Baldur: Whole-Proof Generation and Repair with Large Language Models

Emily First
University of Massachusetts
Amherst, MA, USA
efirst@cs.umass.edu

Markus N. Rabe
Augment Computing
Palo Alto, CA, USA
markus@augmentcode.com

Talia Ringer
University of Illinois
Urbana-Champaign, IL, USA
tringer@illinois.edu

Yuriy Brun
University of Massachusetts
Amherst, MA, USA
brun@cs.umass.edu

ABSTRACT

Formally verifying software is a highly desirable but labor-intensive task. Recent work has developed methods to automate formal verification using proof assistants, such as Coq and Isabelle/HOL, e.g., by training a model to predict one proof step at a time and using that model to search through the space of possible proofs. This paper introduces a new method to automate formal verification: We use large language models, trained on natural language and code and fine-tuned on proofs, to generate whole proofs at once. We then demonstrate that a model fine-tuned to repair generated proofs further increasing proving power. This paper: (1) Demonstrates that whole-proof generation using transformers is possible and is as effective but more efficient than search-based techniques. (2) Demonstrates that giving the learned model additional context, such as a prior failed proof attempt and the ensuing error message, results in proof repair that further improves automated proof generation. (3) Establishes, together with prior work, a new state of the art for fully automated proof synthesis. We reify our method in a prototype, Baldur, and evaluate it on a benchmark of 6,336 Isabelle/HOL theorems and their proofs, empirically showing the effectiveness of whole-proof generation, repair, and added context. We also show that Baldur complements the state-of-the-art tool, Thor, by automatically generating proofs for an additional 8.7% of the theorems. Together, Baldur and Thor can prove 65.7% of the theorems fully automatically. This paper paves the way for new research into using large language models for automating formal verification.

CCS CONCEPTS

• **Software and its engineering → Software verification**; **Formal software verification**; • **Theory of computation → Automated reasoning**; • **Computing methodologies → Neural networks**; **Machine learning approaches**.

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616243

KEYWORDS

Proof assistants, proof synthesis, proof repair, machine learning, large language models, automated formal verification

ACM Reference Format:
Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-Proof Generation and Repair with Large Language Models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616243

1 INTRODUCTION

Formal software verification — proving software correctness and other properties — is one of the most challenging tasks software engineers can undertake. It is highly effective at producing high quality software. For example, CompCert, a C compiler verified using the Coq interactive theorem prover [88], was the only compiler on a list including the ubiquitous GCC and LLVM, in which a comprehensive study found no bugs [106]. Similarly, the seL4 project resulted in an highly reliable operating system microkernel [44]. However, the cost of manual formal verification — writing the proofs — is often prohibitive. For example, the proof of the C compiler is more than three times as long as the compiler code itself [51]. As a result, recent research has focused on automated proof synthesis, which can lead to fully automating formal verification.

There are two promising approaches for automating proof synthesis. The first is to use *hammers*, such as Sledgehammer [71] for the Isabelle proof assistant. Hammers iteratively apply known mathematical facts using heuristics. The second is to use search-based *neural theorem provers*, such as DeepHOL [5], GPT-f [73], TacticZero [100], Lisa [38], Evariste [46], Diva [22], TacTok [23], and ASTactic [105]. Given a partial proof and the current *proof state* (which consists of the current goal to prove and the list of known assumptions), these tools use neural networks to predict the next individual *proof step*. They use the *proof assistant* to evaluate the proposed next proof steps, which returns a new set of proof states. Iterating this procedure results in a tree-like structure, which defines a search through the space of possible proofs. Neural theorem provers rely on diverse neural architectures, such as Wavenet [5, 92], graph neural networks [69], short long-term memory models [22], and language models with the transformer architecture [31, 73].

In this paper, we propose Baldur, a different, simpler approach to proof synthesis. We show that using large language models (LLMs), fine-tuned on proofs, can produce entire proofs for theorems. LLMs are scaled-up transformer models trained on a large amount of text data, including natural language and code, that have proven to be remarkably effective across a wide variety of applications, including question answering, and text and code generation [9, 16]. Here, we show their remarkable effectiveness for whole proof generation.

The main contributions of our work are:

- We develop Baldur, a novel method that generates whole formal proofs using LLMs, without using hammers or computationally expensive search.
- We define a proof repair task and demonstrate that repairing incorrectly generated proofs with LLMs further improves Baldur's proving power when the LLM is given access to the proof assistant's error messages.
- We demonstrate empirically on a large benchmark that Baldur, when combined with prior techniques, significantly improves the state of the art for theorem proving.

We design Baldur to be able to work with any LLM internally, but we evaluate our implementation using two versions of Minerva [52], one with 8 billion parameters and another with 62 billion parameters. By contrast, existing tools that use (L)LMs for theorem proving, either predict individual proof steps [31, 37, 38], or rely on few-shot prompting and require the existence of natural language proofs as hints [39].

We evaluate Baldur on the PISA dataset [38] of Isabelle/HOL theorems and their proofs used in recent state-of-the-art Isabelle/HOL proof synthesis evaluations [37, 38]. The dataset consists of 183K theorems, of which we use 6,336 for measuring effectiveness. Our evaluation answers the following research questions:

RQ1: How effective are LLMs at generating whole proofs?
LLMs outperform small-model-driven search-based methods. Baldur (without repair) is able to generate whole proofs for 47.9% of the theorems completely automatically, whereas search-based approaches prove 39.0% [37].

RQ2: Can LLMs be used to repair proofs?
LLMs can repair proofs, including their own erroneous proof attempts. Baldur proves an additional 1.5% of the theorems when given access to a previous erroneous proof attempt and the error messages produced by the proof assistant, even when controlling for the computational cost of the additional inference. The error message is crucial for this improvement.

RQ3: Can LLMs benefit from using the context of the theorem?
In-context learning is remarkably effective for LLM-based theorem proving. With context, Baldur proves 47.5% of the theorems, but only 40.7% without context for the same model size.

RQ4: Does the size of the LLM affect proof synthesis effectiveness?
Larger LLMs do perform better, suggesting that our approach will continue to improve with further developments in LLM research.

RQ5: How do LLMs compare to other state-of-the-art proof generation methods?
Baldur complements state-of-the-art approaches by proving theorems they do not. Together with Thor [37], a tool that combines a learned model, search, and a hammer, Baldur can prove 65.7% of the theorems, whereas Thor alone proves 57.0%. These findings suggest that LLM- and search-based methods' ideas complement each other and can work together to further improve the automation of formal verification. An ensemble of 10 different fine-tuned Baldur models proves 58.0%.

By leveraging LLMs, Baldur simplifies the proof synthesis pipeline, greatly reducing the complexity and cost of the fine-grained interaction between the prediction model and the proof assistant that search-based methods require. This reduction enables us to leverage the power of LLMs, which would be prohibitively computationally expensive if synthesis required as many LLM queries as search-based methods. Further, those calls would require re-encoding with each step the additional information the LLM might need, whereas our approach allows us to make a single call and process the context only once, sampling multiple proofs of multiple proof steps, at once.[1] Overall, our study strongly suggest that LLMs are a very promising direction of research for automating formal verification, and identifies several new avenues for future explorations.

2 THE BALDUR APPROACH

Prior approaches to proof synthesis employ a neural model to predict the next *proof step* given the current *proof state*. The proof step predictions then guide a search strategy, such as best-first search or depth-first search. Throughout the search, the proof assistant needs to check each proof step prediction to determine whether it is valid. This means that existing proof synthesis tools require a tight interaction between the neural network and the proof assistant. As we move to using LLMs, this results in complex systems, as LLMs need to run on specialized hardware (GPUs or TPUs), while proof assistants run on CPUs.

We explore a simpler, yet effective method: fine-tuning LLMs to generate complete proofs. This simplification avoids the fine-grained interaction between neural model and the proof assistant, allowing us to run the jobs of generating proofs and checking completely separately. Besides reducing complexity, this can also improve efficiency, because (1) it enables us to use large batch sizes, which can significantly improve hardware utilization during inference (cf. [74]), and (2) when providing additional context to the model, the context now does not have to be reprocessed for each proof step, but only once per proof.

We fine-tune LLMs on proof data to generate entire proofs and explore the impact of giving the LLMs additional information. Our approach and implementation include the following:

- We fine-tune an LLM to generate an entire proof given only the theorem statement. We call this model the *proof generation model* (Section 2.1).
- We provide a model a proof attempt that did not check along with the corresponding *error message* from the proof assistant

[1] Alternatively path advanced caching strategies in the prediction servers of large language models could address this problem. This is beyond the scope of our work.

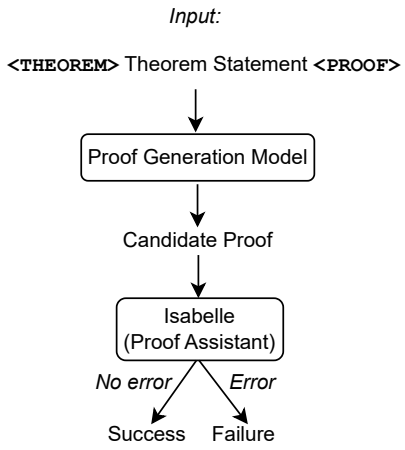

Figure 1: An example of using the proof generation model to generate a proof.

so that the model may attempt to find a better proof. We call this model the *proof repair model* (Section 2.2).
- We provide text from the same *theory file* that the problem was taken from. We add only the lines from the theory file that immediately precede the theorem we want to prove. We call this added information the *theory file context* and we add it to the proof generation model (Section 2.3).
- The LLM that we fine-tune at the core of all of this is Minerva [52], which is pretrained on a mathematics corpus. We describe our Baldur-specific implementation details for how we use this model (Section 2.4).

These fine-tuned LLMs and their interaction with the Isabelle proof assistant make up our tool Baldur. This section details the Baldur approach, which includes creating training datasets and leveraging LLMs to generate and repair proofs.

2.1 Proof Generation

Existing proof generation methods using neural models generate the proof one step at a time. In contrast, our approach generates the entire proof, as illustrated with a single example in Figure 1. We use only the theorem statement as input to our *proof generation model*. We then sample a proof attempt from this model and perform proof checking using Isabelle. If Isabelle accepts the proof attempt without an error, then we have proven the theorem. Otherwise, we can try sampling another proof attempt from the proof generation model. Explicitly, the input and output of our proof generation model is as follows:
- **Input:** theorem statement.
- **Output:** candidate proof.

Example. To illustrate the power of the proof generation approach in our tool Baldur, we first consider, as an example, the theorem `fun_sum_commute`.

```
lemma fun_sum_commute:
  assumes "f 0 = 0" and "∧x y. f (x + y) = f x + f y"
  shows "f (sum g A) = (∑a∈A. f (g a))"
```

The theorem states that for an additive function f where $f(0) = 0$, and an arbitrary function g, applying f on the sum of the set resulting from applying g on each element in a given set is equal to the sum of applying g followed by f to each element in that set. In this context, the sum over an infinite set is taken to be zero. This theorem is from a project in the Archive of Formal Proofs called Polynomials, specifically in the file `Utils.thy`.

The human-written proof distinguishes between two cases: when the set is finite and when it is not. Induction is used for the finite set case.

```
proof (cases "finite A")
  case True
    thus ?thesis
    proof (induct A)
      case empty
      thus ?case by (simp add: assms(1))
    next
      case step: (insert a A)
      show ?case by (simp add:
        sum.insert[OF step(1) step(2)]
        assms(2)
        step(3))
    qed
  next
    case False
      thus ?thesis by (simp add: assms(1))
qed
```

If we were to derive a training example from this example, the input would be theorem statement and the target would be this human-written proof.

Our tool Baldur, using the proof generation model, is able to generate the following correct proof for this statement.

```
  by (induct A rule: infinite_finite_induct)
  (simp_all add: assms)
```

Baldur recognizes that induction is necessary and applies a special induction rule called `infinite_finite_induct`, following the same overarching approach as the human-written proof, but much more succinctly. It is interesting to note that Sledgehammer, the hammer for Isabelle, cannot prove this theorem by default, as it requires induction.

Training Data Creation. To train the proof generation model, we construct a new proof generation dataset. Existing datasets for training models in neural theorem provers contain examples of individual proof steps. Each training example includes, at minimum, the proof state (the input) and the next proof step to apply (the target). Given a dataset that contains individual proof steps, we want to create a new dataset so that we can train models to predict entire proofs at once. So we extract the proof steps of each theorem from the dataset and concatenate them to reconstruct the original proofs. We use this data to generate training examples for the proof generation model, where the input consists of the theorem statement and the target consists of the proof.

In particular, this means that we drop the *proof states* from the dataset, which make up most of the text in the dataset. We argue that for Isabelle proofs this is not necessarily a problem, as Isabelle uses a declarative proof language that is designed to be human-readable. This is in contrast to other proof assistants, such as Coq, where the proofs are typically written in a procedural style that is not easy to interpret for humans without using the proof assistant to generate the intermediate proof states.

Inference. We fine-tune an LLM on our data to predict the entire proof given only a theorem statement. To synthesize a proof using the fine-tuned LLM, we provide a potentially unseen theorem statement and sample a fixed number of sequences (typically 16 or 64) from the language model, where a sequence is an entire proof attempt. We tune the sampling temperature from a small set (between 0.0 and 1.4 in increments of 0.2), which is a multiplicative factor on the log probabilities of the distribution of tokens sampled in each step.

Proof checking. After sampling proofs from the model, we check all of them with the proof assistant. This means that we first load the context in which the theorem was originally proven and then replace the original proof of the theorem with the one we sampled from the model. If Isabelle accepts any of the sampled proofs, we report the theorem as proven.

2.2 Proof Repair

If a proof is not accepted, Isabelle returns an error message that is intended to help humans with debugging their proof script. Existing proof generation methods, however, have no way to leverage error messages.

Building off our proof generation approach, we explore the use of error messages to improve neural theorem provers by developing a proof repair approach. Starting with just the problem statement, we apply the proof generation model from Section 2.1 to sample a proof attempt. If Isabelle accepts the proof attempt, we can stop. Otherwise, we use the error message returned by the proof checker and the incorrect proof attempt to construct an example to serve as input to the *proof repair model*. As depicted in Figure 2, we use the theorem statement, the incorrect proof, and the error message as input to our proof repair model. We then sample the proof attempt from this model, and perform proof checking in the same way as the proof generation approach. Explicitly, the input and output of our proof repair approach pipeline are as follows:

- **Input:** theorem statement, incorrect proof, error message.
- **Output:** candidate proof.

Example. Starting from the theorem `fun_sum_commute`, we illustrate an example of the proof repair approach in our tool Baldur. We apply the proof generation model to obtain more proof attempts. The following is a proof attempt generated by Baldur, which fails in the proof checker.

```
proof (induct A)
case (insert x A)
thus ?case
  by (simp add: assms(2))
qed simp
```

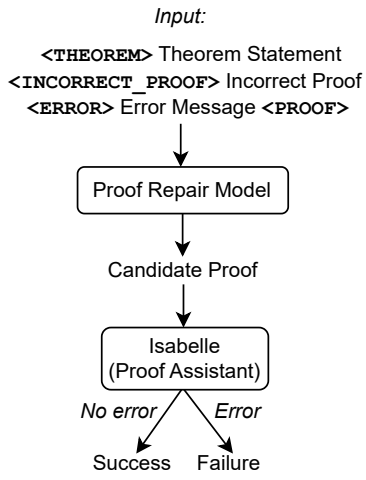

Figure 2: An example of using the proof repair model to repair an incorrect proof.

Baldur attempts to apply an induction, but fails to first break down the proof into two cases (finite vs. infinite set). Isabelle returns the following error message:

```
Step error: Unable to figure out induct rule
At command "proof" (line 1)
```

The error message details where the error occurs (line 1) and that the issue is regarding the induct rule. With these strings as input, using the proof repair model, Baldur can attempt to generate a correct proof for this statement. If we want to instead derive a proof repair training example from these strings, we concatenate the theorem statement, the failed proof attempt, and the error message to serve as the input, and we use the correct human-written proof (recall from previous section) as the target.

Training Data Creation. To train the proof repair model, we need to generate a proof repair training set. Figure 3 details the training data creation process. Using the proof generation model, we sample one proof with temperature 0 for each problem in the original training set used to train the proof generation model. Using the proof assistant, we record all failed proofs and their error messages. We then proceed to construct the new proof repair training set. For each original training example, we concatenate the theorem statement, the (incorrect) candidate proof generated by the proof generation model, and the corresponding error message to obtain the input sequence of the new training example. For the target sequence, we reuse the ground truth proof from the original training example. We fine-tune the pretrained LLM on the proof repair training set to obtain the proof repair model.

2.3 Adding Context

LLMs possess impressive in-context learning abilities (cf. [9, 16]) that allow them to flexibly use information that is provided as part of the input sequence (and, in fact, as part of their own output [68, 97]). In order to explore to what extent in-context learning can help in the theorem proving domain, we extend their inputs with potentially

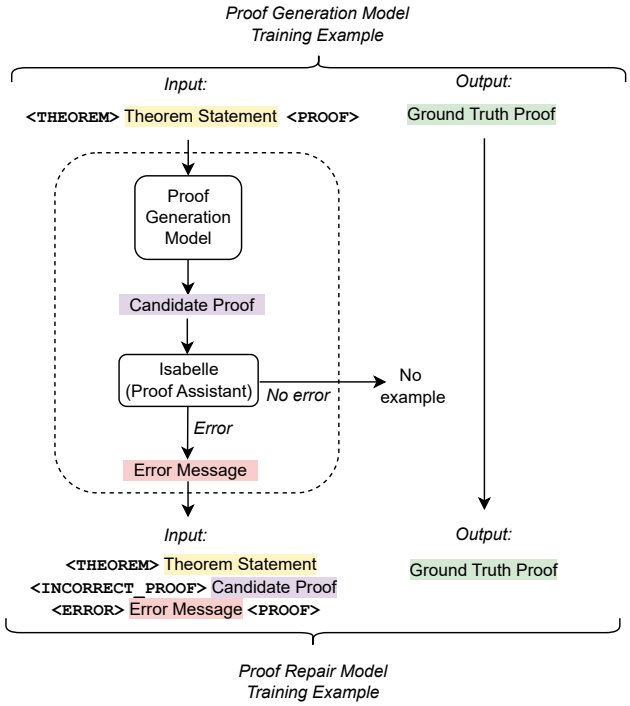

Figure 3: Training data creation for the proof repair model.

helpful context. Adding to our proof generation approach, we use the theory file contexts (the lines preceding the theorem statement) as input to our *proof generation model with context*. Explicitly, the input and output of our proof generation model with context is as follows:

- **Input:** theory file context and theorem statement.
- **Output:** candidate proof.

Example. Continuing the example, the theory file context directly preceding `fun_sum_commute` is the following theorem statement and its associated proof.

```
lemma additive_implies_homogenous:
  assumes "∧x y. f (x + y) = f x +
  ((f (y::'a::monoid_add))::'b::cancel_comm_monoid_add)"
  shows "f 0 = 0"
proof -
  have "f (0 + 0) = f 0 + f 0" by (rule assms)
  hence "f 0 = f 0 + f 0" by simp
  thus "f 0 = 0" by simp
qed
```

The proof generation model with context in Baldur can leverage this additional information. Strings that appear in the theorem statement for `fun_sum_commute`, such as `"f 0 = 0"`, appear again in this context, and so the additional information surrounding them could help the model make better predictions.

Training Data Creation. We add the lines of the theory file that precede the theorem statement to serve as additional context. This means that context can include statements, such as the previous theorems, definitions, proofs, and even natural language comments. To make use of the available input length of LLMs, we first add up to 50 preceding statements from the same theory file. During training, we first tokenize all these statements, and then we truncate the left of the sequence to fit the input length.

Premise Selection. Many proofs make frequent use of definitions and previously proven statements, also known as *premises*. Some neural theorem provers, such as HOList [5], focus entirely on the problem of selecting the right set of premises, which has been shown to be quite successful in theorem proving.

Premise selection is clearly similar to the addition of context in some aspects, but we want to emphasize some key differences: (1) Adding context is an extremely simple technique that only requires rudimentary text processing, (2) by adding the preceding lines of the theory file, the model can only observe a small fraction of the available premises, (3) most of the added context consists of proofs.

2.4 Large Language Model

We use Minerva [52], a large language model pretrained on a mathematics corpus based on the PaLM [16] large language model. Specifically, we use the 8 billion parameter model and the 62 billion parameter model. The Minerva architecture follows the original Transformer architecture [93], but has some noteworthy differences. It is a decoder-only transformer with maximum sequence length of 2,048 tokens. The model uses

- rotary position encodings [86] instead of sinusoidal absolute position embeddings,
- parallel layers [8], which compute the feed forward layer and the attention layer in parallel and add up their results instead of computing them in sequence, and
- multi-query attention, which uses a single key-value pair per token per layer for faster decoding [84].

As this model is not a contribution of this paper, we refer the reader to prior work for lower-level details on the Minerva architecture [16].

Baldur-specific implementation details. The proof generation task naturally consists of an input, which is the theorem statement (potentially augmented with additional information), and the output (target), which is the proof for the theorem. To work with the decoder-only model, we concatenate the inputs and targets, but the loss is only computed over the target during fine-tuning so that the model is learning to conditionally generate the target given the input and not the input itself. The inputs use bidirectional attention while the targets use causal attention as in PrefixLM [76].

As the transformer has a maximum context length of 2048, we pad the sequences with zeros if they are too short, and we need to truncate them if they are too long. Inputs to the model are truncated to the maximum input length by dropping tokens on the left. The rationale for dropping tokens on the left is that the additional context is given before the theorem statement, and can be truncated more safely than the theorem statement itself. Similarly, targets (i.e. the proof to generate) are truncated on the right to the maximum target length.

We used a maximum input length of 1536 and a maximum target length of 512 all experiments but the repair study and the 62b model, which used 1024 and 1024 instead. We use a drop-out rate of 0.1 for both generation and repair models to address overfitting.

During sampling from the language model we restrict the choice of the next token to the 40 tokens with the highest score, also called top-K sampling [21]. We sample sequences with a maximal length of 256 tokens. The model was trained to generate up to 512 tokens, but since most successful proofs are relatively short, this limitation has little impact on the proof rate while saving some compute.

We use a batch size of 32, and fine-tune for up to 100,000 steps, but we observed that the model begins to overfit to the training set after 50,000 to 70,000 steps. For inference, we selected checkpoints from just before the model started to overfit.

3 EVALUATION

This section presents our experiments answering the following research questions:

- RQ1: How effective are LLMs at generating whole proofs?
- RQ2: Can LLMs be used to repair proofs?
- RQ3: Can LLMs benefit from using the context of the theorem?
- RQ4: Does the size of the LLM affect proof synthesis effectiveness?
- RQ5: How do LLMs compare to other state-of-the-art proof generation methods?

To answer these questions, we trained several language models using the approach from Section 2, and evaluated them on the PISA benchmark (see Section 3.2).

3.1 Experimental Setup

Machine specification. For most of the training runs of the 8b model, we used 64 TPUv3 cores distributed across 8 hosts. For training the 62b model, we used 256 TPUv3 cores distributed across 32 hosts. For most inference jobs, we used between 32 inference servers using 8 TPUv3 cores each.

Proof Checker. We use the PISA codebase [38] under a BSD 3-clause license, which allows us to interact with the Isabelle proof assistant to check proofs. To run large jobs of the proof checker, we package it in a Docker container and run it on GCP. We extended the proof checker to discard any proofs that contain "sorry" or "oops", which are keywords that skip proofs, but otherwise pass the proof checker. We apply a timeout of 10 seconds to each proof step in the proof checker.

3.2 PISA Benchmark

We derive our datasets from the PISA dataset [38], which includes the Isabelle/HOL repository under a BSD-style license and the Archive of Formal Proofs (AFP) from October 2021. The AFP is a large collection of Isabelle/HOL proof developments. PISA includes the core higher-order logic library of Isabelle, as well as a diverse library of proofs formalised with Isabelle. This includes mathematics proofs and verification of software and hardware systems. The PISA dataset comes with a 95%/1%/4% split of theorems for the training/validation/test sets, which we follow in this work as well.

For the test set, prior work randomly chose 3,000 theorems from the test set to report their results on. We report our results on the complete test set. Some entries in the dataset are not proper theorems (starting with the keyword "lemmas" instead of "lemma"), which we filter out, as did prior work. This leaves us with a total of 6,336 theorems in our test set (originally 6,633 theorems).

It is worth noting that, as with any LLM-based work, there is the potential for proofs from the test set to have leaked into the LLM pretraining data. Minerva was trained on a dataset consisting of scientific papers from the arXiv preprint server and web pages that include mathematical expressions [52]. While the pretraining data for the Minerva LLM at the base of our models does not include the PISA dataset, it does contain code that may include some Isabelle/HOL proofs found in PISA. This should be kept in mind when interpreting the results.

3.3 RQ1: How effective are LLMs at generating whole proofs?

We aligned our methodology with the methodology described in Thor [37] to enable a comparison between various methods. The Thor paper includes informative baselines for the PISA benchmark, including Sledgehammer, a method relying on heuristic search, and a language model approach using search.

Sledgehammer and the search-based language model approach achieve 25.6% and 39.0%, respectively. In comparison, our naive proof generation approach with an 8b language model achieves a proof rate of 34.8% with 16 samples and of 40.7% with 64 samples. The comparison is even more favorable if we consider the other variants of Baldur, which achieve a proof rate of up to 47.9%.

We observe that the comparison depends on the computational cost that we spend during inference. While comparing the cost required for the two methods is involved, one measure we can use is the amount of computational resources reserved during proof generation. For a single proof, the language model approach using search [37] requires a TPUv3 with 8 cores for 216 seconds,[2] while our methodology also requires a TPUv3 with 8 cores for around 35 seconds to sample 64 proofs — a difference of factor 6. This argument disregards the time spent on proof checking, which is intentional: proof checking is done on CPUs, which is cheap compared to time spent on TPUs. So, disentangling these two workloads can lead to significant reductions in computational cost.

> **RA1**: These results demonstrate that LLMs can generate full proofs just as well as smaller language models augmented with a search strategy.

3.4 RQ2: Can LLMs be used to repair proofs?

We trained models for proof generation and repair as detailed in Section 2. If we sample from the proof generation model once with temperature 0, collect the failed proofs, and then repair once with temperature 0, we generate an additional 266 or 4.2% correct proofs. However, in this comparison, the generate + repair approach uses two samples, while the generate approach has only one sample. For a fair comparison, we have to compare the repair approach to the generate approach with additional inference attempts.

[2] Jiang et al. state in Section 4.1 [37] that 1,000 problems take around 60 TPU hours.

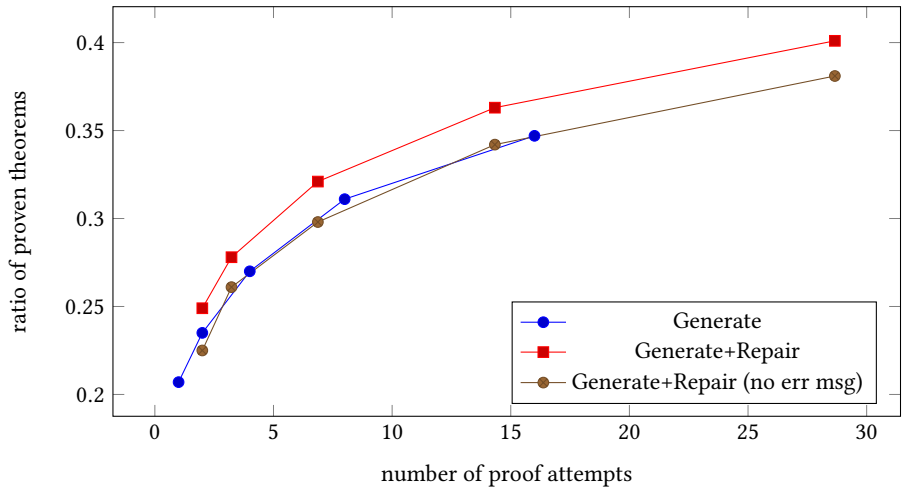

Figure 4: Ratio of theorems proven vs inference cost.

Model	16 samples	64 samples
Baldur 8b generate	34.8%	40.7%
Baldur 8b generate + repair	36.3%*	—
Baldur 8b w/ context	40.9%	47.5%
Baldur 62b w/ context	42.2%	47.9%
Baldur 8b w/ context ∪ Thor	—	65.7%

Figure 5: Proof rate of different models.
***The repair approach uses half the number of samples, and then one repair attempt for each sample.**

Figure 4 plots the proof success rate of the generate approach and the repair approach against the number of proof attempts. Note that the number of samples for the repair approach does not perfectly align with the number of samples for the generate approach. This is because the generate approach tends to produce multiple copies of the same proofs, which we deduplicate before repair, and only generate one repair attempt per failed proof attempt. For each of the number of samples of the generate approach, we tune the temperature in the range of 0.0 to 1.4 in increments of 0.2, and we always use temperature 0 for the repair approach.

The repair approach consistently outperforms the plain proof generation model, which only uses the theorem statement as input. To shed some light on what causes the gains, we trained another repair model with the same information, except without the error message. Figure 4 shows this model's proof success rate; it does not surpass the performance of the plain generation model when normalized for inference cost. This suggests that the information in the error message is crucial for the observed gains of the repair approach.

RA2: LLMs can be used to repair proofs, including their own failed proof attempts, boosting overall proving power.

3.5 RQ3: Can LLMs benefit from using the context of the theorem?

Figure 5 reports the impact of adding theory file context to our plain generation approach. At 64 samples, the proof rate increases from 40.7% to 47.5% for the same model size. Figure 6 plots the proof success rate of the generation model with and without context against the number of proof attempts. We observe that the proof generation models with context consistently outperform the plain generation model. We illustrate the complexity of generated proofs with several examples [24].

To get a better understanding of where these gains are coming from, we inspected 5 randomly sampled examples that the model using context was able to solve, but the plain generation model could not. We determined the lists of problems each model could solve, computed their difference, and then sampled 5 examples uniformly at random. For examples that had multiple correct proofs generated by the model, we selected one at random.

While the sample size is not large enough to make quantitative judgements, it appears that the model frequently makes use of similar proofs in the context. We observe that for 3 of the 5 examples, the model readily **copies and adapts** proofs that exist in its context. For another example, the model made use of a premise that did not occur in its context, which happened to also be used in the ground truth proof, but with a different tactic. In the final example, the model found a simpler proof that did not occur like this in the context. This suggests that the addition of context does not play the same role as premise selection.

RA3: LLMs can benefit from the context in which the theorem occurred in the theory file, both quantitatively by increasing proving power, and qualitatively by copying and adapting nearby proofs.

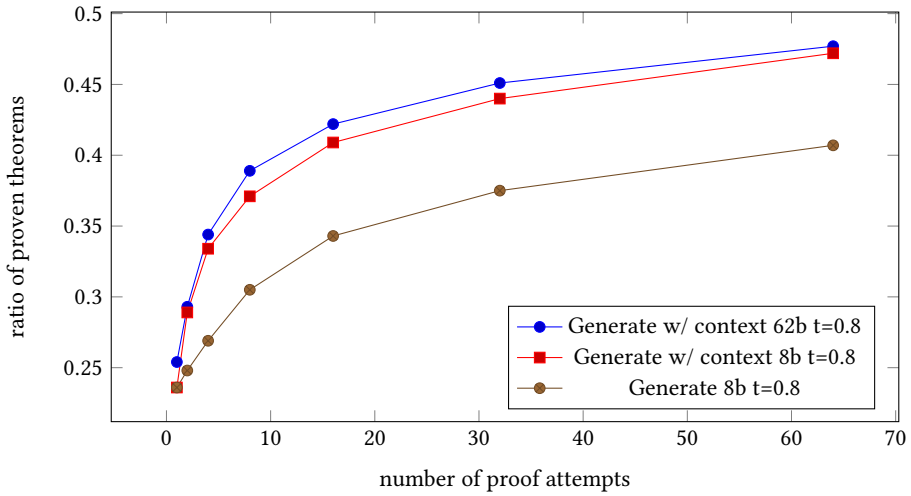

Figure 6: Ratio of theorems proven vs. inference cost for models with different sizes and temperatures.

3.6 RQ4: Does the size of the LLM affect proof synthesis effectiveness?

We fine-tuned and evaluated the 62b version of Minerva on the proof generation task with context. Figure 5 reports that for 16 samples, the large model can prove an additional 1.3% over the 8b model, resulting in a total proof rate of 42.2%. For 64 samples, the large model can prove an additional 0.4% over the 8b model, resulting in a total proof rate of 47.9%.

Figure 6 plots the proof success rate of the generation model with context for the 8b model and the 62b model against the number of proof attempts. We observe that the 62b proof generation model with context outperforms the 8b proof generation model with context. One caveat here is that we were not able to tune hyperparameters as well due to the higher cost of these experiments, so an optimally tuned 62b model may perform even better.

RA4: Theorem proving performance improves with the scale of the language model.

3.7 RQ5: How do LLMs compare to other state-of-the-art proof generation methods?

While comparisons across different neural theorem provers are hard in general, we can compare to Thor [37], one of the most effective approaches available. Thor also relies on language models, but uses smaller models (700m parameters) and uses a different kind of proof step as its prediction target. Instead of using the human ground truth proofs, Thor generates a new training set and aims to solve each proof step by generating a declarative statement, which is then solved using Sledgehammer. That is, Thor disentangles the planning stage of the next proof step, which is the specification of the target state (using a "have" statement) and premise selection, which is done by Sledgehammer. This enables Thor to solve a total of 57% of the problems.

AFP Topic	Test set	Baldur	Thor
Computer Science	4,019	50.0%	57.5%
Logic	966	51.6%	53.6%
Mathematics	2,200	41.9%	50.5%
Tools	102	53.9%	51.8%

Figure 7: Proof rate by AFP topic classification, and the number of theorems in each category. There are only 6,336 theorems in total in the test set, but the projects these theorems appear in can be covered by multiple topics.

By contrast, Baldur solves up to 47.9% of the problems. While there is a significant gap, we argue that the means by which the two techniques improve over plain language modeling are largely orthogonal. Figure 5 reports a large gain from 57% to 65.7% when we consider the union of Baldur and Thor, which supports this hypothesis. Additionally, we find that an ensemble of 10 different fine-tuned Baldur models proves 58.0%.

We compare Baldur's and Thor's proof rates on different types of problems. The AFP is indexed by four overarching topics: computer science, logic, mathematics, and tools. The authors of individual proof developments self-identity which topics their projects fall into. We use these provided topic labels to determine the categories of problems from our test that Baldur and Thor can most effectively solve. Figure 7 shows the breakdown of which theorems in the test set fall into which topics, and Baldur's and Thor's proof success rates on these theorems. In terms of relative performance, Baldur performs better than Thor on problems related to tools and similarly on problems related to logic. We observe that Thor outperforms Baldur on problems related to mathematics and computer science. For mathematics proofs, we hypothesize that premise selection may be particularly useful, and Thor's use of Sledgehammer is likely what gives it a leg up on solving these mathematics problems. Overall, we observe some complementarity in Baldur's and Thor's

effectiveness on problems of different topics, though future work should examine this complementarity in more depth.

> **RA5**: Our findings suggest that LLM-based methods and search-based methods are complementary, and together can lead to large gains in proving power.

4 DISCUSSION: WHAT'S NEXT?

Our evaluation shows that LLMs can generate whole proofs at once, and can repair their own mistakes, forming the basis for an effective and simple approach to proof synthesis. Moving forward, we find three directions particularly promising:

(1) integrating proof generation and proof repair models into a new **learnable proof search** strategy,
(2) investigating **alternative data splits** corresponding to different goals, and
(3) evaluating these techniques across **different proof assistants**.

Learnable Proof Search. While our generate + repair approach to proof synthesis lets us avoid costly proof search procedures, it also lends itself to a new proof search strategy. The search strategy would work as follows:

(1) use the generation model to sample candidate proofs,
(2) use the repair model to attempt to repair those proofs, and
(3) continue to use the repair model to repair the repair-model-generated attempts from (2).

This paves the way for a learnable proof search strategy.

We demonstrate a proof-of-concept of this new proof search strategy. We sample once using the generation model, repair the generated sample using the repair model, and repair the repair model's attempt using the repair model. When using both models, we sample with temperature 0. So the inference cost in this setup is 3 (1 for the first generation, 1 for the first repair, and 1 for the second repair).

The generate + repair approach with inference cost of 2 proves 24.9% of the test set theorems. With a second repair attempt, it proves an additional 1.3%, for a total of 26.2%. The generation approach with inference cost of 3 proves 25.4%, which is 0.8% less than the second repair attempt for the same inference cost.

To make this a more viable proof search strategy, future work needs to focus on generating proof repair training data that better mirrors the required changes for the subsequent repair attempts. When proof checking, the resulting error message is for the first occurring error, typically from the first couple of lines of the predicted proof. So the proof repair model will only learn to address these types of errors. An alternative approach could be, for example, to take the training examples from the proof generation model and use the first few lines of the human-written ground truth proof as a *proof prefix*. We could then concatenate this proof prefix to the end of the input. Since it is a decoder-only model, we can simply sample the model's attempt at the rest of the proof. If the proof prefix concatenated with the rest of the proof does not check, then that can serve as a new training example for the proof repair model.

Alternative Data Splits. The PISA benchmark that we use to evaluate our approach commits to a particular data split between training data and testing data. It is interesting to note, however, that different data splits may themselves correspond to different goals, even fixing the same evaluation task and metric. Moving forward, it may be useful to consider different kinds of data splits corresponding to different goals, even fixing the same dataset and benchmark suite. Here, we consider two different splits: *theorem-wise* and *project-wise*.

PISA uses a random theorem-wise split of the theorems appearing the AFP. This means that for any theorem in the test set, the theorems and (the corresponding proofs) that appear before or after that theorem may be in the training set. This split is useful to evaluate since a forward-looking goal of neural theorem prover researchers is to integrate these tools directly into proof assistants, where they could make use of the full project context. That project context may include human-written proofs of nearby theorems that look similar (or even identical) to one another — automatically repurposing and adapting those proofs can be quite fruitful.

By contrast with PISA, CoqGym [105], the neural theorem prover benchmark suite for the Coq proof assistant, uses a project-wise split, where training and testing data come from entirely different projects. This is useful when the goal is to help proof engineers who start completely new projects and want an automated proof synthesis tool to prove as much as it can. A tool that is trained and evaluated in a setting where it expects that it has seen proofs in a given proof development, as may happen with a theorem-wise split, may not perform as well in this new setting. Explicit consideration for the data split and the goals it achieves may help drive neural theorem proving research even further.

Different Proof Assistants. To make better sense of new strides in neural theorem proving, it makes sense to evaluate the same techniques across many different proof assistants. But this remains challenging. Consider once again the problem of data splits: since prover developments that evaluate on CoqGym [22, 23] follow the same project-wise split as CoqGym, it can be hard to make sense of how those developments compare to those trained and evaluated using theorem-wise data splits, like our own Baldur.

We used an established benchmark of Isabelle/HOL proofs to fairly compare Baldur to prior work and to increase the chances that our results generalize. However, we observed that search-based proof-synthesis tools for other proof assistants tend to prove a smaller fraction of theorems than we have found in our work. For example, Diva [22], the current state of the art for the Coq proof assistant, proves 33.8% of its benchmark automatically. This could be a reflection of size and quality of the available training data or the complexity of the available evaluation data (which, by necessity, is different from what we use because it involves theorems and proofs in different languages), or a more fundamental difference in the complexity of synthesizing proofs in these respective languages.

Future work should allow for direct comparisons by porting the developed techniques across proof assistants. Cross-proof-assistant benchmark suites may help substantially with this, but still have their limitations. For example, MiniF2F [110] implements the same benchmark suite for Math Olympiad problems across many different proof assistants. But math problems are not evenly represented

across proof assistants, which draw different user communities with different emphases. Fair comparisons between proof assistants are hard, but we do believe they are necessary.

5 RELATED WORK

Existing methods for automating formal theorem proving can be classified into two categories, hammers and search-based methods. Hammers, such as CoqHammer [19] and Sledgehammer [71], iteratively use a set of precomputed mathematical facts to attempt to "hammer" out a proof. While hammers are powerful, they lack the ability to employ certain tactics, such as induction, preventing them from proving certain large classes of theorems. Search-based methods use a prediction model that, given some information about a partially written proof, the target theorem being proven, and the current proof state, predicts a set of next likely proof steps. The methods then use metaheuristic search [32] to attempt to synthesize a proof. They iterate querying the prediction model for the likely next steps and using the proof assistant to get feedback on those steps and prune non-promising paths, generating a search tree of possible proofs. The proof assistant also determines when the proof is complete. The tools mostly differ in the prediction model they use, which are typically learned automatically. For example, ASTactic uses only the proof state [105], TacTok uses the proof state and the partially written proof script [23], Diva (which combines the use of many models) also uses the proof term [22], and Passport also uses identifier information [83]. Other search-based techniques include Tactician [7], Proverbot9001 [82], and GamePad [35] for Coq; TacticToe [26] for HOL4; and DeepHOL [5, 69] for HOL Light. Prior work has found that hammers and search-based methods are complementary, each often proving theorems the other cannot [22, 23, 105], though effective user interfaces are needed to help proof engineers use these tools [2]. Thor [37] combines a search-based method with a hammer, using both a prediction model and Sledgehammer in its search. By contrast, our Baldur uses an LLM to generate an entire proof at once and then to one-shot repair it.

The most closely related work to ours is LISA [38], which fine-tunes a pretrained language model on a large Isabelle/HOL proof corpus, and uses it inside of a search procedure to predict proof steps. GPT-f [73] likewise combines a generative language model with proof search to target the Metamath proof language. A Monte-Carlo tree search approach outperforms GPT-f in Lean [46].

TacticZero [100] learns not just tactics but also proof search strategies for end-to-end proof synthesis, rather than relying on a single fixed proof search strategy like other neural theorem proving approaches. The approach works by way of deep reinforcement learning, and improves over the previous state of the art on a benchmark for the HOL4 theorem prover.

A related problem to neural theorem proving is *autoformalization*: the automatic translation of natural language specifications and proofs into formal, machine-checkable specifications and proofs. LLMs have shown promise for autoformalization of specifications, and automatically generated proofs of the resulting autoformalized specifications have been used to improve a neural theorem prover on a widely used benchmark suite in Isabelle/HOL [101]. ProofNet [4] introduces a dataset and benchmark suite for autoformalization in Lean, based on undergraduate mathematics, and shows preliminary promising results autoformalizing proofs on that benchmark using Codex [14] with few-shot learning. Autoformalization of both theorems and proofs in Coq shows promise on a small preliminary benchmark suite [18]. Autoformalization for specification logics in verification is also promising [30].

The Draft, Sketch, and Prove method (DSP) [39] presents a hybrid between theorem proving and autoformalization, which, similar to our approach, makes use of LLMs for theorem proving. It provides informal proofs as drafts for the LLM to translate into a formal proof sketch, which is then proven via Sledgehammer. In contrast, we use fine-tuning for LLMs, do not make use of Sledgehammer, and do not rely on the availability of natural language proofs.

Pretrained language models can be used to answer natural-language mathematics questions [67]. Large language models, such as Minerva [52] and PaLM [16], have been evaluated on natural language mathematics benchmarks, such as GSM8k [17] and MATH [33]. The ProofNet [4] benchmark suite mentioned above includes informal proofs alongside formal proofs as a benchmark.

We introduce the proof repair task, with error messages. This is a new machine learning task for formal proofs. We show that solving this task improves neural theorem proving performance. Proof engineers perform proof repair constantly during formal proof development [79]. Automating this task first arose with the advent of symbolic tools for automatic proof repair in the Coq proof assistant [77], and has since made its way into tools for other proof systems [55]. Our work is among the first to explore proof repair in a machine learning context, and the first we are aware of to use error messages for a proof repair task, and to use repair to improve performance of proof synthesis.

There are numerous other tasks that machine learning tools for proofs consider that may either help users with proof development directly, or improve neural theorem proving performance themselves. For example, PaMpeR [62] predicts proof methods alongside explanations in Isabelle/HOL. ACL2(ml) [34] generates helper lemmas and suggests similar theorems in ACL2. Other popular proof-related tasks leveraging machine learning include premise selection and datatype alignment [78]. Nonfunctional and data-centered properties [25, 60, 61] can also benefit from formal verification [6], but more research is necessary both on manual and automated approaches to verifying such properties. Probabilistic verification has successfully provided guarantees for such properties for machine learning systems [27, 57, 89, 96].

Our approach can help minimize human effort in formal verification by automatically synthesizing proofs for some theorems. Other tools that assist humans writing formal verification proofs can similarly save time, and can be complementary to our work for theorems Baldur cannot prove fully automatically. iCoq [10, 11], and its parallelized version PiCoq [70], find failing proof scripts in evolving projects by prioritizing proof scripts affected by a revision. iCoq tracks fine-grained dependencies between Coq definitions, propositions, and proof scripts to narrow down the potentially affected proof scripts. QuickChick [47], a random testing tool for Coq, searches for counterexamples to executable theorems, helping a programmer to become more confident that a theorem is correct. Roosterize [63, 65] can suggest names for lemmas, and language models can also help automatically format proofs [64], both improving readability and maintainability. Mutation analysis can identify

weak specifications, when mutating definitions does not break their proofs [12, 36]. The mutation operators could, hypothetically, be applied in repair and in providing feedback for developers as to why a proof has broken.

The automated program repair field studies the task of taking a program with a bug, evidenced by one or more failing tests, and automatically producing a modified version of the program that passes all the tests [50]. Generate-and-validate repair techniques use search-based techniques or predefined templates to generate many syntactic candidate patches, validating them against the tests (e.g., GenProg [49], Prophet [53], AE [98], HDRepair [48], ErrDoc [91], JAID [13], Qlose [20], and Par [43], ssFix [102], CapGen [99], SimFix [41], Hercules [81], Recoder [111], among others). Techniques such as DeepFix [29] and ELIXIR [80] use learned models to predict erroneous program locations, as well as the patches. It is possible to learn how to repair errors together by learning how to create errors, which can increase the amount of available training data, but poses an additional challenge of learning to approximate making human-like errors [107]. Unfortunately, these automated program repair techniques often overfit to the available tests and produce patches that, while passing all the tests, fail to encode the developers' intent [59, 66, 75, 85]. Improving the quality of the resulting repairs can be done via improving fault localization strategies [3, 40, 45, 54, 58, 87, 103], patch generation algorithms (e.g., heuristic-based [41, 49, 53, 72, 91, 99], constraint-based [1, 28, 42, 56, 94], and learning-based [15, 29, 80]), and patch validation methodologies [90, 95, 104, 108, 109]. By contrast, in Baldur's domain of theorem proving, it is impossible to produce a proof that appears to prove the theorems, but actually fails to do so, because the theorem prover acts as an absolute oracle for the correctness of the proof. As a result, it may be more difficult to produce a proof in the first place, but if techniques in this domain do produce proofs, they are guaranteed to be correct.

6 CONTRIBUTIONS

This paper is the first to fine-tune large language models to generate entire proofs of theorems without the need for proof search or hammers. We demonstrate that this approach is more effective and more efficient than prior methods that use one-step-at-a-time search-based generation, and that it is complementary to existing search-based and hammer-based approaches: Together, our Baldur and prior tools can fully automatically synthesize proofs for 65.7% of the theorems in a large Isabelle/HOL benchmark, establishing a new state of the art. We further demonstrate that generate-and-repair improves proof synthesis when the language model is given access to the error messages produced by erroneous proofs.

This work opens new avenues of research into (1) using LLMs to automate theorem proving and simplify formal verification of software properties, (2) repair approaches, both for proofs and, potentially, more traditional automated program repair tasks, and (3) the use of context (e.g., failed synthesis attempts and error messages) in proof generation. Our very encouraging results suggest a bright future for automated proof generation and repair using LLMs.

DATA AVAILABILITY

This work uses T5X, which is publicly available: https://github.com/google-research/t5x. The scripts to launch training and inference, and to process the training data and results rely on proprietary Google infrastructure, which inhibits us from publicly releasing the code. Our evaluation uses the PISA codebase and dataset, which are also publicly available: https://github.com/albertqjiang/Portal-to-ISAbelle.

ACKNOWLEDGMENTS

This work was performed at Google., Inc. We thank Stella Biderman, Ernest Davis, and others who provided feedback on an earlier draft of this paper. This work is supported by the Defense Advanced Research Projects Agency under grant no. DARPA HR0011-22-9-0063, and by the National Science Foundation under grant no. CCF-2210243.

REFERENCES

[1] Afsoon Afzal, Manish Motwani, Kathryn T. Stolee, Yuriy Brun, and Claire Le Goues. 2021. SOSRepair: Expressive Semantic Search for Real-World Program Repair. *IEEE TSE* 47, 10 (October 2021), 2162–2181. https://doi.org/10.1109/TSE.2019.2944914

[2] Arpan Agrawal, Emily First, Zhanna Kaufman, Tom Reichel, Shizhuo Zhang, Timothy Zhou, Alex Sanchez-Stern, Talia Ringer, and Yuriy Brun. 2023. Proofster: Automated Formal Verification. In *ICSE Demo* (14–20). Melbourne, Australia.

[3] Fatmah Yousef Assiri and James M Bieman. 2017. Fault Localization for Automated Program Repair: Effectiveness, Performance, Repair Correctness. *Software Quality Journal* 25, 1 (2017), 171–199. https://doi.org/10.1007/s11219-016-9312-z

[4] Zhangir Azerbayev, Bartosz Piotrowski, and Jeremy Avigad. 2022. ProofNet: A benchmark for autoformalizing and formally proving undergraduate-level mathematics problems. In *Workshop MATH-AI: Toward Human-Level Mathematical Reasoning*. New Orleans, LA, USA.

[5] Kshitij Bansal, Sarah M. Loos, Markus N. Rabe, Christian Szegedy, and Stewart Wilcox. 2019. HOList: An Environment for Machine Learning of Higher Order Logic Theorem Proving. In *ICML*, Vol. 97. PMLR, Long Beach, CA, USA, 454–463. http://proceedings.mlr.press/v97/bansal19a.html

[6] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella-Béguelin. 2013. Probabilistic Relational Reasoning for Differential Privacy. *ACM TOPLAS* 35, 3 (Nov. 2013), 9:2–9:49. https://doi.org/10.1145/2492061

[7] Lasse Blaauwbroek, Josef Urban, and Herman Geuvers. 2020. The Tactician. In *Intelligent Computer Mathematics*. 271–277.

[8] Sidney Black et al. 2022. GPT-NeoX-20B: An Open-Source Autoregressive Language Model. In *BigScience Episode #5 — Workshop on Challenges & Perspectives in Creating Large Language Models*. 95–136. https://doi.org/10.18653/v1/2022.bigscience-1.9

[9] Tom B. Brown et al. 2020. Language Models Are Few-Shot Learners. In *NeurIPS*.

[10] Ahmet Celik, Karl Palmskog, and Milos Gligoric. 2017. ICoq: Regression proof selection for large-scale verification projects. In *ASE*. Urbana-Champaign, IL, USA, 171–182. https://doi.org/10.1109/ASE.2017.8115630

[11] Ahmet Celik, Karl Palmskog, and Milos Gligoric. 2018. A Regression Proof Selection Tool for Coq. In *ICSE Demo Track*. Gothenburg, Sweden, 117–120. https://doi.org/10.1145/3183440.3183493

[12] Ahmet Celik, Karl Palmskog, Marinela Parovic, Emilio Jesús Gallego Arias, and Milos Gligoric. 2019. Mutation Analysis for Coq. In *ASE*. San Diego, CA, USA, 539–551. https://doi.org/10.1109/ASE.2019.00057

[13] Liushan Chen, Yu Pei, and Carlo A. Furia. 2017. Contract-based program repair without the contracts. In *ASE*. Urbana, IL, USA, 637–647.

[14] Mark Chen et al. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). https://arxiv.org/abs/2107.03374

[15] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE TSE* 47, 9 (2019), 1943–1959. https://doi.org/10.1109/TSE.2019.2940179

[16] Aakanksha Chowdhery et al. 2022. PaLM: Scaling Language Modeling with Pathways. *CoRR* abs/2204.02311 (2022). https://doi.org/10.48550/arXiv.2204.02311 arXiv:2204.02311

[17] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training Verifiers to Solve Math Word Problems. *CoRR* abs/2110.14168 (2021). arXiv:2110.14168 https://arxiv.org/abs/2110.14168

[18] Garett Cunningham, Razvan C. Bunescu, and David Juedes. 2023. Towards Autoformalization of Mathematics and Code Correctness: Experiments with Elementary Proofs. *CoRR* abs/2301.02195 (2023). https://doi.org/10.48550/arXiv.2301.02195

[19] Łukasz Czajka and Cezary Kaliszyk. 2018. Hammer for Coq: Automation for Dependent Type Theory. *Journal of Automated Reasoning* 61, 1-4 (2018), 423–453. https://doi.org/10.1007/s10817-018-9458-4

[20] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program Repair with Quantitative Objectives. In *CAV*. Toronto, ON, Canada, 383–401.

[21] Angela Fan, Mike Lewis, and Yann Dauphin. 2018. Hierarchical Neural Story Generation. In *ACL*. Melbourne, Australia, 889–898. https://doi.org/10.18653/v1/P18-1082

[22] Emily First and Yuriy Brun. 2022. Diversity-Driven Automated Formal Verification. In *ICSE* (22–27). Pittsburgh, PA, USA, 749–761. https://doi.org/10.1145/3510003.3510138

[23] Emily First, Yuriy Brun, and Arjun Guha. 2020. TacTok: Semantics-Aware Proof Synthesis. *PACMPL OOPSLA* 4 (November 2020), 231:1–231:31. https://doi.org/10.1145/3428299

[24] Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-Proof Generation and Repair with Large Language Models. *CoRR* abs/2303.04910 (2023). https://arxiv.org/abs/2303.04910.

[25] Sainyam Galhotra, Yuriy Brun, and Alexandra Meliou. 2017. Fairness Testing: Testing Software for Discrimination. In *ESEC/FSE* (6–8). Paderborn, Germany, 498–510. https://doi.org/10.1145/3106237.3106277

[26] Thibault Gauthier, Cezary Kaliszyk, Josef Urban, Ramana Kumar, and Michael Norrish. 2021. TacticToe: Learning to Prove with Tactics. *Journal of Automated Reasoning* 65, 2 (February 2021), 257–286. https://doi.org/10.1007/s10817-020-09580-x

[27] Stephen Giguere, Blossom Metevier, Yuriy Brun, Bruno Castro da Silva, Philip S. Thomas, and Scott Niekum. 2022. Fairness Guarantees under Demographic Shift. In *ICLR* (25–29). https://openreview.net/forum?id=wbPObLm6ueA

[28] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2018. Automated Clustering and Program Repair for Introductory Programming Assignments. In *PLDI*. Philadelphia, PA, USA, 465–480. https://doi.org/10.1145/3192366.3192387

[29] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *AAAI*. San Francisco, CA, USA, 1345–1351.

[30] Christopher Hahn, Frederik Schmitt, Julia J. Tillman, Niklas Metzger, Julian Siber, and Bernd Finkbeiner. 2022. Formal Specifications from Natural Language. *CoRR* abs/2206.01962 (2022). https://doi.org/10.48550/arXiv.2206.01962

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

[32] Mark Harman. 2007. The Current State and Future of Search Based Software Engineering. In *ICSE*. 342–357. https://doi.org/10.1109/FOSE.2007.29

[33] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring Mathematical Problem Solving With the MATH Dataset. *CoRR* abs/2103.03874 (2021). https://arxiv.org/abs/2103.03874

[34] Jónathan Heras and Ekaterina Komendantskaya. 2014. ACL2(ml): Machine-learning for ACL2. *Electronic Proceedings in Theoretical Computer Science* 152 (04 2014). https://doi.org/10.4204/EPTCS.152.5

[35] Daniel Huang, Prafulla Dhariwal, Dawn Song, and Ilya Sutskever. [n. d.]. GamePad: A Learning Environment for Theorem Proving. In *ICLR), year = 2019, url = https://openreview.net/forum?id=r1xwKoR9Y7,*.

[36] Kush Jain, Karl Palmskog, Ahmet Celik, Emilio Jesús Gallego Arias, and Milos Gligoric. 2020. MCoq: Mutation Analysis for Coq Verification Projects. In *ICSE Demo Track*. Seoul, South Korea, 89–92. https://doi.org/10.1145/3377812.3382156

[37] Albert Jiang, Konrad Czechowski, Mateja Jamnik, Piotr Milos, Szymon Tworkowski, Wenda Li, and Yuhuai Tony Wu. 2022. Thor: Wielding Hammers to Integrate Language Models and Automated Theorem Provers. In *NeurIPS*.

[38] Albert Qiaochu Jiang, Wenda Li, Jesse Michael Han, and Yuhuai Wu. 2021. LISA: Language models of ISAbelle proofs. In *Conference on Artificial Intelligence and Theorem Proving (AITP)*. Aussois, France, 17.1–17.3.

[39] Albert Q. Jiang, Sean Welleck, Jin Peng Zhou, Wenda Li, Jiacheng Liu, Mateja Jamnik, Timothée Lacroix, Yuhuai Wu, and Guillaume Lample. 2022. Draft, Sketch, and Prove: Guiding Formal Theorem Provers with Informal Proofs. *CoRR* abs/2210.12283 (2022). https://doi.org/10.48550/arXiv.2210.12283

[40] Jiajun Jiang, Yingfei Xiong, and Xin Xia. 2019. A manual inspection of Defects4J bugs and its implications for automatic program repair. *Science China Information Sciences* 62, 10 (2019), 200102.

[41] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping Program Repair Space with Existing Patches and Similar Code. In *ISSTA*. Amsterdam, The Netherlands, 298–309. https://doi.org/10.1145/3213846.3213871

[42] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing Programs with Semantic Code Search. In *ASE* (9–13). Lincoln, NE, USA, 295–306. https://doi.org/10.1109/ASE.2015.60

[43] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *ICSE*. San Francisco, CA, USA, 802–811. https://doi.org/10.1109/ICSE.2013.6606626

[44] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. SeL4: Formal Verification of an OS Kernel. In *SOSP*. Big Sky, Montana, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[45] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. 2019. iFixR: Bug Report Driven Program Repair. In *ESEC/FSE*. 314–325. https://doi.org/10.1145/3338906.3338935

[46] Guillaume Lample, Marie-Anne Lachaux, Thibaut Lavril, Xavier Martinet, Amaury Hayat, Gabriel Ebner, Aurélien Rodriguez, and Timothée Lacroix. 2022. HyperTree Proof Search for Neural Theorem Proving. *CoRR* abs/2205.11491 (2022). https://doi.org/10.48550/arXiv.2205.11491

[47] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. 2017. Generating Good Generators for Inductive Relations. *PACMPL* 2, POPL (Dec. 2017), 45:1–45:30. https://doi.org/10.1145/3158133

[48] Xuan Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *SANER*, Vol. 1. 213–224. https://doi.org/10.1109/SANER.2016.76

[49] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE TSE* 38 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[50] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *CACM* 62, 12 (Nov. 2019), 56–65. https://doi.org/10.1145/3318162

[51] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *CACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

[52] Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay V. Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. 2022. Solving Quantitative Reasoning Problems with Language Models. *CoRR* abs/2206.14858 (2022). https://doi.org/10.48550/arXiv.2206.14858

[53] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *POPL*. St. Petersburg, FL, USA, 298–312. https://doi.org/10.1145/2837614.2837617

[54] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. 2020. Can Automated Program Repair Refine Fault Localization? A Unified Debugging Approach. In *ISSTA*. Virtual Event, USA, 75–87. https://doi.org/10.1145/3395363.3397351

[55] Paolo Masci and Aaron Dutle. 2022. Proof Mate: An Interactive Proof Helper for PVS (Tool Paper). In *NASA Formal Methods Symposium*. Springer, 809–815.

[56] Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2018. Semantic Program Repair Using a Reference Implementation. In *ICSE*. 129–139. https://doi.org/10.1145/3180155.3180247

[57] Blossom Metevier, Stephen Giguere, Sarah Brockman, Ari Kobren, Yuriy Brun, Emma Brunskill, and Philip S. Thomas. 2019. Offline Contextual Bandits with High Probability Fairness Guarantees. In *NeurIPS* (9–14). Vancouver, BC, Canada, 14893–14904. http://papers.neurips.cc/paper/9630-offline-contextual-bandits-with-high-probability-fairness-guarantees

[58] Manish Motwani and Yuriy Brun. 2023. Better Automatic Program Repair by Using Bug Reports and Tests Together. In *ICSE* (14–20). Melbourne, Australia.

[59] Manish Motwani, Mauricio Soto, Yuriy Brun, René Just, and Claire Le Goues. 2022. Quality of Automated Program Repair on Real-World Defects. *IEEE TSE* 48, 2 (February 2022), 637–661. https://doi.org/10.1109/TSE.2020.2998785

[60] Kıvanç Muşlu, Yuriy Brun, and Alexandra Meliou. 2013. Data Debugging with Continuous Testing. In *ESEC/FSE New Ideas Track* (18–26). Saint Petersburg, Russia, 631–634. https://doi.org/10.1145/2491411.2494580

[61] Kıvanç Muşlu, Yuriy Brun, and Alexandra Meliou. 2015. Preventing Data Errors with Continuous Testing. In *ISSTA* (12–17). Baltimore, MD, USA, 373–384. https://doi.org/10.1145/2771783.2771792

[62] Yutaka Nagashima and Yilun He. 2018. PaMpeR: Proof Method Recommendation System for Isabelle/HOL. In *ASE*. Montpellier, France, 362–372. https://doi.org/10.1145/3238147.3238210

[63] Pengyu Nie, Karl Palmskog, Junyi Jessy Li, and Milos Gligoric. 2020. Deep Generation of Coq Lemma Names Using Elaborated Terms. In *IJCAR*. Paris, France, 97–118.

[64] Pengyu Nie, Karl Palmskog, Junyi Jessy Li, and Milos Gligoric. 2020. Learning to Format Coq Code Using Language Models. In *The Coq Workshop*. Aubervilliers, France.

[65] Pengyu Nie, Karl Palmskog, Junyi Jessy Li, and Milos Gligoric. 2021. Roosterize: Suggesting Lemma Names for Coq Verification Projects Using Deep Learning. In *ICSE Demo Track*. Madrid, Spain, 21–24. https://doi.org/10.1109/ICSE-Companion52605.2021.00026

[66] Kunihiro Noda, Yusuke Nemoto, Keisuke Hotta, Hideo Tanida, and Shinji Kikuchi. 2020. Experience Report: How Effective is Automated Program Repair for Industrial Software?. In *SANER*. 612–616.

[67] Kimia Noorbakhsh, Modar Sulaiman, Mahdi Sharifi, Kallol Roy, and Pooyan Jamshidi. 2021. Pretrained Language Models are Symbolic Mathematics Solvers too! *CoRR* abs/2110.03501 (2021). https://arxiv.org/abs/2110.03501

[68] Maxwell I. Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. 2021. Show Your Work: Scratchpads for Intermediate Computation with Language Models. *CoRR* abs/2112.00114 (2021).

[69] Aditya Paliwal, Sarah M. Loos, Markus N. Rabe, Kshitij Bansal, and Christian Szegedy. 2020. Graph Representations for Higher-Order Logic and Theorem Proving. In *AAAI and EAAI*. New York, NY, USA, 2967–2974.

[70] Karl Palmskog, Ahmet Celik, and Milos Gligoric. 2018. PiCoq: Parallel Regression Proving for Large-Scale Verification Projects. In *ISSTA*. Amsterdam, Netherlands, 344–355. https://doi.org/10.1145/3213846.3213877

[71] Larry Paulson and Tobias Nipkow. 2023. The Sledgehammer: Let Automatic Theorem Provers write your Isabelle scripts! https://isabelle.in.tum.de/website-Isabelle2009-1/sledgehammer.html.

[72] Justyna Petke and Aymeric Blot. 2018. Refining Fitness Functions in Test-Based Program Repair. In *Workshop on Automated Program Repair (APR)*. Seoul, Republic of Korea, 13–14. https://doi.org/10.1145/3387940.3392180

[73] Stanislas Polu and Ilya Sutskever. 2020. Generative Language Modeling for Automated Theorem Proving. *CoRR* abs/2009.03393 (2020). https://arxiv.org/abs/2009.03393

[74] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2022. Efficiently Scaling Transformer Inference. *CoRR* abs/2211.05102 (2022). https://doi.org/10.48550/arXiv.2211.05102

[75] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems. In *ISSTA*. Baltimore, MD, USA, 24–36. https://doi.org/10.1145/2771783.2771791

[76] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21 (2020), 140:1–140:67. http://jmlr.org/papers/v21/20-074.html

[77] Talia Ringer. 2021. *Proof Repair*. Ph. D. Dissertation. University of Washington.

[78] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. 2019. QED at Large: A Survey of Engineering of Formally Verified Software. *Foundations and Trends® in Programming Languages* 5, 2–3 (2019), 102–281.

[79] Talia Ringer, Alex Sanchez-Stern, Dan Grossman, and Sorin Lerner. 2020. REPLica: REPL Instrumentation for Coq Analysis. In *International Conference on Certified Programs and Proofs (CPP)*. New Orleans, LA, USA, 99—-113. https://doi.org/10.1145/3372885.3373823

[80] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. 2017. ELIXIR: Effective object oriented program repair. In *ASE*. 648–659.

[81] Seemanta Saha, Ripon K. Saha, and Mukul R. Prasad. 2019. Harnessing Evolution for Multi-Hunk Program Repair. In *ICSE* (29–31). Montreal, QC, Canada, 13–24. https://doi.org/10.1109/ICSE.2019.00020

[82] Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, and Sorin Lerner. 2020. Generating Correctness Proofs with Neural Networks. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2020)*. London, UK, 1–10. https://doi.org/10.1145/3394450.3397466

[83] Alex Sanchez-Stern, Emily First, Timothy Zhou, Zhanna Kaufman, Yuriy Brun, and Talia Ringer. 2023. Passport: Improving Automated Formal Verification Using Identifiers. *ACM TOPLAS* 45, 2, Article 12 (June 2023), 12:1-12:30 pages. https://doi.org/10.1145/3593374

[84] Noam Shazeer. 2019. Fast Transformer Decoding: One Write-Head is All You Need. *CoRR* abs/1911.02150 (2019). https://doi.org/10.48550/arXiv.1911.02150

[85] Edward K. Smith, Earl Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the Cure Worse than the Disease? Overfitting in Automated Program Repair. In *ESEC/FSE*. 532–543. https://doi.org/10.1145/2786805.2786825

[86] Jianlin Su, Yu Lu, Shengfeng Pan, Bo Wen, and Yunfeng Liu. 2021. RoFormer: Enhanced Transformer with Rotary Position Embedding. *CoRR* abs/2104.09864 (2021). https://doi.org/10.48550/arXiv.2104.09864

[87] Shuyao Sun, Junxia Guo, Ruilian Zhao, and Zheng Li. 2018. Search-Based Efficient Automated Program Repair Using Mutation and Fault Localization. In *COMPSAC*, Vol. 1. 174–183. https://doi.org/10.1109/COMPSAC.2018.00030

[88] The Coq Development Team. 2017. Coq, v.8.7. https://coq.inria.fr.

[89] Philip S. Thomas, Bruno Castro da Silva, Andrew G. Barto, Stephen Giguere, Yuriy Brun, and Emma Brunskill. 2019. Preventing Undesirable Behavior of Intelligent Machines. *Science* 366, 6468 (November 2019), 999–1004. https://doi.org/10.1126/science.aag3311

[90] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F. Bissyandé. 2020. Evaluating Representation Learning of Code Changes for Predicting Patch Correctness in Program Repair. In *ASE*. Melbourne, Australia, 981–992. https://doi.org/10.1145/3324884.3416532

[91] Yuchi Tian and Baishakhi Ray. 2017. Automatically diagnosing and repairing error handling bugs in C. In *ESEC/FSE*. Paderborn, Germany, 752–762.

[92] Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew W. Senior, and Koray Kavukcuoglu. 2016. WaveNet: A Generative Model for Raw Audio. *CoRR* abs/1609.03499 (2016). https://doi.org/10.48550/arXiv.1609.03499

[93] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *NeurIPS*.

[94] Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Search, align, and repair: Data-driven feedback generation for introductory programming exercises. In *PLDI*. Philadelphia, PA, USA, 481–495. https://doi.org/10.1145/3296979.3192384

[95] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. 2020. Automated Patch Correctness Assessment: How Far Are We?. In *ASE*. Melbourne, Australia, 968–980. https://doi.org/10.1145/3324884.3416590

[96] Aline Weber, Blossom Metevier, Yuriy Brun, Philip S. Thomas, and Bruno Castro da Silva. 2022. Enforcing Delayed-Impact Fairness Guarantees. *CoRR* abs/2208.11744 (2022). https://doi.org/10.48550/arXiv.2208.11744

[97] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed H. Chi, Quoc Le, and Denny Zhou. 2022. Chain of Thought Prompting Elicits Reasoning in Large Language Models. *CoRR* abs/2201.11903 (2022). https://doi.org/10.48550/arXiv.2201.11903

[98] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. 2013. Leveraging Program Equivalence for Adaptive Program Repair: Models and First Results. In *ASE*. Palo Alto, CA, USA, 356–366. https://doi.org/10.1109/ASE.2013.6693094

[99] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *ICSE*. Gothenburg, Sweden, 1–11. https://doi.org/10.1145/3180155.3180233

[100] Minchao Wu, Michael Norrish, Christian Walder, and Amir Dezfouli. 2021. TacticZero: Learning to Prove Theorems from Scratch with Deep Learning. In *NeurIPS*. https://openreview.net/forum?id=edmYVRkYZv

[101] Yuhuai Wu, Albert Q. Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. 2022. Autoformalization with Large Language Models. *CoRR* abs/2205.12615 (2022). https://doi.org/10.48550/ARXIV.2205.12615

[102] Qi Xin and Steven P. Reiss. 2017. Identifying Test-suite-overfitted Patches through Test Case Generation. In *ISSTA*. 226–236. https://doi.org/10.1145/3092703.3092718

[103] Deheng Yang, Yuhua Qi, and Xiaoguang Mao. 2018. Evaluating the Strategies of Statement Selection in Automated Program Repair. In *International Conference on Software Analysis, Testing, and Evolution (SATE)*. 33–48. https://doi.org/10.1007/978-3-030-04272-1_3

[104] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better test cases for better automated program repair. In *ESEC/FSE*. Paderborn, Germany, 831–841. https://doi.org/10.1145/3106237.3106274

[105] Kaiyu Yang and Jia Deng. 2019. Learning to prove theorems via interacting with proof assistants. In *ICML*. 6984–6994.

[106] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *PLDI*. San Jose, CA, USA, 283–294. https://doi.org/10.1145/1993498.1993532

[107] Michihiro Yasunaga and Percy Liang. 2021. Break-it-fix-it: Unsupervised learning for program repair. In *ICML*. 11941–11952.

[108] He Ye, Matias Martinez, and Martin Monperrus. 2021. Automated patch assessment for program repair at scale. *Empirical Software Engineering (EMSE)* 26, 2 (2021), 1–38. https://doi.org/10.1007/s10664-020-09920-w

[109] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. 2019. Alleviating patch overfitting with automatic test generation: A study of feasibility and effectiveness for the Nopol repair system. *Empirical Software Engineering (EMSE)* 24, 1 (2019), 33–67. https://doi.org/10.1007/s10664-018-9619-4

[110] Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. 2022. miniF2F: A cross-system benchmark for formal Olympiad-level mathematics. In *ICLR*. https://openreview.net/forum?id=9ZPegFuFTFv

[111] Qihao Zhu, Zeyu Sun, Yuan an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *ESEC/FSE*. 341–353. https://doi.org/10.1145/3468264.3468544

Received 2023-02-02; accepted 2023-07-27