Abstract

In this work, we propose to train a foundation model (or transformer model) from human Coq scripts to learn to make abstractions like the ones humans make. Then using this model (capable of creating abstractions from code) we propose a method to use it to automatically extract a DSL that compresses a given Coq code data set the most. This compression objective is chosen because smaller terms are easier to synthesize in general – a property we conjecture would help most at synthesis time when predicting helper lemmas (or helper terms). We hope to evaluate 1. The ability of a synthesizer to complete partial proofs using helper lemmas using a learned DSL from data 2. The ability of the model to reason compositionally and 3. The ability of the synthesizer to learn common human heuristics to synthesize terms.

1 Introduction

Mathematics is a powerful tool that has benefited humanity for millennia. Proficiency in this subject has far-reaching implications for society because physics, mechanical engineering, computer science, statistics, machine learning, and the scientific method itself depend on it. For example, being able to automate mathematical proofs would allow us to create safe and interpretable A.I. because a system with provable guarantees is a system we understand and can trust. Therefore in this proposal, we suggest a novel way of implementing the concept of sketching for automated theorem provers using powerful ideas from cognitive science and artificial neural networks; these ideas are; compositionality, learning-to-learn, learning as model building and curriculum-learn.

The principle of compositionality highlights that human knowledge, learning and reasoning are all organized as reusable (high-level) concepts. Research in cognitive science suggests that compositionality is crucial to learning because it enables efficient reasoning. If an intelligent system (artificial or not) processes concepts blindly at a too low level of granularity, it can suffer from combinatorial explosion as there are simply too many possible solutions to naively check. If, however, the system is able to identify the main and re-occurring high-level concepts, it can benefit from the abstraction of unimportant low-level details -- and thus reach a conclusion more efficiently. I propose to implement this with sketching where a sketcher neural network learns to outline high-level concepts such that a synthesizer can efficiently fill out the low-level detail efficiently. In addition, with modern meta-learning techniques, the system can be made fully differentiable.

Another observation from cognitive science is that humans are able to learn rich representations from few examples and leverage past experience in future learning tasks. Thus, with modern learning-to-learn approaches, one can accelerate the learning abilities of these cognitively inspired theorem provers. In combination with compositionality, learning-to-learn further enables fast learning via shared representations. This also avoids repetitive re-learning in the system and allows for greater power of generalization since it’s able to use shared ideas in different scenarios.

One of the most powerful ways that humans learn is by using structured data (like textbooks, Wikipedia, etc.) that already outline what the concepts are, therefore humans learn incrementally without having to discover everything from scratch. We suggest that new high-level concepts can be learned incrementally in the curriculum-learning paradigm. In the domain of theorem proving, not only are there thousands of textbooks outlining the order but there are also mathematical libraries with proofs already in program form ready to be used in this learning paradigm. We suggest that this outline of the concepts is a source of semi-supervised data for the system to learn the compositionality of mathematics.

The last promising idea from cognitive science we will outline is Learning as Model Building. One of the reasons humans are so effective at learning programming and mathematics is because they learn beyond pattern recognition. They are able to build intuitive models that capture relationships and similarities between concepts. In other words, when people think about solving a new unseen problem they do not go through all the formal rules and “deduce the solution”. Instead, we reason by analogies and predictions from the model we have built about the (real or mathematical) world. Those predictions simulate the world and sketch the solution. This is exactly what the proposed sketcher would implement. In addition, with the unambiguous reward of provability, these systems can be trained in a self-supervised manner with reinforcement learning.

In this proposal, we will focus on compositionality. We propose to explore the ability of a foundation model to learn to reason compositionally by training it to generate abstractions from human ones and induce a Domain Specific Language (DSL). The DSL is the main avenue where the model will in the future re-use compositionally the abstractions it learned to generate. The hope is that the model is able to effectively reason compositionally by learning how humans reason compositionally through their abstractions. We propose to evaluate the model’s ability to reason compositionally by its ability to synthesize helper terms and lemmas in partial proofs in Coq. This is the ideal benchmark because the ability to synthesize terms in any mathematics and code is rich in examples of abstraction re-usability.

2 Methods

2.1 Problem Statement: Helper Term Synthesis

Recall that the goal is to explore how foundation models learn compositionally via their creation and use of abstractions, DSLs, and the ability to synthesize terms in Coq. Most importantly we want the synthesized helper terms to be useful in completing proofs. One possible application of such a synthesizer is to aid a prover that gets stuck in a proof but requires might make progress with the conjecture of a helper term (or lemma). Thus, we propose to pre-train the model to predict holes in partial proofs. Thereafter -- due to the importance of DSL in synthesis -- we propose to use the model in three steps: 1. First train the model to generate abstraction from human abstraction 2. After the model has been trained to generate abstraction to use it to induce its own DSL, and 3. To evaluate the system by completing partial proofs by using the DSL it induced. Overall, the goal will be to assess the capabilities of a model to induce and use a DSL in the synthesis of helper terms for partial proofs.

We define a synthesis task as follows: the ability of the model to synthesize terms in partial proofs. Therefore, a data point in our task is as follows:

Where is the target theorem to be proven, is the proof state before a helper lemma was needed and is the partial proof term up to the location before the helper term was needed. Then model would be trained to synthesize the helper term . This data set would be extracted by 1. executing tactic script up until the proof required the synthesis of a helper term e.g., if it’s explicitly written or as an identifier 2. Executing tactic scripts up to every tactic. Note we decided to exclude the partial tactic script because we conjecture that all the information in the partial proof is encoded in the partial proof term. In addition, it allows our model to conjecture proof term holes without the need for a tactic-based prover.

We propose to pre-train a foundation model with the above task before training it to generate abstractions. We also propose that the model is evaluated with the above task with a DSL that it generates on its own. More details will be described in the evaluation section.

2.2 Abstraction Training: an end-to-end learning method to teach a model to generator abstractions

Humans rely heavily on re-usability via shared libraries in the generation of any type of code. In addition, without such a library synthesis of many formal languages is intractable – as previous program synthesis work has heavily relied on handcrafted DSLs. Therefore, evidence suggests that a synthesis system would benefit greatly from the creation of its own abstractions and library of re-usable code via a DSL. We propose to do this by teaching a system explicitly to learn to create abstractions from human ones with a foundation model. We suggest to model Coq code with the standard BPE tokenization [TODO: ask Christian about the type of tokenization he uses for skip-trees]. This approximately means that the model would use common substrings as tokens.

We proceed to explain our novel task for training our model to learn to generate abstractions. We propose to **train a foundation model to predict what subsequence in its input to mask as the abstraction and use the ground truth of human abstractions as the training mask**. Note that this is the inverse task of skip-tree training. Skip-trees training masks part of the input to the foundation model and it’s tasked with predicting it. We propose instead to do the reverse – to give the entire term and the model must predict what subsequence to mask. We’d like to emphasize that the masks for abstraction training came from human code. The hope is that the model would therefore learn to make abstractions on its own by learning from human abstractions. We also suggest predicting the content and the identifier name of the abstraction whenever it is available. We call this training procedure abstraction training and call a foundation model trained in this fashion an Abstraction Generator (AbsGen).

We provide a visualization of the abstraction training task in figure 1. We also provide the original figure from the skip-tree training from the original paper in the appendix to ease the comparison with our proposed method.

Diagram

Description automatically generated

Figure 1: Visualization of the training for end-to-end abstraction learning. The number 1 indicates what abstraction the model has to predict indicated by the word mask. The number 2 indicates the content inside the mask (abstraction) the model has to predict. The number 3 indicates the name of the identifier the model has to predict. Blue colors indicate the term represented as a sequence. The green indicates the same decoder used for each task. The orange indicates the encoder model. For the decoder to know what task (abstraction, generation of term, or ID prediction it has to solve) the embedding of the task is given <abs>, <gen>, or <id>. Alternatively, we could use three decoders (or something else).

2.3 Generating the Data Set

Recall the goal is to train a foundation model to learn to reason compositionally by inducing its own DSL and using it at test time for the synthesis of helper terms for partial proofs. We propose to use human abstractions as data to train our model as proposed in figure 1. Concretely, given a set of Coq scripts, for each defined identifier, we will locate its use in the scripts and extract the surrounding term it was used in to generate an example for abstraction training. For example, if the identifier and *were* used in we would first inline the definitions of the two abstractions to generate the entire input term and then task the model to predict what tokens in the input were used as abstractions by the human and remaining tasks as in figure 1.

For future work, we propose to think about how to use entire proof terms from completed proofs to generate additional examples for abstraction training. But currently propose to use helper terms extracted from partial execution of scripts as pre-training data.

2.4 DSL Generation from an Abstractor Generator

After training a model using our abstraction training to obtain an Abstraction Generator we generate a candidate DSL from data. Recall the goal is to obtain an Abstractor Generator capable of generating abstractions that are useful in proof completion. This is implicitly obtained because the model was trained on abstractions already useful in completed human proofs. In addition, proof completion likely becomes easier if synthesis becomes easier. Thus, we propose to generate a DSL from the Abstractor Generator that compresses the entire training and validation data the most. Compression improves synthesis because the model is tasked with generating simpler terms and leveraging the re-usability of common terms in the DSL. The way we propose to do this is by passing each term from the train and validation data mode to the model as in figure 1 but in inference mode. This only generates abstractions from each term passed and we would not train the model. Then we choose the top K abstractions that compress the data the most according to the following objective:

This simply says that given a set of terms S (our top K) compute the size of the term l in the validation and train set. The validation set is important to encourage abstractions that generalize to unseen term proofs. The final term in the above equation indicates the inclusion of the target theorems themselves from training (and proof terms from our partial proof data set) to encourage further generalization of the abstractions we generate. For preliminary results, we can optimize the above objective for the top K terms that minimize it using classical subset selection algorithms.

After the generation, the DSL generator model must be fine-tuned using the new DSL since the new abstractions were not present in the initial training.

For future work, we hope to have a ranking function trained from data to predict which DSL might be good (e.g. with a position evaluator) and a neural combinatorial optimizer for DSL selection.

3 Evaluation

Recall a synthesizer can be helpful when aiding a prover in making progress in a proof via the prediction of helper terms or lemmas. Therefore, the evaluation will have the same format as the training data as described in section 2.1 -- but with an unseen set of theorem with partial proofs. In addition, we want to assess how good the DSL generated from our abstraction generator is in aiding in completing partial proofs to the synthesizer. Therefore, we will test and evaluate the synthesizer in completing partial proofs with different types of DSLs:

1. The synthesizer with no DSL
2. The synthesizer with a DSL generated from the Abstractor Generator (using the train and validation set)
3. The synthesizer with a DSL generated using handcrafted DSL by humans e.g. choosing a DSL based a subset of the library the human used

We plan to report both

1. The number of proofs that were closed with the synthesizer’s aid
2. And the number of partial proofs holes that were filled successfully

The first one measure the synthesizer's ability to truly aid in completing entire proofs and the second is the potential usefulness of the synthesizer in moving proofs forward.

In addition, we conjecture that to build a good library of abstractions humans have to implicitly know how to identify equivalences. Furthermore, foundation models seem to have surprising emergent behaviors that they have not been explicitly trained on. Therefore, we also propose to

1. Design a test to measure the model’s ability to learn equivalences
2. The model’s ability to learn human heuristics to synthesize terms

Number one is still open and to be decided on how to measure. To measure the synthesizer’s ability to learn heuristics we can construct a data set of (seen & unseen) partial proofs that the synthesizer has to complete and then qualitatively inspect the structure of the terms the model synthesized against the ones humans used.

For future work, we also think it will be fascinating to evaluate the synthesizer's ability to meta-learn and create useful DSL in a few-shot learning scenario. This can be done similar to how few-shot learning is done in vision. The test set is instead divided into different tasks (e.g. organized by subjects, algebra, analysis, compilers, etc.). Instead, divide those into two sets (usually called the train/support set and test/query set). Then during testing allow the model to use the support set for adaptation to generate a DSL not only based on the full training set but with a small sample from the target domain. Then proceed with the evaluation of completion of partial proofs as usual but on the query set for each task. Then report the usual two errors previously reported but across all the tasks.

Timeline

We predict that generating the partial proof data set would take about 2 weeks. To generate the abstraction data set (from human abstractions) would take about 2-3 weeks. This one seems more subtle in identifying abstractions used across projects which might be more nuanced and with more unknowns. Then to implement the entire pipeline and test it should take 2 weeks. To train the model should be around 2-4 weeks depending on the size of the data set and model. The evaluation should be done constantly every time we get a model so this time is included in the training. If we decide to extend the generator to use beam search as a synthesis method it likely would take one more week.

Related Work

Related work on synthesis for theorem proving is Gpt-f and skip trees. The main difference with those methods is that they do not attempt to train their transformer model to generate abstractions in any way. Refactor is a paper by Tony Wu et al. that focuses on “Human mathematicians are often good at recognizing modular and reusable theorems that make complex mathematical results within reach” which is related but have not read it in enough detail to know the difference at this point. Last is EC^2 and DreamCoder which focus on synthesis and DSL learning. Their focus is on learning from scratch instead leveraging a (large) seed data set from humans. They also do not use foundation models and focus on more classical techniques overall – especially in the DSL learning (where they use version spaces and fragment grammars).

Appendix

Skip-Tree training

Diagram

Description automatically generated

Figure X: we provide the figure for skip-tree training as a reference to more easily contrast it with our proposed end-to-end abstraction training.

Discussion

1. Less data of human abstraction vs skip-tree which can mask whatever randomly
2. How to provide more data for abstraction training? Note: this is not an issue for other languages or Codex – since they can use all of GitHub as data.

Cognitive Biased inspired baseline

TODO: model we discussed based on 4 compression steps:

* Compress DSL
  + A.
  + B.
  + C.
  + D.

Possible objective: