Aesop Gets the Fables: Towards a Hybrid Proof Search Using Ingredient Prediction by Language Models

Siddhartha Gadgil
Indian Institute of Science
Bangalore, India
gadgil@iisc.ac.in

Ayush Agrawal
Microsoft Research
Bangalore, India
ayushagrawal1801@gmail.com

Anand Rao Tadipatri
Indian Institute of Science Education and Research
Pune, India
anand.tadipatri@students.iiserpune.ac.in

Navin Goyal
Microsoft Research
Bangalore, India
navingo@microsoft.com

Abstract

We describe a system for a hybrid proof search in Lean 4 based on predicting ingredients: premises, lemmas and instantiations, using language models. The predictions are used with Aesop and other Lean automation for proof searches.

We integrate our system with Lean, with the goal that it can be used ergonomically to complete proofs once they are within the reach of AI powered automation.

We extract ingredient data not only from definitions and theorems, but also from sub-proofs. This gives a large dataset of over a million statements with ingredients. We find that a language model, `codegen-350M-multi`, fine-tuned with our data can predict ingredients with reasonable accuracy. We also analyse frequencies of tactics used in Lean's Mathematical library mathlib; these suggest that our approach is promising.

Keywords Lean theorem prover, interactive theorem provers, language models, proof search.

1 Introduction

A hybrid proof search combines (rule based) algorithms and machine learning to attempt to prove theorems in automated theorem proving systems or to provide automation in interactive theorem proving systems. Algorithms are used for *efficient deduction* based on *unification*, *term rewriting*, *simplification* etc. and for *search heuristics*. Machine learning complements this by suggesting useful *starting points* for deductions and searches.

In this work, we use build a hybrid proof search system for the Lean Interactive Theorem Prover [4], using the rule-based automation of Aesop [10] as well as (directly and indirectly) the automation that is part of Lean or its Mathematical library [11] mathlib such as its *unification* engine, *simplifier*, rewriting based on *discrimination trees* etc. We use language models not only for *premise prediction* (which has been done earlier in many contexts), but more generally *ingredient prediction*. Thus, we use large language models to predict:

- *Premises*: Names of theorems from the library that are likely to be useful in the proof.
- *Lemmas*: Statements of theorems that are likely to be useful in the proof.
- *Instantiations*: Terms in the proof that are not identifiers, proofs or types. These are typically arguments of functions or theorems.
- *Conditions*: Propositions that are sub-terms of a proof and are not identifiers. These can be used as conditions in `if` statements and `by_cases` and similar tactics.

To train language models we extract data from the mathlib library of formalized mathematics, using proof-terms. We use not only theorems and their proofs from the library, but also sub-terms of the proof-terms which are themselves proofs. This gives us a dataset of over a million statements with ingredients. This dataset is in the public domain and can be used for other purposes. Our extraction is based on custom *delaborators* for Lean – delaborators convert kernel expressions to surface syntax, with our custom delaborators recording additional information.

We use the heuristic that sub-expressions that are not represented in the surface syntax (such as `Eq.mp` and `HAdd.hAdd`) are unlikely to be useful in proofs. Hence, for the most part we consider only expressions that are represented in the surface syntax. However, we do retain some information that is suppressed in syntax, such as types of variables in function declaration (in practice this information is often looked up by hovering or in the infoview in interactive Lean development).

While premises are globally defined identifiers, lemmas, instantiations, and conditions depend on a *local context* (we often call this simply the *context*). We extract this as part of the data and train models to predict the context as well as the lemma or instantiation. We train CodeGen350m to predict all the above ingredients.

Premise selection has been used successfully in first-order logic (FOL) provers, as well as some higher-order logic systems like Isabelle. However, in Lean (and similar systems),

it is not obvious how to use premises. Further, our goal is to also use predictions of lemmas, instantiations and conditions. Another component of our work is to build automation using Aesop to use these predictions. To do this we also write some tactics that are useful for this purpose, for example to make use of the predicted local context correctly.

While eventually AI systems may work autonomously, it is likely that in the near future they will be most useful as interactive assistants, in particular completing proofs once the goal is within their capability. To this end, we have implemented code to run searches in the background and cache results, which are provided to the user as suggestions. This code is generic and can be used with any automation (indeed by default it is used with the `aesop` tactic).

By design, this work has many loosely connected components: data extraction, trained prediction models, search using predicted ingredients, new tactics, and background running of searches. Some of these need to be scaled and tuned to be effective. We hope that some of these components can be use in combination with work by others.

1.1 Contributions

To the best of our knowledge, our approach of an *ingredient prediction* based search is novel. Other works either predict tactics directly or predict premises (and in some case commands) from a prior list. We feel ingredient predictions are a good way to use the generative power of language models without demanding good predictions of tactics.

Our approach to extracting data gives a large dataset with over a million entries. As mathlib has a standard style and very little duplication, we believe this is a very good dataset for training language models for theorem proving. Further, this can be used, for instance, to analyse relations between different branches of mathematics.

The rule based automation and integration with Lean may be independently useful.

1.2 Related Work

Language models have been fruitfully employed in various automated theorem proving settings ranging from FOL [1, 2], HOL (`Isabelle`) [5, 7, 8, 12] to very expressive logics like the Calculus of Inductive Constructions (`Lean`) [6, 9, 14, 15]. In [1, 2] resolution based methods use deep learning models such as transformers as clause selection functions. Search based methods involving reinforcement learning such as expert iteration and Alphazero [16] have been proposed in [6, 9, 14]. In these systems, language models are trained to output a proof step given a tactic state and act as a value function for searches. These methods are further proposed to be combined with tools called hammers that aid in proof automation by using the power of external theorem provers [7]. `Sledgehammer` [3] is a widely used hammer in Isabelle. These hammers are further enriched by exploiting deep learning techniques for premise selection [12]. In Isabelle, it is shown that amalgamation of these "neural hammers" along with the neuro-symbolic search techniques achieves remarkable proof rates [12]. In [17], apart from the next proof-step prediction task, premises selected using contrastive learning are used in the prelude to improve the proof search. Premise selection techniques using contrastive learning [18] and classical machine learning [13] are proposed as recommender systems to users. However, the avenues of combining these to enable proof automation are not clear.

Hammers are not yet fully developed for `Lean` owing to its much greater complexity. However, we foresee that `Aesop` [10] has potential to provide crucial proof automation when aided with useful ingredients of the proof. We therefore not only utilize premises but also lemmas, instantiations and conditions in the proofs that we collectively call ingredients. In next proof step prediction task as proposed in [6, 9, 14], a language model has to output the full step constituting correct tactic with correct arguments. A minor mistake (such as direct rewrite in place of reverse rewrite) can result in an invalid step discarded from the search process. The real use of an oracle like LMs comes in predicting "ingredients" that have to be reasoned or introduced ex nihilo. Therefore, we exploit LMs to only output these ingredients and offload their correct usage to a better symbolic automation tactic (`Aesop`) that can take care of using these aptly. We not only focus on backward style proofs as in extant works [9] but also on forward style through the introduction of lemmas. Baldur [5] proposes full proof generation using language models ($8B$), but their adoption could be limited owing to the large compute needed. Further, it is unclear whether full proof generation would be helpful when proving new/complex theorems.

2 Background

Lean 4 [4] is a theorem prover and programming language based on the calculus of constructions with inductive types. Thus, mathematical objects such as numbers and spaces, functions, propositions and proofs are all *terms* in Lean. Each term has a *type*, which is itself a term. Propositions are a special class of types, and the proof of a proposition A is a term a with type A (denoted $a : A$).

2.1 Definitions, Propositions and Proofs

Definitions introduce terms into the context and assign a name to them. Theorems are essentially special definitions, with the type of the term being defined the proposition that is proved and the value a proof of this proposition. Lean's type checking and termination checking ensure that proofs of theorems are correct.

An example of a theorem is in Figure 1, where a theorem with name `exists_infinite_primes` is defined. The type of this theorem is the proposition that there are infinitely many primes, and the value is a proof of this proposition.

```
theorem exists_infinite_primes (n : ) :  p, n  p
    Prime p :=
  let p := minFac (n ! + 1)
  have f1 : n ! + 1  1 := ne_of_gt <| succ_lt_succ
    <| factorial_pos _
  have pp : Prime p := minFac_prime f1
  have np : n  p :=
    le_of_not_ge fun h =>
      have h : p  n ! :=
        dvd_factorial (minFac_pos _) h
      have h : p  1 :=
        (Nat.dvd_add_iff_right h).2 (minFac_dvd _)
      pp.not_dvd_o h
  p, np, pp
```

Figure 1. Proof of infinitude of primes in mathlib

The most significant difference between proofs and arbitrary terms in Lean is that two proofs of the same proposition are equal by definition. Formally, given a proposition `A`, if `x` and `y` both have type `A`, then `x = y` by definition.

2.2 Expressions and Syntax

Terms are represented internally by kernel *expressions*. The strings representing terms (including propositions and proofs) are transformed into such kernel expressions in two steps. In the first step, a *parser* converts the string into surface *Syntax*. In the second step, an *elaborator* converts the syntax into a type-correct kernel expression. Typically, proofs are not explicitly described as terms as in Figure 1, but are generated by a *tactic sequence* as we describe in Section 2.3.1.

Crucially for us, kernel expressions are converted to pretty-printed strings in two steps, reversing the above. First, a *delaborator* converts the kernel expression to syntax, and then a *pretty printer* converts the syntax to a string.

The complexity and details of the kernel expression of a term are dictated by the foundations, and often contain details irrelevant to mathematical understanding or proving. As a thumb rule, we largely ignore components of kernel expressions that are not represented in syntax.

2.3 Ingredients

As mentioned in the introduction, we consider ingredients of various kinds. Indeed, in the proof in Figure 1 the most non-obvious idea is to consider the *term* $n! + 1$ when n is given. This term is not mentioned in the statement of the theorem, but is used in the proof. Premise prediction will not help with this aspect in this case (though we do need to predict premises such as `minFac_prime`), but predicting instantiations will.

Concretely, we consider the following kinds of ingredients:

- *Premises*: Names (identifiers) of theorems from the library that are likely to be useful in the proof.
- *Lemmas*: Statements of theorems that are likely to be useful in the proof.
- *Instantiations*: These are terms in the proof that are not identifiers, proofs or types. These are typically arguments of functions or theorems.
- *Conditions*: These are propositions that are sub-terms of a proof that are not identifiers. These can be used as conditions in `if` statements and `by_cases` and similar tactics.

In training and prediction, we group together instantiations and conditions as these are both non-atomic terms, with a Boolean indicating whether the term is a proposition. It is easy to separate predicted terms into the two groups, and the context information, syntax etc. are similar in the two cases (and we often refer instantiations when we mean both instantiations and conditions, where there is no cause for confusion).

In Example 1, some of the premises are `le_of_not_ge`, `Nat.Prime.not_dv` `Nat.factorial`, `Nat.dvd_factorial` etc.

While premises are globally defined identifiers in the environment of a theorem, lemmas, instantiations and conditions depend on a *local context*. This consists of two kinds of local declarations: introducing variables with specified types and *let* declarations making a local definition. Note that assumptions are simply variables whose type is the proposition being included in the hypotheses.

The only instantiation in Example 1 is `n! + 1`, but this appears in three different contexts (differing in assumptions). Some of the lemmas are: `0 < n!`, in a context including the variable `n`, and `(p  1)  (p  n! + 1)`, in a context including the variable `n` and the let declaration `p :  := Nat.minFac (n! + 1)`.

2.3.1 Tactics and proofs

Most complex Lean proofs are typically generated by a sequence of tactics as, for example, in Figure 2. A tactic is a meta-program which, given a *goal state* returns a new goal state or fails. The goal state consists of a local context, which consists local variables and let declarations, and a goal, which is determined by a type. The goal is the proposition to be proved and is represented by a *meta-variable*. Local variables whose types are propositions are assumptions. A tactic assigns an expression to the current goal meta-variable, in general involving new meta-variables created by the tactic. In particular, a solution of the goals in the new goal state gives a solution of the original goal state.

Formally, a tactic proof is of the form `by tacticSeq`, where `tacticSeq` is a sequence of tactics forming an indented block or with tactics separated by semicolons (or a mix of these). Indeed, `by tacticSeq` is valid *syntax* for a term for any non-empty sequence of tactics `tacticSeq`; however this may not elaborate to a term. To (attempt) to elaborate to a term, we

```
theorem not_bddAbove_setOf_prime : ¬BddAbove { p |
    Prime p } := by
  rw [not_bddAbove_iff]
  intro n
  obtain p, hi, hp := exists_infinite_primes n.succ
  exact p, hp, hi
```

Figure 2. A proof using tactics

form an initial goal state from the expected type, i.e., the statement of the theorem. If a tactic is successful on its given goal state (using the initial goal state for the first tactic), it returns a new goal state which is taken as the goal state for the next tactic. If the tactic sequence is successful, the goal state is empty, and the term is a proof of the theorem.

Some powerful tactics like `simp` and `aesop`, if successful on a goal state, can be replaced by a simpler and more precise tactic sequence. Associated to these are *search tactics* `simp?`, `aesop?`, etc. which provide a *Try this* suggestion if successful on a goal state, suggesting a tactic sequence that can be used instead.

3 Techniques

3.1 Extraction of ingredients

It is straightforward in Lean 4 to obtain definitions, theorems and their proofs in the environment, and hence from mathlib and Lean's core and standard library by importing these into the environment. We exclude definitions and theorems in namespaces corresponding to code for use in the compiler, interpreter, IO etc. We extract ingredients from theorems and their proofs. Further, we also extract ingredients from sub-terms of the proof-terms which are themselves proofs. This gives us a dataset of over a million statements with ingredients. This dataset is in the public domain and can be used for other purposes.

The theorem statements (types) and proofs (terms) are kernel expressions. As kernel expressions have to be valid in the foundations of Lean, they are typically very complicated. In Lean, *delaborators* transform an expression into surface syntax, which is then pretty-printed. Many sub-expressions and identifiers (such as `Eq.mp` and `HAdd.hAdd`) are suppressed in the syntax even if they are present in the kernel expression. We use the heuristic that sub-expressions and identifiers that are not represented in the surface syntax are unlikely to be useful in proofs. Similarly, we only consider subterms of the proof-terms that are represented in the surface syntax; these correspond to syntax *nodes* whose *kind* belongs to the *syntax category* `term`.

The Syntax is a tree-like structure, more precisely an inductive type. Certain leaves correspond to identifiers. Thus, to extract *premises* of a theorem, we can generate syntax for the proof-term using Lean's standard delaborators and extract the identifiers from leaves.

However, the syntax does not contain enough information to extract *lemmas* and *instantiations* with context. More importantly, we do not have information about sub-terms that are proofs. Hence, we use *custom delaborators* that records additional information. These are based on Lean's own delaborators and involve copy-pasting and modifying.

3.1.1 Sub-proofs via delaborators

Concretely, depending on a setting Lean's delaborators can show a sub-term `p` whose type is a proposition `prop` either like a term or as `(_ : prop)` suppressing the details of the proof (which are irrelevant as in Lean two proofs of the same proposition are definitionally equivalent). In our situation, we need to record *both* the proof and the proposition, as also the fact that the term is a proof. For this we introduce special syntax, with a custom elaborator and a custom delaborator for the syntax. While extracting ingredients, we use delaborators including the custom delaborator to generate syntax, and then extract the information from the syntax. We also have to use modified copies of all delaborators to make correct recursive calls.

3.1.2 Context for instantiations and lemmas

Instantiations and lemmas have to be extracted with their local contexts. These are determined by matching with syntax for functions, dependent function types, and let statements. However, the types of variables in these cases are often suppressed in syntax when Lean can deduce them while elaborating the parent expression (and users can hover or look in the infoview for this information). However, the sub-proof may not have enough information to deduce the types, and in any case this suppressing makes the task of the language model harder, as also its output less useful for proof searches. We hence modify the delaborators to include this information.

3.1.3 Named lemmas

Often in Lean, auxiliary lemmas are stated as separate lemmas before the main result and proved. A proof search cannot expect to find these in the context, but must predict their statements. Hence, we include statements of lemmas that are identifiers in a proof as lemmas to be predicted (excluding those in some namespaces associated to the compiler).

3.1.4 Recursive data extraction

With the modified delaborators, extracting ingredients is done recursively from definitions and proofs by constructing a syntax tree and visiting its nodes. The custom syntax and delaborators have enough information to recognize when nodes correspond to sub-proofs and what the proposition and proof are at such nodes.

Training for premises
theorem-statement <PREMISES> premise_1; . . . ; premise_n

Training for lemmas
theorem-statement <LEMMA> lemma_1
theorem-statement <LEMMA> lemma_2
. . .
theorem-statement <LEMMA> lemma_n

Training for terms
theorem-statement <CONTEXT> context <TERM> term_1
theorem-statement <CONTEXT> context <TERM> term_2
. . .
theorem-statement <CONTEXT> context <TERM> term_n

Figure 3. Training statements for premises, lemmas, and terms (instantiations and conditions).

3.2 Training

3.2.1 Data split

To avoid contamination due to overlap between proofs and their sub-proofs, and sub-proofs of the same proof, we split the theorems and definitions of mathlib into three sets, with sub-proofs assigned to the same group as their parent. About 80% of the theorems and definitions were assigned to the training group, with 10% assigned to each of test and validation groups. We then generated sub-proofs and extracted ingredients.

The Training data consisted of about 1.5 million unique theorems while the test and validation data consisted of about 185 thousand and 188 thousand unique theorems.

The formatting of the different training tasks is shown in Figure 3. The training tasks involved predicting premises, lemmas, and terms, with terms including instantiations and conditions. Terms and lemmas have associated to them a (local) context.

Owing to the limited context length of the models, we design the training datapoints in as follows:

- Premises (identifiers) are output as a string, with identifiers separated by semicolons. Thus, each data-point consists of the theorem statement followed (after a token) by a semi-colon separated list of identifiers.
- For a theorem with n lemmas, we have n training data-points, each consisting of the statement followed (after a token) by a lemma statement. The lemma statement includes a context.
- For a theorem with n terms (i.e., instantiations and conditions) we have n data-points, each consisting of the theorem statement, followed by (after a separation token) the context, followed by (after a separation context) the term.

With this design, we get about $8.8M$ training data-points (premises: $\sim 1M$, lemmas: $\sim 6.5M$, terms: $\sim 1.3M$) and about $1.1M$ test data-points (premises: $\sim 0.12M$, lemmas: $\sim 0.81M$, terms: $\sim 0.17M$).

We finetune Codegen (`codegen-350M-multi`) with $350M$ trainable parameters using next token prediction loss. A single model is trained for all the three tasks. The model is trained for 5 epochs with the effective batch size of 64 and the learning rate of $1e-4$. We use Adam's optimizer ($\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 1e-8$) and $fp16$ precision for training. We use 4 A100 $80GB$ GPUs for finetuning.

3.2.2 Inference

Given, a theorem statement and optionally the context, we follow the theorem statements with the appropriate disambiguation tokens: <PREMISES>, <LEMMA> and <TERM>, and seek the completion to get premises, lemma and term predictions. For the Codegen model, we generate 4 outputs using beam search for each of the test datapoints.

3.3 Proof search using ingredients

Our approach to using ingredients is based on an analysis of the frequency of use of various tactics in mathlib as well as our experience formalising in Lean. Thus, we see that most tactic applications use relatively few tactics such as `rw`, `apply`, `simp`, `intro` and `cases`. Most of these correspond to Aesop rules, with the main task being the prediction of correct premises (there is no Aesop rule for rewrite, but we make a custom one). Further, lemmas and instantiations are used both explicitly – in tactics like `have`, `use` and `refine` and implicitly by proving an auxiliary lemma or making an auxiliary definition. Further, in many proofs powerful tactics like `linarith`, `ring`, `norm_cast`, and `norm_num` provide automation to find proofs, so we include these as part of the search configuration. We give more details of a tactic census and conclusions for our approach in Section 4.3, including comparison with tactic-prediction and premise-selection based approaches.

We use Aesop to perform a *best first* proof search using ingredients predicted by our language models. Namely, Aesop *rules* are created based on ingredients, and these are used to search for proofs. Rules are similar to tactics except that applying a rule to a goal state may result in multiple goal states, along with expressions for the metavariable for the original goal in terms of the metavariables(so each new goal state implies the given goal state). A rule, when successfully applied, also gives tactic scripts corresponding to the new goal states, so that application of a given tactic script gives the corresponding goal state.

In practice, as Aesop expects globally defined names to be used in building rules, in some cases instead of associating rules to ingredients we store these, or tactics derived from these, in memory (using `IO.Ref`) and use these in custom rules.

We have implemented various rules to use ingredients in the Aesop-based search. More experimentation is needed to find good heuristics for weights associated to these rules, and to decide whether a given ingredient should be used for a given rule. We describe some of these rules below. In addition to using ingredients suggested by a language model, we use some ingredients given by heuristics from the goal.

Some rules in Aesop allow premises to be used directly. They can be used in the *apply*, *simp* and *forward* rules. In addition, those of appropriate type can be used in the *cases* and *constructor* rules. We use (some of) the predicted premises in these rules.

The most important use of premises is in *rewriting*. Indeed, the `rw` tactic is the most common tactic used in mathlib proofs, accounting for about 20% of the tactic uses. In addition to rewriting the goal, tactics are used to rewrite hypotheses. We implement a rule to give new goal states by rewriting, including reverse rewriting, the goal and each hypothesis by each premise in a list. Thus, we get a new goal state for each legal rewrite of either the goal or a hypothesis by a premise. We further refine this by implementing *targetted rewriting*, where we use Lean's *conv* tactic mode to pick sub-expressions to rewrite, and create a new goal state for each successfully rewritten sub-expression.

To use a term suggested by the language model in the proof search, we introduce it into the local context. Terms in the context are used by Aesop's forward (and destruct) rules, so such terms will be used for instantiations. However, the term should only be introduced if (each declaration in) the suggested context is contained in the local context, with all declarations in the suggested context being present in the local context (it is fine if there are additional local declarations). Further, the suggested term and its context are given as strings by the language model, so need to be parsed and elaborated. We introduce custom tactics to check that the given context is contained in the local context, to parse and elaborate the term, and to introduce the term (with a generated new name) into the local context if we have the required containment of declarations and successful elaboration. We remark that if the context is not contained in the local context, the term is not likely to elaborate, or the elaboration could introduce meta-variables leading to low-level errors.

A second way that specific terms, namely those that are propositions, can be used are with the `by_cases` tactic. This splits into cases where the proposition is true and where it is false. The code for this is a minor modification of the above.

A suggested lemma is via a custom tactic `using`, which factors the goal `A` given a type `B`, giving goals `B` and `B → A`. As with terms, lemmas are suggested with a context, and the statement is unlikely to elaborate unless the local context has the necessary declarations. However, it is natural to consider declarations in the context of the lemma that are not in the local context as additional hypotheses, and to elaborate the type with respect to these declarations introduces as *for all* or *let* declarations. Thus, for lemmas we attempt to parse and elaborate *relative* to the local context, rather than demanding containment of the suggested context in the local context. We could have followed a similar heuristic in the case of terms using *lambda* in place of *for all* but feel that the heuristics we use are more natural.

```
theorem not_bddAbove_setOf_prime : ¬BddAbove { p |
    Prime p } := by
  aided_by auto_tactic do
    rw [not_bddAbove_iff]
    intro n
    obtain p, hi, hp := exists_infinite_primes n.succ
    exact p, hp, hi
```

Figure 4. A proof using auto tactics

3.4 Integration with interactive proving

Various experts[1] have commented that a very useful role of AI in theorem proving is to complete proofs once they are within the reach of AI. Indeed, this will be similar to copilot taking over and supplying the boilerplate once the structure of a program is written, but with the advantage that correctness is checked before making a suggestion. To this end, as part of this project we have implemented a *tactic combinator* which runs a chosen tactic in the background after each tactic step.

We introduce syntax `aided_by auto_tactic do tactics` where `auto_tactic` is a tactic or tactic sequence that is run in the background after each tactic step. This is valid syntax for a tactic even with `tactics` being an empty sequence. An example of such syntax is Figure 4. We also use macros to introduce more convenient syntax, such as `by# tacticSeq` which is essentially equivalent to the above with `auto_tactic` being `aesop`.

In interactive usage, the aided-by block is thus parsed as a tactic and an elaboration is attempted after each tactic in the sequence is typed. At each such stage a *Task* is spawned (to run in the background) to search for a proof using the tactic `auto_tactic`. If the tactic succeeds, the result is cached. When the same stage is elaborated again (as the next tactic is typed) a *Try this* suggestion is provided, as a hyperlink and code action, using the cached result. For example, in Figure 4, at the time a user finishes typing `intro n`, thus giving a syntactically valid block, two previous searches would have been launched earlier, one with no tactics and one with the single tactic `rw [not_bddAbov_iff]`. If either of these have succeeded the user is offered a *Try this* hyperlink and code-action to complete the proof. Otherwise, if goals remaind after `intro n` (as they do in this case) a fresh task is spawned to

[1]For example, Adam Kalai in personal conversations.

seek a proof for this goal state. A short configurable delay (50 milliseconds by default) is introduced for cases where the proof is found quickly, and then the user can continue just as in an ordinary tactic block.

4 Analysis and Results

To assess both the present performance of our system and its potential when scaled up, we generated and analysed data:

- The quality of ingredient prediction in our current language model.
- The extent to which our proof process works given the correct ingredients.
- Use of tactics in mathlib to see the extent to which proofs using them are present in our proof search, and also to compare our approach with two alternative approaches – tactic prediction and premise selection.

4.1 Language model predictions

The goal of our language model is to predict ingredients present in the proof. Additional ingredients predicted by the language model are not especially harmful unless they dilute the ingredients too much. We thus use two measures of the correctness of predictions, *coverage* and *efficiency*; traditional metrics used in NLP are not particularly suitable for our task. Coverage is the fraction of ingredients that are predicted by the language model. Efficiency is the fraction of the ingredients predicted by the language model that are correct. We also measure the fraction of cases where we have full coverage, so the automation has all the ingredient information. For the measures reported here, we are only considering exact matches as strings. In the case of lemmas and terms, it is sufficient to have definitionally equal terms, but we do not consider this here as checking this is computationally expensive (the code for this check has been implemented as part of a related project on autoformalisation).

The following are the results for the test set after the fifth epoch of fine-tuning **codegen-350m**.

- **Premises**
  - Count: 112473
  - Average coverage: 0.66
  - Average efficiency: 0.39
  - Covered: 49430
  - Fraction fully covered: 0.44
- **Terms**
  - Count: 67927
  - Average coverage: 0.73
  - Average efficiency: 0.38
  - Covered: 40948
  - Fraction fully covered: 0.60
- **Lemmas**
  - Count: 98426
  - Average coverage: 0.42
  - Average efficiency: 0.34
  - Covered: 28007
  - Fraction fully covered: 0.28

The average coverage is fairly high. We can hope that this can be improved by using a larger model. The average efficiency is also high compared to what is needed, as most wrong ingredients fail, and fail fast due to techniques like *discrimination trees* used by Lean and Aesop. Indeed, we would like to experiment with trading better coverage for a lower efficiency by tuning hyperparameters.

Learning progress of the LM.

We used the validation data to analyse the learning over time in our model. We used the analysis metrics – coverage and efficiency, as described in Section 4.1 (in our preliminary experiments, we found that validation loss is not a good indicator of the performance of the model). Since inference over the entire validation set is computationally expensive, we used a subset of the validation set. We randomly sampled 1000 datapoints for each of the three ingredient-tasks. We computed the analysis metrics for validation at each epoch as shown in Figure 6. We found that the model is not overfitting to the training data till $5th$ epoch as the validation metrics are improving with the decrease in the training loss. Our tests of the model (reported above) used the model's checkpoint at $5th$ epoch.

Effect of generating more sequences.

We also analysed the effect of increasing the number of return sequences from the language model on coverage at the cost of efficiency. As we did for studying the learning progress, we randomly sampled 1000 datapoints for each of the three ingredient-tasks from the test set. We then computed coverage metrics (average coverage and fraction of the datapoints fully covered) for 4, 8, 16, and 32 return sequences as shown in Figure 7. While we see a modest improvement ($\sim 10 - 11\%$ in both average coverage and fraction of full coverage) for premises, lemmas (55% in average coverage and $\sim 71\%$ in fraction of full coverage) and terms ($\sim 21\%$ in average coverage and $\sim 32\%$ in fraction of full coverage) show major improvements. This could be due to the way we formatted our tasks. Since a single sequence allows the language model to output all the premises, increasing the number of sequences can show decreasing returns. This shows that our hybrid proof search could benefit from more generations if the automation focuses mainly on the plausible ingredients provided amongst the provided ingredients, i.e., the cost of wrong additional suggestions is low (this is the case with rewrites, for example).

On hallucinations.

Language models are infamous for hallucinating i.e. fabricating non-existing outputs. In the context of the premises task, we define hallucination as predicting premises that are

not shown in the training data. On an average, our language model *incorrectly* hallucinates 11% of premises given a theorem statement i.e. these premises are not part of mathlib. Interestingly, around 12% of the correct premises predicted are not shown to the language model during finetuning. These *useful* hallucinations suggest that the language model is able to learn the naming pattern from the training data in addition to outputting relevant premises.

4.2 Proof search with ingredients

As the search with ingredients is complex and needs a lot of tuning based on experimentation, as a start to understanding automation we considered those theorems and definitions whose only premises begin with `Eq` or `Iff` and which had no lemmas, instantiations or conditions. In most cases such premises are generated by tactics and other automation like unification and type-class inference. We hence considered a simple search where the premises were use only with the *apply* rule, both as an unsafe rule and a safe rule.

We found that 78% of the test results were proved, with 75% of these proved without using any ingredients (as we expect to usually be the case for this special situation).

To achieve this success we analysed failures in earlier rounds to improve our search. It turned out that two causes of failure were subtleties of Aesop which improved inference avoided (transparency in introduction rules, simplifications blocking application of unsafe rules). A third was a bug in Aesop which we reported and was fixed[2].

These results are encouraging. While they are perhaps not good enough yet to use in production, we hope that an interactive improvement process like the above will give sufficiently powerful automation (we believe that a 75% rate is enough for a useful tool, but as our benchmark was not for typical cases we hesitate to claim that the automation is adequate).

4.3 Analysis of tactic use in mathlib

A proof search using language models (or some other form of Deep Learning) depends on the quality of predictions and the automation in using them. The quality of prediction is likely to be better if the prediction task is easier; for instance premises will be predicted better than tactics. On the other hand, the automation in using predictions is likely to be better if the task is easier; for instance tactics are easier to use than premises. Thus, we have a trade-off in the choice of ingredients to predict and use. In addition, if simpler ingredients are predicted and used, additional automation may need to be implemented (as we do with custom tactics and aesop rules)

Here we analyse tactics used in mathlib and data of frequencies of premises to understand how to make this trade-off. Specifically, this data suggests that premise selection is not likely to be adequate in Lean, unless much more powerful automation is developed.

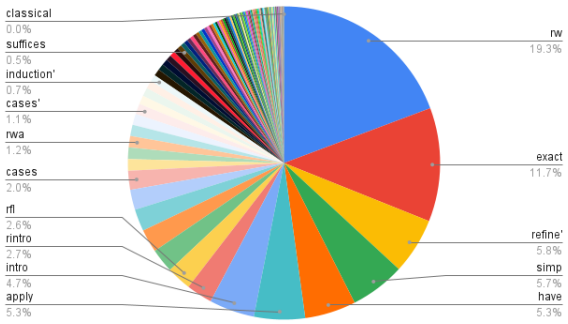


Figure 5. Frequency of tactic use in mathlib

On the other hand, tactic prediction is in some sense arbitrarily general; so perfect tactic predictions can prove anything and close to perfect tactic predictions can prove most things with just a search. But close to perfect tactic prediction is hard. Our analysis below suggests that ingredient prediction is adequate in most cases. Ingredient prediction is likely to be easier than tactic prediction, and indeed our results show that a relatively small model performs fairly well.

The frequencies of tactic calls are plotted in Figure 5 and the top 30 are shown in Table 1. As we see, the top 20 tactics account for 80% of the uses. Moreover, many of these can be rewritten in terms of other tactics. For example, `simpa` is just `simp` followed by `assumption`.

The most common tactic, `rw`, is used with ingredient prediction in our search. Indeed, we use a refined form, so this can be used in even more cases. The tactics `apply`, `intro`, `simp rfl`, `cases`, and `constructor` are naturally integrated into Aesop search with ingredient prediction. A few other powerful tactics, `ext`, `congr` `linarith`, `ring`, `norm_cast`, and `norm_num` are also used in our search by adding tactic rules for a given list of *power tactics* specified as strings (`ext` may in fact be included in aesop's default tactics soon).

Many tactics which we do not use in our search are typically present for brevity and can be readily replaced. The `refine` tactic can usually also be replaced by a sequence of tactics. Note also that `exact` can generally be replaced by `apply`, and in many cases (where we use a complex term) by the `assumption` tactic.

The tactics `have` and `let` introduce lemmas and terms into the local context. In our search, we do this based on predictions of lemmas and terms, which is a major advantage of our approach over pure premise selection. Note that in practice auxiliary lemmas are needed much more than suggested by the frequency of `have` and `let`, as we see by analysing the frequencies of identifiers using our extracted proof data.

Indeed, about 47% of the identifiers were used only in one theorem or definition and about 18% were used twice.

[2] https://github.com/JLimperg/aesop/commit/b6f2b26b6e513b7bcd9ca000ddfde6713cf0cf59

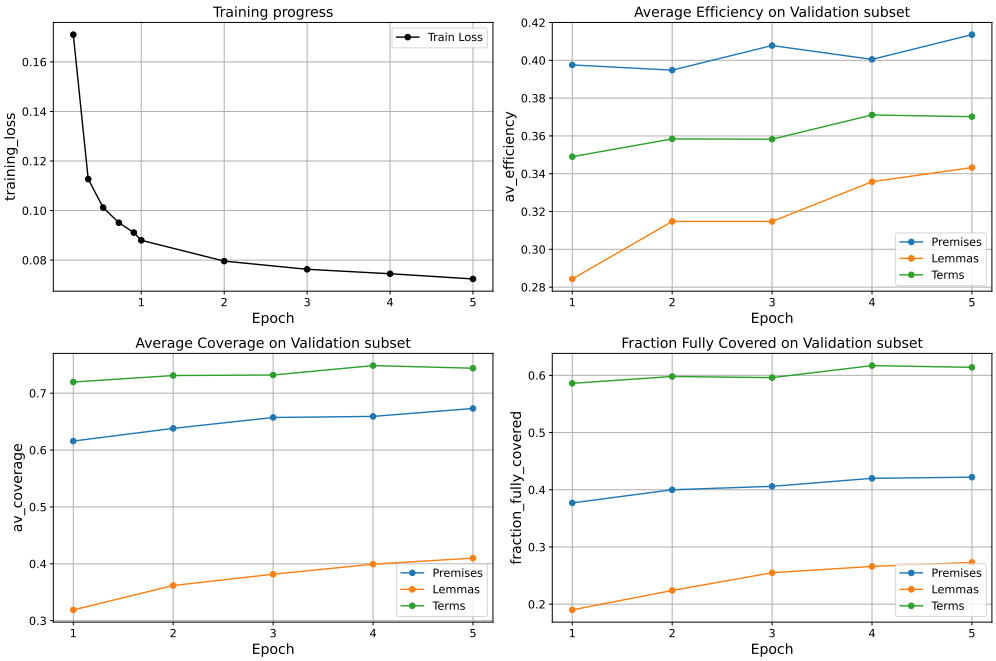


Figure 6. Training loss and validation metrics: all the metrics show a general improvement as the training progresses

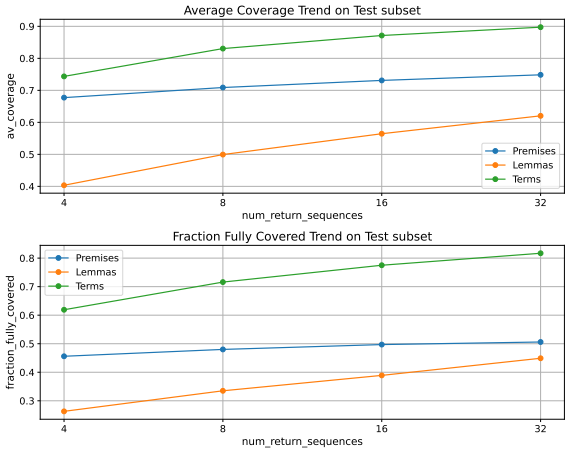


Figure 7. Coverage as the number of return sequences is increased

These are clearly auxiliary statements or definitions, proved or defined for use in a specific theorem (or system generated based on a match in a specific case). It is not reasonable to expect these to be already proved and hence present in the environment, for use with premise selection. Instead, we need to predict these, and use them in the proof search. In our training, we do include named lemmas (as mentioned in Secion 3.1.3) in the lemma prediction data so that the language model can learn to predict both these and lemmas directly included in the proof.

4.3.1 Code illustrations

While we do not as yet have end-to-end demonstrations of our results, the source code in the supplementary material can be run or contains results for a few stages.

Running the commands

```
lake exe cache get
lake exe testpremises Nat.exists_infinite_primes
```

gives as output the data extracted from the proof of infinitude of primes and its sub-proofs. The supplementary material already contains the extracted ingredients in the directory rawdata/premises, with various kinds of ingredients in files named `extra.jsonl` in different subdirectories.

The file `CodeGen/Premiseless.lean` has the successful and failed attempts described in Section 4.2 for the selected set of test premises. In the successful cases the generated proof is filled in and in the failed cases a `#print` statement is given, to facilitate analysis of the failure. This is generated by the program `premiseless.lean`.

The file `background.md` has illustrations of the tactics running in the background. Due to recent upstream breaking of aesop's *Try this*, temporarily the pasted code is misaligned.

To run the full data extraction one can run the program `getpremises.lean`. This, however, takes a few hours on a 32 core machine.

The code for the data structures of ingredients and their serialization is in `LeanAide/PremiseData.lean`. The custom delaborators are in `LeanAde/VerboseDelabs.lean` with some other helpers. The code for extraction on the ingredients is

Table 1. Frequency of tactic use

Tactic	#Used	%Used	Cumulative %
rw	47300	19.31	19.31
exact	76069	11.74	31.05
refine'	90290	5.80	36.85
simp	104191	5.67	42.53
have	117163	5.29	47.82
apply	130083	5.27	53.10
intro	141576	4.69	57.79
rintro	148172	2.69	60.48
rfl	154433	2.56	63.03
ext	160502	2.48	65.51
simpa	166350	2.39	67.90
obtain	171923	2.27	70.17
rcases	177004	2.07	72.25
cases	181809	1.96	74.21
by_cases	184727	1.19	75.40
convert	187645	1.19	76.59
rwa	190538	1.18	77.77
let	193364	1.15	78.92
constructor	196146	1.14	80.06
cases'	198822	1.09	81.15
refine	201036	0.90	82.06
dsimp	203170	0.87	82.93
congr	205167	0.82	83.74
induction'	206931	0.72	84.46
haveI	208678	0.71	85.18
intros	210276	0.65	85.83
erw	211811	0.63	86.45
change	213203	0.57	87.02
use	214516	0.54	87.56
infer_instance	215767	0.51	88.07

in `LeanAide/Premises.lean`. The code for the search using `Aesop` is in `LeanAide/AesopSearch.lean`.

References

[1] Eser Aygün, Zafarali Ahmed, Ankit Anand, Vlad Firoiu, Xavier Glorot, Laurent Orseau, Doina Precup, and Shibl Mourad. 2020. Learning to prove from synthetic theorems. *arXiv preprint arXiv:2006.11259* (2020).

[2] Eser Aygün, Ankit Anand, Laurent Orseau, Xavier Glorot, Stephen M Mcaleer, Vlad Firoiu, Lei M Zhang, Doina Precup, and Shibl Mourad. 2022. Proving theorems using incremental learning and hindsight experience replay. In *International Conference on Machine Learning*. PMLR, 1198–1210.

[3] Jasmin Christian Blanchette and Lawrence C Paulson. 2013. Hammering Away. *A Users Guide to Sledgehammer for Isabelle/HOL. url: http://isabelle. in. tum. de/dist/Isabelle2013-2/doc/sledgehammer. pdf* (2013).

[4] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12699)*, André Platzer and Geoff Sutcliffe (Eds.). Springer, 625–635. https://doi.org/10.1007/978-3-030-79876-5_37

[5] Emily First, Markus N Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: whole-proof generation and repair with large language models. *arXiv preprint arXiv:2303.04910* (2023).

[6] Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward Ayers, and Stanislas Polu. 2022. Proof Artifact Co-Training for Theorem Proving with Language Models. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rpxJc9j04U

[7] 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. *Advances in Neural Information Processing Systems* 35 (2022), 8360–8373.

[8] Albert Qiaochu Jiang, Sean Welleck, Jin Peng Zhou, Timothee Lacroix, Jiacheng Liu, Wenda Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. 2022. Draft, Sketch, and Prove: Guiding Formal Theorem Provers with Informal Proofs. In *The Eleventh International Conference on Learning Representations*.

[9] Guillaume Lample, Timothee Lacroix, Marie-Anne Lachaux, Aurelien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. 2022. Hypertree proof search for neural theorem proving. *Advances in Neural Information Processing Systems* 35 (2022), 26337–26349.

[10] Jannis Limperg and Asta Halkjær From. 2023. Aesop: White-Box Best-First Proof Search for Lean. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Boston, MA, USA) *(CPP 2023)*. Association for Computing Machinery, New York, NY, USA, 253266. https://doi.org/10.1145/3573105.3575671

[11] The mathlib community. 2020. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, Jasmin Blanchette and Catalin Hritcu (Eds.). ACM, 367–381. https://doi.org/10.1145/3372885.3373824

[12] Maciej Mikuła, Szymon Antoniak, Szymon Tworkowski, Albert Qiaochu Jiang, Jin Peng Zhou, Christian Szegedy, Łukasz Kuciński, Piotr Miłoś, and Yuhuai Wu. 2023. Magnushammer: A transformer-based approach to premise selection. *arXiv preprint arXiv:2303.04488* (2023).

[13] Bartosz Piotrowski, Ramon Fernández Mir, and Edward Ayers. 2023. Machine-Learned Premise Selection for Lean. arXiv:2304.00994 [cs.AI]

[14] Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. 2022. Formal Mathematics Statement Curriculum Learning. In *The Eleventh International Conference on Learning Representations*.

[15] Stanislas Polu and Ilya Sutskever. 2020. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393* (2020).

[16] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362, 6419 (2018), 1140–1144. https://doi.org/10.1126/science.aar6404 arXiv:https://www.science.org/doi/pdf/10.1126/science.aar6404

[17] Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. 2023. LeanDojo: Theorem Proving with Retrieval-Augmented Language Models. *arXiv preprint arXiv:2306.15626* (2023).

[18] Eric Yeh, Briland Hitaj, Sam Owre, Maena Quemener, and Natarajan Shankar. 2023. CoProver: A Recommender System for Proof Construction. *arXiv preprint arXiv:2304.10486* (2023).