Prototypical data extraction and interaction for the Isabelle proof assistant

Jonathan Julián Huerta y Munive
Czech Technical University in Prague, Czechia

Abstract

This article describes an openly available prototypical implementation of a framework for extracting theorem-proving data from the Isabelle proof assistant and interacting with the prover via Python and Scala read-eval-print loops (REPL). The framework is tested by generating proof data from 187,210 proofs from Isabelle's Archive of Formal Proofs (AFP). To showcase the framework's flexibility and usefulness for performing Machine Learning (ML) experiments, the data is further split into training, evaluation and testing sets per AFP entry, and serves for training several T5 large-language-models on it. A small-scale experiment varying the data format, model size, and training method reproduces conclusions from the ML literature about the benefits of fine-tuning models on domain-specific corpora. Finally, the REPL serves to further evaluate the models, which produce simple proofs of several AFP theorems from the test data set. Overall, the work presented here contributes evidence for the promising integration between Machine Learning and Interactive Theorem Proving. It is also one of the first steps towards turning Isabelle into a reinforcement learning environment.

2012 ACM Subject Classification Theory of computation → Automated reasoning; Computing methodologies → Natural language generation

Keywords and phrases Proof assistants, LLMs, interactive theorem proving, Isabelle

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Funding *Jonathan Julián Huerta y Munive*: MSCA Postdoctoral Fellowship (DeepIsaHOL) 101102608.

Acknowledgements I want to thank ...

1 Introduction

Machine learning (ML) methods have demonstrated significant potential in assisting proof automation in interactive theorem provers (ITPs) [10, 40, 4, 30, 2, 16, 32, 41, 38]. For instance, users of the Isabelle proof assistant have widely adopted the Sledgehammer tool [5], which combines a k-nearest-neighbour (kNN) algorithm and various automated theorem provers (ATPs) for premise selection. Moreover, there is work in progress for integrating not only *hammers* but more modern ML methods in the most used ITPs such as Coq [4, 30], Lean [32, 41, 38], HOL4 [10, 40], and HOL Light [1, 2]. Yet, several factors hinder the progress of the scientific field concerned with these integrations.

Firstly, some ITPs have received more attention than others. Despite their popularity, well-engineered design, large repositories of proofs, and long-standing history as ITPs, older provers such as Coq or Isabelle have received comparatively less attention than Lean. Fortunately, among the first steps targeting this issue in Coq came from work on the Tactician [4]. This is an OCaml implementation directly integrated with Coq that has allowed ML experiments using kNNs, graph neural networks and large language models (LLMs). However, in Isabelle, the Portal-to-Isabelle (PISA) project [14] that provided a similar integration has not been updated since the beginning of 2023. Its impressive results using LLMs for autoformalisation [16] and premise selection [15, 21] have not been transformed into user tools.

Secondly, comparing the integrations among provers is challenging. This stems from the variety of prover implementations, prover-ML integrations, and learning objectives in

the integration. While some provers use dependent types and proof terms, others use the logic of computable functions (LCF) and type-checking. An ML system optimised for one kind of data may not work in another. Moreover, while some ITP-ML integrations focus on premise selection [21], others tackle proof-method selection [22, 23, 24], autoformalisation [16], conjecturing [25], proof reconstruction [29], or a (partial) combination of all of these [10, 40]. The current solution to compare these diverse integrations is a generic benchmark [43] that formalises several mathematical problems in different provers. Yet, a complementary, community-based, and robust solution is to provide interfaces for ML models on each ITP. Then, developers of ML models can use these interfaces to create evaluations for their models. Implementations such as the Tactician or LeanDojo [41] already turn ITPs into environments and make them part of the models' training and evaluation.

This paper reports first steps in achieving the same for the Isabelle proof assistant. That is, it describes an open implementation, usage, and evaluation of a prototypical framework for extracting theorem-proving data from Isabelle and for interacting with the ITP via Python and Scala read-eval-print loops (REPL). The framework has been tested on a large dataset of proofs from Isabelle's Archive of Formal Proofs (AFP) and it has been used to train several LLMs [28, 6]. The evaluation of the models presented here adds evidence for the promising integration of ML methods and ITPs. Overall, the framework represents a significant step towards turning Isabelle into a reinforcement learning environment [35, 34].

Specifically, the framework provides a data-extraction algorithm (Section 2) that can traverse Isabelle sessions, detect all proofs in their `thy`-files, and output a JSON representation of each proof that includes: all its proof steps; the variables, constants, and type-variables in that proof step; the Isabelle-representation of the step's proof obligation, the user-seen proof state, and the proof methods and keywords available at that proof step. Where previous data-extraction systems for Isabelle focused on lemmas, the work presented here also finds proofs about function- and type-definitions. Moreover, a claimed limitation from previous approaches was the lack of typing data [20]. In contrast, the proof extraction process described here includes variables, constants and type-variables. Notably, this output is more detailed and context-aware than previous Isabelle-focused data-extraction approaches [15, 20, 21].

By creating a REPL for controlled experimentation within Isabelle (Section 3), the project aims to facilitate comparisons and potentially repurpose ML models developed for other provers. To exemplify the framework's usage, this article includes the training of LLMs on the generated data and performs small-scale experiments that reproduce conclusions from the ML literature about the benefits of fine-tuning models after being pre-trained on generic corpus of data (Section 4). The resulting models are capable of suggesting sound proof steps and automatically prove simple theorems from the test set (Section 5).

The article concludes by discussing the limitations of the current implementation (Section 6) and provides a discussion of the approach in the context of previous projects (Section 7). Ultimately, the vision (Section 8) for the contributions presented here is twofold: (1) to interface Isabelle with the best ML models available in order to create useful tools for ITP users, and (2) to explore the limits of proof automation in ITPs by turning Isabelle into a reinforcement learning environment [35, 34]. In the spirit of contributing to open research and foster future collaborations, the source code for this work, weights of the trained models, and examples of the generated data are openly available [12, 11].

2 Data extraction

Isabelle is a 39-year-old proof assistant whose logical core builds on the "LCF approach" that models inferences as functions on abstract datatypes [39]. It is one of the most automated ITPs, has a user-friendly structured proof language (Isar), explicit proof terms, and a user base of at least 523 people that have contributed approximately 284,000 lemmas to Isabelle's Archive of Formal Proofs (AFP) [19]. Its implementation philosophy is that its core, written in the Isabelle/ML (for meta-language) programming language, is in charge of the processing of mathematics, while the interface and other external tools interact with and follow this core's lead via Isabelle/Scala. The work described here partially follows Isabelle's implementation philosophy and defines and re-uses Isabelle/ML functions for the data-extraction process. Generally, this section's contribution is a collection of Isabelle/ML APIs for extracting and interacting with Isabelle's data. These result in a function that takes as input an Isabellle `thy`-file, and returns JSON representations of the file's proofs.

Implementation Firstly, using Isabelle/ML's `thy`-file parser enables the generation of a list of top-level *transitions*. These are Isabelle's official constructs for manipulating the prover's top-level states that carry theory information (e.g. imported theorems), proof context information (e.g. current user configuration) and, when proving, proof-states. Intuitively, top-level transitions τ_i are functions mapping a top-level state s_i to the next one s_{i+1} with optional error messages ϵ_{i+1} if the transition was not valid in the context of the `thy` file (that is, $\tau_i\, s_i = \langle s_{i+1}, \epsilon_{i+1}\rangle$). Alternatively, transitions correspond to user commands and their arguments (for example, `have "F x = y"` or `apply blast`). The APIs developed here augment Isabelle/ML's transitions with the user-text that produced them and call such pairings *actions* a_i. Using Isabelle's keyword classification, it is possible to generate an action-based representation of proofs, definitions or other context blocks by identifying their starting and ending actions in the `thy`-file. The APIs allow for the retrieval of these context blocks. In particular, they can produce a sequence of proofs $\langle p_j\rangle_{j\in J}$ from the sequence of the `thy`-file's actions $\langle a_i\rangle_{i\in I}$, where each proof is itself a sequence of actions. Finally, this work augments proofs' actions a_i with the (previous) top-level state s_{i-1}, the proof's goal, the user-available methods at that point, the list of previous theorems used in the proof, and the constants, variables, and type variables in each proof step. Pretty-printing this information into a properly formatted string becomes the JSON object representation of the proof.

In order to correctly retrieve all the context-dependent information for a given `thy`-file, it is necessary to provide an initial `Theory`. These can be roughly viewed as Isabelle/ML data-carrying entities containing persistent information from the parent `thy`-files. Isabelle creates this information by managing theory-dependencies via `ROOT` files. Such files list the `thy`-files that the user-formalisation comprises and its dependencies. Isabelle keeps a record of some `ROOT` files and uses them to find more dependencies. To simplify the `ROOT`-location process, this work also provides an import-dependency graph of key-value pairs, where the keys are paths to a `thy`-file, and the values are the `Theory` entities produced at the end of each file. Each node in the graph connects a `thy`-file to its parent `Theory`s. To transparently build such a graph, the `scala-isabelle` library [36] enables representing Isabelle/ML constructs as Scala objects. Thus, top-level states, transitions, proof contexts, and theories can be directly manipulated at the Scala level. This is particularly useful for retrieving the list of parent `Theory`'s that each `thy`-file requires to be successfully parsed. Crucially, the `Theory` graph uses Isabelle itself to find dependencies not located in the `thy`-file's current directory. The `Theory` graph allows the creation of initial top-level states for each `thy`-file in the graph. Its combination with the JSON generation produces the data extraction algorithm.

API example The Scala snippet below demonstrates how to use the data-extraction algorithm in the Scala interactive shell. Alternatively, it can be run directly via

```
sbt "run /path/to/a/read/directory/ /path/to/a/writing/directory/"
```

from the project's top directory. Each `Writer` object has a corresponding `minion.scala` module that initialises a working directory and orchestrates the Isabelle theory stack via the `imports.scala` graph. The `minion` has methods for explicit resource management (`isabelle.destroy()`) of Isabelle ensuring efficient cleanup of its state after using it. The `writer` module provides methods to extract the data in JSON format from a single file (`write_data`) and the entire Isabelle session (`write_all`).

```
import isabelle_rl._
import de.unruh.isabelle.control.Isabelle

val logic = "LTL" // Isabelle session name

val writer = new Writer(
    "/path/to/a/read/directory/", // it can have a ROOT, or a .thy file
    "/path/to/a/writing/directory/",
    logic)

writer.write_data("your/file.thy")

writer.write_all()

val minion = writer.get_minion()

implicit val isabelle: Isabelle = minion.isabelle

isabelle.destroy()
```

Listing 1 Interaction with the data-extraction algorithm for Isabelle

Proof data example The listing below abbreviates a generated JSON object of the 5th proof in the `Table.thy` file from the AFP entry `MFOTL_Monitor` [31]. Each generated file includes the proof's steps, available proof scripting (Isar) keywords, backward-reasoning (apply-style) keywords, methods (`methods`), and lemmas the proof depends on (`deps`). Additionally, each proof step includes the user `action`, the user state before said action, the proof-obligation as an Isabelle/ML `term`, extra user-added hypothesis `hyps`, previously `proven` terms within the proof, `variables`, `constants`, and `type variables` in that step.

```
{
  "proof": {
    "steps": [
      {
        "step": {
          "action": "lemma in_empty_table[simp]: \"\<not> x \<in>
              empty_table\"",
          "user_state": "",
          "term": "x \<notin> empty_table \<Longrightarrow> (x \<
              notin> empty_table)",
          "hyps": [],
          "proven": [],
          "variables": [{"Type0": "x :: 'a"}],
          "constants": [
            {"Type0": "Pure.prop :: prop \<Rightarrow> prop"},
            {"Type1": "empty_table :: 'a set"},
```

```
            ...
            {"Type5": "(\<Longrightarrow>) :: prop \<Rightarrow> prop
                \<Rightarrow> prop"}
          ],
          "type variables": [{"Sort0": "'a :: type"}]
        }
      },
      { ... }  // Other steps omitted for brevity
    ],
    "apply_kwrds": [
      {"name": "\\<proof>"},
      {"name": "sorry"},
      {"name": "..."},  // Many apply-style commands omitted
    ],
    "isar_kwrds": [
      {"name": "define"},
      {"name": "assume"},
      {"name": "..."},  // Many Isar-style commands omitted
    ],
    "methods": [
      {"name": "Metis.metis"},
      {"name": "Transfer.transfer_prover_start"},
      {"name": "..."},  // Many methods omitted
    ],
    "deps": [
      {"thm": {"name": "Pure.protectIPure.protectI", "term": "PROP ?A
          \<Longrightarrow> (PROP ?A)"}},
      {"thm": {"name": "HOL.eq_reflectionHOL.eq_reflection", "term":
          "?x = ?y \<Longrightarrow> ?x \<equiv> ?y"}},
      ... // More dependencies omitted
    ]
  }
}
```

Overall, the data generation files comprise approximately 2900 lines of Isabelle/ML code, 1450 lines of Scala, and 100 lines of Python. The hope is that the generated data can be directly used by the scientific community without Isabelle expertise for training and testing their machine learning models. Yet, this article provides tools for more comprehensive evaluations in the upcoming sections.

3 Data interaction

Most popular machine learning (ML) libraries are often Python frontends for efficient implementations of training algorithms and ML models. To transparently connect such models with the APIs from Section 2, this work uses the Py4j library [8] to connect Python with the Java Virtual Machine. Such usage enables the Python level to access and manipulate objects defined in Scala. Thus, following the same approach as in Section 2, one can define Isabelle/ML constructs that can be accessed by Scala and Python, enabling the interaction between Isabelle and the ML models. Specifically, the solution described here comprises read-eval-print loops (REPL) at the Scala and Python levels.

Implementation At the Isabelle/ML level, there is a REPL state built as a stack of records of an integer, an action, a top-level state and an error. Each function at this level

ensures that the initial record is always in the stack. Two kinds of functions are provided: the expected operations such as `init`, `read`, `eval`, `print`, `reset`, `undo`, and information-retrieval functions such as `last_proof_of`, `is_at_proof`, or `last_error`. The `read` function receives a string and a REPL state, converts the input string into actions, and generates the accompanying top-level states and errors. The `eval` function adds these generated data to the stack. Crucially, `read` detects whether any of the actions is an Isabelle proof method that might loop indefinitely. If an action might loop, `read` tries to generate the accompanying state for 10 seconds, and if the state-generation does not return a result in that time, `read` reports a timeout error and uses the input state as the result. The integer on each record in the stack models the number of user inputs and is relevant in, for example, the `undo` function. A user may input the Isabelle text representing a lemma with its proof to `read`. While such behaviour becomes many Isabelle actions, for the REPL, it corresponds to only one user input. Thus, `undo` will go back to the state before the input lemma and not to the previous Isabelle action by removing all records numbered with the latest user input.

At the Scala and Python levels, the REPL initialisation requires users to provide, at most, a "logic" and a `thy`-file to the REPL constructor. The "logic" is really the name of an Isabelle session. `ROOT` files are the official mechanism that Isabelle uses to indicate its sessions, and there is usually one session/`ROOT`-file per AFP entry. The `thy`-file input to the REPL is the name of a file in the scope of the input logic. If the REPL receives no logic and no `thy`-file, it starts by default importing `Main` and with the name `Scratch.thy` mirroring what happens when users start Isabelle in a similar setting. As before, the imports dependency graph allows the correct retrieval of the `Theory` that the REPL needs to start. Finally, instead of repeating the methods from the REPL state, its Scala version simplifies the interface to only having the operations `apply`, `reset`, `undo`, while most of the same information-retrieval methods remain. The Python level merely lifts all the Scala REPL methods to Python. Both levels ensure that the communication with Isabelle is mostly done via simple types such as booleans, strings, and integers.

API Example The listing below demonstrates an interactive proof using the Python REPL interfaced with Isabelle. The REPL instance facilitates sequential interaction with the proof assistant, allowing users to apply proof tactics and observe state transitions.

Listing 2 Interacting with Isabelle via Python REPL

```
>>> import sys
>>> sys.path.append(‘/path/to/this/project/src/main/python‘)
>>> from repl import REPL

>>> repl = REPL("HOL")
REPL and minion initialized.

>>> repl.apply("lemma␣\"\\<forall>x.␣P␣x␣\\<Longrightarrow>␣P␣c\"")
’proof␣(prove)␣goal␣(1␣subgoal):␣1.␣\\<forall>x.␣P␣x␣\\<Longrightarrow>␣P␣c’

>>> repl.apply("apply␣simp")
’proof␣(prove)␣␣goal:␣␣No␣subgoals!’

>>> repl.apply("done")
’’
```

The REPL implementation only requires 231 extra lines of Isabelle/ML, 143 lines of Scala, and 188 lines of Python code, which exhibits the generality and extensibility of the framework. Upcoming sections showcase its usage as an evaluation tool for ML models.

4 Evaluation

In this section, the generated data serves for training various large language models (LLMs) for interactive theorem proving. In conversations, several researchers from the formal methods community have expressed skepticism concerning LLM-usage for such a purpose. One of the most frequently repeated arguments is that pre-training of these models on a large internet corpus makes their usage statistically unsound. That is, many LLMs could have easily "memorised" online repositories of proofs like the AFP. Thus, the framework presented here enables a partial response to those concerns by doing small-scale experiments training these models only on the framework's data. Besides this, the framework is also evaluated by reproducing results from the natural language processing (NLP) community specialising the results to theorem-proving data. Concretely, the questions addressed here are:

1. Does increasing the model size improve its performance on Isabelle data?
2. Does supplying more information about a proof state improve the models' performance?
3. Do LLMs fine-tuned on Isabelle data perform better than those only pre-trained on it?

Machines specifications Training and evaluation happen on a system equipped with 8 NVIDIA Tesla V100-SXM2-32GB-LS GPUs, 503 GB of RAM and 80 CPUs. Despite these resources, data management for training is optimised so that observed RAM usage peaks at 35 GB. Moreover, a maximum of 14 processes are observed to be active during simultaneous model training and evaluation, while fully utilising the GPUs. Training loops typically consume approximately 23 GB of RAM. Partial, smaller evaluations are possible in an M2 Pro Apple Silicon system with 12 cores (8 performance and 4 efficiency) and 16 GB of RAM.

Models This work uses several text-to-text transfer transformer (T5) models [28, 6]. The T5 models have an encoder-decoder architecture, very similar to that of the original transformer [37] with minor changes [28]. For computational and time constraints, this work focuses on the `small` (77M parameters) and `base` (248M parameters) versions of the `google/flan-t5` family of models, leaving the `large` (783M), `xl` (2.85B), `xxl` (11.3B) variations out of the scope of the small-scale experiments here and for future work. Similar considerations apply for other more recent "efficient" models such as Gemma 3 (>1B) [33] and DeepSeek (>1B) [9]. Depending on the data format (see below), the models are kept with their default 512 token-input size or changed to 1024. All models trained here come from the Hugging Face library [13]. Notice that it is unlikely that Isabelle code appears in the claimed Hugging Face pre-training data for this work's fine-tuned models. The reported pre-training datasets for those are `openai/gsm8k` [7], `deepmind/code_contests` [18], and `cimec/lambada` [27]. The first one is a collection of questions and answers on grade school math *word* problems, the second one collects coding problems for frequently used programming languages (e.g. Python, Java, C, C++, ...), and the last one is extracted from several English-language novels.

Training setup This work uses Hugging Face's `accelerate` library for distributed training and evaluation. It usually employs 4 processes and 4 GPUs for the `small` models and 8 processes and 8 GPUs for the `base` models. Due to GPU memory constraints, the `small` models can use batch sizes of 8, while `base` models are limited to batch sizes of 4. The training uses an `AdamW` optimiser with a constant learning rate of 0.00001 and a weight decay of 0.01. All training loops measure the standard binary cross-entropy loss. Since the experiments involve pre-training the T5 models with Isabelle data, the original T5 tokenizer (SentencePiece) [17] is trained on all generated JSON files, producing a vocabulary size of 121,719 tokens. The pre-trained models use this newly trained tokenizer while the fine-tuned

models use Hugging Face's pre-trained tokenizer for T5. Since both tokenizers are different, the number of samples per dataset increases for the fine-tuned versions.

Data preprocessing This work provides a Python library (`proofs.py`) of methods to manipulate the generated JSON objects and preprocess the data. For instance, methods such as `proofs.orig_objective_of` or `proofs.user_proof_up_to` retrieve the required information about the proof. Two versions of the data are then employed. For a given proof state, the simplest version, `s` (for state), includes the user proof up to that point and the user-seen state, both separated by a `USER_STATE` separator. The more complex data version, `spk` (for state, premises, keywords), adds the name and content of the theorems used in the proof as well as the name of the proof methods available at that point. The models then must predict the next Isabelle action (defined in Section 2). Thus, the `spk` format simplistically models a premise and method selection setup. If the input is smaller than the model's input size $n = 512$ (which usually happens in the `s` data format), a pad token is added to fill the n-sized (vector) model input. If the tokenized input is larger, the extra tokens become an extra data point with padding if necessary. To avoid changing the data format too much in the `spk` format, the experiments also consider augmenting some models' input size to $n = 1024$ in the `spk` setup. Finally, for each `thy`-file in an AFP entry, 64% of the file's proofs were reserved for training, 16% for validation, and 20% for testing. When fine-tuning, the recommendation for T5 models is to add a prefix indicating the task at hand. In those cases, the string "isabelle next step: " is prefixed to all model inputs. Table 1 shows the number of samples per split and data format.

Format	Training	Validation	Testing
`s`	1,077,920	280,600	496,400
`finetune_s`	1,502,736	394,368	707,242
`spk`	4,322,842	1,081,944	1,983,106
`spk_trim`	7,694,240	1,924,416	2,358,016

Table 1 Number of samples per data split and data format.

Training For all models, an epoch is defined here as an iteration through all training samples of each AFP entry. Figures 1 and 2 show the evolution in time through 4 epochs of the average training and validation loss (respectively, accuracy) for the `t5_small_finetune_s` model. Each set of dots represents a training epoch and a set of crosses represents its validation counterpart. For readability, the training epochs and their corresponding validations are shown sequentially. Due to computational and time constraints, the models are trained for a maximum of four epochs, except for the `t5_small_spk` with input size 1024 that has considerably more training steps per epoch. The graphs of Figures 1 and 2 also show that after four epochs, the `t5_small_finetune_s` model has not yet memorised the training dataset since the average training loss is not lower than the average validation loss, albeit, they are close.

Evaluation The values in Table 2 show the performance of the T5 models during training. Comparing the `t5_small_s` and `t5_base_s` models (using the simplest "state" data format) to address question **1**, one may argue that increasing the model size indeed reduces the loss and increases accuracy. However, these differences are negligible in comparison to those between pre-training and fine-tuning. Interestingly, by comparing `t5_small_s` and `t5_small_spk` one may suspect that increasing the amount of data per example makes the models perform worse after seeing approximately the same amount of training examples (question **2**). Overall, training the models from scratch produces higher losses and less

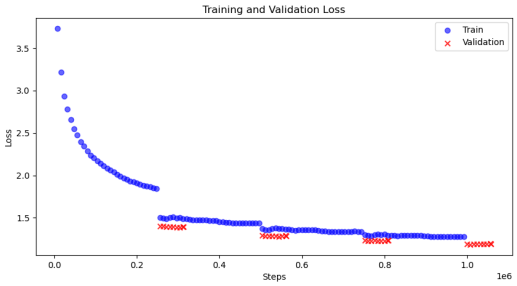


Figure 1 Average training and validation loss: Four epochs are shown with each training epoch followed by its validation.

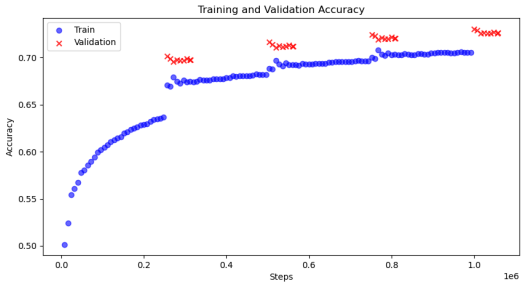


Figure 2 Average training and validation accuracy: Four epochs are shown with each training epoch followed by its validation.

accuracy in the same amount of epochs than fine-tuning them. This adds evidence in support of answering affirmatively to question **3** echoing lessons from the ML literature where the models have more difficulty predicting patterns in the data when they are not pre-trained on a diverse dataset [28]. Notably, `t5_small_spk_trim` has the worst validation loss and accuracy, which suggests that overflowing tokens as done in that case makes training ineffective as opposed to providing the proof context in a single example (`t5_small_spk`).

Model	Train Loss	Train Acc.	Valid Loss	Valid Acc.
t5_small_s (4ep)	3.02	0.445	3.13	0.436
t5_small_finetune_s (4ep)	1.18	0.720	1.14	0.733
t5_small_spk (3ep)	2.96	0.417	3.29	0.375
t5_small_spk_trim (4ep)	2.71	0.438	3.21	0.371
t5_base_s (4ep)	2.84	0.474	3.06	0.441

Table 2 Training model performance.

Table 3 shows the models' performance on the test set according to three metrics. The first column corresponds to the ratio of examples where the models' (best-first) predictions exactly coincided with the user actions. The second column shows the ratio of examples where the models' first-token prediction coincided with that in the user action. Presumably, this first token would correspond to an Isabelle command, e.g. `apply` or `have`, and thus, this measure is designed to indicate model awareness on Isabelle's proof stages. Lastly, the third column indicates the ratio of examples where the models' predictions are entirely incorrect.

Model	Test Exact Acc.	Test 1st-Tok Acc.	Test All Wrong
t5_small_s (4ep)	0.137	0.316	0.303
t5_small_finetune_s (4ep)	0.150	0.302	0.231
t5_small_spk (2ep)	0.042	0.254	0.553
t5_small_spk_trim (4ep)	0.022	0.194	0.371
t5_base_s (4ep)	0.170	0.307	0.299

Table 3 Testing model Performance

Consistently with the training results, the fine-tuned version of the small model achieves the most robust set of values. However, the difference between this version and the pre-trained `t5_small_s` and `t5_base_s` models is not as pronounced as in the training and validation

sets. In particular, the larger model performs slightly better on predicting exact user actions or first tokens, while the simplest `small` model is the best one on the first token prediction metric. These results slightly reinforce the previous evidence towards answering questions **1** and **3**. That is, using larger models and fine-tuning, slightly improves model performance. However, it does not decisively lead to pronounced differences in the models' generalisation capabilities. With respect to supplying more data (question **2**), both the `spk` and `spk_trim` variations consistently showed lower exact accuracy and higher ratios of wrong predictions on the test set. Overall, the values in Table 3 support the conclusion that fine-tuning a small pre-trained model on Isabelle data can lead to better performance compared to training from scratch or naively increasing the input context.

5 Automated proving

This section describes a loop that makes Python an intermediary between the models and Isabelle to automatically prove proofs. The loop further tests the conclusions from Section 4 and evaluates the REPL and, more generally, the framework. The evaluation is accompanied by a qualitative analysis of the produced proofs.

Implementation The loop loads one of the models from Section 4, traverses the proofs on the test set and makes the model predict the next steps based on the replies from Isabelle. The REPL includes functionality asking Isabelle to load the corresponding `thy`-file just before the theorem to prove. In previous experiments without this configuration, some models were able to (cheat) "prove" various theorems by simply calling the simplifier with the already proven theorem as an argument. In a depth-first search setting, when a proof is started, the evaluation loop retrieves model predictions using beam-search, sends the predictions to Isabelle, and decides the next step based on the ITP's reply. If the incoming top-level state is accompanied by an error, the loop counts this as a *failed step*, backtracks the REPL to a state before the error, and tries the next prediction. The models sometimes recommend using Isabelle's `by` command due to its frequency in the data. Since the `by` command should only be used to conclude proofs, and thus, hides the errors produced by the method employed, the loop replaces this `by` commands with `apply` commands. If a no-subgoals state is detected after the application of this replacement, the loop backtracks and applies the original model's produced `by`-prediction, and counts this successful *`by`-application*. If the REPL detects that the proof is *finished*, the loop retrieves it from the REPL and saves it in a `thy`-file. If the `apply`-version of the model prediction does not produce an error and does not finish the proof, the loop counts this as an *incorrect `by`-application*. Nevertheless, non-error-producing actions are counted as *progressing* the proof and the same model-ITP interaction is repeated up to a proof-tree depth. In all experiments, the model provides 5 predictions and the maximum proof-depth explored is 5 actions. Yet, for time and memory constraints the models did not tackle the same number of proofs.

Model	Predictions	Progress	by commands	Correct by	Proofs	Finished
small_s	4,185,571	62.48%	98,369	41.05%	24,071	6.63%
finetune_s	2,055,838	37.29%	97,907	8.29%	24,740	11.46%
small_spk	1,859,629	49.01%	35,885	14.76%	27,597	8.55%
base_s	2,983,954	60.65%	48,141	26.78%	20,252	8.92%

Table 4 Evaluation of the T5 small models on automatically proving theorems

Table 4 shows the models' performance on the test set when iterated through the REPL

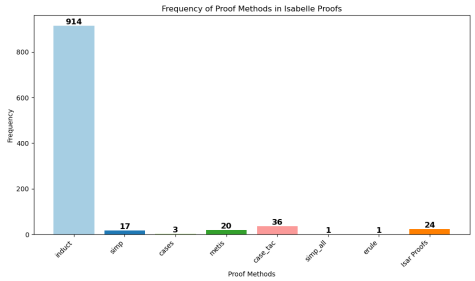
loop. Interestingly, after `by`-conversion to `apply`, the simplest `t5_small_s` model suggests the highest percentage of predictions that make progress in the proof. However, despite progress in several proofs, its fraction of completed proofs is the smallest. This suggests that its model predictions are, although valid, unsound. Qualitatively exploring the loop-produced log files shows that many "successful" `t5_small_s`-actions were repeated applications of the `case_tac` method or, in proof-scripting sections, repeated applications of the Isar-command `fix`. The `case_tac` method receives an argument and splits the proof in two depending on the truth of the argument (e.g. `apply (case_tac "x = 0")`). The model's `case_tac`-actions are usually accompanied by equalities, augmenting the number of proof obligations but not necessarily advancing the proof. Similarly, the `fix` command simply serves to pose universally quantified variables (e.g. `fix x`). Thus, repeating `fix`, without immediately stating a property that its argument holds, does not progress the proof soundly. Despite these limitations, the simplest `t5_small_s` successfully completed 6.82% proofs. Many of them are simple uses of the simplifier, e.g. `by simp`, or simple induction proofs, e.g. `by (induct n, simp_all)`. Yet, there are more complex model-produced proofs that interestingly display the relation between proof states and tactic applications. For instance, the model completed lemma `pre_between_symI`, in the `FaceDivisionProps.thy` of the `Flyspeck-Tame` AFP entry [3] (whose original AFP-proof is `by auto`) by calling the simplifier, then repeatedly removing the conjunctions in the user-state with conjunction elimination (`apply (erule conjE)` in Isabelle), and then finishing with an application of `metis`. Many of the completed proofs were found in less than 10 seconds, but `t5_small_s`' average time for processing proofs is 48.67 seconds barely using 1 GPU for the model generation and less than 12 GB of RAM. This suggests that spending time exploring the proof tree is not conducive to proof completion for the `t5_small_s` model. Yet, more work is required to conclusively analyse these preliminary qualitative results.

Notice that, in general for all models, the total number of `by`-conversions is a small fraction of the total recommended actions: 2.35%, 4.76%, 1.93%, and 1.61% from top to bottom in Table 4. Curiously, the `t5_small_finetune_s` model had the lowest percentage of progressing predictions, while its fraction of `by`-conversions was the largest. Moreover, it has one of the lowest percentages of correctly suggested `by`-statements which is considerably smaller than the corresponding percentage from the `t5_small_s` model. Taken in isolation, these percentages could be perceived as indicating that the `t5_small_finetune_s` model is less effective than the other models. However, its proof-completion rate is approximately 2.5% greater than the second best model. Thus, it is the most effective model for theorem proving. This strongly supports the data from Section 4 and helps to conclude that pre-training on a large corpus and then fine-tuning is more effective than simply training directly on the desired task.

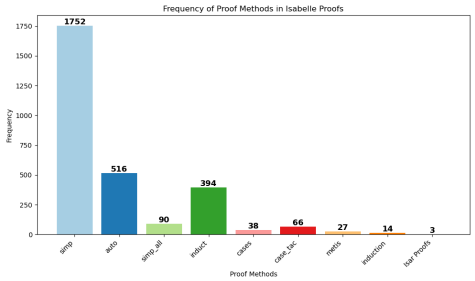
Concerning model size improving the performance of the models (question **1**), Table 4 shows consistent results with Section 4. Namely, model size seems to improve performance but the increased percentage of finished proofs is not as pronounced as that of fine-tuning. Yet, contrastingly with Section 4, the values in Table 4 support the notion that increasing the amount of data seen per sample helps models perform better (question **2**).

Figure 3 shows an approximation of the distribution of methods used by the models in the completed proofs. Namely, it shows the frequency of the first method immediately after using the proof commands `apply` or `by` in the produced proofs. It also includes the frequency of the proofs completed in the proof-scripting language Isar. One can observe that the largest model uses a more diverse pool of methods in the completed proofs, while the success of the best performing model `t5_small_finetune_s` seems to rely on its usage of

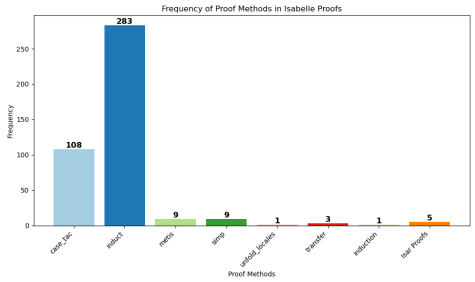
the simplifier. Indeed, exploring its generated `thy`-files, one immediately sees many proofs by definition (supplied to `simp`). Since the proof-depth is not large, most Isar-style proofs consist simply of the starting and ending keywords (that is `proof` and `qed`) enclosing a simple body stating `from assms show ?thesis by method`. Notably, the simplest `t5_small_s` model relies more on Isar proofs than its modified counterparts. A last observation from Figure 3 is that the models successfully employ proof methods that are not as frequent as inductions, simplifications or `auto`-calls. Namely, some proofs were completed with `unfold_locales` for expanding the definitions in a `locale`, `intro_classes` for a similar operation when dealing with Isabelle's classes, the `transfer` method from the homonym package, and the dot `.` method which the `t5_base_s` model appropriately use to close `interpretation` statements.



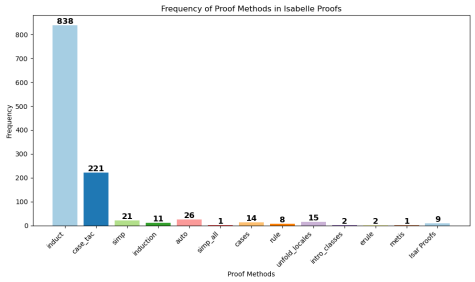
(a) `t5_small_s` model



(b) `t5_small_finetune_s` model



(c) `t5_small_spk` model



(d) `t5_base_s` model

Figure 3 Models' first proof method frequency per line

Finally, better theorem-proving performance seems to also be correlated with time spent per proof, with the worst model spending the most time per proof and vice-versa. Specifically, the `t5_small_s` model spent, on average, 48.67 seconds per proof, the `t5_small_finetune_s` model spent 41.13 seconds per proof, the `t5_small_spk` model spent 29.76 seconds per proof, and the `t5_base_s` model spent 53.47 seconds per proof. This suggests that more capable models could improve waiting times for proof-recommendations in LLM-backed tools but this does not alleviate the processing times due to increases in model size. The results of Table 4 support the old adage that "quality is better than quantity"

This section evidences that the framework's previously described data extraction and data interaction of Sections 2 and 3 respectively are robust tools for training and evaluating ML models. The framework is openly available and an anonymous artefact is provided [**?**].

6 Limitations

This section outlines identified limitations concerning the framework's data extraction and interaction capabilities. These limitations will be addressed in a future non-prototypical version of the framework but are reported here for any current potential users.

A limitation of the framework's data-extraction algorithm is that some generated JSON strings can become excessively large due to the comprehensive nature of the retrieved information. Not even Isabelle's pretty printer can correctly display some of them in the prover's IDE (PIDE). Furthermore, some data can be too large for the communication between Isabelle/ML and `scala-isabelle`. Data size management is an inherent challenge when transferring data between different environments and it highlights trade-offs between transparent and efficient data transfers. These two issues imply that only 187,210 (approximately 66% of the) AFP-proofs were successfully turned into JSON files and that, the data generation often left some idle Isabelle processes when executed. A different data-production algorithm with better serialization that alleviates these issues has been used for an alternative data format, but such an implementation has not yet been adapted for the framework's JSON generation.

The issue above relates to a more general characteristic of the current implementation. That is, the main datatype manipulated in the majority of the framework's data-transfers are strings while more compressed formats for ITP and ML have not been employed yet. This choice reflects an initial focus on data accessibility and ease of implementation with the understanding that more optimised representations can be explored later. Such data compression might also alter the ML generalisation [30] since strings are the de-facto type in datasets for training LLMs. Concretely, an optimised representation of this work's Isabelle/ML-type `Data.T` into a vector of tokens is a worthy future pursuit. An inherent consequence of interacting with Isabelle programmatically through a text-based interface, as opposed to the richer visual feedback provided by a dedicated PIDE, is that the REPL's inputs are not easy to parse or write for humans. This is a minor inconvenience for REPL users, but its primary goal is to facilitate automated interaction with Isabelle as in Section 5.

Finally, the (previously known) facts in the generated JSON representations are only those explicitly used in the proof. This is not a faithful representation of the premise selection task that users do to complete the proving process. More crucially, expanding the framework's generated data to a broader set of available hypotheses could enhance the training of premise selection models, replicate previous successful approaches [21], and it would make comparisons fairer against established methods like Sledgehammer.

7 Related Work

The integration of ML techniques into interactive theorem provers (ITPs) has become an active area of research combining various provers with different strategies.

In the HOL4 prover, the TacticToe [10] tool pioneered the use of a k-nearest neighbour (kNN) classifier (for predicting tactics, theorems, and lists of goals) combined with Monte Carlo tree search (MCTS) for proof exploration. It was later followed by the TacticZero [40] reinforcement learning (RL) work whose neural networks learn to associate goals with tactics and pairings of these with lists of their arguments. The models presented in this article are LLM-based but the data-extraction and interaction tools could be expanded to reproduce both approaches. That is, creating a premise selector [21] and comparing it with Isabelle's kNN-based Sledgehammer is within the framework's reach. Alternatively, turning Isabelle into an RL environment for results similar to TacticZero requires modifying the evaluation loop of Section 5.

For the Coq prover, work on the Tactician [4, 30] provides three different ML algorithms that interact with the prover: a kNN, a graph neural network (GNN), and a from-scratch LLM. There, the models that learn online (while exploring the proof), like the kNN, consistently outperform the offline models, including the LLMs. The conclusions from Sections 4 and 5 complement their evaluations as context-provided data, and pre-training on a larger corpus of data followed by fine-tuning on ITP data clearly improve the LLM performance. Larger, more detailed experiments using the framework presented here might provide better insights into the performance of different ML models. More recent work on Coq focuses on creating datasets for proof-repair [29].

The work described in this article is similar to the HOList project [2] that provides an environment for ML research in a simplified Python version of HOL Light allowing users to test various models, including GNNs [26] for ranking possible actions. A fully RL approach in HOL Light has also demonstrated the potential of this paradigm for theorem proving [1].

The growing interest in the Lean prover within the mathematical community has also attracted significant attention from ML researchers. In particular, a framework inspiring the one described in this article is LeanDojo [41]. The work showcased here follows LeanDojo's contributions of open data extraction and interaction with a well-known ITP.

Finally, Sledgehammer [5] has been widely adopted for premise selection in Isabelle. Later work has also included the use of regression trees for conjecturing and proofs by induction [24, 22, 23]. More recently, the Portal-to-Isabelle (PISA) project explored the use of LLMs for autoformalisation and tactic suggestion in Isabelle, with Magnushammer representing a transformer-based approach to premise selection that has shown promising results. The project aimed at a deeper ML integration but the most recent efforts have focused on enhancing LLMs proof search efficiency rather than enhancing PISA itself [42].

8 Conclusion and future work

This work contributes openly available data-generation and interaction tools to a growing research field aimed at enhancing proof automation in ITPs through the integration of machine learning (ML) techniques. I have presented a data-extraction algorithm for the Isabelle proof assistant, as well as data-interaction tools in the form of read-eval-print-loops (REPLs) in Scala and Python.

By applying the data extraction to a large dataset of 187,210 proofs from the Archive of Formal Proofs, I demonstrated the framework's ability to structure proof data in a format suitable for machine learning (ML). Training T5-based large language models (LLMs) on this data further reinforced conclusions from the ML literature regarding the benefits of fine-tuning on domain-specific corpora. I also tested the framework's ITP interactive capabilities by implementing a REPL-based evaluation for the models. Its results showed that the LLMs are capable of generating meaningful proof steps and even proving some test-set theorems autonomously. These findings complement a growing body of evidence for the feasibility of leveraging LLMs in ITP environments. In the spirit of open research, a snapshot of the current source code, examples of generated data, and model weights are available in a publicly available Zenodo repository [12], and the evolving source code in a GitHub repository [11].

Future work could extend this framework by experimenting with larger, more recent and efficient models, premise selection capabilities, alternative training strategies, and reinforcement learning (RL) techniques that further enhance the integration between ML and ITPs. Additionally, investigating real-time proof assistance scenarios could help bridge the gap between ML-driven theorem proving and practical user interactions within Isabelle.

An envisioned objective is to create tools for ITP users that accelerate their workflows and help make proof assistants more efficient, accessible and widespread. Ultimately, this work establishes a foundation for turning Isabelle into an RL environment and advancing research at the intersection of machine learning, theorem proving and formal verification. Finally, this article is an open invitation to Isabelle or ML experts to collaborate on this vision.

References

1. Kshitij Bansal, Sarah M. Loos, Markus N. Rabe, and Christian Szegedy. Learning to reason in large theories without imitation. *CoRR*, abs/1905.10501, 2019. arXiv:1905.10501.
2. Kshitij Bansal, Sarah M. Loos, Markus N. Rabe, Christian Szegedy, and Stewart Wilcox. HOList: An environment for machine learning of higher order logic theorem proving. In *ICML 2019*, volume 97 of *PMLR*, pages 454–463. PMLR, 2019. URL: http://proceedings.mlr.press/v97/bansal19a.html.
3. Gertrud Bauer and Tobias Nipkow. Flyspeck i: Tame graphs. *Archive of Formal Proofs*, May 2006. https://isa-afp.org/entries/Flyspeck-Tame.html, Formal proof development.
4. Lasse Blaauwbroek, Josef Urban, and Herman Geuvers. The tactician - A seamless, interactive tactic learner and prover for coq. In Christoph Benzmüller and Bruce R. Miller, editors, *CICM 2020*, volume 12236 of *LNCS*, pages 271–277. Springer, 2020. doi:10.1007/978-3-030-53518-6_17.
5. J. C. Blanchette, C. Kaliszyk, L. C. Paulson, and J. Urban. Hammering towards QED. *JFR*, 9(1), 2016.
6. Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, Albert Webson, Shixiang Shane Gu, Zhuyun Dai, Mirac Suzgun, Xinyun Chen, Aakanksha Chowdhery, Sharan Narang, Gaurav Mishra, Adams Yu, Vincent Zhao, Yanping Huang, Andrew Dai, Hongkun Yu, Slav Petrov, Ed H. Chi, Jeff Dean, Jacob Devlin, Adam Roberts, Denny Zhou, Quoc V. Le, and Jason Wei. Scaling instruction-finetuned language models. https://arxiv.org/abs/2210.11416, 2022. doi:10.48550/ARXIV.2210.11416.
7. Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021. URL: https://arxiv.org/abs/2110.14168, arXiv:2110.14168.
8. Barthélémy Dagenais. Py4j. https://github.com/py4j/py4j, 2024.
9. DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL: https://arxiv.org/abs/2501.12948, arXiv:2501.12948.
10. Thibault Gauthier, Cezary Kaliszyk, Josef Urban, Ramana Kumar, and Michael Norrish. Tactictoe: Learning to prove with tactics. *JAR*, 65(2):257–286, 2021. URL: https://doi.org/10.1007/s10817-020-09580-x, doi:10.1007/S10817-020-09580-X.
11. Jonathan Julian Huerta y Munive. DeepIsaHOL, November 2023. URL: https://github.com/yonoteam/DeepIsaHOL.
12. Jonathan Julian Huerta y Munive. Snapshot of contributions from the DeepIsaHOL project, March 2025. doi:10.5281/zenodo.15080049.
13. Hugging Face. Hugging face – the ai community building the future. Online; accessed 23-March-2025. URL: https://huggingface.co/.
14. Albert Q. Jiang, Wenda Li, Jesse Michael Han, and Yuhuai Wu. Lisa: Language models of isabelle proofs. *6th Conference on Artificial Intelligence and Theorem Proving*, 2021.
15. Albert Qiaochu Jiang, Wenda Li, Szymon Tworkowski, Konrad Czechowski, Tomasz Odrzygózdz, Piotr Milos, Yuhuai Wu, and Mateja Jamnik. Thor: Wielding hammers to integrate language models and automated theorem provers. In *NeurIPS 2022*, 2022. URL: http://papers.nips.cc/paper_files/paper/2022/hash/377c25312668e48f2e531e2f2c422483-Abstract-Conference.html.

16 Albert Qiaochu Jiang, Sean Welleck, Jin Peng Zhou, Timothée Lacroix, Jiacheng Liu, Wenda Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *ICLR 2023*. OpenReview.net, 2023. URL: https://openreview.net/pdf?id=SMa9EAovKMC.

17 Taku Kudo and John Richardson. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing, 2018. URL: https://arxiv.org/abs/1808.06226, arXiv:1808.06226.

18 Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, December 2022. URL: http://dx.doi.org/10.1126/science.abq1158, doi:10.1126/science.abq1158.

19 Carlin MacKenzie, James Vaughan, Jacques Fleuriot, and Fabian Huch. The archive of formal proofs. https://www.isa-afp.org/, 2025. [Online; accessed 15-March-2025].

20 Maciej Mikula, Szymon Antoniak, Szymon Tworkowski, Albert Qiaochu Jiang, Jin Peng Zhou, Christian Szegedy, Lukasz Kucinski, Piotr Milos, and Yuhuai Wu. MagnusData. https://github.com/Simontwice/MagnusData, 2023.

21 Maciej Mikula, Szymon Antoniak, Szymon Tworkowski, Albert Qiaochu Jiang, Jin Peng Zhou, Christian Szegedy, Lukasz Kucinski, Piotr Milos, and Yuhuai Wu. Magnushammer: A transformer-based approach to premise selection. *CoRR*, abs/2303.04488, 2023. URL: https://doi.org/10.48550/arXiv.2303.04488, arXiv:2303.04488, doi:10.48550/ARXIV.2303.04488.

22 Yutaka Nagashima. SeLFiE: Modular semantic reasoning for induction in Isabelle/HOL. *CoRR*, abs/2010.10296, 2020. arXiv:2010.10296.

23 Yutaka Nagashima. Faster smarter proof by induction in Isabelle/HOL. In Zhi-Hua Zhou, editor, *IJCAI 2021*, pages 1981–1988. ijcai.org, 2021. URL: https://doi.org/10.24963/ijcai.2021/273, doi:10.24963/IJCAI.2021/273.

24 Yutaka Nagashima and Yilun He. PaMpeR: proof method recommendation system for Isabelle/HOL. In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *ASE 2018*, pages 362–372. ACM, 2018. doi:10.1145/3238147.3238210.

25 Yutaka Nagashima, Zijin Xu, Ningli Wang, Daniel Sebastian Goc, and James Bang. Template-based conjecturing for automated induction in Isabelle/HOL. In Hossein Hojjat and Erika Ábrahám, editors, *FSEN 2023*, volume 14155 of *LNCS*, pages 112–125. Springer, 2023. doi:10.1007/978-3-031-42441-0_9.

26 Aditya Paliwal, Sarah M. Loos, Markus N. Rabe, Kshitij Bansal, and Christian Szegedy. Graph representations for higher-order logic and theorem proving. *CoRR*, abs/1905.10006, 2019. URL: http://arxiv.org/abs/1905.10006, arXiv:1905.10006.

27 Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Ngoc Quan Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernandez. The LAMBADA dataset: Word prediction requiring a broad discourse context. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1525–1534, Berlin, Germany, August 2016. Association for Computational Linguistics. URL: http://www.aclweb.org/anthology/P16-1144.

28 Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020. URL: http://jmlr.org/papers/v21/20-074.html.

29 Tom Reichel, R. Wesley Henderson, Andrew Touchet, Andrew Gardner, and Talia Ringer. Proof repair infrastructure for supervised models: Building a large proof repair dataset. In Adam Naumowicz and René Thiemann, editors, *ITP 2023*, volume 268 of *LIPIcs*, pages

26:1–26:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. URL: https://doi.org/10.4230/LIPIcs.ITP.2023.26, doi:10.4230/LIPICS.ITP.2023.26.

30 Jason Rute, Miroslav Olsák, Lasse Blaauwbroek, Fidel Ivan Schaposnik Massolo, Jelle Piepenbrock, and Vasily Pestun. Graph2Tac: Learning hierarchical representations of math concepts in theorem proving. *CoRR*, abs/2401.02949, 2024. URL: https://doi.org/10.48550/arXiv.2401.02949, doi:10.48550/ARXIV.2401.02949.

31 Joshua Schneider and Dmitriy Traytel. Formalization of a monitoring algorithm for metric first-order temporal logic. *Archive of Formal Proofs*, July 2019. https://isa-afp.org/entries/MFOTL_Monitor.html, Formal proof development.

32 Peiyang Song, Kaiyu Yang, and Anima Anandkumar. Towards large language models as copilots for theorem proving in Lean. *arXiv preprint arXiv: Arxiv-2404.12534*, 2024.

33 Gemma Team. Gemma 3. 2025. URL: https://goo.gle/Gemma3Report.

34 Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U. Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Hannah Tan, and Omar G. Younis. Gymnasium: A standard interface for reinforcement learning environments, 2024. URL: https://arxiv.org/abs/2407.17032, arXiv:2407.17032.

35 Mark Towers, Jordan K. Terry, Ariel Kwiatkowski, John U. Balis, Gianluca de Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Arjun KG, Markus Krimmel, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Andrew Tan Jin Shen, and Omar G. Younis. Gymnasium, March 2023. https://zenodo.org/record/8127025. doi:10.5281/zenodo.8127026.

36 Dominique Unruh. scala-isabelle. https://github.com/dominique-unruh/scala-isabelle, 2024.

37 Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023. URL: https://arxiv.org/abs/1706.03762, arXiv:1706.03762.

38 Sean Welleck and David Renshaw. LLMLean. https://github.com/cmu-l3/llmlean, 2024.

39 Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle framework. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 33–38, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

40 Minchao Wu, Michael Norrish, Christian Walder, and Amir Dezfouli. Tacticzero: Learning to prove theorems from scratch with deep reinforcement learning. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *NeurIPS 2021*, pages 9330–9342, 2021. URL: https://proceedings.neurips.cc/paper/2021/hash/4dea382d82666332fb564f2e711cbc71-Abstract.html.

41 Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. In *Neural Information Processing Systems (NeurIPS)*, 2023.

42 Xueliang Zhao, Wenda Li, and Lingpeng Kong. Subgoal-based demonstration learning for formal theorem proving. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024. URL: https://openreview.net/forum?id=pSnhA7Em1P.

43 Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. MiniF2F: a cross-system benchmark for formal olympiad-level mathematics. *arXiv preprint arXiv:2109.00110*, 2021.