Lemmanaid: Neuro-Symbolic Lemma Conjecturing

Anonymous ACL submission

Abstract

Mathematicians and computer scientists are increasingly using *proof assistants* to formalize and check correctness of complex proofs. This is a non-trivial task in itself, however, with high demands on human expertise. Can we lower the bar by introducing automation for conjecturing helpful, interesting and novel lemmas? We present the first neuro-symbolic lemma conjecturing tool, LEMMANAID, designed to discover conjectures by drawing analogies between mathematical theories. LEMMANAID uses a fine-tuned LLM to generate *lemma templates* that describe the shape of a lemma, and symbolic methods to fill in the details. We compare LEMMANAID against the same LLM fine-tuned to generate complete lemma statements (a purely neural method), as well as a fully symbolic conjecturing method. LEMMANAID consistently outperforms both neural and symbolic methods on test sets from Isabelle's HOL library and from its Archive of Formal Proofs (AFP). Using DeepSeek-coder-6.7B as a backend, LEMMANAID discovers 50% (HOL) and 28% (AFP) of the gold standard reference lemmas, 8-13% more than the corresponding neural-only method. Ensembling two LEMMANAID versions with different prompting strategies further increases performance to 55% and 34% respectively. In a case study on the formalization of Octonions, LEMMANAID discovers 79% of the gold standard lemmas, compared to 62% for neural-only and 23% for the state of the art symbolic tool. Our result show that LEMMANAID is able to conjecture a significant number of interesting lemmas across a wide range of domains covering formalizations over complex concepts in both mathematics and computer science, going far beyond the basic concepts of standard benchmarks such as miniF2F, PutnamBench and ProofNet.

1 Introduction

Learning to construct new, interesting, and useful lemmas is an important and long standing challenge in AI for mathematical reasoning (Lenat, 1976; Fajtlowicz, 1988; Ireland and Bundy, 1996; Colton, 2002), yet it remains an underexplored area (Yang et al., 2025). Automatically discovered lemmas can both aid a human user working on a mathematical formalization and strengthen automated theorem provers. In this work, we examine how the generalization capabilities of LLMs can be combined with symbolic tools, to generate lemmas by a process of *analogy* (Figure 1). The LLM suggests generic *templates* describing learned families of analogous lemmas, which are then instantiated to concrete theories symbolically. Previous work has shown that such families of analogous lemmas indeed occur in proof assistant libraries (Einarsdóttir et al., 2022; Heras et al., 2013).

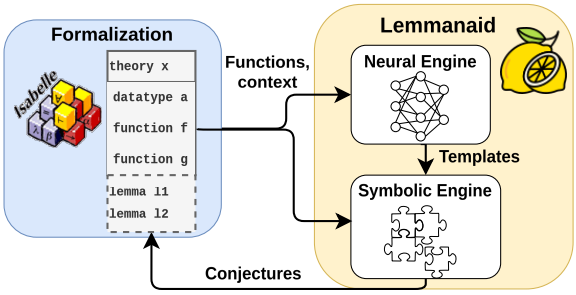

Figure 1: High-level overview of LEMMANAID.

Proof assistants like Lean (de Moura et al., 2015) and Isabelle (Nipkow et al., 2002) are becoming increasingly important among research mathematicians who want complex proofs checked for guaranteed correctness. Following the formalization of Kepler's conjecture (Hales et al., 2017), there have been several projects formalizing recent results in research mathematics (Scholtze, 2020; Commelin and Topaz, 2024; Gowers et al., 2023; Tao, 2023). However, formalizing mathematics is a non-trivial task, which in turn has sparked research in areas where large language models might be of assistance: primarily autoformalization, translating informal proofs to formal language (Wang et al., 2018; Szegedy, 2020; Wu et al., 2022), and formal proof synthesis, suggesting the next steps for the proof assistant (Polu and Sutskever, 2020; Jiang et al., 2022, 2023; First et al., 2023). Here, the LLM and proof assistant complement each other: LLMs can provide a more flexible and powerful proof search while hallucinations in proofs are caught by the proof assistant and easily discarded. We address a third challenge, *conjecturing*, with the aim of providing a

first tool, LEMMANAID, towards generic conjecturing over a broad range of mathematical theories in domains from realistic formalizations by researchers in computer science and mathematics. We move beyond the basic domains appearing in standard benchmarks for LLM-based mathematics reasoning, which have been derived from high school and undergraduate level mathematics competitions and standard textbooks, like miniF2F (Zheng et al., 2022), PutnamBench (Tsoukalas et al., 2024) and ProofNet (Azerbayev et al., 2023).

Conjecturing: Methods and Evaluation A weakness of neural conjecturing is that it sometimes generates repetitive, redundant or trivial lemmas, or hallucinates undefined symbols in the formalization (Urban and Jakubův, 2020; Rabe et al., 2021; Johansson and Smallbone, 2023). Scaling the model is a commonly attempted remedy, albeit computationally expensive and most effective in domains where background libraries have already been formalized, such as the theories needed for e.g. math olympiad problems (Zheng et al., 2022). Symbolic methods, on the other hand, can be designed and programmed to avoid repetition and redundancy. Previous symbolic tools (Smallbone et al., 2017; Einarsdóttir et al., 2021; Singher and Itzhaky, 2021) have been used to successfully discover, for example, lemmas needed in automated (co-)inductive provers (Johansson et al., 2014; Einarsdóttir et al., 2018, 2024; Kurashige et al., 2024) over (co-)recursive datatypes. However, symbolic tools are limited in the shape, size and domain of lemmas they can generate, and do not scale well to larger sets of inputs. To address these shortcomings, we propose a novel neuro-symbolic lemma conjecturing approach and tool: LEMMANAID. In LEMMANAID, an LLM is trained to generate *lemma templates* that describe the shape of a family of analogous lemmas, rather than directly generating complete lemmas (see Figure 1). Symbolic synthesis methods are then used to fill in the details. In this way, we leverage the best of both neural and symbolic methods. The LLM suggests appropriate analogous lemma-patterns likely to be relevant for the theory at hand. The symbolic engine ensures correctness and novelty, while keeping the search space manageable. As far as we are aware, this is the first work focusing on neuro-symbolic lemma conjecturing.

Conjecturing can broadly be divided into two main methodologies, both of which both have been attempted with both symbolic and neural methods (Johansson, 2019): *Top-down* methods find relevant lemmas for a specific proof attempt (Ireland and Bundy, 1996; Heras et al., 2013). This includes recent work using test time reinforcement learning, where custom curricula are produced to train a specialized neural prover (Dong and Ma, 2025; Hubert et al., 2025; Chen et al., 2025). *Bottom-up* methods, to which our work belongs, instead aim to find interesting lemmas from a set of definitions, without any pre-defined top-level goal (Lenat, 1976; McCasland et al., 2017; Urban and Jakubův, 2020; Johansson and Smallbone, 2023). Evaluation metrics are typically different for top-down and bottom-up approaches. For top-down conjecturing, increased proof success rate (of original proofs) is a common measure, while for bottom-up methods, like ours, a notion of *interestingness* of conjectures must be defined. Using human-written formalizations as a gold-standard is a common evaluation strategy (Montano-Rivas et al., 2012; Smallbone et al., 2017; Einarsdóttir et al., 2018; Urban and Jakubův, 2020). For LEMMANAID, we therefore measure the coverage of (unseen test-set) lemmas that can be discovered by LEMMANAID for (a subset of) Isabelle's HOL library[1] and from its Archive of Formal Proofs (AFP)[2].

Contributions In our experiments, we use DeepSeek-coder-1.3b and 6.7b, and Llama-3.2-1b language models. For computational resource reasons we could not evaluate larger models. We show that the LEMMANAID approach compares favorably to the results achievable using purely neural or purely symbolic conjecturing. The main contributions of our work are:

- LEMMANAID, the first neuro-symbolic lemma conjecturing approach that uses an LLM to suggest templates and a symbolic engine to instantiate templates as candidate lemmas.
- An broad evaluation of LEMMANAID on the Isabelle proof assistant's HOL and AFP libraries. This goes beyond evaluations of prior tools that have focused on specific mathematical domains.
- A comparison to existing symbolic method, QuickSpec, and neural LLM-based lemma conjecturing models we create, showing that LEMMANAID outperforms these methods and is complementary.

2 Related Work

Proof Assistants and Autoformalization Proof Assistants, such as Isabelle (Nipkow et al., 2002) and Lean (de Moura et al., 2015), are increasingly being used to check proofs in both mathematics and computer science for correctness. To do so, the user needs to *formalize* their theory, by translating it into the formal language of the proof assistant. They then interact with the system to construct a proof by stringing together calls to *tactics*, each executing and checking some part of the proof. Popular tools like Sledgehammer can automate (parts of) many proofs by selecting a suitable set of previously proved lemmas, and sending the conjecture to an automated first-order prover or SMT-solver (Blanchette et al., 2011).

Formalizing theories in proof assistants is a non-trivial task, which has sparked interest in *autoformalization*: translating definitions, theorems and lemmas

[1] https://isabelle.in.tum.de/dist/library/HOL/index.html

[2] https://www.isa-afp.org

written in natural language to the formal language of a proof assistant (Wang et al., 2018; Szegedy, 2020; Wu et al., 2022). Recent work by Zhang et al. (2025) shows that autoformalization on realistic mathematical definitions and problems, appearing on ArXiV and Wikipedia, is still a challenge for LLMs. They present a new benchmark for autoformalization of mathematical definitions, going beyond the basic definitions of e.g. miniF2F. We see LEMMANAID as an excellent complement to this kind of work: autoformalized definitions could be passed on to LEMMANAID to provide a richer initial formalization.

Proof Synthesis Even with definitions and statements formalized, constructing the required proofs from various tactics is again non-trivial, even with the help of tools like Sledgehammer. This has motivated work on various LLM-driven methods for synthesizing proof scripts for proof assistants such as Isabelle or Lean, either step by step, whole proofs at once, or via proof sketches (Polu and Sutskever, 2020; Wang et al., 2024; Ren et al., 2025; Hubert et al., 2025; Chen et al., 2025; Wang et al., 2025; Lin et al., 2025; Jiang et al., 2022; First et al., 2023; Jiang et al., 2023). Our work does not focus on producing proofs, but on suggesting suitable conjectures similar to the kind of lemmas that humans write down in libraries of formal proofs, which could in the future be combined with, and complement, proof synthesis systems.

Neural Conjecturing and Reinforcement Learning Recent top-down conjecturing methods have successfully been used in *test time reinforcement learning* as a means to generate additional data with the aim of creating a curriculum for training a neural prover (Dong and Ma, 2025; Hubert et al., 2025; Chen et al., 2025). This has contributed to the recent gold medal performances on International Math Olympiad problems (Hubert et al., 2025; Chen et al., 2025). Here, conjectures are typically variants of some given *seed statement* of interest. LEMMANAID, on the other hand, uses a bottom-up conjecturing approach and targets a different use-case. It aims to make novel suggestions directly from *definitions*, interesting to a human doing a formalization, rather than generating variants of a given statement.

Other methods combining neural conjecturing and reinforcement learning have demonstrated success in specific domains, such as inductive proofs about synthesized programs that generate integer sequences (Gauthier and Urban, 2025). The LEGO prover attempts to introduce intermediate proof statements as reusable lemmas (Wang et al., 2024) for other problems in the same domain. Poesia et al. (2024) treat conjecturing as a reinforcement learning game in simple propositional logic, arithmetic and group theory, with well formed conjectures generated neurally via constrained decoding. Tsoukalas et al. (2025) introduce a system combining symbolic rules and reinforcement learning to form conjectures in elementary number theory and on finite fields. The system is evaluated against a gold-standard set of conjectures, while simultaneously developing an interestingness function over conjectures. In contrast, LEMMANAID targets the broad range of theories from research in computer science and mathematics represented in proof assistant libraries, and aims to avoid domain specific learning by generating generic templates as an intermediate step.

Templates for Synthesizing Conjectures Following the observation that many mathematical theories share analogical lemmas of similar shapes, Buchberger et al. (2006) first proposed to use *templates* as human-provided guidance in mathematical theory exploration, by conjecturing interesting lemmas by analogy to known shapes. This has been implemented in a range of symbolic lemma conjecturing systems, including some targeting Isabelle/HOL (Montano-Rivas et al., 2012; Heras et al., 2013; McCasland et al., 2017; Einarsdóttir et al., 2021; Nagashima et al., 2023). A similar technique called *sketching* has also been applied in the domain of program synthesis (Solar-Lezama, 2009). In the above works, templates have typically been provided by the human user. Nye et al. (2019) used a neural network to propose program sketches which were then filled in by a symbolic program synthesizer, thus sharing some features with our work.

3 The LEMMANAID Approach

We have implemented a tool, LEMMANAID, for template-based conjecturing in Isabelle/HOL. The motivation comes from the observation that many lemmas in formalizations share a similar high-level structure and are analogous to each other (Buchberger et al., 2006; Einarsdóttir et al., 2022). The goal of LEMMANAID is to aid a proof assistant in identifying such analogies. We envision a user working on a new mathematical formalization; having defined some functions, types and other concepts, and perhaps a few theorems about these. This collection of definitions, which we refer to as a (partial) *theory*, serves as input to the conjecturing system. LEMMANAID then outputs conjectures that are likely to be useful in this context, allowing the user to make progress in their formalization, or better understand the behavior of the theory they have defined so far (see Figure 1). This happens in two stages: First (neural part), the partial theory is given as input to an LLM which outputs templates likely to be relevant. Second (symbolic part), LEMMANAID searches over possible instantiations of those templates in the current theory, to produce concrete conjectures.

We make all code, experimental scripts, data, and models publicly available online[3]. See Appendix B for details about computing resources and replication parameters used in our evaluation.

[3] https://anonymous.4open.science/r/Lemmanaid-ACL

3.1 LEMMANAID's Symbolic Engine

Template Language A *template* is an abstraction of a mathematical statement with concrete operators replaced by *holes*. The template thus captures the overall structure of the statement. As an example, consider the following lemmas about the Octonionic product and sum functions from the Isabelle ATP formalization about the octonions, an eight-dimensional extension of complex numbers (Koutsoukou-Argyraki, 2018):

lemma *octo_product_noncommutative:*
$\neg(\forall x\ y :: octo.(x * y = y * x))$

lemma *octo_distrib_left:*
$a * (b + c) = a * b + a * c$ `for` $a\ b\ c :: octo$

lemma *octo_assoc_plus:*
$a + (b + c) = (a + b) + c$ `for` $a\ b\ c :: octo$

If we abstract away the function symbols operating on octonions, and rename the variables according to our template abstraction method, we obtain the three templates:

$$\neg(\forall\ y_0\ y_1.\ ?H_1\ y_0\ y_1 = ?H_1\ y_1\ y_0) \quad (1)$$

$$?H_1\ x_1\ (?H_2\ x_2\ x_3) = ?H_2\ (?H_1\ x_1\ x_2)\ (?H_1\ x_1\ x_3) \quad (2)$$

$$?H_1\ x_1\ (?H_1\ x_2\ x_3) = ?H_1\ (?H_1\ x_1\ x_2)\ x_3 \quad (3)$$

Note how the function symbols $*$ and $+$ operating on octonions have been replaced by the holes $?H_1$ and $?H_2$, and the variables a, b, c, x, y standardized to y_0, y_1, x_1, x_2, x_3. Note that logical symbols, such as negation $\neg$, universal quantifier $\forall$ and equals sign $=$ remain in the template structure. Technically, templates are implemented as instances of Isabelle/HOL's term datatype[4]. Our template language represents analogous lemma statements, that have been abstracted and normalized in the following way: *Function symbols* are replaced with a hole, represented as $?H_k$ where k is a positive integer label. Note that the type of the hole is also an abstraction of the type of the original symbol, with concrete types replaced by type variables. In our examples, the holes in the templates above have types that match any function with two arguments. Each occurrence of a particular function symbol is replaced by the same hole label, even in the case of polymorphic functions where different occurrences of the symbol may have different types. *Variables* are renamed to x_k (or y_k in the case of bound variables), where k is a non-negative integer label, and their types set to match those of the corresponding holes. As mentioned, a small set of generic logic symbols have been specified to remain in the template. This is to avoid over-generalization and allow templates to act as representatives for meaningful families of analogous lemmas. See Appendix A for details about these symbols.

[4]See section 2.2 of the Isabelle/Isar Implementation Manual https://isabelle.in.tum.de/dist/Isabelle2025/doc/implementation.pdf

Template Instantiation We can *instantiate* or *fill* the holes in a template by any operator with a matching type. For instance, template 1 can be instantiated using *any* binary operator, such as subtraction on real numbers or concatenation on lists, to obtain a new conjecture. If the operator indeed is non-commutative, we obtain a correct lemma analogous to the lemma *octo_product_noncommutative*. In this way, templates are generalizations of lemma statements: each template matches many different lemmas that may apply in different theories.

LEMMANAID's symbolic engine takes as input a template and a list of relevant function names from the current theory. It then searches over the possible instantiations of the holes in the template using the given functions (note that unlike holes the variables are fixed at instantiation, and not replaced by any further terms). Each indexed hole is replaced by one of the given function symbols while ensuring that the resulting candidate lemma is well typed. This may lead to a number of different candidate lemmas being produced. While using templates restricts the search, we still also set a time-out to deal with rare cases where the template contains very many holes or we have a large number of potential functions to instantiate them with. We also experimented with using an LLM to also instantiate the templates, but found it performed worse than the symbolic engine as well as incurring a greater computational cost (see Appendix C).

Revisiting our example, consider again template 3 (associativity). Suppose we give this template to LEMMANAID's symbolic engine, along with the functions $+$ (addition), $-$ (subtraction), sin, cos and $\hat{}$ (exponentiation) for real numbers, and the functions len (the length of a list), rev (reversing a list) and @ (concatenate two lists). In the hole-filling step, it will come up with the following candidates:

$$(x_1 + x_2) + x_3 = x_1 + (x_2 + x_3)$$
$$(x_1\ @\ x_2)\ @\ x_3 = x_1\ @\ (x_2\ @\ x_3)$$
$$(x_1 - x_2) - x_3 = x_1 - (x_2 - x_3)$$
$$(x_1^{x_2})^{x_3} = x_1^{x_2^{x_3}}$$

Note how sin, cos, len, and rev cannot be used to fill the hole $?H_1$, since $?H_1$ is applied to two arguments and therefore requires a binary operator. Also note that only the instantiations with $+$ and @ result in valid lemmas, while the other two conjectures are false. The user can for instance apply Isabelle's counterexample checker to easily identify these.

3.2 LEMMANAID's Neural Engine

LEMMANAID's neural engine uses a fine-tuned LLM to suggest templates given a theory context, which are then passed to LEMMANAID's symbolic engine (§3.1). For fine-tuning, we create labeled training data (input-output pairs) following §3.1: we extract the template of each lemma appearing in a corpus of human-written

theory files, and use a string representation of the template as the desired output. For each output template, its corresponding input is the list of symbols appearing in the body of the original lemma, as well as contextual information about the symbols. For a given symbol, both its type and any known associated definition convey information about how the symbol can be used. Although definitions provide more complete descriptions of symbols, types are more succinct and they often provide sufficient information to form well-typed templates and lemmas. Section 4.2 evaluates the effectiveness of types and definitions as inputs. Revisiting the examples about octonions, the lemma *octo_distrib_left* would result in the data-point of the shape:

Input: [*Symbols:* *, +] [*Types of* *, +] [*Defs of* *, +];
Output: [$?H_1\ x_1\ (?H_2\ x_2\ x_3) =$
$?H_2\ (?H_1\ x_1\ x_2)\ (?H_1\ x_1\ x_3)$]

To obtain the list of symbols present in a given lemma, we first construct a theory file with low-level ML code to interact with Isabelle/Isar top-level and retrieve the lemma. To extract the symbols, we recursively iterate over applications in the term storing any encountered constants, using Isabelle-client (Shminke, 2022). The list of symbols and their information is the minimal context necessary to recover a given lemma. However, additional context, such as relevant existing lemmas, may further guide the conjecturing process and is an interesting direction for future work.

3.3 Direct Neural Conjecturing

While LEMMANAID employs a neural engine to generate templates and a symbolic engine to instantiate those templates into lemmas, there is another approach one can take, which is to use a neural engine to directly generate lemmas. To create such a baseline, we adapt LEMMANAID's neural engine, by instead fine-tuning it on tuples of the form (symbols, context, lemma) instead of a template. Revisiting the running example again, the lemma *octo_distrib_left* now results in a data-point of the shape:
*Input: [Symbols: *, +] [Types of *, +] [Defs of *, +] ;*
Output: [$(a * (b + c) = a * b + a * c$*]*

Two possible string representations of the lemma in the output are: (1) the *lemma command* as appearing in the source code of the theory file (what a human would write), and (2) a string representation of the internal *lemma object* within the proving engine. We noticed early in our experimentation that an LLM is almost never able to predict the lemma command as it is less structured than the lemma object, and so we use the lemma object representation. While there is some novelty in predicting lemma objects directly as opposed to lemma commands, the main purpose of doing direct neural lemma conjecturing is to create a suitable baseline for our experiments.

3.4 Dataset

We train and evaluate LEMMANAID on mathematical libraries from the Isabelle proof assistant. First, the Isabelle/HOL library, which contains formalizations based on higher order logic for a range of mathematics (e.g. number theory, analysis, algebra, set theory). Second, we also use the Isabelle Archive of Formal Proofs (AFP), which is a large collection of formalizations from the research community including mathematics, computer science and logic. At the time of writing, the AFP includes about 285,200 lemmas in close to 900 theories. We extract templates for the lemmas in the above libraries as described in §3.1. In total, this results in a dataset of 62,816 data-points from the HOL library and 210,898 data-points from the AFP[5].

4 Evaluation

We set out to answer the following research questions:

RQ1 How does our neuro-symbolic approach, LEMMANAID, compare with neural approaches and existing symbolic approaches?

RQ2 How does different contextual information (such as types and definitions) impact performance?

RQ3 What characterizes the conjectures LEMMANAID is able to generate (how many can be proved/have counter examples)?

4.1 Experimental Setup

Baselines There are no existing neural-based tools or models for Isabelle to compare against so we train our own neural baselines (Section 3.3), to suggest lemma objects. Models trained to predict the lemma object are denoted by "neural" in tables in the evaluation. The state-of-the-art symbolic tool for lemma conjecturing is QuickSpec (Smallbone et al., 2017). QuickSpec is limited in that it can only generate equational lemmas as its conjecturing algorithm is based around enumerative synthesis of terms and the construction of equivalence classes via automated testing.

Benchmarks We derive our datasets from the Isabelle/HOL library and the Archive of Formal Proofs (recall §3.4) and create multiple train/validation/test sets from these libraries. We create a file-wise split of the HOL library so that we may evaluate the in-distribution capabilities of LLM-based approaches for lemma conjecturing tasks. The training, validation and test sets are called HOL-train (57,576 datapoints), HOL-val (500 datapoints), and HOL-test (4,740 datapoints), respectively.

Next, we supplement HOL-train and HOL-val with all projects from the AFP2024 that are published prior to 2024 to create HOL+AFP-train and HOL+AFP-val. We then create a new test set called AFP-test comprised of 27 AFP projects published in 2024 (and thus disjoint from those in HOL+AFP-train and HOL+AFP-val). More information about results on the different projects

[5] Some theories could not be processed by the batch method we used, due to technical issues with theory imports. Hence, our dataset does not cover everything from the AFP.

in AFP-test can be found in E.2. Training models on HOL+AFP-train and evaluating on HOL-test allows us to understand the effect of more training data as opposed to training on only HOL-train. On the other hand, training models on HOL+AFP-train and evaluating on AFP-test allows us to evaluate an out-of-distribution task (as AFP projects tend to differ greatly in topic, content, and style). Importantly, evaluating on AFP-test allows us to mitigate the risk of test leakage as the models we use have publicly reported pretraining cutoff dates in 2023, and all projects in AFP-test are published in 2024 and after. AFP-test consists of 18,975 datapoints. Note that risk for test-data leakage is bigger for the purely neural baseline, as LEMMANAID's neural model produces our custom template format which is not explicitly present in the original data source.

We separately test on the Octonions project (Koutsoukou-Argyraki, 2018) from the AFP, which contains 350 lemmas, and leave it out of all training sets in order to compare LEMMANAID, direct neural lemma conjecturing models, and symbolic tool, QuickSpec. We choose the Octonion theory for this comparison as it consists of equational lemmas, which is the domain QuickSpec supports. Recall that QuickSpec can only conjecture equational lemmas on computable functions which limits its applicability to a smaller subset of Isabelle theories.

We enumerate the lemmas appearing in a given project by compiling and processing a theory and counting the number of lemma objects that exist. We do this by using the `Find_Theorems` tool in Isabelle which retrieves theorems from a proof context, and we only keep the theorems defined in the active theory.

Metrics In each evaluation task, there is one *gold-standard lemma*, and the method being assessed generates a set of suggested lemmas. We define *lemma success rate* as the percentage of these tasks for which the given method is able to successfully generate (as part of the set of lemmas it generates) the gold-standard lemma (where we compare lemmas syntactically). This overall metric measures the performance of a method *end-to-end*. Note that the purely neural model and LEMMANAID are given the same neural computational budget per task. However, LEMMANAID's symbolic engine may generate several instantiations from the same template (this is however computationally cheap in comparison to the neural inference), and we consider the instantiation task to be successful if one of the generated instantiations matches the gold-standard lemma (syntactically).

Models and Inference We evaluate three instances of LEMMANAID using DeepSeek-coder-1.3B, DeepSeek-coder-6.7B and Llama-3.2-1B as the pretrained model that we fine-tune. These models are selected as representatives of open-weight base models performant on code-related tasks. One benefit of using smaller models in LEMMANAID is that it supports a more realistic setup for actual users wanting to use a conjecturing tool and run it locally on their machine. See Appendix B for some details about computing resources and replication parameters used in our evaluation. At inference time, we use greedy decoding to obtain a template prediction from LEMMANAID's LLM. We also use beam search with beam size equal to 4. For our RQ1 evaluation, we use both decoding strategies for all neural models, including neural baselines, and thus all methods with neural models have the same LLM inference budget of 5.

Checking Outputs To check the correctness of a suggested template, we use the exact match (string similarity) of the predicted template and the abstracted template of the *gold-standard lemma* we want to recover. This is sufficient for templates given the way in which we define the template language. Exact match is not sufficient for comparing lemma objects because it requires identical variable names and therefore does not account for alpha equivalence of the terms. We want to count a generated lemma as matching the gold-standard even if it uses different variable names. We parse the gold-standard lemma object and the predicted lemma object in the context of a given theory, as some symbols are only defined in that context. We perform term comparison, where we traverse the tree, checking for equivalences on left and right, respectively. Since alpha renaming is already implemented in Isabelle, we use this for variable comparison. Note that this cannot be done easily outside of Isabelle because we must know a given variable's scope. Since instantiation of a predicted template via LEMMANAID's symbolic engine produces multiple candidate lemmas, we iterate over them and check each against the gold-standard lemma. LEMMANAID's symbolic engine's instantiation timeout is set to 60s.

4.2 RQ1–2: Discovery of Gold Standard Lemmas

To evaluate LEMMANAID and compare it against neural-only baselines and symbolic methods, we train LEMMANAID's neural model and the baseline models on either HOL-train or HOL+AFP-train, obtain multiple variants, and evaluate them across different test sets (Table 1). For each model, we compare the results obtained when various contextual information is included in the input: definitions, type information, or both (as described in 3.2). Note that adding only the definitions, but not types, led to worse performance in initial experiments with DeepSeek 1.3B, so was not attempted for the other LLMs. For each model, we get 5 predictions: 1 using greedy decoding, and 4 using beam search (with beam size of 4). We break down the results for each decoding method in Appendix D.

We see that LEMMANAID outperforms the respective neural baselines on all test sets, with the best model (using DeepSeek 6.7B and HOL+AFP training) reaching 50% on HOL-test and 28.2% on AFP-test. For the smaller 1B models, the DeepSeek backend appears to consistently outperform Llama-3.2. We also see that

Table 1: Percentage of gold-standard lemmas discovered for LEMMANAID and a neural only version. We include results for the symbolic QuickSpec tool only on Octonions which is in its scope. Results are for 5 predictions (greedy decoding & beam search with beam size 4). Note that due to limited computational resources and poor results using DeepSeek 1.3B, we did not further explore the prompting technique with only definitions for the other models.

	HOL-train			HOL+AFP-train		
	HOL-test	AFP-Test	Octonions	HOL-test	AFP-Test	Octonions
DeepSeek-coder-1.3B						
LEMMANAID (types + defs)	**39.3%**	**19.8%**	57.4%	**41.7%**	**27.7%**	**66.9%**
LEMMANAID (types)	37.4%	19.4%	**58.3%**	35.5%	27.1%	55.7%
LEMMANAID (defs)	33.0%	7.1%	54.0%	35.4%	12.8%	61.4%
Neural (types + defs)	29.3%	9.6%	33.7%	30.0%	13.5%	44.0%
Neural (types)	32.3%	10.3%	42.9%	29.0%	12.6%	47.1%
Neural (defs)	29.5%	5.4%	41.4%	25.4%	6.9%	44.0%
LEMMANAID Combined	47.6%	25.9%	67.7%	49.6%	35.3%	75.7%
Neural Combined	41.1%	14.8%	53.1%	38.3%	18.6%	58.0%
Combined	52.6%	27.7%	70.9%	52.7%	37.2%	79.1%
DeepSeek-coder-6.7B						
LEMMANAID (types + defs)	**46.1%**	**25.9%**	**68.0%**	**50.0%**	27.4%	76.6%
LEMMANAID (types)	43.0%	22.1%	66.9%	47.5%	**28.2%**	**79.1%**
Neural (types + defs)	40.9%	16.8%	52.6%	28.9%	20.0%	61.7%
Neural (types)	37.3%	14.7%	51.4%	36.8%	19.0%	52.9%
LEMMANAID Combined	51.7%	28.9%	76.0%	54.9%	34.2%	83.1%
Neural Combined	45.3%	20.1%	58.9%	42.5%	25.3%	65.1%
Combined	56.4%	31.2%	79.4%	58.0%	39.1%	84.6%
Llama-3.2-1B						
LEMMANAID (types+defs)	31.5%	18.0%	57.4%	28.5%	21.5%	46.9%
LEMMANAID (types)	**33.9%**	**18.6%**	**59.7%**	**34.7%**	**23.4%**	**60.6%**
Neural (types+defs)	22.1%	7.3%	33.1%	18.4%	10.1%	54.6%
Neural (types)	22.2%	8.0%	41.4%	16.2%	9.8%	57.4%
LEMMANAID Combined	39.1%	20.7%	62.6%	38.3%	29.9%	65.4%
Neural Combined	28.0%	8.7%	46.9%	23.1%	12.3%	38.3%
Combined	42.4%	21.8%	63.4%	40.8%	31.5%	66.6%
QuickSpec	—	—	22.8%	—	—	22.8%

neural methods are somewhat complementary to LEMMANAID: taken as an ensemble they discover even more lemmas, 58% on HOL-test and 39.1% on AFP-test (using DeepSeek 6.7B and HOL+AFP training). The inclusion of type information is greatly beneficial to both LEMMANAID and the neural baseline method. We see that in some cases the success rate is higher when only type information is included and definitions are excluded, while in others including both is beneficial, although the differences are relatively small. We see that on AFP-test, the performance drops for all variants compared to their results on HOL-test. This is unsurprising, as the lemmas in AFP projects are more diverse than those in HOL. Also, the input to the LLMs may not always include enough contextual information for retrieving definitions in AFP theories requiring long-distance dependencies, though we always account for HOL as a dependency.

For Octonions, LEMMANAID models (best reaching 79.1%) outperform their respective neural baselines (61.7%) while both greatly outperform QuickSpec (reaching just 22.8%). Similar to other datasets, an ensemble of LEMMANAID and neural models provides a significant improvement, with overall lemma success rate up to 84.6%. Not shown in Table 1 is that QuickSpec generates several thousand lemmas, giving it an extremely poor precision of less than 1%. This is because it has limited heuristics for judging which lemmas are interesting, and only skips lemmas that are logical consequences of existing lemmas. It also skips several useful and simple lemmas, such as $inner\ e_1\ x = Im_1\ x$, as they are considered trivial consequences of other lemmas. This is further discussed in Appendix H.

4.3 RQ3: Analysis of Generated Conjectures

In §4.2, we counted the number of gold standard lemmas discovered. Here, we break down these results further.

Applicability to Realistic Formalizations Appendix E shows a breakdown per projects/theories in AFP-test and HOL-test, with the number of test lemmas from each and the success rate of the best-performing LEMMANAID and neural ensembles with DeepSeek 6.7B. We note that LEMMANAID performs better on almost every test theory in the test sets. We also list the topic of the different AFP formalizations in our test set, showing that they range over a variety of different topics from Computer Science, Mathematics and Logic. The results confirm that LEMMANAID's approach to conjecturing by analogy successfully produces interesting lemmas across a wide range of real world formalizations.

Generated Conjectures We take a closer look at the full set of conjectures generated by the best-performing instance of LEMMANAID (DeepSeek 6.7B and HOL+AFP training), under greedy decoding. We separate the generated conjectures into three main categories (true, false and open) using the Sledgehammer tool for automated proofs (Blanchette et al., 2011), and counter-example finding tools Nitpick (Blanchette and Nipkow, 2010) and Quickcheck (Bulwahn, 2012). *True conjectures* consist of 1) the gold-standard lemmas, and 2) other conjectures proved by Sledgehammer. *False conjectures* are those for which we can find a counter-examples. *Open conjectures* are those that can neither be proved by Sledgehammer, nor have a counter-example.

For HOL-test (containing 4,740 gold-standard lemmas), LEMMANAID generates 10,314 conjectures, however 5040 of these comes from a degenerate case[6]. Removing these, applying deduplication (removing 164 duplicates) leaves 5,110 conjectures for which Table 2 shows a breakdown: 1043 are novel true statements which don't exist in the target theories, and 1256 are open conjectures which require non-trivial reasoning. Appendix F contains some examples of these, and Appendix G additional statistics.

Table 2: Generated conjectures for HOL-test.

Category	Size
Gold-standard	1721 (33.7%)
Other True Statements	1043 (20.4%)
False Conjectures	1090 (21.3%)
Open Conjectures	1256 (24.6%)
Total	5110

5 Conclusion

LEMMANAID is a novel neuro-symbolic bottom-up conjecturing tool for the Isabelle proof assistant. LEMMANAID works by (neurally) learning analogical families of conjectures, represented as templates which are symbolically instantiated in new theories. LEMMANAID outperforms both neural baselines and a symbolic tool, QuickSpec, on test sets from Isabelle's HOL library and from its Archive of Formal Proofs, discovering 50% of gold standard lemmas for HOL-test and 28% for AFP-test, using deepseek-6.7b as backend. Prompting strategies including both type information and definitions seem to work slightly better for the deepseek backends, while only including type information was favorable for Llama-3.2, although differences are small. Ensembling the different strategies further improves results, showing that the different approaches, including the neural baseline, discovers slightly different conjectures.

This work is a first step toward demonstrating the usability of neuro-symbolic analogical conjecturing for proof assistants on a realistic selection of theories from real formalizations in computer science, mathematics and logic, going beyond benchmark theories from high-school and undergraduate mathematics competitions. In the future, LEMMANAID could be integrated in a workflow supporting for example auto-formalization, especially complementing works on auto-formalization of mathematical definitions.

[6]These come from a case of a single template consisting of 7 OR statements resulting in 5040 equivalent conjectures.

Limitations

Due to limitations in computational resources, we have evaluated the Lemmanaid framework with smaller LLM backends, ranging between 1B to 6.7B, for two different model families, DeepSeek and Llama 3. Each instance of Lemmanaid requires fine-tuning both the template-generation model, and a direct neural conjecturing model. This in turn, was done for two (or three) different prompting strategies (see e.g. table 1), including either both type and function definitions, or only one of them. Further prompting experiments, and wider beam searches were beyond the scope of our computational budget. Furthermore, for the largest model (Deepseek-coder-6.7b) we suspect additional computational resources might have revealed slightly better hyper-parameter settings, which with additional fine-tuning could have improved those results further.

We also conducted some brief preliminary experiments to compare with commercial LLMs, which however was difficult to do in a controlled and fair setting to compare. We found that the commercial models tended to either expose suspected test-data leakage, reproducing the target Isabelle-formalization almost verbatim, or didn't succeed at all, in line with results in (Johansson and Smallbone, 2023).

Finally, as highlighted in §4.3, LEMMANAID's symbolic engine could be further improved to prune the number of symmetric instantiations, leading to duplicates and equivalent conjectures. This happens especially in the presence of templates with several logic combinators (or, and). In addition, also described in in §4.3, the conjectures generated by LEMMANAID has only been attempted to be proved by the fully automated tactic Sledgehammer. While powerful, any conjecture requiring a more specialized tactic-based proof remain open. In this paper, our focus was a comparison across a wide range of formalizations against their gold-standard lemmas, why we leave deeper analysis and interactive proof attempts of open conjectures in each theory, as future work.

References

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

Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. 2011. Extending sledgehammer with smt solvers. *Journal of Automated Reasoning*, 51:109 – 128.

Jasmin Christian Blanchette and Tobias Nipkow. 2010. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *International conference on interactive theorem proving*, pages 131–146. Springer.

Bruno Buchberger, Adrian Craciun, Tudor Jebelean, Laura Kovács, Temur Kutsia, Koji Nakagawa, Florina Piroi, Nikolaj Popov, Judit Robu, Markus Rosenkranz, and Wolfgang Windsteiger. 2006. Theorema: Towards computer-aided mathematical theory exploration. *Journal of Applied Logic*, 4:470–504.

Lukas Bulwahn. 2012. The new quickcheck for isabelle: Random, exhaustive and symbolic testing under one roof. In *International Conference on Certified Programs and Proofs*, pages 92–108. Springer.

Luoxin Chen, Jinming Gu, Liankai Huang, Wenhao Huang, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Kaijing Ma, Cheng Ren, Jiawei Shen, Wenlei Shi, Tong Sun, He Sun, Jiahui Wang, Siran Wang, Zhihong Wang, Chenrui Wei, and 17 others. 2025. Seed-prover: Deep and broad reasoning for automated theorem proving. *Preprint*, arXiv:2507.23726.

Simon Colton. 2002. The HR program for theorem generation. In *Automated Deduction—CADE-18*, pages 285–289, Berlin, Heidelberg. Springer Berlin Heidelberg.

Johan Commelin and Adam Topaz. 2024. Abstraction boundaries and spec driven development in pure mathematics. *Bull. Amer. Math. Soc.*, 61.

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The lean theorem prover (system description). In *Automated Deduction - CADE-25*, pages 378–388, Cham. Springer International Publishing.

Kefan Dong and Tengyu Ma. 2025. STP: Self-play LLM theorem provers with iterative conjecturing and proving. In *Forty-second International Conference on Machine Learning*.

Sólrún Halla Einarsdóttir, Márton Hajdu, Moa Johansson, Nicholas Smallbone, and Martin Suda. 2024. Lemma discovery and strategies for automated induction. In *Automated Reasoning*, pages 214–232, Cham. Springer Nature Switzerland.

Sólrún Halla Einarsdóttir, Moa Johansson, and Johannes Åman Pohjola. 2018. Into the infinite - theory exploration for coinduction. In *Proceedings of AISC 2018*, pages 70–86.

Sólrún Halla Einarsdóttir, Moa Johansson, and Nicholas Smallbone. 2022. Lol: A library of lemma templates for data-driven conjecturing. In *Work-in-progress papers presented at the 15th Conference on Intelligent Computer Mathematics (CICM 2022) Informal Proceedings*, page 22.

Sólrún Halla Einarsdóttir, Nicholas Smallbone, and Moa Johansson. 2021. Template-based theory exploration: Discovering properties of functional programs by testing. In *Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages*, IFL '20, page 67–78, New York, NY, USA. Association for Computing Machinery.

S. Fajtlowicz. 1988. On conjectures of graffiti. *Discrete Math.*, 72(13):113–118.

Emily First, Markus Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-Proof Generation and Repair with Large Language Models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 1229–1241, New York, NY, USA. Association for Computing Machinery.

Thibault Gauthier and Josef Urban. 2025. Learning conjecturing from scratch. In *Automated Deduction - CADE-30*.

W. T. Gowers, Ben Green, Freddie Manners, and Terence Tao. 2023. On a conjecture of marton. *Preprint*, arXiv:2311.05762.

Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dand, John Harrison, Le Troung Hoan, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Tat Thanf Nguyen, and et al. 2017. A formal proof of the Kepler conjecture. *Forum of Mathematics, Pi*, 5:e2.

Jonathan Heras, Ekaterina Komendantskaya, Moa Johansson, and Ewen Maclean. 2013. Proof-pattern recognition and lemma discovery in ACL2. In *Proceedings of LPAR*.

T. Hubert, R. Mehta, and L. et al. Sartran. 2025. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*.

Andrew Ireland and Alan Bundy. 1996. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16:79–111.

Albert Qiaochu Jiang, Wenda Li, Szymon Tworkowski, Konrad Czechowski, Tomasz Odrzygóźdź, Piotr Miłoś, Yuhuai Wu, and Mateja Jamnik. 2022. Thor: Wielding hammers to integrate language models and automated theorem provers. In *Advances in Neural Information Processing Systems*.

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

Moa Johansson. 2019. Lemma discovery for induction. In *Intelligent Computer Mathematics*, pages 125–139, Cham. Springer International Publishing.

Moa Johansson, Dan Rosén, Nicholas Smallbone, and Koen Claessen. 2014. Hipster: Integrating theory exploration in a proof assistant. In *Proceedings of CICM*, pages 108–122. Springer.

Moa Johansson and Nicholas Smallbone. 2023. Exploring mathematical conjecturing with large language models. In *17th International Workshop on Neural-Symbolic Learning and Reasoning, NeSy 2023*.

Angeliki Koutsoukou-Argyraki. 2018. Octonions. *Archive of Formal Proofs*. https://isa-afp.org/entries/Octonions.html, Formal proof development.

Cole Kurashige, Ruyi Ji, Aditya Giridharan, Mark Barbone, Daniel Noor, Shachar Itzhaky, Ranjit Jhala, and Nadia Polikarpova. 2024. Cclemma: E-graph guided lemma discovery for inductive equational proofs. *Proc. ACM Program. Lang.*, 8(ICFP).

Douglas B. Lenat. 1976. *AM, an artificial intelligence approach to discovery in mathematics as heuristic search*. Ph.D. thesis, Stanford University.

Yong Lin, Shange Tang, Bohan Lyu, Jiayun Wu, Hongzhou Lin, Kaiyu Yang, Jia Li, Mengzhou Xia, Danqi Chen, Sanjeev Arora, and Chi Jin. 2025. Goedel-prover: A frontier model for open-source automated theorem proving. *Preprint*, arXiv:2502.07640.

Daniel Matichuk, Toby Murray, and Makarius Wenzel. 2016. Eisbach: A proof method language for isabelle. *J. Autom. Reason.*, 56(3):261–282.

R. L. McCasland, A. Bundy, and P. F. Smith. 2017. MATHsAiD: Automated mathematical theory exploration. *Applied Intelligence*.

Omar Montano-Rivas, Roy McCasland, Lucas Dixon, and Alan Bundy. 2012. Scheme-based theorem discovery and concept invention. *Expert systems with applications*, 39(2):1637–1646.

Yutaka Nagashima, Zijin Xu, Ningli Wang, Daniel Sebastian Goc, and James Bang. 2023. Template-based conjecturing for automated induction in isabelle/hol. In *Fundamentals of Software Engineering*, pages 112–125, Cham. Springer Nature Switzerland.

Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, Berlin, Heidelberg.

Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. 2019. Learning to infer program sketches. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *PMLR*, pages 4861–4870.

Gabriel Poesia, David Broman, Nick Haber, and Noah Goodman. 2024. Learning formal mathematics from intrinsic motivation. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

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

Markus N. Rabe, Dennis Lee, Kshitij Bansal, and Christian Szegedy. 2021. Mathematical reasoning via self-supervised skip-tree training. In *Proceedings of ICLR*.

Z. Z. Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, Z. F. Wu, Zhibin Gou, Shirong Ma, Hongxuan Tang, Yuxuan Liu, Wenjun Gao, Daya Guo, and Chong Ruan. 2025. Deepseek-prover-v2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. *Preprint*, arXiv:2504.21801.

Peter Scholtze. 2020. Liquid tensor experiment.

Boris Shminke. 2022. Python client for isabelle server. *Preprint*, arXiv:2212.11173.

Eytan Singher and Shachar Itzhaky. 2021. Theory exploration powered by deductive synthesis. In *Computer Aided Verification*, pages 125–148, Cham. Springer International Publishing.

Nicholas Smallbone, Moa Johansson, Koen Claessen, and Maximilian Algehed. 2017. Quick specifications for the busy programmer. *Journal of Functional Programming*, 27.

Armando Solar-Lezama. 2009. The sketching approach to program synthesis. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, APLAS '09, page 4–13, Berlin, Heidelberg. Springer-Verlag.

Christian Szegedy. 2020. A promising path towards autoformalization and general artificial intelligence. In *Intelligent Computer Mathematics*, pages 3–20, Cham. Springer International Publishing.

Terrence Tao. 2023. Formalizing the proof of pfr in lean4 using blueprint: a short tour.

George Tsoukalas, Jasper Lee, John Jennings, Jimmy Xin, Michelle Ding, Michael Jennings, Amitayush Thakur, and Swarat Chaudhuri. 2024. Putnambench: Evaluating neural theorem-provers on the putnam mathematical competition. *Preprint*, arXiv:2407.11214.

George Tsoukalas, Rahul Saha, Amitayush Thakur, Sabrina Reguyal, and Swarat Chaudhuri. 2025. Learning interestingness in automated mathematical theory formation. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Josef Urban and Jan Jakubův. 2020. First neural conjecturing datasets and experiments. In *Proceedings of CICM*.

Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, Jianqiao Lu, Hugues de Saxcé, Bolton Bailey, Chendong Song, Chenjun Xiao, Dehao Zhang, Ebony Zhang, Frederick Pu, Han Zhu, and 21 others. 2025. Kimina-prover preview: Towards large formal reasoning models with reinforcement learning. *Preprint*, arXiv:2504.11354.

Haiming Wang, Huajian Xin, Chuanyang Zheng, Zhengying Liu, Qingxing Cao, Yinya Huang, Jing Xiong, Han Shi, Enze Xie, Jian Yin, Zhenguo Li, and Xiaodan Liang. 2024. LEGO-prover: Neural theorem proving with growing libraries. In *The Twelfth International Conference on Learning Representations*.

Qingxiang Wang, Cezary Kaliszyk, and Josef Urban. 2018. First experiments with neural translation of informal to formal mathematics. In *Intelligent Computer Mathematics*, pages 255–270, Cham. Springer International Publishing.

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

Kaiyu Yang, Gabriel Poesia, Jingxuan He, Wenda Li, Kristin E. Lauter, Swarat Chaudhuri, and Dawn Song. 2025. Position: Formal mathematical reasoning—a new frontier in AI. In *Forty-second International Conference on Machine Learning Position Paper Track*.

Lan Zhang, Marco Valentino, and Andre Freitas. 2025. Autoformalization in the wild: Assessing LLMs on real-world mathematical definitions. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 1720–1738, Suzhou, China. Association for Computational Linguistics.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. 2022. minif2f: a cross-system benchmark for formal olympiad-level mathematics. In *International Conference on Learning Representations*.

A Template language and symbols

Templates are implemented as instances of Isabelle/HOL's term datatype[7]. While theory-specific symbols are replaced by holes, general logical symbols remain in the template to not make it overly abstract and obscuring the analogies we wish to uncover. The symbols that are part of the template language and not abstracted away are:

- All constants whose names begin with 'HOL.' These are the functions defined in Isabelle/src/HOL/HOL.thy including equality, True, False, Not, All and Ex (quantifiers), conjunction and disjunction.
- All constants whose names begin with 'Pure.' Basic logical constructs including implication (Pure.imp), Pure.all, Pure.eq, defined in Isabelle/src/Pure/logic.ML
- Bounded quantifiers for sets (these are rendered the same as the regular quantifiers defined in HOL).
- Set membership.

[7] See section 2.2 of the Isabelle/Isar Implementation Manual https://isabelle.in.tum.de/dist/Isabelle2025/doc/implementation.pdf

- Pairs/Cartesian products as defined in src/HOL/Product_Type.

- Inequality symbols: less, greater, less or equal, greater or equal Defined in "Orderings.ord_class.less_eq" and "Orderings.ord_class.less" (greater and greater_eq are defined in terms of less and less_eq and are translated in the term structure so we don't expect them to appear in templates).

Implementation details of the exact symbols and how to extract them are available in our anonymous code repository.

B Computing Resources and Details for Replication

We run the majority of our training and evaluation on 8 NVIDIA RTX 2080 Ti (11GB VRAM). The Deepseek 6.7B models are trained and evaluated on 4 NVIDIA Tesla A40s, each with 48GB of VRAM. We use 8-bit quantization when loading models for both fine-tuning and inference. We train models with DistributedData-Parallel (DDP) for 12 epochs and $8e-4$ learning rate. We use a maximum sequence length of 1024 with 200 tokens reserved for the output. We use an effective batch size 16.

C Template instantiation by an LLM vs LEMMANAID's symbolic engine

As discussed in Section 3.1, LEMMANAID instantiates its predicted templates symbolically. As an alternative approach, we also experimented with trining an LLM (DeepSeek-coder-1.3b) to instantiate templates. We define the *instantiation rate* as measuring the percentage of instantiation tasks that the respective methods can perform successfully.

We observe that LEMMANAID's symbolic engine has a much greater instantiation rate (89.1%) than that of the LLM (66.9%), see Table 3. However, interestingly, the LLM recovers some lemmas that we symbolically fail to recover, so when we consider lemmas recovered either symbolically or neurally, the rate is 92.2%.

Table 3: Instantiation rate for LLM (fine-tuned on HOL-train) and LEMMANAID for gold-standard templates (HOL-test).

Method	Instantiation Rate
LLM	66.9%
LEMMANAID	89.1%
Combined	92.2%

Timeouts account for 96% of the cases where LEMMANAID's symbolic engine fails to instantiate a template. The remaining small number of failures are due to runtime errors, such as referencing private (hidden) symbols or exceptions during a theory import. Since the LLM is able to overcome some of the failures of the symbolic engine, a potential hybrid system could have both an LLM and symbolic engine instantiating templates. Further examination of the strengths and weaknesses of each approach could help achieve an optimal complementary combination.

D Results found using greedy decoding vs beam search

We break down the results from Table 1 into Table 4 and Table 5 to show the performance of different decoding strategies. The results in Table 4 are for greedy decoding, which selects the token with the highest probability at each generation step. The results in Table 5 are for beam search, which obtains multiple high-probability predictions for a given model. We see that beam search helps to improve results for both LEMMANAID and the neural baseline ensembles. However, LEMMANAID is still performant in a greedy setting, showing that the approach can achieve good results with a smaller inference budget.

E Breakdown of Results per Theory

E.1 HOL-test

For the lemmas included in our test set HOL-test, Table 6 shows the number of test lemmas from each session and the lemma success rates (in percentages) of LEMMANAID, neural-only lemma prediction, and their combination. The results shown are the ensembles of the different variants of LEMMANAID and the neural baselines, trained on HOL+AFP-train using DeepSeek-coder-6.7b. We created a file-wise split of the HOL library.

For the 29 different sessions, we see that the LEMMANAID ensemble outperforms the neural ensemble on 25 of them, while the neural ensemble performs better in one case, and in the remaining three both methods get the same rate of accurate predictions (0% in two of the three). On average the success rate of the LEMMANAID ensemble is 10.34% higher than that of the neural ensemble. Their combined success rate is on average 3.00% higher than that of the LEMMANAID ensemble, showing that the approaches are somewhat complementary but mostly overlapping.

A clear outlier is the Eisbach session where both methods are completely unsuccessful, this is because the lemmas in that set are all simple example lemmas used to demonstrate the use of the Eisbach proof method language (Matichuk et al., 2016). These examples [8] only contain basic logic symbols which we discard from our input symbol set, so our models don't have any input to base their predictions on.

[8] https://isabelle.in.tum.de/library/HOL/HOL-Eisbach/Examples.html

Table 4: Neuro-symbolic and neural methods trained on HOL-train and HOL+AFP-train lemma success rates for different test sets and different input features (type information and/or definitions). QuickSpec lemma success rate for Octonions. Here, the decoding strategy is greedy.

	Lemma Success rate					
	HOL-train			HOL+AFP-train		
Method	HOL-test	AFP-Test	Octonions	HOL-test	AFP-Test	Octonions
Deepseek-coder-1.3b						
LEMMANAID (types + defs)	**28.6%**	14.5%	47.1%	**29.0%**	**19.6%**	51.4%
LEMMANAID (types)	27.2%	**14.8%**	**48.0%**	24.3%	**19.6%**	42.6%
LEMMANAID (defs)	23.6%	4.7%	44.3%	24.2%	8.0%	**53.1%**
Neural (types + defs)	22.8%	7.0%	29.1%	25.5%	11.0%	37.1%
Neural (types)	19.9%	7.7%	26.6%	23.2%	9.7%	41.7%
Neural (defs)	22.0%	3.4%	31.4%	20.3%	4.7%	36.3%
LEMMANAID Combined	37.1%	20.6%	58.0%	37.3%	26.8%	61.4%
Neural Combined	31.7%	11.3%	44.6%	32.8%	15.3%	50.6%
Combined	42.5%	22.7%	62.0%	42.9%	29.6%	66.6%
Deepseek-coder-6.7b						
LEMMANAID (types+defs)	**34.7%**	**19.6%**	**60.6%**	**36.3%**	9.8%	**62.0%**
LEMMANAID (types)	31.5%	17.9%	54.0%	34.3%	**18.8%**	45.7%
Neural (types+defs)	29.7%	11.1%	46.9%	15.8%	3.1%	33.4%
Neural (types)	28.1%	10.7%	42.9%	8.6%	14.1%	46.0%
LEMMANAID Combined	40.1%	22.9%	66.0%	42.2%	21.3%	69.1%
Neural Combined	34.7%	14.9%	50.9%	17.6%	15.2%	52.3%
Combined	45.0%	25.7%	70.3%	44.1%	26.2%	74.3%
Llama-3.2-1b						
LEMMANAID (types+defs)	16.2%	9.0%	41.4%	16.8%	9.4%	20.0%
LEMMANAID (types)	**17.8%**	**9.1%**	**43.4%**	**19.3%**	**11.1%**	22.9%
Neural (types+defs)	11.5%	3.4%	15.4%	9.5%	4.1%	**36.3%**
Neural (types)	11.2%	3.8%	21.4%	10.1%	4.3%	34.6%
LEMMANAID Combined	23.2%	13.5%	49.7%	24.2%	18.8%	52.6%
Neural Combined	16.3%	5.1%	30.0%	14.5%	7.3%	18.3%
Combined	27.4%	14.7%	51.7%	27.5%	20.2%	54.0%
QuickSpec	—	—	22.8%	—	—	22.8%

Table 5: Neuro-symbolic and neural methods trained on HOL-train lemma success rates for different test sets and different input features (type information and/or definitions). QuickSpec lemma success rate for Octonions. Here, the decoding strategy is beam search (beam size = 4).

	Lemma Success rate					
	HOL-train			HOL+AFP-train		
Method	HOL-test	AFP-Test	Octonions	HOL-test	AFP-Test	Octonions
Deepseek-coder-1.3b						
LEMMANAID (types + defs)	**37.4%**	**18.7%**	**52.6%**	**39.8%**	**26.2%**	**63.7%**
LEMMANAID (types)	35.6%	17.8%	49.7%	33.3%	25.7%	52.9%
LEMMANAID (defs)	31.6%	6.6%	48.6%	33.4%	12.1%	58.3%
Neural (types + defs)	28.2%	9.1%	31.4%	27.0 %	12.3%	43.4%
Neural (types)	31.9%	10.0%	42.3%	27.0%	11.7%	44.0%
Neural (defs)	29.2%	5.3%	41.4%	23.3%	6.4%	40.6%
LEMMANAID Combined	46.4%	25.0%	66.0%	48.0%	33.8%	74.6%
Neural Combined	40.6%	14.4%	52.9%	36.1%	17.3%	56.3%
Combined	52.0%	27.0%	69.4%	51.4%	35.8%	78.9%
Deepseek-coder-6.7b						
LEMMANAID (types+defs)	**46.0%**	**25.8%**	**68.0%**	**49.9%**	27.4%	76.6%
LEMMANAID (types)	42.9%	16.5%	66.6%	47.2%	**28.2%**	**79.1%**
Neural (types+defs)	40.9%	16.8%	52.6%	21.8%	20.0%	61.7%
Neural (types)	37.1%	14.7%	51.1%	36.7%	18.9%	52.9%
LEMMANAID Combined	51.6%	28.4%	76.0%	53.9%	34.1%	83.1%
Neural Combined	45.3%	20.0%	58.9%	41.7%	25.1%	65.1%
Combined	56.4%	30.9%	79.4%	57.4%	38.9%	84.6%
Llama-3.2-1b						
LEMMANAID (types+defs)	31.1%	17.8%	57.4%	27.2%	21.2%	46.6%
LEMMANAID (types)	**33.5%**	**18.4%**	**59.7%**	**34.0%**	**23.1%**	**60.6%**
Neural (types+defs)	21.6%	7.0%	32.0%	18.0%	9.7%	54.3%
Neural (types)	21.7%	7.8%	41.1%	14.3%	9.3%	56.9%
LEMMANAID Combined	38.6%	20.2%	62.6%	37.4%	29.0%	64.3%
Neural Combined	27.3%	8.2%	46.0%	21.9%	11.5%	36.3%
Combined	41.7%	21.2%	63.4%	39.7%	30.5%	66.0%
QuickSpec	—	—	22.8%	—	—	22.8%

Table 6: Results for the different sessions of the HOL library.

HOL session	Lemmas	LEMMANAID	Neural	Combined
Algebra	49	**73.47%**	67.35%	81.63%
Analysis	1009	**41.72%**	28.84%	44.80%
Auth	350	**61.71%**	60.00%	64.29%
Bali	346	**40.46%**	21.10%	41.62%
Cardinals	93	**46.24%**	34.41%	46.24%
Computational_Algebra	50	**66.00%**	62.00%	66.00%
Data_Structures	91	**74.73%**	70.33%	76.92%
Datatype_Examples	145	**53.10%**	23.45%	55.17%
Decision_Procs	240	**58.75%**	47.08%	60.42%
Eisbach	23	0.00%	0.00%	0.00%
ex	75	**41.33%**	37.33%	42.67%
Hoare	15	**80.00%**	66.67%	80.00%
HOLCF	93	**61.29%**	47.31%	63.44%
IMP	284	**89.08%**	88.73%	95.07%
Imperative_HOL	61	**40.98%**	27.87%	45.90%
IMPP	26	**50.00%**	26.92%	53.85%
IOA	20	**45.00%**	35.00%	50.00%
Lattice	55	**61.82%**	54.55%	69.09%
Library	1054	**54.74%**	39.47%	58.25%
Matrix_LP	25	**24.00%**	20.00%	24.00%
Metis_Examples	1	0.00%	0.00%	0.00%
MicroJava	178	**47.75%**	34.27%	51.12%
Nominal	302	**81.79%**	61.26%	84.44%
Nonstandard_Analysis	36	**75.00%**	72.22%	80.56%
Number_Theory	47	**61.70%**	**61.70%**	72.52%
Quotient_Examples	18	**55.56%**	38.89%	55.56%
Proofs	6	**33.33%**	16.67%	33.33%
TLA	7	**57.14%**	28.57%	57.14%
UNITY	41	14.63%	**19.51%**	24.39%

E.2 AFP-test

For the formalizations included in our test set AFP-test, Table 7 shows the formalization topic chosen by the project authors, the number of gold-standard lemmas from each project and the lemma success rates (in percentages) of LEMMANAID, neural-only lemma prediction, and their combination. The results shown are the ensembles of the different variants of LEMMANAID and the neural baselines, trained on HOL+AFP-train using DeepSeek-coder-6.7b.

We can see that out of the 27 tested formalization projects, the LEMMANAID ensemble outperforms the neural ensemble on 19 of them, while the neural ensemble performs better for two, and in the remaining 6 cases the performance is equally good (0% in five out of six cases). Note that most of the projects with a success rate of 0% contain a small number of target lemmas (less than 10), with the outlier being the Actuarial Mathematics formalization. In this case, the syntax used in the formalization meant that none of the predicted lemmas were considered syntactically equivalent, however a more thorough evaluation would have shown some of them to be semantically equivalent to the target lemmas. On average the LEMMANAID ensemble has a 7.18% better performance than the neural one. Their combined success rate is on average 3.52% higher than the one for the LEMMANAID ensemble.

F Examples of Generated Conjectures

The analysis presented in section 4.3 divides the set of conjectures generated by LEMMANAID into *true*, *false*, and *open* conjectures. Here, we present some examples of conjectures in each category.

True Statements If Sledgehammer succeeds in finding a proof of a given conjecture, it's considered a true statement. True statements included both gold-standard lemmas formalized by experts and other novel true statements not previously formalized. The following two conjectures were discovered by LEMMANAID: the first lemma is identical to the gold-standard lemma $totient\_imp\_prime$ and the second one doesn't match any formalized lemmas.

lemma "$totient\ x_1 = x_1 - 1 \Longrightarrow 0 < x_1 \Longrightarrow prime\ x_1$"

lemma "$1 \leq x_1 \Longrightarrow 1 \leq totient\ x_1$"

False Statements Conjectures for which Nitpick or Quickcheck can find a counter-example are considered false statements. For the conjecture below, Quickcheck finds a counter-example when $x_1 = 2$.

lemma "$2 \leq x_1 \Longrightarrow even\ (totient\ x_1)$"

Open Conjectures LEMMANAID generates a number of conjectures which can't be automatically categorized. This is expected due to the incompleteness of the automated reasoning tools we use. These conjectures require

Table 7: Information about the different formalization projects in AFP-test.

AFP entry	Topic(s)	Lemmas	LEMMANAID	Neural	Combined
ConcurrentHOL	CS/Concurrency	481	**39.71%**	28.27%	43.24%
Sumcheck Protocol	CS/Security CS/Algorithms	62	**67.74%**	62.90%	74.19%
Broadcast_Psi	CS/Concurrency	145	**82.07%**	69.66%	85.52%
AutoCorres2	CS/PL CS/Semantics & reasoning Tools	12721	**34.72%**	22.48%	37.86%
Substitutions Lambda-Free	Logic/Rewriting	55	**50.91%**	21.82%	52.73%
Doob_Convergence	Math/Prob. theory	2	0.00%	0.00%	0.00%
Orient_Rewrite Rule_Undecidable	Logic/Rewriting	148	**60.14%**	42.57%	62.16%
Uncertainty Principle	Math/Physics	3	0.00%	0.00%	0.00%
Derandomization with Conditional Expectations	CS/Algorithms	15	40.00%	40.00%	40.00%
Verified QBF Solving	CS/Algorithms Logic/General logic	171	**53.80%**	43.86%	57.31%
IMP Noninterference	CS/PL CS/Security	50	**40.00%**	38.00%	42.00%
Actuarial Mathematics	Math/Games & econ.	168	0.00%	0.00%	0.00%
LL(1) Parser Generator	CS/Algorithms CS/PL	205	38.05%	**46.83%**	55.61%
Schönhage-Strassen Multiplication	CS/ALgorithms Math/Algebra	128	**41.41%**	26.56%	43.75%
Isabelle_DOF	CS/Semantics & reasoning	41	**53.66%**	34.15%	53.66%
Interval Analysis	Math/Analysis	694	**55.19%**	46.83%	58.50%
MFOTL Checker	CS/Data mgmt systems CS/Algorithms Logic/General logic	2081	14.22%	**24.89%**	30.71 %
Decomposition of totally ordered hoops	Math/Algebra	1	0.00%	0.00%	0.00%
Approximate Model Counting	CS/Algorithms	112	**32.14%**	17.86%	34.82%
Wieferich-Kempner Theorem	Math/Number theory	8	0.00%	0.00%	0.00%
PNT_with Remainder	Math/Number theory	155	**35.48%**	25.81%	40.00%
Continued Fractions	Math/Analysis	428	**37.62%**	35.75%	43.69%
CondNormReasHOL	Logic/Phil. aspects Logic/General logic	34	**11.76%**	5.88%	11.76%
Region Quadtrees	CS/Data structures	182	**41.76%**	34.07%	46.15%
Karatsuba	CS/Algorithms	431	**50.58%**	32.25%	55.22%
Pick's Theorem	Math/Geometry	334	**19.46%**	14.37%	22.46%
Kummer Congruence	Math/Number theory	120	**34.17%**	25.83%	38.33%

further complex reasoning to decide if that are true or false. One example of an open conjecture generated by LEMMANAID is the monotonicity of Euler's function. Nitpick and Quickcheck fail to find a counterexample such as $x_1 = 21, x_2 = 22$.

lemma "$x_1 \leq x_2 \Longrightarrow totient\ x_1 \leq totient\ x_2$"

G Additional Test Set Statistics

Statistics on HOL-test, AFP-test, and Octonions, and the successes of LEMMANAID and neural baselines on each dataset are shown in Table 8. We count the number of lemmas in each test set (Lemmas), how many of them are equational (Eq), and look into the length (in characters) of lemmas and their templates as well as the number of different symbols that appear. LEMMANAID and neural models here are trained on HOL-train and use greedy decoding, using DeepSeek-coder-6.7b for both the neural prediction and the neural engine of LEMMANAID.

H Partial QuickSpec output on Octonions

As reported in the main text, QuickSpec generated close to 10,000 lemmas on the Octonions example, so we do not include the full output here. However, we include the output on a small subset of the Octonions theory to illustrate some of the problems.

QuickSpec takes as input a list of function and constant symbols out of which it will build the terms. We started by giving it the functions (with $\times$, inverse and 1). As the inverse of 0 is not defined, we had to instruct QuickSpec to only test using non-zero Octonions, a limitation not shared by LEMMANAID. For the same reason we could not give it the constant symbol 0 or the functions $+$ and $-$, which would allow it to construct a zero octonion. The output of QuickSpec is shown in Figure 2. So far the number of lemmas is manageable and 9 of the 18 are found in the AFP theory, giving a precision of 50%.

Then we ran QuickSpec again, adding the extra functions $+$, 0, inner product $(\cdot)$ and norm, but removing the inverse operation (as discussed above). The results are still reasonable but, especially with the inner product function, most of the lemmas are uninteresting, not found in the AFP theory, and generated only because they happen to be true (see Figures 3 and 4). In general, it seems that the more function symbols there are, the more QuickSpec suffers from combinatorial blowup and generating uninteresting lemmas.

In the complex numbers, the functions $\mathrm{Re}, \mathrm{Im} : \mathbb{C} \to \mathbb{R}$ extract the real and imaginary parts of a number respectively. In the octonions, there are eight analogous functions $\mathrm{Re}, \mathrm{Im}_1 \ldots \mathrm{Im}_7 : \mathbb{O} \to \mathbb{R}$. When we added these functions, QuickSpec generated 775 laws, the vast majority extremely uninteresting for a human. An example of a typical generated law is $\mathrm{Re}(x + \mathrm{Im}_2\ y \times z) = \mathrm{Re}(x + y \times \mathrm{Im}_2\ z)$.

Corresponding to the complex number i there are seven octonions $e_1 \ldots e_7$ which (together with 1) act as unit numbers. Unfortunately when we add these as constants then QuickSpec starts to generate many thousands of irrelevant lemmas. Examples include:

$$(e_5 - e_4) \times (e_1 + e_4) = (1 + e_1) \times (e_4 - e_1)$$

and

$$e_4 \times (\mathrm{Im}_1\ x \times \mathrm{Im}_2\ y) = \mathrm{Im}_7(\mathrm{Im}_1\ x \times (y \times e_4))$$

The problem is that QuickSpec does not attempt to judge which lemmas are relevant to a human user.

Figure 2: Output from QuickSpec on Octonions for multiplication and inverse.

```
== Functions ==
(*) :: It -> It -> It
  1 :: It

== Laws ==
  1. x * 1 = x
  2. 1 * x = x
  3. (x * x) * y = x * (x * y)
  4. (x * y) * x = x * (y * x)
  5. (x * y) * y = x * (y * y)
  6. x * (y * (x * y)) = (x * y) * (x * y)
  7. x * (y * (y * x)) = (x * y) * (y * x)
  8. x * (y * (y * y)) = (x * y) * (y * y)
  9. x * ((y * z) * x) = (x * y) * (z * x)
 10. (x * (y * x)) * z = x * (y * (x * z))
 11. ((x * y) * z) * y = x * (y * (z * y))

== Functions ==
inv :: It -> It

== Laws ==
 12. inv 1 = 1
 13. inv (inv x) = x
 14. x * inv x = 1
 15. inv x * inv y = inv (y * x)
 16. inv x * (x * y) = y
 17. x * (y * inv x) = (x * y) * inv x
 18. (inv x * y) * x = inv x * (y * x)
```

Table 8: Number of lemmas in each test set, and how many were discovered by LEMMANAID and the corresponding neural method (using DeepSeek-6.7b and greedy decoding), including how many were equational. The table also shows information about template and lemma lengths (number of characters) and the number of different symbols appearing.

	Lemmas	Eq	Template Length					Lemma Length					# Symbols	
			Min	25%	Med	75%	Max	Min	25%	Med	75%	Max	Mean	Max
HOL-test	4740	2242	9	49	82	127	9014	7	57	91	150	10454	4.2	100
LEMMANAID	1643	885	11	37	55	85	300	10	45	69	100	368	3.3	15
Neural	1407	750	11	37	55	85	433	11	45	68	98	390	3.3	15
AFP-test	18975	13202	7	39	63	130	10492	7	58	80	141	7530	4.1	31
LEMMANAID	3712	2989	7	25	31	45	220	12	39	55	73	286	2.8	14
Neural	2111	1526	7	23	33	45	184	10	36	52	71	253	2.8	12
Octonions	350	262	17	29	49	75	688	11	33	55	79	391	3.8	21
LEMMANAID	212	152	17	23	43	67	121	11	29	47	65	133	2.7	11
Neural	164	105	19	23	49	67	269	15	25	47	66	279	2.7	11.0

Figure 3: Output from QuickSpec on Octonions with $+$, 0, inner product $(\cdot)$

```
== Functions ==
(*) :: It -> It -> It
  0 :: It
  1 :: It

== Laws ==
  1. x * 0 = 0
  2. x * 1 = x
  3. 0 * x = 0
  4. 1 * x = x
  5. (x * x) * y = x * (x * y)
  6. (x * y) * x = x * (y * x)
  7. (x * y) * y = x * (y * y)
  8. x * (y * (x * y)) = (x * y) * (x * y)
  9. x * (y * (y * x)) = (x * y) * (y * x)
 10. x * (y * (y * y)) = (x * y) * (y * y)
 11. x * ((y * z) * x) = (x * y) * (z * x)
 12. (x * (y * x)) * z = x * (y * (x * z))
 13. ((x * y) * z) * y = x * (y * (z * y))

== Functions ==
(+) :: It -> It -> It

== Laws ==
 14. x + y = y + x
 15. x + 0 = x
 16. (x + x) * y = x * (y + y)
 17. (x + y) + z = x + (y + z)
 18. x * (y + 1) = x + (x * y)
 19. (x + 1) * y = y + (x * y)
 20. (x + x) * (x * y) = (x * x) * (y + y)
 21. (x + x) * (y * x) = (x * y) * (x + x)
 22. (x + x) * (y * y) = (x * y) * (y + y)
 23. (x * y) + (x * z) = x * (y + z)
 24. (x * y) + (z * y) = (x + z) * y
 25. (x + 1) * (x + x) = (x + x) * (x + 1)
 26. x * (y * (x + y)) = (x * y) * (x + y)
 27. x * (y + (y * x)) = (x * y) * (x + 1)
 28. x * (y + (y * y)) = (x * y) * (y + 1)
 29. (x * (x + y)) * y = x * ((x + y) * y)
 30. ((x + y) * x) * y = (x + y) * (x * y)
 31. (x + (x * x)) * y = (x + 1) * (x * y)
 32. (x + (y * x)) * y = (y + 1) * (x * y)
 33. (x + (x + x)) * y = x * (y + (y + y))

== Functions ==
(·) :: It -> It -> It

== Laws ==
 34. x · y = y · x
 35. x · 0 = 0
 36. 1 · 1 = 1
 37. (x · y) * z = z * (x · y)
 38. x · (y * x) = x · (x * y)
 39. x · (y + y) = y · (x + x)
 40. x · (y · y) = y · (x * y)
 41. x · (y · 1) = y · (x · 1)
 42. 1 · (x * y) = 1 · (y * x)
 43. 1 · (x · y) = x · y
 44. 1 · (x + 1) = 1 + (x · 1)
 45. (x · y) + (x · z) = x · (y + z)
 46. (x * y) · (x * z) = (x · x) * (y · z)
 47. (x * y) · (z * y) = (x · z) * (y · y)
 48. (x * y) · (y + x) = (y * x) · (y + x)
 49. (x * y) · (z · w) = (y * x) · (z · w)
 50. (x · y) · (z · w) = (x · y) * (z · w)
 51. (x + x) * (x · 1) = (x * x) + (x · x)
 52. (x · y) * (z · 1) = z · (x · y)
 53. (x * y) · (y + 1) = (y * x) · (y + 1)
 54. (x · y) · (z · 1) = z · (x · y)
 55. (1 + 1) · (1 + 1) = (1 + 1) * (1 + 1)
 56. x * (y * (z · w)) = (x * y) * (z · w)
 57. (x * (y · z)) * w = (x * w) * (y · z)
 58. (x + (y · z)) * x = x * (x + (y · z))
 59. x · (y * (z * x)) = (y * z) · (x · x)
 60. x · (y * (z · w)) = (x · y) * (z · w)
 61. x · ((x * y) * z) = (y * z) · (x · x)
 62. x · ((y * x) * z) = x · (y * (x * z))
 63. x · ((x + y) * y) = x · (y * (x + y))
 64. x · (y + (z * x)) = x · (y + (x * z))
 65. x · (y + (x · x)) = x · (y + (x * x))
 66. x · (y + (x · y)) = (x + 1) · (x · y)
 67. x · (y + (y · y)) = y · (x + (x * y))
 68. x · (y · (z * y)) = z · (y · (x * y))
 69. x · (y · (z · w)) = y · (x · (z · w))
 70. x · (y + (y + y)) = y · (x + (x + x))
 71. x · (y * (z · 1)) = z · (x · y)
 72. 1 · ((x * y) * z) = 1 · (x * (y * z))
 73. (x * (y · 1)) · z = y · (x · z)
 74. x · (y + (y · 1)) = y · (x + (x · 1))
```

Figure 4: Continued output from Figure 3, Octonions with $+$, 0, inner product $(\cdot)$ and norm

```
== Functions ==
norm :: It -> It

== Laws ==
 75. norm 0 = 0
 76. norm 1 = 1
 77. norm x = x · x
 78. norm (x + (x * x)) = norm (x + norm x)
 79. (x · y) * (z · 1) = z · (x · y)
 80. (x · y) · (z · 1) = z · (x · y)
 81. norm (norm x + (y * z)) = norm (norm x + (z * y))
 82. (norm x + norm y) * z = z * (norm x + norm y)
 83. x · (y * (z · 1)) = z · (x · y)
 84. (x * (y · 1)) · z = y · (x · z)
 85. norm x * (1 + norm y) = norm x + norm (x * y)
 86. norm x · (1 + norm y) = norm x + norm (x * y)
 87. norm (x * (x + norm x)) = norm (norm x * (x + 1))
```