Lemmanaid: Neuro-Symbolic Lemma Conjecturing

Yousef Alhessi [*1] **Sólrún Halla Einarsdóttir** [*2] **George Granberry** [2] **Emily First** [1] **Moa Johansson** [2]
Sorin Lerner [1] **Nicholas Smallbone** [2]

Abstract

Automatically conjecturing useful, interesting and novel lemmas would greatly improve automated reasoning tools and lower the bar for formalizing mathematics in proof assistants. It is however a very challenging task for both neural and symbolic approaches. We present the first steps towards a practical neuro-symbolic lemma conjecturing tool, LEMMANAID, that combines Large Language Models (LLMs) and symbolic methods, and evaluate it on proof libraries for the Isabelle proof assistant. We train an LLM to generate *lemma templates* that describe the shape of a lemma, and use symbolic methods to fill in the details. We compare LEMMANAID against an LLM trained to generate complete lemma statements as well as previous fully symbolic conjecturing methods. LEMMANAID outperforms both neural and symbolic methods on test sets from Isabelle's HOL library and from its Archive of Formal Proofs, discovering between 29-39.5% of the gold standard human written lemmas. This is 8-15% more lemmas than the neural-only method. By leveraging the best of both symbolic and neural methods we can generate useful lemmas for a wide range of input domains, facilitating computer-assisted theory development and formalization.

1. Introduction

Learning to construct new, interesting, and useful lemmas for proof assistants is an important yet underexplored area in AI for mathematical reasoning (Yang et al., 2024). Such lemmas can aid a human user working on a mathematical formalization, as well as strengthen automated theorem provers. In this work, we examine how LLMs can be used for lemma generation, and how they can be combined with symbolic tools for optimal results. Our aim is to to provide a first tool towards generic conjecturing over a broad range of mathematical theories, which is practically useful for users of proof assistants.

A weakness of LLMs is that they sometimes generate repetitive or redundant lemmas, fail to discover more novel and useful lemmas, or hallucinate undefined symbols in the formalization. Furthermore, there are no correctness guarantees on the LLM's output, so the generated lemmas may simply be false. These challenges have been encountered in previous work on neural conjecturing (Urban & Jakubův, 2020; Rabe et al., 2021; Johansson & Smallbone, 2023). Symbolic methods, on the other hand, can be designed and programmed to avoid repetition and redundancy. However, symbolic methods will only generate lemmas that fit a predefined specific search space, and tend to scale poorly to a larger search space. Previous symbolic tools (Smallbone et al., 2017; Einarsdóttir et al., 2021; Singher & 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). However, these 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. An LLM is trained to generate *lemma templates* that describe the shape of a family of analogous lemmas, rather than directly generating complete lemmas. Symbolic synthesis methods are then used to fill in the details. Previous work has shown that such families of analogous lemmas indeed occur in proof assistant libraries (Einarsdóttir et al., 2022; Heras et al., 2013). 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.

[*]Equal contribution [1]Department of Computer Science and Engineering, University of California, San Diego, USA [2]Department of Computer Science and Engineering, Chalmers University of Technology & University of Gothenburg, Gothenburg, Sweden. Correspondence to: Sólrún Halla Einarsdóttir <slrn@chalmers.se>.

The second AI for MATH Workshop at the 42^{nd} International Conference on Machine Learning, Vancouver, Canada.

For evaluation, we measure the coverage of (unseen test-set) lemmas that can be discovered by LEMMANAID from Isabelle's HOL library[1] and from its Archive of Formal Proofs (AFP)[2]. Using human-written formalizations as a gold-standard gives us a good approximation of how many *interesting* lemmas LEMMANAID can produce for a novel theory (taking only basic definitions of the theory as input), an evaluation strategy used in many prior lemma conjecturing works, e.g. (Montano-Rivas et al., 2012; Smallbone et al., 2017; Einarsdóttir et al., 2018; Urban & Jakubův, 2020). In our experiments, we use a DeepSeek-coder-1.3b language model. We choose a small LLM for several reasons: in addition to computational and environmental concerns, our ultimate aim is to create something that is accessible for regular proof-assistant users without need for huge compute resources, i.e. a model which can run locally for inference. We further 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 predict templates and a symbolic engine to instantiate templates as lemmas.
- An evaluation of LEMMANAID on the Isabelle proof assistant's HOL and AFP libraries. This goes much beyond evaluations of prior tools that have focused on specific 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.

To ensure reproducibility of our results and enable others to build on our work, we make all code, experimental scripts, data, and models publicly available online[3].

2. Related Work

Proof Assistants 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).

Autoformalization and Proof Synthesis Formalizing theories in proof assistants is a non-trivial task, which has sparked interest in *autoformalization*: translating definitions, theorems and lemmas written in natural language to the formal language of a proof assistant (Wang et al., 2018; Szegedy, 2020; Wu et al., 2022). Furthermore, 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 incrementally (Jiang et al., 2023; Ren et al., 2025), or by generating whole proofs at once (First et al., 2023). The LEGO-prover furthermore attempts to introduce intermediate proof statements as reusable lemmas (Wang et al., 2024). 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. As such it is orthogonal to much previous work, but could in the future be combined with, and complement, proof synthesis systems.

Templates for Synthesizing Conjectures Following the observation that many mathematical theories share analogical lemmas of similar shapes, Buchberger et al. (2006) proposed to use *templates* as guidance in mathematical theory exploration, allowing efficient conjecturing of many (but not all) 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). Unlike our work, where we use an LLM to suggest templates based on context, the templates in symbolic systems are typically pre-defined or provided in interaction with the user.

Neural Conjecturing and Reinforcement Learning Recent work on neural conjecturing and reinforcement learning has demonstrated success in specific domains —famously in Euclidian plane geometry through the AlphaGeometry system (Trinh et al., 2024), as well as in reasoning about programs that generate integer sequences from the Online Encyclopedia of Integer Sequences (OEIS) (Gauthier & Urban, 2023; 2025). Poesia et al. (2024) treated conjecturing as a reinforcement learning game in simple propositional logic, arithmetic and group theory, with well formed con-

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

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

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

jectures generated neurally via constrained decoding. In contrast, we target the broad range of theories represented in proof assistant libraries, and want to avoid domain specific learning by generating generic templates as an intermediate step. The STP system (Dong & Ma, 2025), performs a type of conjecture generation by producing variants of a given seed-statement, with the aim of generating additional training data for a self-playing theorem prover. LEMMANAID, on the other hand, does not rely on having an available seed-statement to start from, and also targets a different use-case, namely to make suggestions to a human doing a formalization.

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 those. 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. Secondly (symbolic part), LEMMANAID searches over possible instantiation of those templates in the current theory, to produce concrete conjectures.

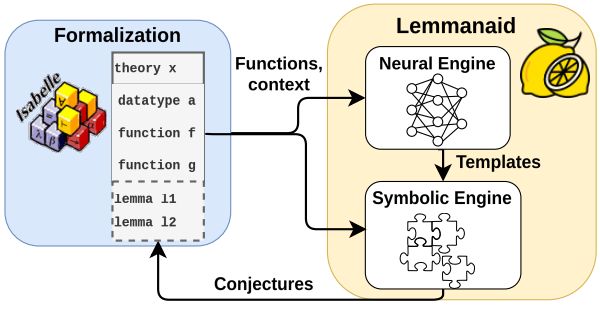

Figure 1. High-level overview of LEMMANAID.

Section 3.1 introduces the notion of template-based conjecturing, our template language, and how templates capture information about families of analogous lemmas. We also describe the symbolic engine of LEMMANAID and how it instantiates templates to produce conjectures. Section 3.2 discusses how we train LEMMANAID's neural engine, a fine-tuned LLM that predicts templates, which are subsequently fed to the symbolic engine. Section 3.3 details direct lemma conjecturing baselines that we implement for Isabelle. Finally, Section 3.4 describes the datasets we use for training and evaluation.

3.1. Template-based conjecturing via 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

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

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.1 for details about these symbols.

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.

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:

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

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. Template Prediction via LEMMANAID's Neural Engine

LEMMANAID's neural engine uses a fine-tuned LLM to predict templates given a theory context. Template predictions are inputted 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, 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 datapoint 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 Lemma 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 datapoint 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 206,304 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? How does different contextual information (such as types and definitions) impact performance?

RQ2 How does an LLM compare to LEMMANAID's symbolic engine in its ability to fill in lemma templates?

RQ3 How well does an LLM understand the lemma template language?

RQ4 What kinds of lemmas is LEMMANAID able to generate?

4.1. Experimental Setup

Baselines While prior work (Gauthier & Urban, 2023; 2025) has explored direct neural lemma conjecturing, there are no existing neural-based tools or models for Isabelle to compare against. We train our own neural baselines (Section 3.3), and go beyond prior work by predicting the 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 (AFP) (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 in AFP-test can be found in A.3. 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 16,362 datapoints.

We separately test on the Octonions project (Koutsoukou-Argyraki, 2018), which contains 350 lemmas, from the AFP 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

[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.

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).

	HOL-train			HOL+AFP-train		
	HOL-test	AFP-Test	Octonions	HOL-test	AFP-Test	Octonions
LEMMANAID (types + defs)	**38.8%**	**25.4%**	53.7%	38.6%	**29.1%**	47.4%
LEMMANAID (types)	35.6%	**25.4%**	**58.5%**	**39.5%**	26.5%	**52.0%**
LEMMANAID (defs)	33.0%	13.8%	45.7%	33.9%	14.7%	41.4%
Neural (types + defs)	30.7%	13.8%	34.6%	29.1%	14.9%	39.1%
Neural (types)	31.5%	14.4%	43.4%	28.9%	14.6%	41.1%
Neural (defs)	26.7%	8.6%	30.6%	28.5%	10.4%	36.3%
LEMMANAID Combined	46.5%	30.9%	68.0%	48.2%	34.6%	65.7%
Neural Combined	39.7%	19.5%	49.1%	38.9%	20.8%	56.6%
Combined	50.7%	32.9%	71.4%	52.0%	37.4%	74.0%
QuickSpec	—	—	22.8%	—	—	22.8%

in the active theory. The implications of this enumeration are discussed in A.2.

Metrics We define multiple metrics used throughout our evaluation. In our approach, we have a set of lemma-prediction tasks. In each such task, there is one *gold-standard lemma*, and the method being assessed generates a set of predicted lemmas. We define *lemma success rate* as the percentage of these lemma prediction 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*. For LEMMANAID's neural engine, we want to measure *template success rate*, which is the percentage of template-prediction tasks for which it predicts the correct gold-standard template. For the symbolic engine, given a template, the symbolic engine generates many instantiations, and we consider the instantiation task to be successful if one of the generated instantiations matches the gold-standard lemma (syntactically). The *instantiation rate* measures the percentage of instantiation tasks that the symbolic engine can perform successfully. We also use this metric in RQ2 to measure an LLM's ability to instantiate templates. To answer RQ3, we need to measure how well LLMs are able to generate templates from lemma objects. In this case, a task consists of predicting a template from a lemma object. We define *abstraction rate* as the percentage of such lemma-template-from-object prediction tasks the LLM is able to perform.

Models and Inference For all tasks, we use deepseek-coder-1.3b-base as the pretrained model that we fine-tune. In early experiments, we explored the use of Llama-3.2-1b, but found that all methods performed slightly worse. 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 A.4 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. For all other RQs, greedy decoding is used for LLM inference.

Checking Outputs To check the correctness of a predicted 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: Neuro-symbolic vs Neural vs Symbolic

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 all models, we compare the results obtained when various contextual information is included in the input: definitions, type information, or both (as described in 3.2). 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 A.5.

We see that LEMMANAID outperforms the respective neural baselines on all test sets. We also see that neural methods are somewhat complementary to LEMMANAID, taken as an ensemble they discover even more lemmas. 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 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 include enough contextual information as retrieving definitions in AFP theories requires retrieving across dependencies, though we always account for HOL as a dependency. We see that when trained on HOL+AFP-train, performance improves on AFP-test, in particular for LEMMANAID where we see an increase from 25.4% to 29.1%, while the neural baseline only increases from 14.4% to 14.9%.

For Octonions, LEMMANAID models outperform their respective neural baselines while both greatly outperform QuickSpec. Similar to other datasets, an ensemble of LEMMANAID and neural models provides a significant improvement, with overall lemma success rate up to 74%. However, QuickSpec is complementary to both models trained on HOL-train and HOL+AFP-train, improving overall lemma success rate to 74.6 % and 76.9% respectively.

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 no 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 shows the limitations of the purely symbolic approach in selecting relevant lemmas, and is further illustrated in A.6.

4.3. RQ2: Template instantiation by LEMMANAID's symbolic engine vs an LLM

As discussed in Section 3.1, LEMMANAID instantiates its predicted templates symbolically. We also noted some limitations of this instantiation and how it is subject to timeouts. Alternatively, we can train an LLM to instantiate templates. Table 2 displays the template instantiation rate for LEMMANAID's symbolic engine and an LLM trained to instantiate templates when both are given the gold-standard lemma template and the symbols appearing in the gold-standard lemma. We observe that LEMMANAID's symbolic engine has a much greater instantiation rate (89.1%) than that of an LLM (66.9%). 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 2. 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.

4.4. RQ3: Template abstraction performance of LLM

Table 3. Template abstraction rate for LLMs (fine-tuned on HOL-train) with different inputs evaluated on HOL-test.

LLM Input	Abstraction Rate
Lemma Object	88.9%
+ Symbols + Types	92.0%
+ Symbols + Types + Defs	90.3%
Combined	94.5%

To assess how well LLMs understand the language of lemma templates, we train an LLM to abstract templates from lemma objects. We have three variants of this task: (1) only lemma objects as input, (2) lemma objects, symbols, and their types as inputs, and (3) which is similar to (2) with the addition of definitions (Table 3). Models perform

Table 4. Statistics on HOL-test, AFP-test, and Octonions, and the successes of LEMMANAID and neural baselines on each dataset. LEMMANAID and neural models here are trained on HOL-train and use greedy decoding.

	Lemma Count	Equational	Template Length					Lemma Length					# Symbols	
			Min	25%	Median	75%	Max	Min	25%	Median	75%	Max	Mean	Max
HOL-test	4740	2242	9	49	82	127	9014	7	57	91	150	10454	4.2	100
Lemmanaid Success	1323	734	11	33	53	77	289	10	43	65	90	325	3.0	13
Neural Success	1135	629	11	37	55	77	289	11	45	65	92	347	3.1	15
AFP-test	16362	10582	7	37	72	142	10794	7	56	88	152	7530	4.2	31
Lemmanaid Success	2420	1894	7	23	29	43	210	10	37	52	72	246	2.7	10
Neural Success	1594	1142	11	23	33	45	173	12	34	48	68	192	2.7	13
Octonions	350	262	17	29	49	75	688	11	33	55	79	391	3.8	21
Lemmanaid Success	143	107	17	23	39	59	109	11	25	43	60	133	2.5	6
Neural Success	106	66	19	19	44	67	109	15	24	47	73	133	2.4	7

similarly, and there are some non-overlapping examples that each model uniquely predicts so the combination has 93.7% abstraction rate. Notably, a point of confusion for models is operators with custom syntax. The use of custom syntax introduces ambiguity about how the arguments of a function are ordered. Custom syntax can possibly make some arguments implicit, which introduces further ambiguity. Overall, these results indicate that LLMs can understand the template language well.

4.5. RQ4: Qualitative analysis

To assess the utility of our tools, we study the characteristics of HOL-test, Octonions and AFP-test, as well as the subsets of lemmas that LEMMANAID and neural baselines successfully generate. For each subset, we consider the number of equational lemmas, the length (in characters) of the templates and lemmas, and the number of symbols appearing in the lemmas (Table 4). Both LEMMANAID and the neural baselines demonstrate an ability to predict equational and non-equational lemmas. They successfully discover lemmas and templates of different lengths but show greater ability to handle shorter ones. This is to be expected because some contextual information for longer lemmas would be cutoff due to our max sequence length (where we limit the number of input and output tokens) and they are likely more complex for an LLM to reason about. See A.7 for some examples that LEMMANAID synthesizes that other methods do not. See A.8 for some examples of how LEMMANAID succeeds and fails.

5. Conclusion

LEMMANAID is a novel neuro-symbolic tool for conjecturing lemmas for mathematical formalization. LEMMANAID outperforms both neural baselines we create and symbolic tool, QuickSpec, on test sets from Isabelle's HOL library and from its Archive of Formal Proofs. LEMMANAID discovers between 29-39.5% of the gold standard human written lemmas in test sets, 8-15% more lemmas than the neural baseline using the same training setup. LEMMANAID can discover useful lemmas for a wide range of formalization domains in mathematics, computer science, and logic.

We note that our experimental setup most likely underreports results, as we measure matches with one specific gold-standard lemma. It is entirely possible that LEMMANAID sometimes comes up with a different gold-standard lemma from the same theory, or even additional lemmas that are valid and useful but not present in the existing formalization. Our evaluation is a first step towards demonstrating the usefulness of neuro-symbolic conjecturing for proof assistants. So far we have not yet explored its full potential. For instance, for future work LEMMANAID could easily be placed in a workflow utilizing the many tools available in Isabelle/HOL for e.g. counterexample checking and automated proofs: Sledgehammer, simp or even neural proof synthesis methods.

Acknowledgments

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. The computation was enabled by resources provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS), partially funded by the Swedish Research Council through grant agreement no. 2022-06725.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Blanchette, J. C., Böhme, S., and Paulson, L. C. Extending sledgehammer with smt solvers. *Journal of Automated Reasoning*, 51:109 – 128, 2011. URL https://api.semanticscholar.org/CorpusID:5389933.

Buchberger, B., Craciun, A., Jebelean, T., Kovács, L., Kutsia, T., Nakagawa, K., Piroi, F., Popov, N., Robu, J., Rosenkranz, M., and Windsteiger, W. Theorema: Towards computer-aided mathematical theory exploration. *Journal of Applied Logic*, 4:470–504, 12 2006. doi: 10.1016/j.jal.2005.10.006.

de Moura, L., Kong, S., Avigad, J., van Doorn, F., and von Raumer, J. The lean theorem prover (system description). In Felty, A. P. and Middeldorp, A. (eds.), *Automated Deduction - CADE-25*, pp. 378–388, Cham, 2015. Springer International Publishing. ISBN 978-3-319-21401-6.

Dong, K. and Ma, T. STP: Self-play LLM Theorem Provers with Iterative Conjecturing and Proving, 2025. URL https://arxiv.org/abs/2502.00212.

Einarsdóttir, S. H., Johansson, M., and Pohjola, J. Å. Into the infinite - theory exploration for coinduction. In *Proceedings of AISC 2018*, pp. 70–86, 01 2018. ISBN 978-3-319-99956-2. doi: 10.1007/978-3-319-99957-9_5.

Einarsdóttir, S. H., Smallbone, N., and Johansson, M. 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, pp. 67–78, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450389631. doi: 10.1145/3462172.3462192. URL https://doi.org/10.1145/3462172.3462192.

Einarsdóttir, S. H., Johansson, M., and Smallbone, N. 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*, pp. 22, 2022.

Einarsdóttir, S. H., Hajdu, M., Johansson, M., Smallbone, N., and Suda, M. Lemma discovery and strategies for automated induction. In Benzmüller, C., Heule, M. J., and Schmidt, R. A. (eds.), *Automated Reasoning*, pp. 214–232, Cham, 2024. Springer Nature Switzerland. ISBN 978-3-031-63498-7.

First, E., Rabe, M., Ringer, T., and Brun, Y. 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, pp. 1229–1241, New York, NY, USA, November 2023. Association for Computing Machinery. doi: 10.1145/3611643.3616243. URL https://doi.org/10.1145/3611643.3616243.

Gauthier, T. and Urban, J. Learning program synthesis for integer sequences from scratch. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37 (6):7670–7677, Jun. 2023. doi: 10.1609/aaai.v37i6.25930. URL https://ojs.aaai.org/index.php/AAAI/article/view/25930.

Gauthier, T. and Urban, J. Learning conjecturing from scratch, 2025. URL https://arxiv.org/abs/2503.01389.

Heras, J., Komendantskaya, E., Johansson, M., and Maclean, E. Proof-pattern recognition and lemma discovery in ACL2. In *Proceedings of LPAR*, 2013. doi: 10.1007/978-3-642-45221-5_27.

Jiang, A. Q., Welleck, S., Zhou, J. P., Lacroix, T., Liu, J., Li, W., Jamnik, M., Lample, G., and Wu, Y. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=SMa9EAovKMC.

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

Johansson, M., Rosén, D., Smallbone, N., and Claessen, K. Hipster: Integrating theory exploration in a proof assistant. In *Proceedings of CICM*, pp. 108–122. Springer, 2014.

Koutsoukou-Argyraki, A. Octonions. *Archive of Formal Proofs*, September 2018. ISSN 2150-914x. https://isa-afp.org/entries/Octonions.html, Formal proof development.

Kurashige, C., Ji, R., Giridharan, A., Barbone, M., Noor, D., Itzhaky, S., Jhala, R., and Polikarpova, N. Cclemma: E-graph guided lemma discovery for inductive equational proofs. *Proc. ACM Program. Lang.*, 8(ICFP), August 2024. doi: 10.1145/3674653. URL https://doi.org/10.1145/3674653.

McCasland, R. L., Bundy, A., and Smith, P. F. MATHsAiD: Automated mathematical theory exploration. *Applied Intelligence*, Jun 2017. ISSN 1573-7497. doi: 10.1007/s10489-017-0954-8. URL https://doi.org/10.1007/s10489-017-0954-8.

Montano-Rivas, O., McCasland, R., Dixon, L., and Bundy, A. Scheme-based theorem discovery and concept invention. *Expert systems with applications*, 39(2):1637–1646, 2012.

Nagashima, Y., Xu, Z., Wang, N., Goc, D. S., and Bang, J. Template-based conjecturing for automated induction in isabelle/hol. In Hojjat, H. and Ábrahám, E. (eds.), *Fundamentals of Software Engineering*, pp. 112–125, Cham, 2023. Springer Nature Switzerland. ISBN 978-3-031-42441-0.

Nipkow, T., Wenzel, M., and Paulson, L. C. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, Berlin, Heidelberg, 2002. ISBN 3540433767.

Poesia, G., Broman, D., Haber, N., and Goodman, N. Learning formal mathematics from intrinsic motivation. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=uNKlTQ8mBD.

Rabe, M. N., Lee, D., Bansal, K., and Szegedy, C. Mathematical reasoning via self-supervised skip-tree training. In *Proceedings of ICLR*, 2021.

Ren, Z. Z., Shao, Z., Song, J., Xin, H., Wang, H., Zhao, W., Zhang, L., Fu, Z., Zhu, Q., Yang, D., Wu, Z. F., Gou, Z., Ma, S., Tang, H., Liu, Y., Gao, W., Guo, D., and Ruan, C. Deepseek-prover-v2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition, 2025. URL https://arxiv.org/abs/2504.21801.

Shminke, B. Python client for isabelle server, 2022. URL https://arxiv.org/abs/2212.11173.

Singher, E. and Itzhaky, S. Theory exploration powered by deductive synthesis. In Silva, A. and Leino, K. R. M. (eds.), *Computer Aided Verification*, pp. 125–148, Cham, 2021. Springer International Publishing. ISBN 978-3-030-81688-9.

Smallbone, N., Johansson, M., Claessen, K., and Algehed, M. Quick specifications for the busy programmer. *Journal of Functional Programming*, 27, 2017.

Solar-Lezama, A. The sketching approach to program synthesis. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, APLAS '09, pp. 4–13, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 9783642106712. doi: 10.1007/978-3-642-10672-9_3. URL https://doi.org/10.1007/978-3-642-10672-9_3.

Szegedy, C. A promising path towards autoformalization and general artificial intelligence. In Benzmüller, C. and Miller, B. (eds.), *Intelligent Computer Mathematics*, pp. 3–20, Cham, 2020. Springer International Publishing. ISBN 978-3-030-53518-6.

Trinh, T. H., Wu, Y., Le, Q. V., He, H., and Luong, T. Solving olympiad geometry without human demonstrations. *Nature*, 625:476–482, 2024. URL https://doi.org/10.1038/s41586-023-06747-5.

Urban, J. and Jakubův, J. First neural conjecturing datasets and experiments. In *Proceedings of CICM*, 2020. doi: 10.1007/978-3-030-53518-6_24.

Wang, H., Xin, H., Zheng, C., Liu, Z., Cao, Q., Huang, Y., Xiong, J., Shi, H., Xie, E., Yin, J., Li, Z., and Liang, X. LEGO-prover: Neural theorem proving with growing libraries. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=3f5PALef5B.

Wang, Q., Kaliszyk, C., and Urban, J. First experiments with neural translation of informal to formal mathematics. In Rabe, F., Farmer, W. M., Passmore, G. O., and Youssef, A. (eds.), *Intelligent Computer Mathematics*, pp. 255–270, Cham, 2018. Springer International Publishing. ISBN 978-3-319-96812-4.

Wu, Y., Jiang, A. Q., Li, W., Rabe, M. N., Staats, C. E., Jamnik, M., and Szegedy, C. Autoformalization with large language models. In Oh, A. H., Agarwal, A., Belgrave, D., and Cho, K. (eds.), *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=IUikebJ1Bf0.

Yang, K., Poesia, G., He, J., Li, W., Lauter, K., Chaudhuri, S., and Song, D. Formal mathematical reasoning: A new frontier in ai, 2024. URL https://arxiv.org/abs/2412.16075.

A. Appendix

A.1. Template language and symbols

Templates are implemented as instances of Isabelle/HOL's term datatype[6]. 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.
- 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.

A.2. Lemma Enumeration

Our lemma enumeration strategy is best described through an example. From the `Cross_Product_7.thy` file from the AFP's Octonions project, the lemma `vector_7` was defined by the user as follows:

```
lemma vector_7 [simp]:
 "(vector [x1,x2,x3,x4,x5,x6,x7] ::('a::zero)^7)$1 = x1"
 "(vector [x1,x2,x3,x4,x5,x6,x7] ::('a::zero)^7)$2 = x2"
 "(vector [x1,x2,x3,x4,x5,x6,x7] ::('a::zero)^7)$3 = x3"
 "(vector [x1,x2,x3,x4,x5,x6,x7] ::('a::zero)^7)$4 = x4"
 "(vector [x1,x2,x3,x4,x5,x6,x7] ::('a::zero)^7)$5 = x5"
 "(vector [x1,x2,x3,x4,x5,x6,x7] ::('a::zero)^7)$6 = x6"
 "(vector [x1,x2,x3,x4,x5,x6,x7] ::('a::zero)^7)$7 = x7"
```

Given the way we count lemmas (or what we consider a datapoint in the entirety of the project), we consider this to be 7 lemmas. This stems from wanting our definition of a lemma to have one lemma object and one template associated with it. Isabelle differentiates between a "singleton" fact and an "indexed" fact, so we treat each as a singleton fact. In this example, these 7 lemmas are clearly very similar, but there is nothing enforcing this similarity, except perhaps conventions of Isabelle users. Furthermore, Isabelle allows one to also define multiple lemmas at the same time using a "lemmas" keyword where conventionally the lemmas are still similar but less similar than indexed facts typically are. Because this problem (one command, multiple facts) occurs a number of times across theories, we decided to treat every fact separately.

A.3. Projects in AFP-test

For the formalizations included in our test set AFP-test, Table 5 shows the 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.

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

Table 5. Information about the different formalization projects in AFP-test.

AFP entry	Topic(s)	Lemmas	LEMMANAID	Neural	Combined
ConcurrentHOL	CS/Concurrency	481	34.10%	24.53%	37.63%
Sumcheck Protocol	CS/Security CS/Algorithms	62	56.45%	43.55%	58.06%
Broadcast_Psi	CS/Concurrency	145	64.14%	40.69%	66.21%
AutoCorres2	CS/PL CS/Semantics & reasoning Tools	11638	33.17%	16.66%	34.96%
Substitutions Lambda-Free	Logic/Rewriting	55	54.55%	25.45%	56.36%
Doob_Convergence	Math/Prob. theory	2	0.00	0.00	0.00
Orient_Rewrite Rule_Undecidable	Logic/Rewriting	148	47.97%	38.51%	50.00%
Uncertainty Principle	Math/Physics	3	0.00	0.00	0.00
Derandomization with Conditional Expectations	CS/Algorithms	15	46.67%	33.33%	46.67%
Verified QBF Solving	CS/Algorithms Logic/General logic	171	44.44%	39.18%	49.71%
IMP Noninterference	CS/PL CS/Security	50	34.00%	34.00%	36%.00
Actuarial Mathematics	Math/Games & econ.	168	0.00	0.00	0.00
LL(1) Parser Generator	CS/Algorithms CS/PL	205	36.59%	27.32%	37.56%
Schönhage-Strassen Multiplication	CS/ALgorithms Math/Algebra	128	33.59%	27.34%	37.50%
Isabelle_DOF	CS/Semantics & reasoning	41	48.78%	31.71%	53.66%
Interval Analysis	Math/Analysis	694	50.29%	36.89%	52.74%
MFOTL Checker	CS/Data mgmt systems CS/Algorithms Logic/General logic	151	43.05%	30.46%	47.02%
Decomposition of totally ordered hoops	Math/Algebra	1	0.00	0.00	0.00
Approximate Model Counting	CS/Algorithms	112	16.96%	18.75%	21.43%
Wieferich-Kempner Theorem	Math/Number theory	8	0.00	0.00	0.00
PNT_with Remainder	Math/Number theory	155	28.39%	21.29%	32.26%
Continued Fractions	Math/Analysis	428	32.71%	29.67%	37.15%
CondNormReasHOL	Logic/Phil. aspects Logic/General logic	34	11.76%	2.94%	11.76%
Region Quadtrees	CS/Data structures	182	35.71%	34.07%	41.21%
Karatsuba	CS/Algorithms	431	48.03%	30.63%	51.97%
Pick's Theorem	Math/Geometry	334	14.67%	14.07%	16.77%
Kummer Congruence	Math/Number theory	120	34.17%	20.00%	37.50%

A.4. Computing Resources and Details for Replication

We run the majority of our training and evaluation on 8 NVIDIA RTX 2080 Ti (11GB VRAM). We use 8-bit quantization when loading models for both fine-tuning and inference. We train models with DistributedDataParallel (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.

A.5. Results found using greedy decoding vs beam search

Table 6. 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)	27.9%	15.0%	40.9%	25.0%	19.5%	25.7%
LEMMANAID (types)	26.0%	13.6%	44.9%	26.5%	18.4%	28.0%
LEMMANAID (defs)	24.2%	7.3%	36.0%	22.0%	8.9%	26.3%
Neural (types + defs)	24.0%	9.7%	30.3%	20.8%	10.1%	30.6%
Neural (types)	23.4%	10.4%	35.7%	21.7%	11.0%	36.0%
Neural (defs)	21.3%	6.4%	26.0%	20.6%	6.8%	29.4%
LEMMANAID Combined	36.2%	20.5%	55.4%	35.4%	24.7%	43.4%
Neural Combined	31.9%	15.1%	42.3%	30.3%	15.5%	40.3%
Combined	41.1%	25.0%	58.9%	41.0%	29.0%	60.3%
QuickSpec	—	—	22.8%	—	—	22.8%

Table 7. 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)	37.1%	21.6%	50.0%	36.2%	27.3%	44.9%
LEMMANAID (types)	33.4%	22.5%	56.6%	36.4%	24.6%	49.7%
LEMMANAID (defs)	31.3%	10.4%	38.6%	31.2%	13.4%	36.0%
Neural (types + defs)	25.7%	10.4%	23.7%	25.7 %	13.4%	31.4%
Neural (types)	23.6%	13.8%	40.0%	22.9%	11.7%	41.1%
Neural (defs)	21.5%	5.3%	21.1%	25.0%	9.3%	32.0%
LEMMANAID Combined	45.4%	28.5%	67.1%	46.4%	33.2%	64.6%
Neural Combined	37.8%	17.8%	47.1%	36.6%	19.3%	55.4%
Combined	49.3%	30.9%	70.9%	50.4%	36.1%	72.9%
QuickSpec	—	—	22.8%	—	—	22.8%

We break down the results from Table 1 into Table 6 and Table 7 to show the performance of different decoding strategies. The results in Table 6 are for greedy decoding, which selects the token with the highest probability at each generation step. The results in Table 7 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 less inference budget.

A.6. 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 was as follows:

```
== 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)
```

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:

```
== 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))

== 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))
```

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 $Re, Im : \mathbb{C} \to \mathbb{R}$ extract the real and imaginary parts of a number respectively. In

the octonions, there are eight analogous functions $Re, Im_1 \ldots Im_7 : \mathbb{O} \to \mathbb{R}$. 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 $Re(x + Im_2\, y \times z) = Re(x + y \times Im_2\, z)$.

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 are $(e_5 - e_4) \times (e_1 + e_4) = (1 + e_1) \times (e_4 - e_1)$, and $e_4 \times (Im_1\, x \times Im_2\, y) = Im_7(Im_1\, x \times (y \times e_4))$. The problem is that QuickSpec does not attempt to judge which lemmas are relevant to a human user.

A.7. Example Lemmas Discovered by LEMMANAID

There are numerous examples of LEMMANAID predicting correct templates (and then recovering the gold-standard lemmas), but it is sometimes difficult to explain exactly *why* LEMMANAID succeeded when other methods failed. In Octonions, for example, LEMMANAID recovers the following lemma, which neither the neural baseline nor QuickSpec recovers:

```
 lemma Im7_tendsto_lowerbound:
"\<lbrakk>
   (f \<longlongrightarrow> limit) net;
   \<forall>\<^sub>F x in net. b \<le> Im7 (f x);
   net \<noteq> bot
 \<rbrakk>
   \<Longrightarrow> b \<le> Im7 limit"
```

The input (below) associated with this lemma is truncated when sent to the LLM due to our max sequence length, but LEMMANAID is still able to recover the correct template.

```
###symbols
  Orderings.bot_class.bot
  Octonions.octo.Im7
  Filter.eventually
  Topological_Spaces.topological_space_class.tendsto
###defs
class bot =
  fixes bot :: 'a ("\<bottom>")
codatatype octo =
  Octo (Ree: real) (Im1: real) (Im2: real) (Im3: real) (Im4: real)
       (Im5: real) (Im6: real) (Im7: real)
definition eventually :: "('a \<Rightarrow> bool) \<Rightarrow> 'a filter
  \<Rightarrow> bool"
  where "eventually P F \<longleftrightarrow> Rep_filter F P"
class topological_space = "open" +
  assumes open_UNIV [simp, intro]: "open UNIV"
  assumes open_Int [intro]: "open S \<Longrightarrow> open T \<Longrightarrow>
        open (S \<inter> T)"
  assumes open_Union [intro]: "\<forall>S\<in>K. open S \<Longrightarrow>
        open (\<Union>K)"
  ...
```

LEMMANAID's neural engine predicts the following (correct) template:

```
    \<lbrakk>
      ?H1 x_1 x_2 x_3;
      ?H2 (\<lambda>y_0. x_4 \<le> ?H3 (x_1 y_0)) x_3;
      x_3 \<noteq> ?H4
    \<rbrakk>
      \<Longrightarrow> x_4 \<le> ?H3 x_2
```

Templates are typically shorter than their corresponding lemma objects, which may help with better reasoning in some cases for neural models. The following is an example lemma object from HOL-test that our neural baseline could not predict directly:

```
\<lbrakk>
  continuous_map ?X ?Y ?f; compact_space ?X;  Hausdorff_space ?Y ;
  ?f ` topspace ?X = topspace ?Y;  inj_on ?f (topspace ?X)
\<rbrakk>
  \<Longrightarrow> homeomorphic_map ?X ?Y ?f
```

The above lemma object is 230 characters, but the template extracted from this lemma is only 134. The associated template, which LEMMANAID's neural engine is able to correctly predict and then its symbol engine is able to successfully instantiate, looks as follows:

```
\<lbrakk>
  ?H1 x_1 x_2 x_3; ?H2 x_1; ?H3 x_2;
  ?H4 x_3 (?H5 x_1) = ?H6 x_2; ?H7 x_3 (?H5 x_1)
\<rbrakk>
  \<Longrightarrow> ?H8 x_1 x_2 x_3
```

A.8. Successes and failures of LEMMANAID

We provide a few examples of conjecturing problems attempted by LEMMANAID, covering some successful and failed attempts.

Success Let's consider the lemma `BExp.bval_not` which features in `IMP/BExp` theory in `HOL`:

```
lemma bval_not[simp]: "bval (not b) s = (\<not> bval b s)"

|gold-standard| ?H1 (?H2 x_1) x_2 = (\<not> ?H1 x_1 x_2)

|prediction| ?H1 (?H2 x_1) x_2 = (\<not> ?H1 x_1 x_2)
```

LEMMANAID predicts an identical template to the gold-standard, and the template can be successfully instantiated to generate a conjecture which is equivalent to `bval_not`.

Success with Many Symbols LEMMANAID is able to handle conjecturing problems with a lot of symbols, which can be tricky for symbolic tools which exhaustively enumerate conjectures. For example, consider the lemma `logderiv_zeta_region_estimate` from an AFP formalization of prime number theory with remainder term:

```
lemma logderiv_zeta_region_estimate:
  assumes "s \<in> logderiv_zeta_region"
  shows "\<parallel>logderiv zeta s\<parallel> \<le> C\<^sub>2 *
        (ln (\<bar>Im s\<bar> + 3))\<^sup>2"

|gold-standard| x_1 \<in> ?H1 \<Longrightarrow>
?H2 (?H3 ?H4 x_1) \<le> ?H5 ?H6 (?H7 (?H8 (?H9 (?H10 (?H11 x_1))
(?H12 (?H13 ?H14)))))

|prediction| x_1 \\<in> ?H1 \\<Longrightarrow>
?H2 (?H3 ?H4 x_1) \\<le> ?H5 ?H6 (?H7 (?H8 (?H9 (?H10 (?H11 x_1))
(?H12 (?H13 ?H14)))))
```

Generating the correct template here requires successfully relating 14 symbols, and reasoning over their corresponding types and definitions.

Off-target (But Valid) Prediction LEMMANAID sometimes generates helpful templates that differ from gold-standard templates. Such templates are still useful, as they can be instantiated into valid lemmas, but our evaluation considers such templates as misses. Consider attempting to conjecture the lemma `bval_and`:

```
lemma bval_and[simp]: "bval (and b1 b2) s = (bval b1 s \<and> bval b2 s)"

|gold-standard| ?H1 (?H2 x_1 x_2) x_3 = (?H1 x_1 x_3 \<and> ?H1 x_2 x_3)

|prediction| \<lbrakk> ?H1 x_1 x_2; ?H1 x_3 x_2 \<rbrakk> \<Longrightarrow>
?H1 (?H2 x_1 x_3) x_2
```

In this case, the predicted template is instantiated to a different, valid lemma, namely `\<lbrakk> bval b1 s; bval b2 s \<rbrakk> \<Longrightarrow> bval (and b1 b2) s`. This conjectured lemma is weaker lemma than the gold-standard, but it is equivalent to assuming the right-hand side of the equality to prove the left-hand side. Counter-example checking and other property testing techniques could be leveraged in these cases to filter valid candidates.

Invalid Prediction Consider the totient function, which counts the number of positive integers up to an input `n` which are relatively prime to `n`. One property of the totient is:

```
lemma totient_le: "totient n \<le> n"

|gold-standard| ?H1 x_1 \<le> x_1

|prediction| x_1 \\<le> ?H1 x_1
```

In this case, LEMMANAID predicted a similar template to the gold-standard template, though the lemma created from instantiating this template is invalid. Again, counterexample checking can help filter invalid lemma candidates.