hU : IsOpen U) (hxU : x ∈ U) :
  ∃ (V : Set X), x ∈ V ∧ IsOpen V ∧ closure V ⊆ U ∧ IsCompact (closure V)
```

No library lemma is needed, because the two statements are related by a propositional rearrangement alone.

A pair settled by two bridges. The following pair differs only in where the division by 90 sits:

```lean
-- reference (USAMO 1996 P1)
theorem Usa1996P1_refs :
    (1 / (90:ℝ)) * ∑ n ∈ Finset.range 90, (2 * (n+1)) * Real.sin ((2 * (n+1)) * Real.pi
        / 180)
    = 1 / Real.tan (Real.pi / 180)
-- candidate: the average restated termwise (a variant of the autoformalized statement)
theorem Usa1996P1_alt :
    ∑ n ∈ Finset.range 90,
      (2 * (n+1)) * Real.sin ((2 * (n+1)) * Real.pi / 180) / 90
    = 1 / Real.tan (Real.pi / 180)
```

Matching the two statements against the library's rewrite index finds `one_div_mul_eq_div`, which turns $\frac{1}{a} \cdot b$ into $\frac{b}{a}$, and `Finset.sum_div`, which distributes division over a finite sum. Neither equation mentions the trigonometric content, and rewriting with them in turn takes one statement to the other. `SemantEq-Eval` settles the pair in both directions by exactly the two-step chain above at search depth two, but not at the default depth of one (Appendix F). The autoformalized candidate in the recall suite also reindexes the sum, and the search does not settle it. The chain never proves the trigonometric identity itself, which it carries across unchanged.

Changing the domain of a variable. A chain moves a bound variable to a new domain with two changes of variables. It generalizes over a variable of the new domain and specializes the old variable at a term built from it, and bridges rewrite the body in between. Write $S = \forall n{:}\mathbb{N}.\ P\ \uparrow n$ and $T = \forall z{:}\mathbb{Z}.\ 0 \le z \to P\, z$. The chain from S to T passes through the subtype of non-negative integers and uses the library's own biconditional `Subtype.forall`:

$\forall n{:}\mathbb{N}.\ P\ \uparrow n$	
$\forall w{:}\{z \mid 0 \le z\}.\ \forall n{:}\mathbb{N}.\ P\ \uparrow n$	generalize w
$\forall w.\ P\ \uparrow (w.\mathrm{val.toNat})$	specialize $n := w.\mathrm{val.toNat}$
$\forall w.\ P\,(w.\mathrm{val})$	bridge `Int.toNat_of_nonneg`, condition from w's type
$\forall z.\ \forall h{:}0 \le z.\ P\,(\langle z, h\rangle.\mathrm{val})$	bridge `Subtype.forall`
$\forall z{:}\mathbb{Z}.\ 0 \le z \to P\, z$	definitional unfolding.

The reverse direction cannot apply `Subtype.forall` backwards and specialize w at $\langle \uparrow n,\ p\rangle$, because the proof p of $0 \le \uparrow n$ would be the library theorem `Int.natCast_nonneg`, and a term that a step introduces may contain no library proof. Two bridges give a chain instead. T unfolds to $\forall z \in \mathrm{Ici}\, 0.\ P\, z$, the bridge `Nat.range_cast_int` rewrites $\mathrm{Ici}\, 0$ to the range of the cast from $\mathbb{N}$, and `Set.forall_mem_range` rewrites the result to S. A model proof that takes the direct route instead meets a hypothesis of the antecedent with a library fact, which `SemantEq-ProofAudit` admits under its tolerance, and Appendix J shows the same case for a positivity hypothesis.

E TWO FAILURES OF EVALUATION BY PROOFS

Both pairs below are labelled not equivalent in ProofNetVerif. The label concerns whether the candidate states the intended claim, so a pair can be labelled not equivalent even when both statements are provable. The listings show simplified proof steps from the `BEq+` strategies, without setup code or tactic wrappers. In each direction, `base_theorem` names the antecedent, the statement on the other side of the pair, and the goal is the statement to prove.

The first reference states the associativity of multiplication in `ZMod n`, Lean's type for the integers modulo n. The candidate states an identity about remainders of natural numbers, which follows from `Nat.mul_mod` with the equality reversed.

Reference (associativity in $\mathbb{Z}/n\mathbb{Z}$)

```lean
theorem exercise_1_1_4 (n : Nat)
    (a b c : ZMod n) :
  (a * b) * c = a * (b * c)
```

Candidate (labelled not equivalent)

```lean
theorem candidate (n : Nat)
    (a b c : Nat) :
  (a*b) % n * (c%n) % n
    = a*b*c % n
```

Simplified proof steps, reference ⇒ candidate

```Lean 4
convert base_theorem using 0
simp_all_arith!
exact fun n a b c =>
  base_theorem n a b c
```

Simplified proof steps, candidate ⇒ reference

```Lean 4
convert base_theorem using 0
simp_all_arith!
noncomm_ring
```

The `convert` step turns the goal into a comparison between the goal and the antecedent as propositions. Simplification removes the remainder identity but leaves an associativity obligation. In the direction on the left, the last line proves that obligation, which is the reference itself, with `base_theorem`. In the direction on the right, `noncomm_ring` proves it without the antecedent. The candidate thus has a direct proof by `Nat.mul_mod` and the reference a direct proof by `mul_assoc`, and proofs of the two statements taken separately do not show that the candidate states the intended claim (Section 2).

The second reference states that disjoint closed sets in a metric space have disjoint open neighbourhoods, which `SeparatedNhds` expresses. The candidate concludes only that the sets are disjoint, from the assumption that their intersection is empty.

Reference (Rudin 2.19a)

```Lean 4
theorem exercise_2_19a
    {X : Type*} [MetricSpace X]
    (A B : Set X)
    (hA : IsClosed A)
    (hB : IsClosed B)
    (hAB : Disjoint A B) :
  SeparatedNhds A B
```

Candidate (labelled not equivalent)

```Lean 4
theorem candidate
    {X : Type*} [MetricSpace X]
    {A B : Set X}
    (hA : IsClosed A)
    (hB : IsClosed B)
    (h : A ∩ B = ∅) :
  Disjoint A B
```

Simplified proof steps, reference ⇒ candidate

```Lean 4
have this :=
  base_theorem A B hA hB
    (Set.disjoint_iff_inter_eq_empty
      .mpr h)
exact SeparatedNhds.disjoint this
```

Simplified proof steps, candidate ⇒ reference

```Lean 4
have this : Disjoint A B := by
  apply_rules [base_theorem]
exact NormalSpace.normal
  A B hA hB this
```

In the direction on the left the antecedent is used but not needed. The argument passed to it, `Set.disjoint_iff_inter_eq_empty.mpr h`, already proves the goal `Disjoint A B` from the goal's own hypothesis, and `SeparatedNhds.disjoint` only turns the antecedent's conclusion back into that goal. In the direction on the right the antecedent is not used at all. `apply_rules` proves the intermediate claim from the hypothesis `hAB`, Lean prints that step as `have this := hAB`, and `base_theorem` does not appear in the proof term. `NormalSpace.normal` then proves the reference from the hypotheses that the sets are closed and disjoint. The right direction shows that a name in a script need not appear in the proof term, and the left direction shows that even an occurrence in the proof term can be unnecessary.

F Details of SemanteqEval

Direct use. The first stage applies the antecedent to the goal and lets the logical rules match the two, which settles a direction when the goal is the antecedent up to renaming, unfolding, or instantiation.

Normalisation. The normalisation stage rewrites both statements with a fixed set that carries no mathematical content: the library's cast normalisation extension, whose lemmas are equations and biconditionals by construction, and a curated pool of logical and set-membership identities such as `not_forall`, `forall_and`, `exists_prop`, `and_imp`, `Set.mem_setOf_eq`, `Set.ext_iff`, `Finset.mem_range`, and `Subtype.ext_iff`. Every member is an equation or a biconditional, so the stage performs rearrangement and bridge rewriting only. The antecedent itself is available as a rewrite rule, which is how a direction closes when the goal is a rephrasing of the antecedent's conclusion.

Logical rearrangement. Rearrangement is performed by a closer that asserts the antecedent into the local context, destructures conjunctions, existentials, biconditionals and disjunctions on both sides, instantiates universally quantified hypotheses at matching local objects, and searches backwards over the logical rules with a small depth bound. No library fact enters except through the normalisation set above.

Discovery. Bridges are found by indexing: the two statements are matched against the library's rewrite index, every fact whose left or right side matches a subterm is collected, and the pool is filtered to facts whose conclusion is an equation or a biconditional. The pool is ranked and capped at forty candidates. Each candidate is materialised as an attempt that rewrites the goal or the antecedent at the matching site, in either orientation. Discovery runs only after the cheap stages have failed, since those stages settle about three quarters of all directions at a small fraction of the time.

Search strength. Three parameters bound the search (Table 5). D is the depth of chained rewrites, the number of bridges applied in sequence before the closer runs; W_k doubles k times the two per-step bridge budgets, the candidate bridges tried and the lemmas discovered, which are 80 and 64 by default; DW_k doubles k times the number of distinct intermediate statements retained at each depth and carried to the next, 24 by default, and raises the budget spent per depth level by a factor of four each time. The default is $D = 1$ with the base budgets.

Proof-term grading. Every constant of an accepted proof term is classified, with local auxiliaries expanded through the kernel environment so that nothing hides inside a tactic's private lemma. Admitted are the antecedent; non-propositional constants, which are vocabulary; eliminators; schematic constants, whose type is built from sorts, variables and the core logical connectives and which therefore carry no domain content; facts whose conclusion is an equation, a biconditional, a heterogeneous equation or a disequality, or a class instance; and a fixed set of sixteen numeral facts, such as `le_refl`, `zero_le_one`, `two_ne_zero` and `Nat.zero_le`, which discharge the conditions of cast bridges and can relate no two statements on their own. Any other constant fails the proof. The same classification filters the discovery pool before any rewrite is attempted, so closer and audit agree by construction.

No proof ex falso. The proof term is walked with a negation depth. Entering a lambda increases the depth only when its body proves `False` and the introduced hypothesis actually occurs in that derivation, which is the shape of an honest refutation. A `False`-typed subterm at depth zero, and any of the eliminators `False.elim`, `absurd` and their kin at depth zero, fail the proof. A self-contradictory antecedent paired with its own rearrangement is thus still accepted, since its proof rearranges components and never derives `False`, while the pair below is refused in both directions:

```lean4
-- reference
theorem r : ∃ n : ℕ, Even n ∧ ¬ Even n
-- candidate
theorem c : ∃ x : ℝ, x = 0 ∧ x ≠ 0
```

Relevance audit. Before any search runs, this audit asks which parts of the goal actually need the antecedent. The goal is split into obligations, namely its conjuncts, the two sides of a biconditional, the body of an existential with its witness generalised again, and the domain of every hypothesis at any depth, and the split is taken on both the raw goal and its normalised form. An obligation that the antecedent asserts as one of its own components is kept. Every other obligation is attempted with the antecedent withheld, by the same proof procedure that each stage of the ladder in §2.2 runs, with the same lemma pool and the same time budget. If any such attempt succeeds, that part of the goal is provable on its own, and the direction is refused without being searched. The pair below is refused this way, because its second conjunct is provable by reflexivity.

```lean4
-- reference
theorem r : Irrational (Real.sqrt 2)
-- candidate
theorem c : Irrational (Real.sqrt 2) ∧ (2 + 2 = 4)
```

Antecedent-free replay. Every accepting attempt records the transformation it applied, the lemma, orientation and site. The identical transformation is replayed with the antecedent absent from the context, and an acceptance whose replay still closes the goal is discarded.

Determinism. Every judgement is a function of the two statements alone. Per-attempt budgets are absolute rather than shares of a shrinking remainder, the relevance probe is evaluated once per direction before the ladder, and no stage depends on wall-clock time. Two runs on the same statements give the same verdict on any machine.

A known false acceptance. On the validation split, which Table 1 does not score, `SemantEq-Eval` wrongly accepts one pair, Munkres 18.8b. The candidate adds a preorder on the space that its conclusion never uses. In the direction from the candidate, the search supplies the specialization preorder of the space, a structure the goal does not carry, which the change-of-variables clause of Definition 1 is meant to exclude. The instantiation check of `SemantEq-ProofAudit` refuses the same proof term, and the search has no such check.

G Details of SemantEq-ProofAudit, and what it accepts beyond the definition

Reading a proof term. `SemantEq-ProofAudit`, which we call the auditor, reads the proof term that Lean elaborates from a tactic script, not the script itself. The proof term is the fully explicit proof that Lean's kernel checks. The auditor walks the term from the root and gives every position one of three roles, shown in Figure 4. First, the *goal derivation* consists of the positions that produce the goal, namely the root, the body of every leading lambda and of every local binding, and the head and the arguments of each application. When the proof converts along an equation or a biconditional, the proof being converted also belongs to the goal derivation. Second, the *antecedent arguments* are the arguments handed to the antecedent, or to a fact obtained from it, when it is applied. Third, the *justification* of a conversion, labelled side condition in Figure 4, is the equation or biconditional along which a rewrite by `rw` or `simp`, or a use of `Iff.mp` or `Iff.mpr`, converts the proof. A constant is judged by the role of the position it occupies and by the shape of its statement, not by its name. On the goal derivation only the antecedent, logical rules and bridges are admitted, and any other library fact fails the proof. A library bridge is admitted there only as a step of a conversion, or as an argument of a rule that combines equations, such as `Eq.trans` or `congrArg`. Applied anywhere else on the goal derivation, with no mention of the antecedent in its application, it fails the proof, because its conclusion then states part of the goal outright instead of rewriting the antecedent into it. The rule concerns library facts, not local lemmas. Library facts at the other two roles fall under the tolerance described below. The checks for unproved holes, for proof ex falso and on applications of the antecedent run at every position, justifications included.

Use of the antecedent in every part. The auditor requires the antecedent to do work in each part of the proof, and it finds the parts by their position in the proof term. Not every mention of the antecedent counts as a use. A mention does not count when it is the discarded argument of `Function.const`, the value of a `have` or `let` whose name never occurs again, or part of a subterm whose statement is `True` or a closed equation with two sides that are equal by definition. A part whose proof has no use of the antecedent, and whose statement does not mention the antecedent, fails the proof. The parts are the following: (i) the whole proof, and on the goal derivation the rest of the proof after each `have` or `let`; (ii) each branch of a case split on anything not derived from the antecedent, and each branch of a case split on the antecedent, or on a fact derived from it, when nothing that the split produces is used; (iii) a proof passed as a lambda to a logical rule on the goal derivation, and the pointwise proof in an extensionality step such as `funext` or `Set.ext`; and (iv) every other component of a conjunction or a biconditional built on the goal derivation, unless it lies under a rewrite that the antecedent justifies. The rules above do not apply inside a justification, where a related rule applies instead. The related rule refuses a rewrite that turns a part of the goal into `True`, or drops it from a conjunction, by means of a proof of the part that does not use the antecedent.

Local lemmas. A local lemma proved from the antecedent is expanded in place, and its proof is audited at the role of the position where the lemma is used. A local lemma proved without the

antecedent, and any lemma in the file whose proof is left open, counts as an outside fact. Its statement is classified by shape and admitted or refused at the role where it is used, like a library fact. When the shape admits it, as an equation or a biconditional, its proof is read as well, and each direction of the proof is checked as an implication proof in its own right, with the direction's hypothesis standing in for the antecedent. A direction that proves a part of the goal without using its hypothesis fails the proof. The rule refuses a proof that states, as a local lemma, the equivalence of two statements that the library proves, proves each side outright, and rewrites along it. It also catches an added claim inside such a lemma. Every library fact that the proof of a local lemma uses is recorded like any other.

Instantiation of the antecedent. Each time the proof applies the antecedent by name, the auditor checks the arguments of the application against the goal's own objects, which it fixes by aligning the two statements. It first counts the data variables that each statement binds, meaning the variables that are not proofs. When the goal binds fewer, it also counts those that appear once the goal's conclusion is unfolded. The goal's own objects are the first data variables that the proof introduces for the goal, as many as the larger count, together with the objects that the proof obtains from them by destructuring a hypothesis or by a local definition. A data variable introduced later is not one of the goal's own objects. The checks on type and data arguments concern only an argument that appears in the statement that the application delivers. First, a type argument, meaning an argument whose own type is a universe such as `Type` or `Prop`, must belong to the goal. If it contains a free variable, it must be one of the goal's own objects or a type that the goal statement mentions. If it is closed, it must share a constant other than a logical connective with the goal statement, and `Prop` itself passes only when the goal statement mentions `Prop`. Neither rule applies when the goal statement quantifies existentially over a type. Second, every data variable in a data argument must be one of the goal's own objects, and a data argument that the delivered statement uses as a type must itself be one of the goal's own objects or appear in the goal statement. Proofs and closed terms are not restricted. Third, a structure argument, meaning an instance in Lean's sense, is checked when it contains a free variable, its class concerns one of the goal's own objects, and it is not itself one of the goal's own instances. It passes if it is equal by definition to the goal's instance of the same class or appears in the goal statement. Otherwise, if its class can have more than one instance, it fails when the goal carries its own instance of the class for the same object. If its class has at most one instance, it fails unless Lean's instance search can build it with classical decidability available, or the goal statement assumes finiteness with a `Finite` or `Fintype` instance. Fourth, a witness that the proof writes out for an existential hypothesis of the antecedent passes if it contains a variable not derived from the antecedent, or if it is a closed term that the goal statement mentions. A closed witness foreign to the goal statement fails the proof, e.g., a fresh constant when the goal's own hypothesis already provides the object, and so does a witness built only from variables derived from the antecedent. The checks do not follow a local name given to a partial application of the antecedent, so an argument passed through such a name is not checked.

Replay without the antecedent. The instantiation checks are backed by a replay, which runs only on a proof that passes every other check. The replay withholds the antecedent and keeps, by value, the local definitions and local facts that the proof builds without it. It then asks a weak prover, Lean's `assumption` followed by `exact?`, to close the goal from what remains, and a proof that uses the antecedent, another assumption or the goal itself does not count. The replay first uses the definitions and facts at the head of the proof, up to the first one that involves the antecedent. If the prover fails there, the replay follows the proof's introductions, its rewrites that do not use the antecedent, and the branches of its case splits. It probes at most four of the goals it reaches. If the prover closes such a goal with the proof's constructions, but cannot close it, or the original goal, from the library alone, the antecedent did no work and the proof is refused. If the prover closes the goal from the library alone, the replay neither refuses nor records the direction.

The tolerance, and a stricter audit policy. `SemantEq-Eval` meets the hypotheses of a bridge only with local hypotheses, its fixed rewrite set and the numeral facts of Appendix F, and on the goal derivation the auditor admits only the antecedent, logical rules and bridges. The auditor tolerates library facts in two other positions. At an antecedent argument, a library fact may meet a hypothesis of the antecedent. A change of domain often leaves such a hypothesis, e.g., $0 \leq \uparrow n$ in Appendix D, and refusing the fact would reject a faithful proof for a gap in the library rather than for a fault of the

proof. Inside a justification, a library fact of any shape is tolerated, because the current auditor does not distinguish a proof of a bridge's hypothesis from other content there. In both positions, every use of a fact that is neither a bridge nor a logical rule is recorded. A use at an antecedent argument is marked *severe* when the fact, together with its arguments, draws on no local hypothesis and on nothing derived from the antecedent, even if it mentions the goal's variables. A use that draws on a hypothesis of the goal stays within what the goal grants, while a severe use meets a hypothesis of the antecedent with a fact that the goal never states. The stricter policy counts every severe use as a failure, at the top level and inside the proofs of local lemmas alike. The auditor offers this policy as a mode, and Table 4 reports the campaign of §4.2 under it.

Table 4: The campaign of Table 3 under the stricter audit policy, in which a severe use, a library fact meeting a hypothesis of the antecedent while drawing on no local hypothesis, fails the proof wherever it occurs. Pairs are counted as both directions, one direction, or neither, and precision and recall are over pair-level acceptance, as in Table 3.

	Equivalent (407)			Non-equivalent (104)				
Prompt style	both	one	neither	both	one	neither	Precision	Recall
Expedient	303	45	59	0	46	58	100.0	74.4
Unrestricted	317	56	34	0	50	54	100.0	77.9
Faithful	**323**	56	28	0	59	45	100.0	**79.4**
Rewrite	323	56	28	0	64	40	100.0	79.4

Across the four styles, 199 proof directions carry a tolerated use at a hypothesis of the antecedent, 125 through a library constant and 74 only through a local lemma proved without the antecedent. The severe mark applies to 16, 13, 10, and 14 of them in the faithful, rewrite, expedient, and unrestricted styles, and 34 of the 53 severe uses draw on no local object at all. The facts involved are the openness of a fixed interval, the continuity of the identity on the reals, the openness of the whole space, the compactness of a singleton, and numeral facts such as the primality of a specific number. On the equivalent pairs, the stricter policy removes 7, 6, 6, and 5 pairs from the count certified in both directions, in the faithful, rewrite, expedient, and unrestricted styles. In each pair removed, the candidate carries a hypothesis that the library proves on its own, such as the continuity of the identity, and the direction that drops the hypothesis meets it with the library fact. Two further pairs lose their only certified direction, one of them in three styles, and in the fourth that pair is among those that lose both directions, because a numeral fact meets a hypothesis of the antecedent inside the proof of a local lemma, a use that the default policy records but only the stricter policy fails. On the non-equivalent pairs, the severe direction was the only certified one, so the pairs concerned move from one direction to neither. Under either policy, no non-equivalent pair is certified in both directions. Counting every tolerated use at a hypothesis of the antecedent as a failure, severe or not, lowers the faithful count further, to 303 of 407, but it also fails the uses that draw on a hypothesis of the goal.

Differences from the audits of `SemantEq-Eval`. The auditor has no relevance audit that asks the library whether a part of the goal is provable on its own. It judges the proof it is given, and it refuses a part of the goal whose proof never depends on the antecedent, which covers a candidate that only adds a claim the library proves on its own whenever the model proves that claim without the antecedent. It does not grade the justification of a rewrite, so a library fact can enter there, although a part of the goal collapsed to a truth value inside a justification is refused. The auditor reports every axiom a proof uses beyond the three standard ones but does not refuse on them. One accepted proof, in the unrestricted style, uses `native_decide` to show that `ZMod 2` has two elements, a true side fact that `decide` also proves. In the other direction, the search does not check how the antecedent is instantiated, and the auditor does.

H DATASETS AND ADJUDICATION

ProofNetVerif. The benchmark provides 3,752 reference and candidate pairs with human labels in a validation and a test split. We score the test split, 1,452 pairs of which 1,303 elaborate under our toolchain (Appendix K), under its published labels and under the adjudicated labels described next. We reviewed the test pairs labelled equivalent, and candidates for review came from a language-model screen and from the results of our two programs. First, a language model, Claude Fable 5, screened the pairs labelled equivalent as a pre-filter. Second, we reviewed the pairs that no configuration of `SemantEq-Eval` accepts and that the campaign of §4.2 certifies in at most one direction. A person made every final decision. The review changed 102 labels, all from equivalent to non-equivalent, since pairs labelled not equivalent were not reviewed. The released provenance shows that the review of 88 of the 102 was prompted by the second source. The other 14 came from the language-model screen and from manual rulings. At least eight changes follow from our criterion rather than from an error in the benchmark. Four Artin 2.8.6 candidates state that the centre of the product equals the product of the centres, where the reference states an isomorphism, and four Rudin 1.4 candidates add a linear order (Appendix J). Only two relabelled pairs are ones that `BEq+` accepts. Both are Munkres 29.4, and each candidate states that $\mathbb{N} \to [0, 1]$ with the product topology is not locally compact, which is false, since that space is compact. We report both label sets, and they give the same ranking of the systems. Our criterion is equivalence with the reference, in the sense that the candidate makes the same claim as the reference. The criterion is a human judgement, and it is stricter than provable equivalence, as the last case of Appendix J shows. ProofNetVerif labels whether a candidate faithfully formalizes the informal statement, so a candidate can be faithful to the text without making the same claim as the reference, e.g., when the reference states a more general result (Appendix J). We did not review the validation split, and Table 1 does not score it.

The ablation set. The 100 non-equivalent pairs of Table 2 comprise 50 ProofNetVerif test-split pairs and 50 constructed pairs. The ProofNetVerif half contains all ten pairs `BEq+` accepts on the adjudicated test split under our toolchain, and 40 pairs sampled from the remaining non-equivalent pairs. The constructed half has four classes, each the smallest form of a failure met elsewhere in the paper. In the 18 *added claim* pairs, the candidate adds to a real statement a claim that the reference does not make, e.g., a conjoined fact, an extra hypothesis, a tightened bound or a stronger conclusion. When the added claim is a library fact, both implications are provable, so an evaluator that finds both proofs can refuse the pair only by checking how they use the antecedent. Added claims are the most common defect among the 102 relabelled benchmark pairs, and the relevance audit targets them. In the 14 *neighbouring theorem* pairs, the candidate is a different true theorem about the same objects, phrased like the reference, such as Lagrange's theorem against Cauchy's. Such pairs show the first failure of §2, and the two `BEq+` acceptances of Appendix E are examples. In the 14 *unrelated true fact* pairs, the candidate is a fact provable on its own, e.g., a decidable arithmetic identity, a restatement of a library theorem, or a statement about a decoy instance, and the introduction gives an example. One of these pairs is the pair found by Simon Sorg that Poiroux et al. (2025) report in their Appendix A.6.1. `BEq+` rejects it because of the guard added for it, and `SemantEq-Eval` needs proof-term grading to reject it. In the 4 *vacuous hypotheses* pairs, the hypotheses contradict each other or contain two unrelated contradictions, like the vacuous statements that benchmark audits report, and the explosion check targets them.

Recall suite. Forty pairs: fifteen hand-written alternative statements, each realising one named transformation of its reference, and twenty-five candidates autoformalized blind from the natural-language statement, without sight of the reference, for problems from Compfiles, ProofNet# and PutnamBench, each with a hand-proved equivalence witness. Two pairs were removed on review, one as one-directional, its reference under-formalizing the problem, and the IMO 1983 pair, whose candidate, as the revision loop with the auditor in Appendix J showed, claims what the reference does not, leaving 38.

The certification set. The model was run on 511 pairs: the 471 test-split pairs labelled equivalent before adjudication that elaborate on our toolchain, and the 40 recall pairs. After adjudication these split into 407 equivalent pairs, 369 from ProofNetVerif and the 38 recall pairs, and 104 non-equivalent pairs, the 102 relabelled pairs and the two removed recall pairs.

Table 5: Effect of search strength on recall and cost over the *equivalent set*. D is the depth of chained rewrites; W_k and DW_k double k times the per-step bridge budgets and the number of intermediate statements retained at each depth. Recall is not comparable with Table 1, which does not contain the recall suite. The full grid is Table 6.

Configuration	Median (s)	Mean (s)	Max (s)	Total CPU	New	Recall
BEq+	1.0	6.3	74	0.71 h	—	60.2%
D1 (base)	2.0	30.5	489	**3.45 h**	—	**65.4%**
D1+W1	2.0	43.2	541	4.88 h	+2	65.8%
D2	2.0	50.7	606	5.73 h	+1	**66.1%**
D8+W2+DW2	2.0	271.3	3126	**30.67 h**	0	66.1%

I FULL CAPABILITY GRID

Table 5 is the excerpt of the grid that Section 3.3 discusses, and Table 6 gives every configuration.

Table 6: Effect of search strength on recall and cost over the 407 adjudicated-equivalent pairs, all configurations. D, W and DW are defined in Appendix F. "New" counts pairs accepted by the configuration and by none cheaper.

Configuration	Median (s)	Mean (s)	Max (s)	Total CPU	New	Recall
Reference point						
BEq+	1.0	6.3	74	0.71 h	—	60.2%
D1 (base)	2.0	30.5	489	**3.45 h**	—	**65.4%**
D1+W1	2.0	43.2	541	4.88 h	+2	65.8%
D2	2.0	50.7	606	5.73 h	+1	**66.1%**
D2+W1	2.0	65.0	642	7.34 h	0	66.1%
D2+W1+DW1	2.0	67.0	834	7.58 h	0	66.1%
D4	2.0	85.6	751	9.68 h	0	66.1%
D4+W1	2.0	100.2	808	11.33 h	0	66.1%
D4+W1+DW1	2.0	113.7	880	12.85 h	0	66.1%
D8+W1	2.0	148.7	1174	16.82 h	0	66.1%
D4+W2+DW2	2.0	172.2	2634	19.47 h	0	66.1%
D8+W2+DW2	2.0	271.3	3126	**30.67 h**	0	66.1%

J CASE STUDIES OF REJECTED MODEL PROOFS

The non-equivalent pairs are instructive because the model often proves them anyway, in one direction or with a proof that looks complete. The rejected proof then shows where the two statements differ. We show three, each refused by a different rule, and then an autoformalized pair whose label we changed after a revision loop with the auditor. The three campaign proofs are printed as the model wrote them, except that the candidate's theorem name, `dummy` in ProofNetVerif, is printed as `candidate`. The proofs in the last case were written in our development sessions, as described there.

Instantiating the antecedent at a foreign type. The reference formalizes Rudin's exercise 1.4, that a lower bound of a nonempty set is at most an upper bound, for a partial order. The candidate states the same shape for a linear order and an abstract element pair, which is closer to Rudin, whose ordered sets are total. The reference is the more general statement, so the two formal statements are not equivalent, although the candidate is a faithful reading of the exercise. The model proves the reference from the candidate anyway.

Lean 4

```
-- ===== assumed statement (antecedent) =====
theorem candidate_assumed {α : Type*} [LinearOrder α] {E : Set α} (hE : E.Nonempty)
    {α_bound : α} (hα : ∀ e ∈ E, α_bound ≤ e) {β_bound : α} (hβ : ∀ e ∈ E, e ≤ β_bound)
    : α_bound ≤ β_bound := by
  sorry

-- ===== goal statement, PROVED from the antecedent =====
theorem exercise_1_4
(α : Type*) [PartialOrder α]
(s : Set α)
(x y : α)
(h₀ : Set.Nonempty s)
(h₁ : x ∈ lowerBounds s)
(h₂ : y ∈ upperBounds s)
: x ≤ y := by
  classical
  obtain ⟨a, ha⟩ := h₀
  have h := candidate_assumed
    (E := {(x ≤ a : Prop)})
    (α_bound := True)
    (β_bound := (x ≤ y))
    (Set.singleton_nonempty (x ≤ a : Prop))
    (by
      intro p hp
      subst p
      exact fun _ => h₁ ha)
    (by
      intro p hp
      subst p
      exact fun hxa => hxa.trans (h₂ ha))
  exact h True.intro
```

The candidate is a theorem about an arbitrary linearly ordered set, and the proof instantiates that set at `Prop`, with the order's elements standing in for propositions. The goal statement never mentions `Prop`, but `Prop` appears in the statement that the application of the antecedent delivers. The check on type arguments (Appendix G) therefore refuses the proof, and it also flags the argument `True`, whose type `Prop` is a universe. The structure check plays no part in the refusal, because it examines only instances on the goal's own objects.

Instantiating the antecedent at a foreign object. Munkres 22.5 states that the restriction of an open map to an open subspace is an open map. The candidate states only that the image of the open set is open, which is a single instance of the map being open and says nothing about the subspace. The model proves the reference from the candidate.

Lean 4

```
-- ===== assumed statement (antecedent) =====
theorem candidate_assumed {X Y : Type*} [TopologicalSpace X] [TopologicalSpace Y] (p :
    X → Y) (hf : IsOpenMap p) (A : Set X) (hA : IsOpen A) : IsOpen (p '' A) := by
  sorry

-- ===== goal statement, PROVED from the antecedent =====
theorem exercise_22_5 {X Y : Type*} [TopologicalSpace X]
  [TopologicalSpace Y] (p : X → Y) (hp : IsOpenMap p)
  (A : Set X) (hA : IsOpen A) : IsOpenMap (p ∘ Subtype.val : A → Y) := by

  intro U hU
  simpa [Function.comp_def, Set.image_image] using
    candidate_assumed p hp (Subtype.val '' U) (hA.isOpenMap_subtype_val U hU)
```

The candidate's open-set argument is filled with `Subtype.val '' U`, where `U` is a set that exists only after the goal's own conclusion is unfolded. The proof thus applies the antecedent to an object that it builds itself, and it uses the candidate as a general schema rather than as the statement that corresponds to the reference. Both statements bind six data variables, namely the two spaces, their topologies, the map and the set, so the auditor takes the first six data variables that the proof introduces as the goal's own objects. The set `U` is the seventh, introduced by `intro` once `IsOpenMap` is unfolded. The argument `Subtype.val '' U` contains `U` and appears in the statement that the antecedent delivers, so the check on data arguments (Appendix G) refuses the proof.

Not using the antecedent at all. Artin 2.8.6 states that the centre of a product group is isomorphic to the product of the centres. The candidate states that the centre of the product, taken as a set, equals the product of the two centres as sets. An isomorphism between two groups and an equality of two sets are different claims, and the pair is not equivalent. The model proves the candidate from the reference in two lines.

Lean 4

```
-- ===== assumed statement (antecedent) =====
theorem exercise_2_8_6_assumed {G H : Type*} [Group G] [Group H] :
  Nonempty (center (G × H) ≃* (center G) × (center H)) := by
  sorry

-- ===== goal statement, PROVED from the antecedent =====
theorem candidate {G H : Type*} [Group G] [Group H] : (center (G × H) : Set (G × H)) =
    (center G : Set G) ×ˢ (center H : Set H) := by
  rcases exercise_2_8_6_assumed (G := G) (H := H) with ⟨_e⟩
  exact Set.center_prod
```

The antecedent is extracted and bound to `_e`, which never appears again, and the goal is then closed by the library theorem `Set.center_prod` alone. The first line can be deleted and the proof still compiles. The auditor refuses the proof under two rules of Appendix G, each of which suffices on its own. First, the proof destructures the antecedent and never uses what it extracts. The branch of the case split must then use the antecedent itself, and it does not. Second, `Set.center_prod` is an equation, so the grading admits it as a bridge. The proof, however, applies it on the goal derivation outside any rewrite and without the antecedent. A bridge used in such a position states the goal outright, and the auditor refuses it.

Such rejected proofs prompted most of the label review (Appendix H). The benchmark labels all three pairs above equivalent, and the model's attempts, read as evidence rather than as certificates, show where the statements of each pair differ. A person decided every label.

A label changed after a revision loop with the auditor. The auditor can also help to judge a statement, by revising a proof from the auditor's stated reasons and examining the faults that remain. IMO 1983 Problem 1 asks for all functions on the positive reals with a given functional equation and a limit condition. The Compfiles reference formalizes the domain with the subtype of positive reals, so positivity is carried by the type. The candidate was autoformalized blind from the natural-language statement, without sight of the reference. It works over the plain reals and restores positivity as a conjunct on the condition side of its biconditional. Both read as formalizations of the problem. A hand-written Lean proof establishes that the two are logically equivalent, and a strong model asked to compare the statements called them equivalent, on the ground that positivity follows from membership in the solution set. The pair therefore entered our recall suite labelled equivalent.

Lean 4

```
-- reference: positivity carried by the type ℝ+
abbrev PosReal : Type := { x : ℝ // 0 < x }

notation "ℝ+" => PosReal

def SolutionSet : Set (ℝ+ → ℝ+) := { fun x ↦ 1 / x }
theorem Imo1983P1_refs (f : ℝ+ → ℝ+) :
    f ∈ SolutionSet ↔
    ((∀ x y, f (x * f y) = y * f x) ∧
     (∀ ε, ∃ x, ∀ y, x < y → f y < ε))
-- candidate: plain ℝ, positivity spelled out as a conjunct
def Imo1983P1_solution : Set (ℝ → ℝ) := {f | ∀ x : ℝ, 0 < x → f x = 1 / x}
theorem Imo1983P1 (f : ℝ → ℝ) :
    ((∀ x : ℝ, 0 < x → 0 < f x) ∧
      (∀ x y : ℝ, 0 < x → 0 < y → f (x * f y) = y * f x) ∧
      Tendsto f atTop (nhds 0)) ↔
    f ∈ Imo1983P1_solution
```

The proofs of every prompt style were rejected in both directions, so we ran the auditor in a loop on the direction from the reference. After each rejection, an assistant language model in our development sessions wrote a new proof from the auditor's stated reasons alone. Four rounds removed every other fault that the auditor reported. One fault remained, which we judged to come from the state-

ment rather than from the proof. The candidate's biconditional has positivity on its condition side, so its direction from membership in the solution set says that every solution is positive. The claim is true, and the library proves it. The reference never states it, however, because its functions are positive by type. The two placements are different claims, which coincide here only because the particular solution set entails positivity. A constraint that one statement carries in a type is re-encoded faithfully as a hypothesis, not as a conjunct. A control supports the diagnosis. On the candidate with positivity moved into a hypothesis and nothing else changed, both directions are certified, the direction from the reference at the first attempt and the reverse after three rounds. The reverse direction meets a hypothesis of the antecedent with a positivity fact under the tolerance of Appendix G, the same kind of step as the reverse direction in Appendix D. The hand-written witness and the strong model judged whether the conjunct is true, and the auditor judged where it comes from. The case marks a boundary of our criterion, which distinguishes a condition that the reference carries in a type from the same condition stated in the candidate's conclusion. Under our labelling criterion (Appendix H), the candidate makes a claim that the reference does not make, so we relabelled the pair non-equivalent and removed it from the recall suite, and all counts in the paper reflect the ruling. The final proof of the loop and the two proofs of the variant are printed below.

Lean 4

```
-- reference ⇒ candidate, the proof after four rounds of revision (still rejected)
-- ===== assumed statement (antecedent) =====
theorem Imo1983P1_refs_assumed (f : ℝ+ → ℝ+) :
    f ∈ SolutionSet ↔
    ((∀ x y, f (x * f y) = y * f x) ∧
     (∀ ε, ∃ x, ∀ y, x < y → f y < ε)) := by
  sorry

-- ===== goal statement, PROVED from the antecedent =====
theorem Imo1983P1 (f : ℝ → ℝ) :
    ((∀ x : ℝ, 0 < x → 0 < f x) ∧
      (∀ x y : ℝ, 0 < x → 0 < y → f (x * f y) = y * f x) ∧
      Tendsto f atTop (nhds 0)) ↔
    f ∈ Imo1983P1_solution := by
  constructor
  · rintro ⟨hf_pos, hf_eq, hf_lim⟩
    let g : ℝ+ → ℝ+ := fun x => ⟨f x, hf_pos x x.property⟩
    have hiff := Imo1983P1_refs_assumed g
    have hg_eq : ∀ x y, g (x * g y) = y * g x := by
      intro x y
      apply Subtype.ext
      exact hf_eq x y x.property y.property
    have hg_tail : ∀ ε, ∃ x, ∀ y, x < y → g y < ε := by
      intro ε
      obtain ⟨N, hN⟩ := (Metric.tendsto_atTop.1 hf_lim) ε ε.property
      refine ⟨⟨max N 1, lt_of_lt_of_le zero_lt_one (le_max_right N 1)⟩, ?_⟩
      intro y hxy
      change f y < ε
      have hyN : N ≤ (y : ℝ) := le_trans (le_max_left N 1) (le_of_lt hxy)
      simpa [Real.dist_eq, abs_of_pos (hf_pos y y.property)] using hN y hyN
    have himp := hiff.mpr
    have hg_mem := himp ⟨hg_eq, hg_tail⟩
    change ∀ x : ℝ, 0 < x → f x = 1 / x
    intro x hx
    exact congrArg Subtype.val (congrFun hg_mem ⟨x, hx⟩)
  · intro hf_mem
    change ∀ x : ℝ, 0 < x → f x = 1 / x at hf_mem
    let g : ℝ+ → ℝ+ := fun x => 1 / x
    have hiff := Imo1983P1_refs_assumed g
    have hmp := hiff.mp
    have hg := hmp rfl
    refine ⟨?_, ?_, ?_⟩
    · intro x hx
      have hval : ((g ⟨x, hx⟩ : ℝ+) : ℝ) = 1 / x := rfl
      rw [hf_mem x hx, ← hval]
      exact (g ⟨x, hx⟩).2
    · intro x y hx hy
      have h := congrArg Subtype.val (hg.1 ⟨x, hx⟩ ⟨y, hy⟩)
      rw [hf_mem y hy, hf_mem x hx,
        hf_mem (x * (1 / y)) (mul_pos hx (one_div_pos.mpr hy))]
```

```
      exact h
    · apply Metric.tendsto_atTop.2
      intro ε hε
      obtain ⟨x, hx⟩ := hg.2 ⟨ε, hε⟩
      refine ⟨(x : ℝ) + 1, ?_⟩
      intro y hy
      have hy_pos : 0 < y := lt_of_lt_of_le x.property (le_trans
          (le_add_of_nonneg_right zero_le_one) hy)
      have hxy : x < (⟨y, hy_pos⟩ : ℝ+) := by
        change (x : ℝ) < y
        exact lt_of_lt_of_le (lt_add_one (x : ℝ)) hy
      have htail := hx ⟨y, hy_pos⟩ hxy
      have htail' : (1 / y : ℝ) < ε := by
        exact_mod_cast htail
      rw [hf_mem y hy_pos, Real.dist_eq, sub_zero,
        abs_of_pos (one_div_pos.mpr hy_pos)]
      exact htail'
```

Lean 4

```
-- variant candidate (positivity as a hypothesis), reference ⇒ candidate (PASS)
-- ===== assumed statement (antecedent) =====
theorem Imo1983P1_refs_assumed (f : ℝ+ → ℝ+) :
    f ∈ SolutionSet ↔
    ((∀ x y, f (x * f y) = y * f x) ∧
     (∀ ε, ∃ x, ∀ y, x < y → f y < ε)) := by
  sorry

-- ===== goal statement, PROVED from the antecedent =====
theorem Imo1983P1 (f : ℝ → ℝ) (hf_pos : ∀ x : ℝ, 0 < x → 0 < f x) :
    ((∀ x y : ℝ, 0 < x → 0 < y → f (x * f y) = y * f x) ∧
      Tendsto f atTop (nhds 0)) ↔
    f ∈ Imo1983P1_solution := by
  constructor
  · rintro ⟨hf_eq, hf_lim⟩
    let g : ℝ+ → ℝ+ := fun x => ⟨f x, hf_pos x x.property⟩
    have hiff := Imo1983P1_refs_assumed g
    have hg_eq : ∀ x y, g (x * g y) = y * g x := by
      intro x y
      apply Subtype.ext
      exact hf_eq x y x.property y.property
    have hg_tail : ∀ ε, ∃ x, ∀ y, x < y → g y < ε := by
      intro ε
      obtain ⟨N, hN⟩ := (Metric.tendsto_atTop.1 hf_lim) ε ε.property
      refine ⟨⟨max N 1, lt_of_lt_of_le zero_lt_one (le_max_right N 1)⟩, ?_⟩
      intro y hxy
      change f y < ε
      have hyN : N ≤ (y : ℝ) := le_trans (le_max_left N 1) (le_of_lt hxy)
      simpa [Real.dist_eq, abs_of_pos (hf_pos y y.property)] using hN y hyN
    have himp := hiff.mpr
    have hg_mem := himp ⟨hg_eq, hg_tail⟩
    change ∀ x : ℝ, 0 < x → f x = 1 / x
    intro x hx
    exact congrArg Subtype.val (congrFun hg_mem ⟨x, hx⟩)
  · intro hf_mem
    change ∀ x : ℝ, 0 < x → f x = 1 / x at hf_mem
    let g : ℝ+ → ℝ+ := fun x => 1 / x
    have hiff := Imo1983P1_refs_assumed g
    have hmp := hiff.mp
    have hg := hmp rfl
    refine ⟨?_, ?_⟩
    · intro x y hx hy
      have h := congrArg Subtype.val (hg.1 ⟨x, hx⟩ ⟨y, hy⟩)
      rw [hf_mem y hy, hf_mem x hx,
        hf_mem (x * (1 / y)) (mul_pos hx (one_div_pos.mpr hy))]
      exact h
    · apply Metric.tendsto_atTop.2
      intro ε hε
      obtain ⟨x, hx⟩ := hg.2 ⟨ε, hε⟩
      refine ⟨(x : ℝ) + 1, ?_⟩
      intro y hy
```

```
      have hy_pos : 0 < y := lt_of_lt_of_le x.property (le_trans
          (le_add_of_nonneg_right zero_le_one) hy)
      have hxy : x < (⟨y, hy_pos⟩ : ℝ+) := by
        change (x : ℝ) < y
        exact lt_of_lt_of_le (lt_add_one (x : ℝ)) hy
      have htail := hx ⟨y, hy_pos⟩ hxy
      have htail' : (1 / y : ℝ) < ε := by
        exact_mod_cast htail
      rw [hf_mem y hy_pos, Real.dist_eq, sub_zero,
        abs_of_pos (one_div_pos.mpr hy_pos)]
      exact htail'
```

Lean 4

```
-- variant candidate (positivity as a hypothesis), candidate ⇒ reference (PASS)
-- ===== assumed statement (antecedent) =====
theorem Imo1983P1_assumed (f : ℝ → ℝ) (hf_pos : ∀ x : ℝ, 0 < x → 0 < f x) :
    ((∀ x y : ℝ, 0 < x → 0 < y → f (x * f y) = y * f x) ∧
      Tendsto f atTop (nhds 0)) ↔
    f ∈ Imo1983P1_solution := by
  sorry
-- ===== goal statement, PROVED from the antecedent =====
theorem Imo1983P1_refs (f : ℝ+ → ℝ+) :
    f ∈ SolutionSet ↔
    ((∀ x y, f (x * f y) = y * f x) ∧
     (∀ ε, ∃ x, ∀ y, x < y → f y < ε)) := by
  let F : ℝ → ℝ := fun x => if hx : 0 < x then ↑(f ⟨x, hx⟩) else 0
  have hF_pos : ∀ x : ℝ, 0 < x → 0 < F x := by
    intro x hx
    have hFx : F x = (f ⟨x, hx⟩ : ℝ) := dif_pos hx
    rw [hFx]
    exact (f ⟨x, hx⟩).2
  have hiff := Imo1983P1_assumed F hF_pos
  have hmp := hiff.mp
  have hmpr := hiff.mpr
  constructor
  · intro hf_mem
    have hF_mem : F ∈ Imo1983P1_solution := by
      show ∀ x : ℝ, 0 < x → F x = 1 / x
      intro x hx
      have hFx : F x = (f ⟨x, hx⟩ : ℝ) := dif_pos hx
      have hval : (f ⟨x, hx⟩ : ℝ) = ((1 / (⟨x, hx⟩ : ℝ+) : ℝ+) : ℝ) :=
        congrArg Subtype.val (congrFun hf_mem ⟨x, hx⟩)
      have hdiv : ((1 / (⟨x, hx⟩ : ℝ+) : ℝ+) : ℝ) = 1 / x := by
        simp
      rw [hFx, hval, hdiv]
    have hconj := hmpr hF_mem
    refine ⟨?_, ?_⟩
    · intro x y
      have hkey := hconj.1 (x : ℝ) (y : ℝ) x.2 y.2
      have hFy : F (y : ℝ) = (f y : ℝ) := dif_pos y.2
      have hFx : F (x : ℝ) = (f x : ℝ) := dif_pos x.2
      have hFxy : F ((x : ℝ) * (f y : ℝ)) = (f (x * f y) : ℝ) :=
        dif_pos (mul_pos x.2 (f y).2)
      rw [hFy, hFx, hFxy] at hkey
      exact Subtype.ext hkey
    · intro ε
      obtain ⟨N, hN⟩ := Metric.tendsto_atTop.1 hconj.2 (ε : ℝ) ε.2
      have hx₀ : (0 < max N 1) = True := eq_true (lt_of_lt_of_le one_pos (le_max_right
          N 1))
      refine ⟨⟨max N 1, of_eq_true hx₀⟩, ?_⟩
      intro y hxy
      have hdy := hN (y : ℝ) (le_of_lt (lt_of_le_of_lt (le_max_left N 1) hxy))
      have hFy : F (y : ℝ) = (f y : ℝ) := dif_pos y.2
      have hdist : dist (F (y : ℝ)) 0 = (f y : ℝ) := by
        rw [Real.dist_eq, sub_zero, hFy, abs_of_pos (f y).2]
      rw [hdist] at hdy
      exact hdy
  · rintro ⟨hf_eq, hf_tail⟩
    have hF_eq : ∀ x y : ℝ, 0 < x → 0 < y → F (x * F y) = y * F x := by
      intro x y hx hy
```

```
    have hFy : F y = (f ⟨y, hy⟩ : ℝ) := dif_pos hy
    have hFx : F x = (f ⟨x, hx⟩ : ℝ) := dif_pos hx
    have hFxy : F (x * (f ⟨y, hy⟩ : ℝ)) = (f (⟨x, hx⟩ : ℝ+) * f ⟨y, hy⟩) : ℝ) :=
      dif_pos (mul_pos hx (f ⟨y, hy⟩).2)
    have hkey : (f (⟨x, hx⟩ : ℝ+) * f ⟨y, hy⟩) : ℝ) = (⟨y, hy⟩ : ℝ+) * f ⟨x, hx⟩ :
        ℝ) :=
      congrArg Subtype.val (hf_eq ⟨x, hx⟩ ⟨y, hy⟩)
    have hmul : (⟨y, hy⟩ : ℝ+) * f ⟨x, hx⟩ : ℝ) = y * (f ⟨x, hx⟩ : ℝ) := rfl
    rw [hFy, hFx, hFxy, hkey, hmul]
  have hF_lim : Tendsto F atTop (nhds 0) := by
    rw [Metric.tendsto_atTop]
    intro δ hδ
    obtain ⟨x, hx⟩ := hf_tail ⟨δ, hδ⟩
    refine ⟨(x : ℝ) + 1, ?_⟩
    intro y hy
    have hy_pos : 0 < y := lt_of_lt_of_le (lt_trans x.2 (lt_add_one (x : ℝ))) hy
    have hxlt : (x : ℝ) < y := lt_of_lt_of_le (lt_add_one (x : ℝ)) hy
    have hfy := hx ⟨y, hy_pos⟩ hxlt
    have hFy : F y = (f ⟨y, hy_pos⟩ : ℝ) := dif_pos hy_pos
    have hdist : dist (F y) 0 = (f ⟨y, hy_pos⟩ : ℝ) := by
      rw [Real.dist_eq, sub_zero, hFy, abs_of_pos (f ⟨y, hy_pos⟩).2]
    rw [hdist]
    exact hfy
  have hF_mem := hmp ⟨hF_eq, hF_lim⟩
  show f = fun x => 1 / x
  funext x
  have hfx : F (x : ℝ) = (f x : ℝ) := dif_pos x.2
  have hFx1 : F (x : ℝ) = 1 / (x : ℝ) := hF_mem (x : ℝ) x.2
  have hdiv : ((1 / x : ℝ+) : ℝ) = 1 / (x : ℝ) := by
    simp
  have hfinal : (f x : ℝ) = ((1 / x : ℝ+) : ℝ) := by
    rw [← hfx, hFx1, hdiv]
  exact Subtype.ext hfinal
```

K Reproducibility

Toolchain. `SemantEq-Eval` and `SemantEq-ProofAudit` run on Lean `v4.30.0-rc1` with Mathlib at revision `0c154d67`. Every statement in the paper is elaborated under this toolchain, and the elaboration filter of Appendix H refers to it.

The BEq+ port. The released implementation of `BEq+` is vendored verbatim and never edited. A copy carries exactly two toolchain-compatibility changes, each marked in the source. The first maps the retired tactic `simp_all_arith!` to its current spelling `simp_all! +arith +decide`, which is the same simplifier with the same options and the only retired tactic the method uses. The second retunes the memory containment of the per-pair Lean server for the larger current environment. Nothing about the method changes. The ported copy produces the `BEq+` numbers of our tables.

The model campaign. Proofs were written by OpenAI Codex CLI 0.151.0 with model `gpt-5.6-sol` at reasoning effort `medium`, six pairs in parallel and one session per pair and prompt style, on 30 August 2026, against Lean `v4.28.1` with Mathlib at revision `1f9fffd5`, the current release at the time. The returned proofs were then checked by `SemantEq-ProofAudit` under our toolchain, and a direction whose proof no longer elaborates there counts as a rejection. Of the 4,088 proof directions, the model left 340 unproved, and 62 no longer elaborate on our toolchain. Per style the model used between 80 and 85 million input tokens, of which 87% were cache hits, and about 1.5 million output tokens, at a wall time of about two hours.

Hardware and cost. Every run of `SemantEq-Eval` and `SemantEq-ProofAudit` was produced on a single consumer laptop CPU with no GPU. The costs of Table 6 are total CPU time over the 407 pairs, 3.45 hours for the base configuration and 30.67 hours for the strongest. Every judgement is deterministic, as described in Appendix F, so a re-run reproduces every verdict, while the timings depend on the machine.

Release. The code of both programs, the adjudicated labels of the ProofNetVerif test split with the provenance of each change, the ablation set and the recall suite, the four prompts, and every model-written proof with its verdict are released with the paper.