DEEP LEARNING RECOMMENDATIONS FOR THE ACL2 INTERACTIVE THEOREM PROVER

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Kyle Thompson

June 2023
ABSTRACT

Deep Learning Recommendations for the ACL2 Interactive Theorem Prover

Kyle Thompson

Due to the difficulty of obtaining formal proofs, there is increasing interest in partially or completely automating proof search in interactive theorem provers. Despite being a theorem prover with an active community and plentiful corpus of 170,000+ theorems, no deep learning system currently exists to help automate theorem proving in ACL2. We have developed a machine learning system that generates recommendations to automatically complete proofs. We show that our system benefits from the copy mechanism introduced in the context of program repair. We make our system directly accessible from within ACL2 and use this interface to evaluate our system in a realistic theorem proving environment.
ACKNOWLEDGMENTS

Thanks to:

- Professor Dennis Sun for superb guidance over the past year and a half. This thesis would not be possible without him.

- Professor Lubomir Stanchev and Jonathan Ventura for serving on my thesis committee and helping me develop through coursework and projects.

- Dr. Eric Smith for leading the project. Dr. Smith also wrote the ACL2 end of the advice tool which makes our system accessible to ACL2 users.

- The Defense Advanced Research Projects Agency (DARPA) who funded the Proof Engineering, Adaptation, Repair, and Learning for Software (PEARLS) project.

- Professor Ruben Gamboa helping write the Data Collection chapter of this thesis, and working with Professor Matt Kaufmann to curate a dataset of broken ACL2 proofs.

- Eric McCarthy and Jeff Kitson for providing the systems support needed to create the HTTP interface between ACL2 and our system.

- My parents, Leigh and Kirk who have given me tremendous love and support.

- Many other family members and friends that have also loved and supported me during my work on this thesis.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF TABLES</th>
<th>viii</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>ix</td>
</tr>
</tbody>
</table>

## CHAPTER

1. **Executive Summary** ............................................. 1
2. **Background** .................................................... 4
3. **Literature Review** ............................................... 11
4. **Data Generation** ................................................ 13
   - 4.1 Removing Hypotheses ....................................... 14
   - 4.2 Removing Hints ................................................ 15
   - 4.3 Removing Books ............................................. 16
   - 4.4 Removing Definitions and Theorems ....................... 17
   - 4.5 Partitioning .................................................. 17
5. **Modeling** .......................................................... 19
   - 5.1 Simple Example ............................................. 19
   - 5.2 High Level Architecture ................................... 21
   - 5.3 Input Representation ....................................... 23
   - 5.4 Output Representation ..................................... 26
6. **Results** .......................................................... 28
   - 6.1 RQ1: Do copies help? ....................................... 28
   - 6.2 RQ2: How about real world performance? .................. 31
   - 6.3 Additional Experiments ..................................... 33
      - 6.3.1 Subtokenization ......................................... 33
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Number of Failed Proof Attempts During Data Collection</td>
<td>14</td>
</tr>
<tr>
<td>4.2</td>
<td>Training Pairs By Hint Type</td>
<td>16</td>
</tr>
<tr>
<td>6.1</td>
<td>Testing Set Accuracy Comparison</td>
<td>30</td>
</tr>
<tr>
<td>6.2</td>
<td>Testing Set Accuracy Comparison</td>
<td>34</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>ACL2 Proof Attempt Tree</td>
<td>8</td>
</tr>
<tr>
<td>5.1</td>
<td>Graph2ToCoppo High Level Architecture</td>
<td>22</td>
</tr>
<tr>
<td>5.2</td>
<td>Tree Representation of a Translated ACL2 Expression</td>
<td>23</td>
</tr>
<tr>
<td>5.3</td>
<td>Graph Representation of a Translated ACL2 Expression with Subtokenization</td>
<td>25</td>
</tr>
<tr>
<td>5.4</td>
<td>Final Input Representation</td>
<td>25</td>
</tr>
<tr>
<td>5.5</td>
<td>Output Copy Generation</td>
<td>26</td>
</tr>
<tr>
<td>6.1</td>
<td>Graph2ToCoppo vs Graph2To Validation Accuracy over Time</td>
<td>30</td>
</tr>
<tr>
<td>6.2</td>
<td>Validation Accuracy of Graph2ToCoppo by Subtokenization Method over Time</td>
<td>35</td>
</tr>
<tr>
<td>6.3</td>
<td>Accuracy of Graph2ToCoppo (GGNN) by Action Type</td>
<td>36</td>
</tr>
<tr>
<td>6.4</td>
<td>Accuracy of Graph2ToCoppo (GGNN) by Number of Tokens in Ground Truth Fix</td>
<td>37</td>
</tr>
</tbody>
</table>
Chapter 1

EXECUTIVE SUMMARY

Formal proofs are proofs written in such detail that they are machine-checkable. That is, there is an algorithm that can verify that each step of the proof is logically sound. A formal proof of a theorem provides the utmost assurance of its validity. Thus, these proofs are highly desirable both for mathematical results, and safety-critical systems. Formal proofs of complicated theorems are mainly generated via Interactive Theorem Provers (ITPs). ITPs give the human a great deal of control over the search for a proof as opposed to Automated Theorem Provers (ATPs) which require little to no human interaction. Writing formal proofs using ITPs requires deep expertise and large amounts of labor. For example, David Russinoff’s verification of the the division, multiplication, and square root algorithms on the AMD-K7 microprocessor in ACL2 required five months of work, 250 definitions, and 3000 lemmas [39]. There are even more extreme examples in other theorem provers such as six year long effort to prove the Feit-Thompson Theorem in Coq [11], or the proof of the Kepler Conjecture in HOL which took over 20 person years [12].

Multiple works have investigated whether or not machine-learning can be used to address these challenges in the context of many different proof assistants with encouraging success. Machine learning has advanced the number of theorems that can be proven automatically in Coq [47, 8, 40, 41], Isabelle [31, 46, 20, 9], HOL Light [4, 34], HOL4 [10], Lean [13, 35], and Metamath [35]. Such systems use a wide variety of machine learning architectures but normally include a decoder to enable the system to produce possibly complicated proof steps. The decoder might grow an abstract syntax tree [47, 8, 41] or can produce a sequence of tokens [31, 36, 35].
In this work, we bring ideas from the literature on program repair to theorem proving. Specifically, we are interested in the copy mechanism introduced with Graph2Diff neural networks [44]. The copy mechanism allows a model to copy a symbol from the input and place it directly in the output even if the model did not encounter the symbol while training. This copy mechanism addresses the tendency for source code to include rare identifiers that often lead to out of vocabulary tokens or weak signals to the model [2, 15, 27]. Since ITPs are typically built atop of programming languages, modeling theorem proving inherits the difficulties of modeling source code. We show that a decoder with a copy mechanism outperforms a decoder that simply produces a sequence of tokens.

We construct our theorem proving system in ACL2 because it has a very large corpus of theorems. In the ACL2 community books, there are over 170,000 theorems. The only proof assistant with more theorems is Isabelle, where the Archive of Formal Proofs dataset contains 204,000 theorems [31]. Additionally, ACL2 makes an effort to prove simple goals automatically without the user having to write an explicit proof. We conjecture that this makes proofs shorter than other proof assistants on average and thus more amenable to proof generation through machine learning. To construct a dataset to train our models, we systematically and automatically break the 170,000 theorems from the ACL2 community books. When we break a theorem, we know that the inverse of the action used to break the theorem must fix the theorem. Therefore, we can train our model on a corpus of broken theorems and their corresponding fixes. Our system is available to ACL2 users through an HTTP machine interface. Through this interface, ACL2 users can automatically attempt to complete a broken proof using our system with just two lines of ACL2 code. We call this interface the advice tool.
We provide two evaluations of our system. In one evaluation, we query the model on each example in a hold-out set and determine if the model recommends the exact same fix that is given in the example. We use this evaluation to show that copies can improve performance of machine learning systems for theorem proving. Though this evaluation is simple to conduct, it does not give our system full credit because multiple fixes may complete a broken theorem. In a more real-world evaluation, we use the advice tool to test how many broken theorems from a hold-out testing set our system can prove in an actual ACL2 session.

The contributions of this thesis are:

- The first deep-learning system in ACL2 to automate the process of writing proofs.
- A method of generating a large corpus of training examples by breaking the theorems in the ACL2 community books.
- A demonstration that a copy mechanism [44] can be beneficial for theorem proving.
- A user friendly tool for accessing and evaluating our models within an ACL2 session.

The remainder of this thesis is organized as follows. We give background information about formal proofs and ACL2 in 2. We discuss other systems that use deep learning to automate theorem proving in Chapter 3. In Chapter 4, we describe how we extract training data from the theorems in ACL2’s community books. Then in Chapter 5 we describe how we use out training data to create a generative model for proof advice. In Chapter 6 we evaluate the performance of our model on theorems from a held-out testing set. We make some final remarks about this work in Chapter 7.
Chapter 2

BACKGROUND

This chapter introduces formal proofs and ACL2 in detail. Recall that a formal proof of a theorem is a machine-checkable sequence of explicit proof techniques applied to the theorem. To understand the difference between an informal proof and a formal proof, let us consider proofs of the following theorem.

**Odd + Odd is Even**

For all natural numbers, $x$ and $y$, if both $x$ and $y$ are odd, then their sum is even.

First, we can examine the properties of an informal proof of this theorem.

**Informal Proof of Odd + Odd is Even**

1. We need to show that $x + y$ is even.
2. If $x$ is odd, then there exists an natural number $k_x$ such that $x = 2k_x + 1$.
3. If $y$ is odd, then there exists an natural number $k_y$ such that $y = 2k_y + 1$.
4. To show that $x + y$ is even, it suffices to show that $2k_x + 1 + 2k_y + 1$ is even.
5. By simplifying the expression, we have to show that $2(k_x + k_y + 1)$ is even.
6. Since $k_x + k_y + 1$ is a natural number, $x + y$ is even.

This informal proof gives a convincing argument that the sum of two odd natural numbers is even. However, if we wanted to make certain that this proof is correct, there are a number of questions we should ask.
• What is the definition of odd? How can we be sure that \( k_x \) and \( k_y \) actually exist?

• How can we be sure that \( 2k_x + 1 + 2k_y + 1 \) actually simplifies to \( 2(k_x + k_y + 1) \)?

• What does it mean to be even? How can we be sure that \( 2(k_x + k_y + 1) \) is actually even?

Most people with even a basic background in mathematics would consider the answers to these questions to be obvious. Thus, they would be satisfied by the informal proof. However, to prove this theorem in a manner that is completely unambiguous, we could use an ITP. Coq is one example of an ITP. First, I present a formal Coq proof of the theorem “Odd + Odd is Even” because Coq proofs are more human-readable than ACL2 proofs. Then, I will show the corresponding ACL2 proof. Note that while Coq proofs are more readable than ACL2 proofs, the proof given below is still very hard to follow without being in a live Coq session.

### Formal Proof of Odd + Odd is Even

**Definition** even\(_2\) (\( n : \text{nat} \)) := \( \exists k, n = 2 \times k \).

**Definition** odd\(_2\) (\( n : \text{nat} \)) := \( \exists k, n = 2 \times k + 1 \).

**Theorem** even\(_{\text{sum}}\): \( \forall (x y : \text{nat}), \)

\( \text{odd}_2 \ x \rightarrow \text{odd}_2 \ y \rightarrow \text{even}_2 \ (x + y) \).

**Proof.**

intros \( x \ y \) \( Hx \ Hy \).

destruct \( Hx \) as \([kx \ Hx] \).

destruct \( Hy \) as \([ky \ Hy] \).

rewrite \( Hx \). rewrite \( Hy \).

\( \exists (kx + ky + 1) \).

simpl. rewrite add\(_{0\ r} \). rewrite add\(_{0\ r} \).
The most obvious difference between the Coq proof and the informal proof is that the
Coq proof is much longer. The Coq proof is longer because it unambiguously shows
that the theorem “Odd + Odd is Even” is true. For example, recall our critique of the
informal proof that it assumes that $2k_x + 2k_y + 2$ simplifies to $2(k_x + k_y + 1)$. In the
corresponding Coq proof, this simplification is carried out using the associative and
commutative properties of addition. These properties themselves had to be proven
before they could be used in this proof.

Now, let us consider what the statement of this theorem might look like in ACL2.
ACL2 is a subset of Common Lisp. This mean’s that in ACL2, when we write function
applications, we write $(f x)$ instead of $f(x)$. As an example, in a language like Java,
we might call a function $foo$ with $foo(1, 2, 3)$. In ACL2 we would instead write
$(foo 1 2 3)$. Furthermore, maybe there is a function $bar$ that takes two arguments.
The expression $(foo (bar 1 2) (bar 3 4) (bar 5 6))$ would apply the function
$foo$ to three arguments defined by the three separate calls to $bar$. With this in mind,
let us examine the statement of the theorem “Odd + Odd is Even” in ACL2.
In general, ACL2 theorems have the form `(defthm <theorem name> <theorem body>)`. The name of our theorem is `even-sum`. It’s body is an implication. The implication can be written in more familiar logical notation as:

\[
\text{integerp}(x) \land \text{integerp}(y) \land \neg\text{evenp}(x) \land \neg\text{evenp}(y) \rightarrow \text{evenp}(x + y)
\]

The antecedent, or left side, of the implication is a conjunction of hypotheses. The hypotheses in this case are that \(x\) and \(y\) are integers, and that \(x\) and \(y\) are odd. The consequent of the implication is that \(x\) plus \(y\) is even. Unlike many other ITPs, we submit this theorem to ACL2 without a proof. ACL2 will try to come up with a proof of the theorem on its own. To come up with a proof, ACL2 will try to apply a number of proof techniques to the theorem. These techniques will automatically use previously proven theorems and previously defined functions to try to prove the current theorem. At a high level, there are two possible outcomes when submitting a theorem to ACL2: ACL2 proves the theorem or ACL2 does not prove the theorem. We call ACL2’s attempt to prove the theorem a *proof attempt*. A successful proof attempt is one in which ACL2 proves the theorem. A failed proof attempt is one in which ACL2 does not prove the theorem. The goal of this work is to use information

---

1Theorem names can be omitted in the case of `thm`
from failed proof attempts to give actionable recommendations that lead to successful proof attempts.

After a failed proof attempt, ACL2 reports diagnostic information to help the user understand why the proof attempt failed. Central to this diagnostic information are a set of checkpoints. Checkpoints are subgoals of a proof attempt that ACL2 could not prove. A failed proof attempt will consist of one or more checkpoints. To better understand checkpoints, it helps to think of ACL2’s proof attempt as a tree. Consider the ACL2 proof attempt depicted in Fig 2.1. From the figure, we can see that each proof technique in ACL2 can lead to one or more subgoals. If applying proof technique to a goal produces zero subgoals, then ACL2 proved the goal. If the

Figure 2.1: ACL2 Proof Attempt Tree
proof technique produces one or more subgoals, ACL2 can prove the goal by finding a proof for each subgoal. Thus, ACL2 can prove a goal only if it can prove all of the goal’s descendants. For example, ACL2 found a proof for Subgoal 3 because ACL2 found a proof for Subgoal 3.1, and Subgoal 3.1 is the only descendant of Subgoal 3. In constrast, ACL2 did not find a proof for Subgoal 2 because ACL2 did not find a proof for Subgoal 2.2, and Subgoal 2.2 is a descendant of Subgoal 2. Sometimes, ACL2 will not be able to find a proof technique to use on a goal. If this is the case, the goal will be a leaf node in the proof attempt tree. At the end of the proof attempt, ACL2 reports these unproven leaf nodes as checkpoints.

Like the body of theorems, checkpoints are ACL2 expressions that represent propositions. For example, in a clean ACL2 environment, ACL2 failes to prove the theorem even-sum. ACL2 reports a single checkpoint.

<table>
<thead>
<tr>
<th>ACL2 Checkpoint For Odd + Odd is Even</th>
</tr>
</thead>
<tbody>
<tr>
<td>`(implies (and (integerp x)</td>
</tr>
<tr>
<td>(integerp y)</td>
</tr>
<tr>
<td>(not (integerp (* 1/2 x)))</td>
</tr>
<tr>
<td>(not (integerp (* 1/2 y)))</td>
</tr>
<tr>
<td>(integerp (+ (* 1/2 x) (* 1/2 y))))</td>
</tr>
</tbody>
</table>

We can see that this unsimplified subgoal is very similar to the body of the theorem even-sum. The differences between the two imply that ACL2 was able to rewrite the theorem using built in functions and lemmas. In other words, ACL2 was able to use functions and lemmas available in a clean ACL2 environment. For example, ACL2 automatically used the definition of evenp, and theorems about commutativity and distributivity to rewrite the theorem. Despite this effort, ACL2 could not further simplify the checkpoint. At this point, it is up to the user to help ACL2 find a
successful proof attempt. The user can take the following actions to lead ACL2 to a successful proof attempt.

- The user can define useful helper-lemmas or let ACL2 know about existing helper lemmas by importing them into the current session.
- The user can give hints to guide ACL2’s proof attempt. These hints modify which proof techniques ACL2 uses and when.
- The theorem the user is trying to prove might not even be true. If this is the case, the user can modify the statement of the theorem to make it true.

If a user runs into a failed proof attempt, they can get advice about how to proceed by querying our deep learning model. Our model takes as input the theorem that was the subject of the failed proof attempt and a single checkpoint from the failed proof attempt. Then, our models generate actions like the ones listed above that can lead ACL2 to a successful proof attempt. These actions can both be given to the user as advice or automatically implemented by our proof advice tool.
Chapter 3

LITERATURE REVIEW

The use of deep learning for proof search initially began by improving a class of tools called hammers. Hammers aid ITP users by translating goals and a set of helpful premises into first order logic so that the goal can be proven by an ATP [45]. Notably, the computational complexity of ATPs is combinatorial in the number of premises available. Thus, premise selection is an extremely important task for hammers [30]. There have been many machine learning approaches both for improving premise selection for hammers, and improving proof search in the hammers’ target ATPs [23, 24, 25, 26, 29]. Some of the first deep learning approaches for automating formal reasoning appeared in the context of these problems [32, 19].

More recently, due to the success of deep learning in domains such as object detection [14], natural language [38], and games [33, 43], there has been a surge of interest in using deep learning systems to directly synthesize proofs scripts in ITPs. This interest inspired a number of deep learning proof systems and theorem proving datasets. For HOL Light there are both the HOList [4] and HolStep [22] datasets. The HOList dataset was used to train the DeepHOL theorem proving agent which is trained using expert iteration to perform proof search. In addition, the dataset was used to show that Graph Neural Networks can construct effective representations of HOL terms [34]. The HOList dataset was also used to investigate the possibility of skip-tree training as a method of generating generic mathematical reasoning capabilities [37].

There has also been a line of work investigating deep learning systems for the Coq theorem prover. GamePad offers an interface to both extract data from and inter-
act with the Coq proof assistant [18]. CoqGym similarly offers an environment for learning to write proof scripts in Coq and was used to create the proof synthesis agent ASTactic [47]. Other deep learning agents in Coq include Prooverbot9001 [40], TacTok [8], Diva [7], Passport [41], and Tactician [5].

Using language models for synthesizing tactics or complete proofs has also seen increased interest from the community. Polu et al. train a proof-oriented version of GPT [38] called GPT-\(f\) by pretraining on data from WebMath [36]. The authors extended their work by training GPT-\(f\) on proofs of increasing difficulty [35]. In addition to generative pretraining, co-training on multiple reasoning tasks has been shown to improve the ability of a language model to synthesis correct proof steps [13]. Finally, there have been very encouraging efforts in using large language models to generate entire proofs [9], incorporate informal proofs [20], and automatically formalize theorems [46].

Though most existing deep learning system for theorem proving have some kind of decoder used to either generate tactics or entire proofs, we are not aware of any systems that use a copy mechanism to handle rare identifiers. Additionally, many of the decoders used by existing systems produce a sequence of tokens [37, 36, 35, 31]. We show that in ACL2, our decoder’s copy mechanism gives a significant performance boost over a decoder that generates a sequence of tokens.

Though there is no prior deep-learning system for automatically completing theorems in ACL2, there have been efforts to decrease the difficulty of using the theorem prover. ACL2(ml) is an Emacs extension that uses unsupervised learning methods to find helpful lemmas and definitions [17]. There have also been some initial experiments in building a hammer for ACL2 [21].
To gather data to train our models, we traverse the ACL2 Community Books, which are public libraries of theorems previously formalized and proved by experts in ACL2. The community books total approximately 9,400 files, 90,000 function definitions, and 170,000 theorems. Unlike most other interactive theorem provers, users in ACL2 never actually explicitly write machine-checkable proofs. Instead, users define a proof strategy for the theorem prover in two ways.

1. Changing the logical world by proving helper lemmas, and providing useful definitions.

2. Guiding the theorem prover’s search with a list of hints.

Given a proof strategy in the form of hints, definitions, and lemmas, ACL2 conducts a fully automated proof attempt. By automatically attempting various proof techniques, ACL2 can prove basic theorems without any hints from the user. However, deeper theorems require hints and helper lemmas to direct the proof attempt.

To gather examples of hints, helper lemmas, and definitions that can remedy failed proof attempts, the data generation tool modifies theorems in order to break them. When ACL2 fails to prove a theorem, we can access the incomplete subgoals from the failed proof attempt. We refer to these subgoals as checkpoints. Each of these checkpoints corresponds to a separate training example, where our model takes as input the broken theorem and one checkpoint and predicts the inverse of the action used to break the theorem, or the fix. For example, if a theorem was “broken” by removing
one of its hypotheses and thereby making the theorem too general for ACL2 to prove, or simply false, then ACL2’s failed proof attempt will produce one or more checkpoints. Each of these checkpoints along with the statement of the broken theorem form the input of a training example, where the target output is the fix of replacing the removed hypothesis.

There are four fundamental ways in which we break theorems when collecting training data: removing a hypothesis from the theorem, removing a hint that was used to prove the theorem, excluding a library that was used to prove the theorem, and removing an individual lemma that was used to prove the theorem. We refer to these four classes of actions used to break theorems as action classes. When one of these actions results in a failed proof attempt, the system generates one training example for every reported checkpoint. The most recent data generation run produced over three million failed proof attempts as seen in Table 4.1.

### 4.1 Removing Hypotheses

An obvious way to break a theorem is to remove each of its hypotheses, one at a time. If a hypothesis is actually necessary, ACL2 will fail to prove the modified theorem and the failed proof attempt will result in one or more checkpoints. Our system then learns to associate similar checkpoints with the action of adding the removed hypothesis.
For instance, many users new to ACL2 expect the following to be true:

\[(\text{equal} \ (\text{rev} \ (\text{rev} \ x)) \ x)\]

Here, \text{rev} computes the reverse of a list. In reality, this property is not true, because \text{rev} always returns a list, even if its input is not. The correct theorem is given by

\[(\text{implies} \ (\text{true-listp} \ x) \ \ (\text{equal} \ (\text{rev} \ (\text{rev} \ x)) \ x))\]

When the hypothesis from this theorem is removed, ACL2 will respond with a checkpoint that looks like

\[(\text{implies} \ (\text{not} \ (\text{consp} \ x)) \ \ (\text{not} \ x))\]

The ACL2 user who tries to prove the original, incorrect theorem will see a similar checkpoint, and the advice tool could suggest adding the \text{true-listp} hypothesis, helping the user submit the correct theorem. It is important to note that out of all of our fixes, adding a hypothesis is the only one that can change the meaning of a theorem. We make sure to take this into account in our evaluation.

4.2 Removing Hints

The ACL2 theorem prover provides significant proof automation, but its heuristics are not always sufficient to prove complicated theorems by itself. Thus users commonly provide hints to the theorem prover, either subtly overriding its default search strategy or suggesting new proof strategies. There are, in fact, many different hints that a
Table 4.2: Training Pairs By Hint Type

<table>
<thead>
<tr>
<th>Hint Type</th>
<th># Checkpoints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Hint</td>
<td>252,025</td>
</tr>
<tr>
<td>Enable Hint</td>
<td>163,619</td>
</tr>
<tr>
<td>Expand Hint</td>
<td>52,822</td>
</tr>
<tr>
<td>Disable Hint</td>
<td>22,304</td>
</tr>
<tr>
<td>Induct Hint</td>
<td>16,669</td>
</tr>
<tr>
<td>By Hint</td>
<td>1,940</td>
</tr>
<tr>
<td>Cases Hint</td>
<td>1,775</td>
</tr>
<tr>
<td>Nonlinearp Hint</td>
<td>439</td>
</tr>
<tr>
<td>Do-not Hint</td>
<td>175</td>
</tr>
</tbody>
</table>

For example, ACL2 is very good at finding induction schemes to prove theorems, but sometimes its heuristics fail to find the right induction scheme. The user can suggest a specific induction scheme, e.g., with induction step \( P(n - 1, x + n) \rightarrow P(n, x) \), and the ML system is trained to associate this induction scheme with checkpoints that occur when the hint is removed.

4.3 Removing Books

Another way to break a theorem is to look at the libraries that were imported when it was originally proven. A library is composed of “books” (files) that contain definitions and rules (theorems) that can then be included to help in future proof attempts. For each of the included books, the tool attempts to prove the theorem by excluding rules loaded from that specific book. In many cases, ACL2 is still able to prove the theorem; perhaps that particular book was loaded to help other theorems in the file, or perhaps ACL2 can simply manage to find a different proof. But in many cases,
the proof attempt fails after a specific book is removed, which creates one or more training examples.

We have found that this is a very effective way to create advice for users, many of whom may not be fully aware of libraries in the community books that may be effective to reason about some of the same functions they are currently using. For instance, users struggling to prove theorems involving modular arithmetic may be advised to include one of the library books that deal extensively with `mod` and `floor`.

4.4 Removing Definitions and Theorems

The final technique we use to break a theorem is to examine the lemmas (i.e., rules) that ACL2 used to prove the theorem and arrange for each of these lemmas to be removed one at a time. For example, the proof of a theorem may depend crucially on the fact that `append` is associative, and ACL2 may have been able to use this fact automatically in the original proof, possibly without the user even being aware of this rule. As before, if the rule is necessary for the proof, the system will associate the rule with each checkpoint generated in the failed proof attempt.

4.5 Partitioning

We partition our training examples into training, testing, and validation sets such that if two examples are derived from the same broken theorem, they are placed in the same set. We use an 80%, 10%, 10% split for training, validation and testing. Additionally, we make an effort to ensure that theorems in separate books but with the same name and body i.e. “copy and pasted theorems” are placed in the same set.
We use the training set to train our model, and we use the validation set to make modeling decisions. We reserve the testing set for reporting the results in this paper.
To model the association between broken theorems, their checkpoints, and their fixes, we use the Graph2Tocopo architecture [6]. The Graph2Tocopo architecture is an encoder-decoder architecture. The encoder takes as input a graph and creates an embedding for each node in the graph. We run experiments using both GGNN encoders [1] and GREAT encoders [16] as is done in the proposal of the Graph2Tocopo framework [6]. The Tocopo [44] decoder generates a sequence of tokens, copies, and pointers. Tokens are no different from tokens in a sequence to sequence model. They are just strings from the model’s output vocabulary. Copies take the label of a node from the input graph and place it in the output. Pointers point to a specific node in the input graph. The decoder generates output elements one at a time given the node embeddings from the encoder in addition to previously generated output elements. In the remainder of this section, we describe how we transform our dataset of examples containing broken theorems and their fixes into input graphs and target output sequences.

5.1 Simple Example

Suppose an ACL2 user wants to prove that \((n + 1)! = (n + 1) \cdot n!\). The user may write this theorem as follows.
When the user submits the theorem to ACL2, they will find that ACL2 cannot prove the theorem automatically. In ACL2’s failed proof attempt, it reports the checkpoint

In other words, ACL2 was not able to simplify the goal beyond rewriting using basic properties of commutativity and distributivity. Although the user sees the checkpoint in the format shown above, ACL2 internally represents the checkpoint as a clause or a disjunction of expressions. Furthermore, ACL2 expands any macros from the checkpoint into their raw form. The representation is called the translated form of the checkpoint. The translated form of our checkpoint is shown below.

---

1This theorem uses the definition of factorial from the book “arithmetic/factorial”.
Although the function `factorial` is in scope, the rule allowing ACL2 to rewrite the goal using its definition is disabled by default. Therefore, the corresponding fix for this theorem is to **enable** the rewrite rule corresponding to `factorial`.

**Example Fixed Theorem**

```
(defthm basic-fact-thm
  (implies (natp n)
    (equal (* (+ n 1) (factorial n))
             (factorial (+ n 1))))
  :hints("Goal" :in-theory (enable factorial)))
```

Note that it would be unnecessary to require our model to rewrite the entire broken theorem since the only ways we will change a theorem are the inverses of the actions used to break them. Therefore, the prefix of each fix indicates its action class (e.g., removal of a hint) and the suffix indicates the object of the action used to break the theorem (e.g., which hint was removed). In our example, the following fix completes the proof.

**Example Fix**

```
:hint-setting-alist (:enable factorial)
```

### 5.2 High Level Architecture

The Graph2Tocopo architecture takes a graph as its input and produces a sequence of tokens as its output. The graph is fed to an encoder. The encoder creates an
embedding for each node in the graph. Each embedding is a vector of real numbers. A node’s embedding encodes information about the node that is pertinent to the model’s predictions. For example, in Fig 5.1, the encoder creates an embedding for FOO, CAT, X, and -REINDEER. In Graph2Tocopo, the encoder is either the GREAT encoder [15] or the GGNN encoder [1]. In either case, the encoder is specialized to create embeddings for nodes in a graph. The embeddings from the encoder are used in the decoder. The decoder’s job is to predict the next token in the target output sequence given the tokens it has already predicted and the node embeddings from the encoder. This is why the target output sequence in Fig 5.1 is “shifted” before being used as input to the decoder. The Graph2Tocopo architecture uses a Tocopo decoder [44]. This decoder enables the model to copy the labels from nodes in the input graph directly into the target output sequence. For example, our model could learn to copy
the contents of the X node and the -REINDEER node directly to the output regardless of their contents.

5.3 Input Representation

To use the Graph2Tocopo architecture, we must first represent our broken theorem and our checkpoint as a single graph. Each node in the graph has a node label and a node type. The node label and node type are later combined into a single embedding for the node. The edges in the graph are labeled with an edge type.

Since ACL2 is a subset of Common Lisp, its syntax is very simple and mainly consists of nested lists. In our case, we can informally think of these nested lists as nested function applications. We represent each function application as a tree. The root node of the tree contains the name of the function, and its children are trees representing the function’s arguments. For example, Fig. 5.2 shows the tree representation of the following subexpression from our translated checkpoint.

\[
\text{(binary+- (factorial n) (binary-* n (factorial n)))}
\]

To limit the size of the model’s vocabulary, it is beneficial to decompose long symbols into subtokens [42]. Therefore, instead of each node in the tree containing a label, it contains a sequence of subtokens. ACL2 programmers use dashes to make their
symbols readable just as Java programmers use camel case and python programmers use underscores. Therefore, we can naturally subtokenize our tokens by splitting on dashes [3]. This allows us to split long and infrequent identifiers into frequent subtokens. For example, instead of the theorem `functional-inversion-of-minus` being represented as a single token in our vocabulary, it is split into the more frequent subtokens `[functional, -inversion, -of, -minus]`.

Fig 5.3 shows the graph representation of the subexpression shown in Fig 5.2 after decomposing tokens into subtokens. The resulting graph has six types of edges:

- **tok2tok** edges represent edges from the original tree. These are the black edges in Fig 5.3.
- **tok2sub** edges are edges from `TOKEN` nodes to subtokens. These are the blue edges in Fig 5.3.
- **sub2sub** edges are edges from a given subtoken to the following subtoken. These are the orange edges in Fig 5.3.
- There is one additional edge type corresponding to the reverse edge for each of the edge types listed above. This way information can flow both up and down the tree.

The nodes in Fig 5.3 are either `TOKEN` nodes or subtoken nodes. `TOKEN` nodes have “token” as both their type and label. Subtoken nodes have “subtoken” as their type, and the subtoken they contain as their label.

We create our final input representation by representing every expression from the broken theorem and translated checkpoint as a graph such as the one in Fig 5.3. However, we make one modification. Since the multiple expressions in a translated checkpoint originate from a single expression, we include a root node so that infor-
Figure 5.3: Graph Representation of a Translated ACL2 Expression with Subtokenization.

Figure 5.4: Final Input Representation
Figure 5.5: Output Copy Generation

Information can flow between the expressions. Both the node type and node label of the root node are “root”. There are two additional edge types: one for an edge from the root node to a checkpoint expression, and one for the reverse edge. Fig 5.4 shows a high-level depiction of our final input representation assuming that the translated checkpoint has three expressions as our the example.

5.4 Output Representation

We must transform our target fixes into sequences of tokens, copies and pointers. Recall that copies take the label from an input node and place it in the output and pointers refer to a specific node in the input. In our fixes, we never need to reference a specific node in the input. Rather, as long as we end up with a sequence of strings that corresponds with the ground truth fix, we are satisfied. Therefore, we use copies and we do not use pointers. Furthermore, the candidates for copying from the input can be derived from a plain sequence of tokens. For example, in Fig 5.5, we can derive that the token FOO in the output can be copied from either of the nodes with label FOO from the input. Likewise, the BAR token in the output can be copied from the node with label BAR from the input. CAT and DOG cannot be considered copy tokens because there are no nodes labeled CAT and DOG from the input.

Recall that our target outputs are of the form <action class> <action object> where the action class indicates whether it was a hypothesis, library, lemma, defini-
tion, or hint that was removed from the theorem and the action object indicates the exact object e.g. the hypothesis that was removed from the theorem. In our example fix from Section 5.1, the action class :hint-setting-alist indicates that the action object (:enable factorial) should be added to the theorem as a hint. In general, the action class is a single symbol while the action object can be an arbitrary expression. Our goal is to decompose the target output into subtokens such that our model can best learn the association between broken theorems, their checkpoints, and their fixes. Towards this end, we attempt to subtokenize the target fix similar to how we subtokenize the broken theorem and checkpoint. Note, however, that in the target fix, we have to subtokenize parentheses and whitespace. We did not have to subtokenize parentheses and whitespace in our input because they are implicit in our tree representation of ACL2 expressions. To achieve a consistent subtokenization, we first split the target fix on parentheses and whitespace, and then we subtokenize the remaining symbols consistently to how we subtokenized the symbols in the input. We then merge adjacent parentheses and whitespace into single subtokens to minimize the number of tokens our model must predict. For example, our example fix, :hint-setting-alist (:enable factorial), would be decomposed into the following subtokens:

[""hint", "-setting", "-alist",
 " (", "enable", " ", "factorial", ")"]

Then, as previously shown, we can easily derive which elements in the output are eligible to be copied from the input graph.
Our evaluation is motivated by the following research questions:

RQ1 Are models that can copy symbols directly from the input into the output more effective at proving ACL2 theorems than models that can solely predict a sequence of tokens?

RQ2 How do our models perform in a real world setting when queried from an ACL2 session?

We also perform additional experiments to validate our choice to subtokenize on dashes instead of using a more conventional subtokenization scheme such as Byte Pair Encoding [42]. Finally, we examine the quality of our model’s predictions over fixes of different lengths and different types of actions to understand where our model performs well.

6.1 RQ1: Do copies help?

First we compare our Graph2Tocopo models which produce a sequence of tokens and copies to models that only produce a sequence of tokens. We will call the ablated architecture that only produces a sequence of tokens Graph2To. To construct Graph2To, we remove the mechanism for our Graph2Tocopo model to produce copies in the output. We perform this ablation by making a few simple changes to the Tocopo decoder. In the Tocopo decoder, each element in the output is assigned an
embedding before running the embeddings through a series of attention layers. The initial embedding is acquired by summing a token embedding, a copy embedding and a pointer embedding. To create a model that only supports tokens in the decoder, we simply use the token embedding as the embedding for each output element instead of summing across tokens, copies and pointers. Additionally, the Tocopo decoder originally makes use of three output heads. One of the output heads predicts tokens, one predicts copies, and one predicts pointers. In Graph2To, we use a decoder with just one output head that predicts tokens. Finally, we modify the Graph2Tocopo loss function to account for the fact that there are no copies or pointers in Graph2To. Since the Graph2Tocopo loss function maximizes the log probability of a correct token, copy, or pointer, Graph2To just maximizes the log probability of a correct token.

We compare Graph2Tocopo to Graph2To using two different encoders. We use the GREAT encoder [16] which is a transformer-based encoder that augments attention weights with biases corresponding to edges in the input graph. We also use the GGNN encoder [1] which is a message-passing architecture in which a node updates its embedding by taking a combination of its neighbors embeddings.

All of our models are trained for approximately two days on a single Tesla T4 GPU. We train our models using a batch size of 8 examples, and using the adam optimizer with a linear warmup over the first 10,000 training steps [28]. Due to memory constraints in our training environment, we limit the size of our input graph to 512 nodes. If an input graph is larger than 512 nodes, we truncate the theorem statement before the checkpoint since the checkpoint often contains more focused information about the failed proof attempt. We limit the size of the model’s output to 256 tokens, though we rarely encounter outputs of that length. Nonetheless, we ensure that our model’s prediction is counted as incorrect whenever the ground truth fix is longer than 256 tokens.
Figure 6.1: Graph2Tocopo vs Graph2To Validation Accuracy over Time.

Table 6.1: Testing Set Accuracy Comparison

<table>
<thead>
<tr>
<th>Model</th>
<th>Top-1 Accuracy</th>
<th>Top Action Type Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph2Tocopo (GGNN)</td>
<td><strong>38.40</strong></td>
<td><strong>46.12</strong></td>
</tr>
<tr>
<td>Graph2To (GGNN)</td>
<td>27.57</td>
<td>35.08</td>
</tr>
<tr>
<td>Graph2Tocopo (GREAT)</td>
<td>26.98</td>
<td>33.88</td>
</tr>
<tr>
<td>Graph2To (GREAT)</td>
<td>17.37</td>
<td>23.89</td>
</tr>
</tbody>
</table>
In Fig 6.1, we show the training behavior of Graph2ToCopo vs the training behavior of Graph2To. We first observe that the GGNN encoder outperforms the GREAT encoder on our dataset of ACL2 proofs. More importantly, we observe that after adjusting for the type of encoder used, the Graph2ToCopo architecture outperforms the Graph2To architecture. We notice that after around 20 hours, the Graph2To validation accuracy begins to decline while the Graph2ToCopo validation accuracy steadily increases. Often it is the case that an ACL2 user must enable rules seen in the goal, or rewrite expressions seen in the checkpoint. Therefore, rather than learning to assemble these expressions from a vocabulary of tokens, it might be easier for the model to directly copy them.

Since we can easily try multiple recommendations from the model inside of an ACL2 session, it makes sense to evaluate the top \( n \) recommendations from the model. If any of the top \( n \) recommendations are correct, then we can say that the model succeeded because the user’s broken theorem will be fixed. However, since we are using an autoregressive model, it is not straightforward to get the \( i \)th best recommendation when \( i \) is greater than one. Therefore, instead of getting the top \( n \) recommendations from the model, we get the top recommendation for each action class and each hint type. Each of these action types has a unique prefix. We can condition the model on these prefixes to get 12 unique predictions from the model. We let \( \text{Top Action Type Accuracy} \) represent the frequency that the ground truth fix is the top fix for one of our action types. We show the testing set accuracy of the four models in Table 6.1.

6.2 RQ2: How about real world performance?

Note that even if our model does not provide a recommendation that is exactly equivalent to the ground truth fix for the broken theorem, the recommendation could
still complete the proof of the theorem. This is because there are multiple viable fixes for a given broken theorem. For example, suppose we would like to prove the following theorem:

(defthm assoc-commute-mul
  (equal
   (* (* a b) c)
   (* (* b c) a)))

In a clean ACL2 environment, ACL2 will fail to prove this theorem. The crucial lemma needed to prove this theorem is named “commutativity-2-of-*” and resides in the book “arithmetic/equalities”. Thus, we could complete the proof by enabling the lemma “commutativity-2-of-*”. However, since this lemma is enabled upon including the book “arithmetic/equalities”, it suffices to include this book to complete the proof. So, there are at least two correct fixes the model could recommend to complete this proof.

To give our model full credit for recommending viable fixes, we also evaluate it within an ACL2 session using our HTTP machine interface. We call this interface the advice tool. The advice tool allows an ACL2 user with a broken theorem to query our model for advice. The user simply calls the advice tool with the command (advice) after a failed proof attempt and the tool will query our model with the broken theorem and checkpoints from the failed proof attempt. The advice tool then attempts to fix the broken proof by using the advice suggested by our system. Note that our advice tool currently does not stack recommendations by the model. Instead, it tries each recommendation independently.

To evaluate our models using our advice tool, we iterate through theorems from the testing set. Each theorem belongs to some book. We replay the book up to the
Theorem of interest. When we get to the theorem of interest, we remove all hints from
the theorem. If the theorem has no hints, we omit the theorem from the evaluation.
Similarly, if the ACL2 proves the theorem without any hints, we omit the theorem
from the evaluation. Otherwise, we query the model 10 times for each theorem. We
rotate through the checkpoints in the failed proof attempt of the broken theorem
and query the model once for each checkpoint until we have made 10 queries. The
first time we visit a checkpoint, we query the model with temperature zero to ensure
we get the best recommendation. On each subsequent visit, we query the model
with temperature one. We then iterate through the recommended changes from the
model to see whether or not they complete the proof. Crucially, we do not accept
recommendations that suggest adding a hypothesis since these recommendations can
change the meaning of the theorem and likely make it easier to prove. Our models
are able to prove 20%\(^1\) of the theorems in our testing set with a budget of 10 queries
per theorem.

6.3 Additional Experiments

We perform a number of additional analyses to support some of our modeling decisions
as well as to further understand our model’s performance.

6.3.1 Subtokenization

The most conventional choice to subtokenize symbols is to use Byte-Pair Encoding
\cite{silver2016}. This choice is made in most language models as well as in a few theorem proving
systems \cite{pelzer2016, pelzer2015, pelzer2014, pelzer2013}. However, we were inspired by the decision made in code2seq
\cite{code2seq} to create subtokens based on Java’s camel case. We show the training behavior

\(^1\)Conservative placeholder.
Table 6.2: Testing Set Accuracy Comparison

<table>
<thead>
<tr>
<th>Model</th>
<th>Top-1 Accuracy</th>
<th>Top-Action Type Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>GGNN (Dashes)</td>
<td>38.40</td>
<td>46.12</td>
</tr>
<tr>
<td>GGNN (BPE)</td>
<td>36.27</td>
<td>44.26</td>
</tr>
<tr>
<td>GREAT (Dashes)</td>
<td>26.98</td>
<td>33.88</td>
</tr>
<tr>
<td>GREAT (BPE)</td>
<td>23.60</td>
<td>30.15</td>
</tr>
</tbody>
</table>

across subtokenization method in Fig 6.2. We show the testing set accuracy across subtokenization method in Table 6.2.

We note that both the validation and testing set accuracy for a scheme that subtokenizes on dashes are higher than a scheme that subtokenizes using Byte-Pair Encoding. We hypothesize that subtokens derived from splitting on dashes are more meaningful than the subtokens derived from byte-pair encoding. For example, recall from Section 5.3 that the lemma `functional-inversion-of-minus` split into the subtokens `[functional, -inversion, -of, -minus]` by our subtokenization scheme. In contrast, our Byte Pair Encoding gives following subtokens: `['function', 'al-', 'inv', 'er', 'si', 'on', '-', 'of-', 'min', 'us']`.

6.3.2 Accuracy By Category

To further understand the performance of our model, we plot the testing set accuracy after grouping by action type and after grouping by the length of the fix. The reported accuracies are for our Graph2Tocopo model with a GGNN encoder. We include 95% confidence intervals in all of our plots.

In Fig 6.3, we plot the accuracy by the action type of the ground truth fix. We find that the model’s efficacy at generating fixes of a certain action type is correlated with the number of training examples of that action type. However, we find two interesting
outliers: use hints and expand hints. We hypothesize that use hints and expand hints tend to be more formulaic than other action types allowing the model to perform well with only a limited number of examples.

To understand how our models handle broken theorems and checkpoints that require fixes of varying lengths, we plot the accuracy of our model by the number of tokens in the ground truth fix in Fig 6.4. We were surprised to find that there is not a strong pattern in the length of a fix and its difficulty to generate. It seems that the action type of the ground truth fix has much more weight in determining its difficulty to generate.

Figure 6.2: Validation Accuracy of Graph2Tocopo by Subtokenization Method over Time.
Figure 6.3: Accuracy of Graph2Tocopo (GGNN) by Action Type
Figure 6.4: Accuracy of Graph2Tocopo (GGNN) by Number of Tokens in Ground Truth Fix
In this work, we present a deep learning system to help automate the theorem proving process in ACL2. We curated a dataset from the 170,000+ theorems in the ACL2 community books. We used this dataset to train models that could recommend hints, hypotheses, helpful books and helpful lemmas to complete broken ACL2 proofs. We found that the Graph2Tocopo architecture was more effective at generating fixes than the Graph2To architecture. We believe this result could generalize to other theorem proving systems that use decoders that do not have a copy mechanism. ACL2 users can directly access our system using our advice tool.


