HyperTree Proof Search for Neural Theorem Proving

Guillaume Lample[*†] **Marie-Anne Lachaux**[*†] **Thibaut Lavril**[*†] **Xavier Martinet**[*†]

Amaury Hayat[§] **Gabriel Ebner**[‡] **Aurélien Rodriguez**[†] **Timothée Lacroix**[*†]

Abstract

We propose an online training procedure for a transformer-based automated theorem prover. Our approach leverages a new search algorithm, HyperTree Proof Search (HTPS), that learns from previous proof searches through online training, allowing it to generalize to domains far from the training distribution. We report detailed ablations of our pipeline's main components by studying performance on three environments of increasing complexity. In particular, we show that with HTPS alone, a model trained on annotated proofs manages to prove 65.4% of a held-out set of Metamath theorems, significantly outperforming the previous state of the art of 56.5% by GPT-f. Online training on these unproved theorems increases accuracy to 82.6%. With a similar computational budget, we improve the state of the art on the Lean-based miniF2F-curriculum dataset from 31% to 42% proving accuracy.

1 Introduction

Over the course of history, the complexity of mathematical proofs has increased dramatically. The nineteenth century saw the emergence of proofs so involved that they could only be verified by a handful of specialists. This limited peer review process inevitably led to invalid proofs, with mistakes sometimes remaining undiscovered for years (e.g. the erroneous proof of the Four Colour Conjecture [1]). Some mathematicians argue that the frontier of mathematics has reached such a level of complexity that the traditional review process is no longer sufficient, envisioning a future where articles are submitted with formal proofs so that the correctness can be delegated to a computer [2].

Unfortunately, very few mathematicians have adopted formal systems in their work, and as of today, only a fraction of existing mathematics has been formalized. Several obstacles have hindered the widespread adoption of formal systems. First, formalized mathematics are quite dissimilar from traditional mathematics, rather closer to source code written in a programming language, which makes formal systems difficult to use, especially for newcomers. Second, formalizing an existing proof still involves significant effort and expertise (the formalization of the Kepler conjecture took over 20 person years to complete [3]) and even seemingly simple statements sometimes remain frustratingly challenging to formalize.

To write a formal proof, mathematicians typically work with Interactive Theorem Provers (ITPs). The most popular ITPs provide high-level "tactics" that can be applied on an input theorem (e.g. the initial goal) to generate a set of subgoals, with the guarantee that proving all subgoals will result in a proof of the initial goal (reaching an empty set means the tactic solves the goal). An example of a proof in Lean [4], an interactive theorem prover, is given in Figure 1 and the corresponding proof hypertree[3] is illustrated in Figure 5 of the Appendix.

[*]Equal contribution. Corresponding authors: `{glample,malachaux,tlacroix}@fb.com`

[†]Meta AI Research [‡]Vrije Universiteit Amsterdam [§] CERMICS École des Ponts ParisTech

[3]A hypergraph is a graph where an edge leads to a set of nodes that is potentially empty in our set-up. A hypertree, in this work, is a hypergraph without cycles. Formal definitions can be found in Appendix C.1

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

```
import data.nat.basic
example (m n k : ℕ) (h₀ : n ≤ m) : n + k ≤ m + k := begin
  induction k,
  {
    exact h₀
  },
  {
    rw nat.succ_le_succ_iff,
    exact k_ih
  }
end
```

First subgoal : $n + 0 \leq m + 0$

Second subgoal : $n + k \leq m + k \Rightarrow n + k + 1 \leq m + k + 1$

Figure 1: **A simple proof of the statement $\mathbf{n \leq m \Rightarrow n + k \leq m + k}$ in Lean.** The *induction* tactic reduces the initial statement to two subgoals, that can be solved independently.

In this paper, we aim at creating a prover that can automatically solve input theorems by generating a sequence of suitable tactics without human interaction. The backward procedure naturally suggests a simple approach where a machine learning model trained to map goals to tactics interacts with an ITP to build the proof of an input goal in a backward fashion. The automated prover builds a hypergraph with the theorem to be proved as the root node, tactics as edges and subgoals as nodes. The prover recursively expands leaves by generating tactics with our model until we find a proof of the initial theorem. A proof is then a hypertree rooted in the initial theorem whose leaves are empty sets.

Unlike Chess or Go, particular challenges arise for tree-search in theorem proving. First, the action space, i.e. the amount of possible "moves" in a given state, is infinite (there is an unlimited number of tactics that can be applied to a given theorem). This requires sampling possible actions from a language model for which training data is scarce. Moreover, if all tactics sampled at a goal fail, we have no information on what region of the probability space to sample next. Second, in the context of theorem proving, we need to provide a proof of all subgoals created by a tactic, whereas AlphaZero[5] for two player games is allowed to focus on the most likely adversary moves.

This paper presents an in-depth study of our approach to overcome these difficulties and the resulting model, Evariste. In particular, we make the following contributions:

- A new MCTS-inspired search algorithm for finding proofs in unbalanced hypergraphs.
- A new environment (Equations) to easily prototype and understand the behavior of the models we train and our proof search.
- A detailed ablation study and analysis of the different components used in our approach on three different theorem proving environments. We study how data is selected for training the policy model after a successful or failed proof-search, what target should be used to train the critic model, and the impact of online training vs. expert iteration.
- State-of-the-art performance on all analyzed environments. In particular, our model manages to prove over 82.6% of proofs in a held-out set of theorems from `set.mm` in Metamath, as well as 58.6% on miniF2F-valid [6] in Lean.

2 Related work

Automated theorem proving has been a long-standing goal of artificial intelligence, with the earliest work dating from the 1950s [7, 8]. We focus here on recent work closest to ours and defer additional related work to Appendix B.

Neural theorem provers. Recent work applying deep learning methods to theorem proving [9–11] are the closest to this work and obtained impressive results on difficult held-out sets for Metamath and Lean. The main differences between their approach and ours are the proof-search algorithm we propose, the training data we extract from proof-searches and our use of online training compared to their expert iterations. Another similar approach, Holophrasm [12], uses a different tree-search algorithm while others [13, 14] learn the search policy along with the tactic model. Unlike previous studies that focus on a single proving environment (e.g. Metamath, Lean, or HOL-Light), we extensively study the performance of our prover on three different formal languages, and found that some conclusions significantly vary based on the considered environment.

MCTS and two player games. AlphaZero [5] demonstrated good performances on two player games, replacing the Monte-Carlo evaluations of MCTS [15] with evaluations from a deep neural network and guiding the search with an additional deep policy. These ideas have been applied to first order logic proving in Kaliszyk et al. [16] with gradient boosted trees as policy and value models. Theorem proving can be thought of as computing game-theoretic value for positions in a min/max tree: to prove a goal, we need one move (max) that leads to subgoals that are all proven (min). This has led to other tree search algorithms such as Proof Number Search [17] or a more recent version, using a neural estimate of proof size: Wu et al. [18]. Noticing heterogeneity in the arities of min or max nodes, we propose a search method that goes down simultaneously in all children of min nodes, such that every simulation can potentially result in a full proof-tree.

3 Online training from proof searches

In this section, we introduce our Hypertree Proof Search (HTPS) algorithm and describe how it is used to generate training data for our model. We then detail our online training method.

3.1 Hypertree Proof Search

Given a main goal g to automatically prove, HTPS is the algorithm that interacts with a policy model P_θ and a critic model c_θ, and the theorem proving environment to find a proof hypertree for g. Proof search progressively grows a hypergraph starting from g, iteratively repeating the three steps illustrated in Figure 2: selection, expansion and back-propagation. The main difference with other search algorithms is our parallel descent in all subgoals of a tactic. A proof is found when there exists a hypertree from the root to leaves that are empty sets.

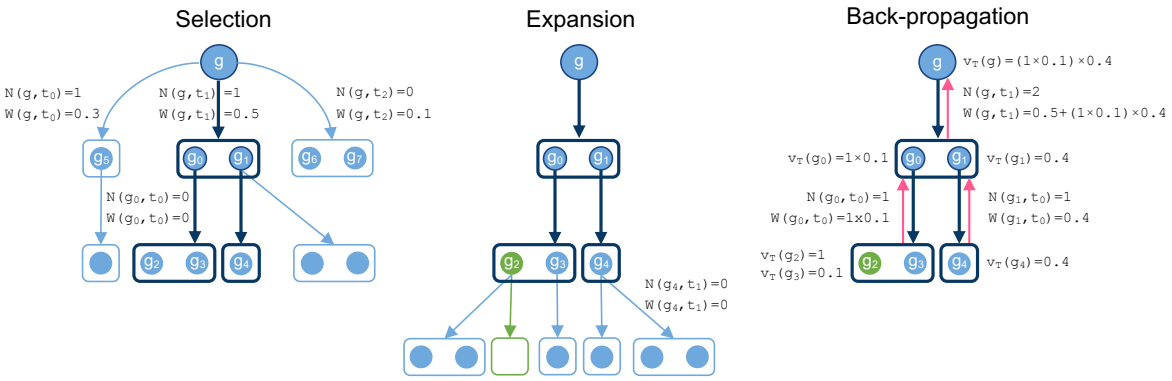

Figure 2: **HyperTree Proof Search**. We aim at finding a proof of the root theorem g. The figure represents the three steps of HTPS that are repeated until a proof is found. Proving either $\{g_5\}$, $\{g_0, g_1\}$, or $\{g_6, g_7\}$ would lead to a proof of g by tactic t_0, t_1, or t_2. Guided by the search policy, we select a hypertree whose leaves are unexpanded nodes. Selected nodes are then expanded, adding new tactics and nodes to the hypergraph. Finally, we evaluate the node values v_T of the hypertree starting from the leaves, using the critic, and back-propagating to the root, updating the visit counts N and total action values W.

We assume a policy model P_θ and critic model c_θ. Conditioned on a goal, the policy model allows the sampling of tactics, whereas the critic model estimates our ability to find a proof for this goal. Our proof search algorithm will be guided by these two models. Additionally, and similar to MCTS, we store the visit count $N(g, t)$ (the number of times the tactic t has been selected at node g) and the total action value $W(g, t)$ for each tactic t of a goal g. These statistics will be used in the selection phase and accumulated during the back-propagation phase of the proof search described below.

Selection The number of nodes in the proof hypergraph grows exponentially with the distance to the root. Thus, naive breadth-first search is infeasible to find deep proofs and some prioritization criteria is required to balance depth and breadth. Similar to *MCTS*, we balance the policy model's prior with current estimates from the critic. In particular, we experiment with two different search policies: PUCT [19] and Regularized Policy (RP) [20] (algorithms are detailed in Appendix C.3).

A key difference between previous work and ours is that our proof search operates on a hypergraph. Thus, whereas *MCTS* goes down a path from the root to an unexpanded node during its selection phase, our algorithm will instead create a partial proof hypertree, leading to a set of either solved or unexpanded nodes. To do so, we recursively follow the arg-max of the search policy from the root, until we reach the leaves of the current hypergraph (the detailed pseudo-code can be found in Section C.4). This selection step is illustrated in Figure 2.

In order to batch calls to the policy and critic models over more nodes to expand, we run several selections sequentially, using a virtual loss [21, 5] to produce different partial proof-trees. Note that solving all unexpanded leaves of any of these trees would immediately lead to a full proof of the root. In the next section, we describe how nodes are expanded.

Expansion To expand a node g, we use the policy model to sample tactics that would make progress on the goal. Tactics are sampled in an auto-regressive fashion (token by token) by the decoder [22], based on the previously generated tokens, and on a representation of the goal provided by the encoder. The generated tactics are then evaluated in the theorem proving environment. Each valid tactic will lead to a set of new subgoals to solve, or to an empty set if the tactic solves the goal. Finally, we add a hyperedge for each valid tactic t_i from the expanded node g to its (potentially empty) set of children for this tactic $\{g_i^0, ..., g_i^k\}$. Note that these children might already be part of the hypergraph. For new nodes, visit counts $N(g, t)$ and total action values $W(g, t)$ are initialized to zero. There are three types of nodes in the hypergraph:

- *Solved*: at least one tactic leads to an empty set, or has all its children solved.
- *Invalid*: all tactics sampled from the policy model were rejected by the environment, or lead to invalid nodes.
- *Unsolved*: neither solved nor invalid, some tactics have unexpanded descendants.

Note that the definitions for *Solved* or *Invalid* are recursive. These status are updated throughout the hypergraph anytime a hyperedge is added. Tactics leading to invalid nodes are removed to prevent simulations from reaching infeasible nodes. Once this is done, we back-propagate values from the expanded nodes up to the root, as described in the next section.

Back-propagation For each expanded goal g in a simulated proof tree T, its value is set to $v_T(g) = 1$ if it is solved, and $v_T(g) = 0$ if it is invalid. Otherwise, its value is estimated by the critic model: $v_T(g) = c_\theta(g)$. This provides v_T for all leaves of T and we can then back-propagate in topographic order (children before parents) through all nodes of T. Interpreting the value of a node as the probability that it can be solved, since solving a goal requires solving all of its children subgoals, the value of a parent is the product of values of its children (we assume that the solvability of subgoals is independent, for simplicity):

$$v_T(g) = \prod_{c \in \text{children}(g,t)} v_T(c)$$

In particular, the value of a goal g is the product of the values of all leaves in T that remain to be solved to obtain a proof of g. Once all values in T are computed, we increment the corresponding visit count $N(g, t)$ in the hypergraph as well as the total action values: $W(g, t) \mathrel{+}= v_T(g)$. For a goal g, the estimated value for tactic t is then the mean of the total action value:

$$Q(g, t) = \frac{W(g, t)}{N(g, t)}$$

A fully detailed back-propagation step is illustrated in Figure 2.

3.2 Online training

Both the policy model P_θ and the critic model c_θ are encoder-decoder transformers [23] with shared weights θ, which are trained online on two different objectives. The policy model P_θ takes as input a tokenized goal and generates tactics. It is trained with a standard seq2seq objective [22], where we minimize the cross-entropy loss of predicted tactic tokens conditioned on the input goal.

Our critic model c_θ is used to predict floating point values representing how likely a goal is to be solved. We start decoding with a special token, restrict the output vocabulary to two tokens PROVABLE and UNPROVABLE, and evaluate the critic with $c_\theta(g) = P(\texttt{PROVABLE}|g, \texttt{CRITIC})$. This objective is identical to a seq2seq objective where the cross-entropy is minimized over the two special tokens.

Our online training uses a distributed learning architecture reminiscent of AlphaZero [19] or distributed reinforcement learning setups [24, 5]. A distributed data parallel trainer receives training data from a set of asynchronous provers that run proof searches on theorems chosen by a controller. Provers, in turn, continuously retrieve the latest model versions produced by the trainers in order to improve the quality of their proof search. This set-up is represented in Figure 7 of the Appendix. Once a prover finishes a proof-search, we extract two types of training samples from its hypergraph:

Tactic samples. At the end of a successful proof search, we extract (goal, tactic) pairs of a minimal proof hypertree of the root node as training samples for the policy model. We use a different minimality criterion depending on the environment: number of proof steps for Metamath and Equations and total tactic CPU time for Lean. We show that this selection has a large impact on performances, other options such as selecting all solved nodes are investigated in Section 5.2.1. The policy model is trained with a standard seq2seq objective [22], where we minimize the cross-entropy loss of predicted tactic tokens conditioned on an input goal.

Critic samples. In the proof search hypergraph, we select all nodes that are either solved, invalid (all tactics failed or led to invalid nodes), or with a visit count higher than a threshold. Then, we use $c(g) = 1$ as the training target for solved nodes. For internal nodes, we use the final estimated action value $c(g) = W(g, t^*)/N(g, t^*)$ where t^* is the tactic that maximizes the search policy at g. Finally, for invalid nodes, we use $c(g) = 0$.

The trainers receive training samples that are stored into two separate finite-size queues, one for each objective. When a queue is full, appending a new sample discards the oldest one. In order to create a batch for a task, we uniformly select samples in the corresponding queue.

During online training, in addition from this generated data, we also sample from the supervised datasets used for fine-tuning our models (see 4.1) which provide high-quality data. All training objectives are weighted equally. An overview of our full training pipeline is given in Appendix D.1.

Our proof-search depends on many hyper-parameters, and the optimal settings might not be the same for all statements, making tuning impractical. Thus, the controller samples these hyper-parameters from pre-defined ranges (see Appendix D.3 for details) for each different proof-search attempt.

4 Experiments

In this section, we provide details about our experimental training and evaluation protocols. We first describe the supervised datasets used to fine-tune our policy models, as well as the tokenization used. Then, we discuss the evaluation datasets and methodology. In Appendix D.2, we provide additional details about our model pretraining and architecture.

We develop and test our methods on three theorem proving environments: Metamath, Lean and Equations. Metamath [25] comes with a database of $35k$ human-written theorems called `set.mm`. We also evaluate our methods on the Lean proving environment, which provides a level of automation that is helpful to solve more complex theorems. Lean comes with a human-written library of $27k$ theorems called Mathlib [26].

4.1 Model fine-tuning and supervised datasets

Starting the HTPS procedure described in Section 3 from a randomly initialized model would be sub-optimal, as no valid tactic would ever be sampled from the policy model. Thus, starting the online training from a non-trivial model is critical. To this end, we first fine-tune our policy model P_θ on a supervised dataset of theorems specific to each environment. We refer to this model as the *supervised* model.

Metamath In Metamath, we extract all proofs from the `set.mm` library, composed of 37091 theorems (c.f. Section D.5 for the version of `set.mm`). The training set is composed of around 1M

Table 1: **Dataset statistics for supervised training.**

	# train theorems	# train proof steps	Avg. goal length
Equations	∞	∞	33.7
Metamath	35k	1M	120.1
Lean	24k	144k	169.3

goal-tactic pairs; more statistics about the training data are provided in Table 1. Tokenization in Metamath is trivial, as statements are composed of space-separated tokens.

Lean Following [11], we extract a supervised dataset from the Mathlib library and co-train with the dataset of proof-artifacts of Han et al. [10] to reduce overfitting. To facilitate experimentation and reproducibility, we use fixed versions of Lean, Mathlib, and miniF2F (c.f. Appendix D.5). Finally, we add another supervised co-training task by converting to Lean a synthetic dataset of theorems generated by the Equations environment (c.f. Appendix E.6). Statistics about the training set are available in Table 1.

Equations Finally, we developed the Equations environment as a simpler analogue to existing proving environments. Its expressivity is restricted to manipulating mathematical expressions (e.g. equalities or inequalities) with simple rules (e.g. $A + B = B + A$, or $A < B \Rightarrow -B < -A$). Unlike Metamath or Lean, the Equations environment does not come with a dataset of manually annotated proofs of theorems. Instead, we generate supervised data on the fly using the random graph generator described in Appendix E.5. As the model quickly reaches perfect accuracy on these synthetic theorems, we only leverage statements from the *Identities* split during online training.

4.2 Evaluation settings and protocol

In Polu et al. [11], the model is fine-tuned on theorems from the training set and expert iteration is done on theorems from different sources: train theorems, synthetic statements, and an extra curriculum of statements without proofs (miniF2F-curriculum). The produced model is then evaluated on unseen statements, namely the validation and test splits of the miniF2F dataset [6].

In this work, we also consider the *transductive* setup: on a corpus of unproved statements available at train time, how many proofs can our method learn to generate? This protocol is also sensible, as allowing the model to learn from a failed proof-search can lead to more focused exploration on the next attempt, proving more statements overall than a model that would not be trained online.

Following [9], we also evaluate the pass@k by running k proof searches on the evaluated statements with the policy and critic obtained by online training. Given the many evaluations presented in this work, we only run them once. We give more details on the hyper-parameters used in Appendix D.3. In the transductive setup, we also report the *cumulative pass rate*, i.e. the proportion of theorems solved at least once during online training.

5 Results

In this section, we present our results and study the moving parts of our pipeline through ablations. We compare our results with GPT-f which represents the state of the art on Metamath and Lean.

Table 2: **Pass rate on Lean environment using 64 trials (pass@64)** Numbers with a † exponent correspond to the cumulative pass-rate since the evaluated statements are part of the online training.

	Supervised	GPT-f	Evariste-1d	Evariste-7d	Evariste
Online training statements	-	miniF2F-curriculum			miniF2F-valid
miniF2F-valid	38.5	47.3	46.7	47.5	58.6†
miniF2F-test	35.3	36.6	38.9	40.6	41.0
miniF2F-curriculum	20.8	30.6	33.6†	42.5†	32.1
Train time (A100 days)	50	2000	230	1620	1360

5.1 Main results

5.1.1 Lean

In Lean, we run our experiments on A100 GPUs with 32 trainers and 200 provers. Each prover runs our Lean API on 48 CPU cores. Unlike Polu et al. [11], we sample statements equally from mathlib-train and miniF2F-curriculum, to avoid giving too much importance to statements from a different domain than the target. Results can be found in Table 2. After 1 day of training, each statement from miniF2F-curriculum has been sampled on average 250 times, and 110 out of the 327 statements have been solved. Our model outperforms GPT-f on miniF2F-test, with an approximately $10\times$ training time speed-up. After 7 days, we solve 139 statements of miniF2F-curriculum (100 for GPT-f), and observe further improvements on miniF2F-valid or miniF2F-test.

For other evaluations, we depart from the set-up of Polu et al. [11], directly using the statements from the miniF2F-valid split in our online training, obtaining 58.6% cumulative pass rate. We then evaluate the final model on miniF2F-test, reaching 41% pass@64, against 36.6% for GPT-f. Without the synthetic data co-training task, the performance drops to 54.9% cumulative pass rate on the miniF2F-valid split, and 38.5% pass@64 on the miniF2F-test split. Examples of proofs found by our model can be found in Appendix F.

5.1.2 Metamath

On Metamath, we train our model on V100 GPUs, with 128 trainers and 256 provers, whereas ablations are run on 16 trainers and 32 provers. We report our results in Table 3 for the supervised model and for a model trained with online training. During online training, we sample statements from the training and from the validation splits of `set.mm` equally.

Online training dramatically improves performances on valid statements, going from a 61% pass@8 to a cumulative pass rate of 82.6%. This improvement cannot solely be explained by the high number of attempts on validation theorems during training. Indeed, the ablation in Figure 3 (right) shows that Evariste significantly outperforms a supervised model with the same number of attempts. The supervised model plateaus at 66% while Evariste keeps improving beyond 74% after 7 days of training, showing that the model is able to learn from previous proof searches through online training.

On test theorems, for which statements were not provided during online training, the accuracy increased by 10% compared to the supervised model, from 55.8% to 65.6% accuracy. The supervised model obtains a pass@32 accuracy of 65.4% (resp. 61.2%) on the validation (resp. test) splits, compared to GPT-f's 56.5% (resp. 56.2%) after expert iteration.

Table 3: **Results on Metamath for a supervised model and Evariste.** We report the pass@8 and pass@32 scores on the validation and test splits. Additionally, for Evariste we also report the cumulative score on the validation set, i.e. the fraction of theorems proved at least one time during online training. Note that for Evariste on Valid, the cumulative and pass@k performances are close since these statements were seen during training.

	Valid			Test	
	cumulative	pass@8	pass@32	pass@8	pass@32
Supervised	N/A	61.0%	65.4%	55.8%	61.2%
Evariste	82.6%	81.0%	81.2%	65.6%	72.4%

5.1.3 Equations

In Equations, we run our main experiment with 32 trainers and 64 provers, whereas ablations are run on 16 trainers and 32 provers. In this environment, the model easily learns the training distribution of our random generator, and solves all synthetically generated problems. Thus, online training is run on the *Identities* statements only. Our main experiment reaches a cumulative pass rate of 91.3% on the *Identities* split, while a supervised model never exceeds 36% even after a similar number of proof attempts. In Appendix 9, we give examples of *Identities* statements proved during online training, as well as the size and depth of proofs found by the model.

In particular, Evariste managed to find the proof of complex mathematical statements, such as $\sinh(x/2) = \sinh(x)/\sqrt{2(1+\cosh(x))}$ and $\tan(3x)(1 - 3(\tan(x))^2) = 3\tan(x) - (\tan(x))^2\tan(x)$ that required 82 and 117 proof steps respectively, showing the abilities of HTPS to

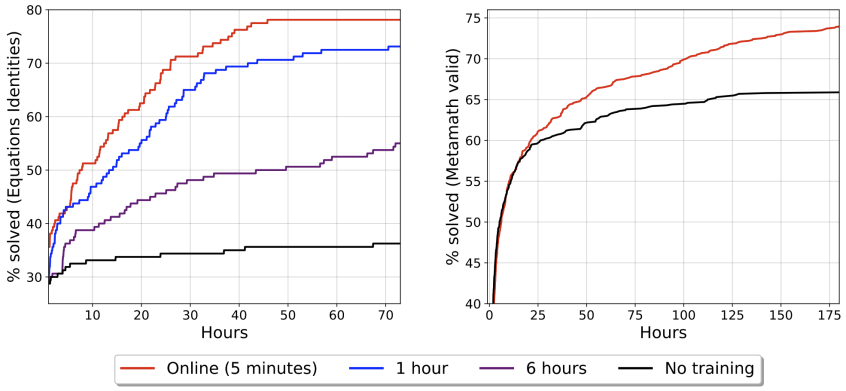

Figure 3: **Comparison between online setup, expert iteration, and fixed model.** We report the cumulative pass rate on the *Identities* (resp. valid) split on Equations (resp. Metamath). Reloading the model more frequently converges faster and to a better performance. When no training is done, the final performance is much lower despite using as many attempts, showing that online training is able to learn from previous proof searches.

prioritize subgoals and guide the search in very large proof graphs. This shows that online training is able to adapt our policy and critic models to a completely new domain, going from automatically generated statements to identities found in math books. Examples to understand the gap between these two domains can be found in Appendix E.

5.2 Ablation study

In this section, we present an ablation study on several components of our system. Since Lean experiments are CPU intensive, we run most of our ablations on the Equations and Metamath environments. On Lean, we ran experiments on a smaller subset of hyper-parameters that consistently performed well on the other environments.

5.2.1 Online training data for tactic objective

Table 4: **Performance of our model for different online training data for tactic objective.** We report the pass@8 score for Metamath and cumulative pass rate for Equations. We either keep all nodes and sample tactics according to the policy, or, extract (minimal) proofs of solved nodes, or (minimal) proofs of the root theorem only. Selecting minimal proofs always improves performance.

Proof Of	All Solved		Root		All Nodes
Type of Proof	All	Min	All	Min	
Metamath (valid)	61.2	65	57.4	**68.6**	51.6
Metamath (test)	57.2	**58.8**	54.8	57.4	54.4
Equations (Identities)	40.6	**78.1**	37.5	71.3	37.5

The way we filter tactics sent to the trainers has a large impact on final performances. We investigated several filtering methods and report the results in Table 4. The first method is similar to the one used in AlphaZero and exposed in [19]: we select all nodes of the proof search hypergraph where the visit count is above a certain threshold and we filter tactics above a given search policy score. At training time, tactics are sampled according to the filtered search policy. With this method the model reaches 51.6% pass@8 on the valid set of Metamath and 37.5% cumulative pass rate on Equations.

We then experimented with other filtering criteria, selecting only goal-tactic pairs that are part of proofs: either a proof of the root node, or of any solved node in the hypergraph. Then, we learn from all possible proofs, or only from proofs that are minimal according to a criteria (number of proof steps for Equations and Metamath, cumulative CPU time for Lean).

Learning only from minimal proofs always leads to improved performance, regardless of the selected roots. Learning from the minimal proofs of all solved nodes, we reach a cumulative pass rate of 78.1% on Equations, compared to 40.6% when learning from all proofs. On Metamath, only learning from the root's minimal proof gives the best result on the valid set, reaching a pass@8 of 68.6%.

5.2.2 Critic

Table 5: **Ablation study on the critic and search hyper-parameters in HTPS.** We report the pass@8 score for Metamath, and the cumulative pass rate for Equations. Evariste, trained with a soft critic and stochastic hyper-parameters, obtains the best performance in both environments. Removing the critic, or using a hard critic leads to reduced performances. In Equations, adding stochasticity in the proof search hyper-parameters increases the performance by 4.3% in Equations, and slightly improves performance in Metamath.

	Evariste	No critic	Hard critic	Fixed search params
Metamath (valid)	68.6	64.8	67.6	**69.8**
Metamath (test)	**57.4**	52.2	57.4	56.2
Equations (*Identities*)	**78.1**	65.6	63.1	73.8

To measure the impact of our critic model, we run an experiment where the proof search is only guided by the policy model. In particular, during the back-propagation phase, we set $v_T(g) = 0.5$ for leaves of T. In that context, our model is no longer trained with a critic objective. We report the results in Table 5. We find that using a critic model improves the performance significantly, by 5.2% and 12.5% on Metamath and Equations respectively.

As mentioned in Section 3, to train the critic objective, we set the training targets as $c(g) = 1$ for solved nodes, $c(g) = 0$ for invalid nodes and $c(g) = W(g, t^*)/N(g, t^*)$ where t^* is the tactic that maximizes the search policy at g, for internal nodes. We also tested a hard critic estimation of the target values, following Polu and Sutskever [9], where $c(g) = 1$ for solved nodes and $c(g) = 0$ for both invalid and internal nodes. We report results in Table 5. For both Metamath and Equations, HTPS critic targets allow Evariste to reach its best performance. In Equations, the model reaches a cumulative pass rate of 78.1%, compared to 63.1% with hard critic estimates. In Equations, using hard critic targets gives worse performances than having no critic model at all, showing that these targets are a bad estimation: setting all internal nodes to zero is too pessimistic.

5.2.3 Fixed proof search parameters

We study the impact of sampling HTPS hyper-parameters for each attempt during online training. We run experiments with fixed, chosen search parameters for Equations and Metamath to compare with random sampling, and report results in Table 5. Evariste achieves better performances than the model trained with fixed search parameters on Metamath test set and Equations *Identities*, reaching 78.1% pass rate compared to 73.8% in Equations *Identities*.

5.2.4 Model update frequency during online training

In our online training procedure, the policy and critic models are updated every five minutes on the provers. We measure the impact of the frequency of these updates by trying different refresh rates: 5 minutes, 1 hour, 6 hours for Equations, and no updates at all for both Equations and Metamath. We report the cumulative pass rate over training hours in Figure 3. The higher the refresh rate, the better the cumulative pass rate over time, confirming the benefits of online training over expert iteration.

6 Conclusion

In this work, we introduce HTPS, an AlphaZero-inspired proof search algorithm for automated theorem proving, along with an online training procedure. We run an extensive study of our pipeline, and present state-of-the-art results on multiple proving environments. We show that online training provides large speed-ups over expert iteration, and allows generalization of the policy and critic models to completely new domains. Despite large number of attempts per theorem, proving the entirety of datasets like miniF2F remains elusive, and generating data from proof-search on the currently available corpora will likely be insufficient in the long term. As manually annotated formal datasets are limited, another way of providing exploration and additional training data (in the spirit of self-play for two player games) is required. Automated generation of new theorems is likely to be one of the future milestones.

Acknowledgments

We thank the Meta AI and FLARE teams for useful comments and discussions throughout this work, notably, Baptiste Rozière, Faisal Azhar, Antoine Bordes, Quentin Carbonneaux, Maxime Darrin, Alexander Miller, Vincent Siles, Joe Spisak and Pierre-Yves Strub. We thank François Charton for his initial contributions to the Equations environment. We also thank Tristan Cazenave for interesting discussions around search algorithms, as well as the members of the Lean community for their help, notably Fabian Glöckle for valuable feedback on this project.

References

[1] Alfred B Kempe. On the geographical problem of the four colours. *American journal of mathematics*, 2(3):193–200, 1879.

[2] Vladimir Voevodsky. Univalent foundations of mathematics. In *International Workshop on Logic, Language, Information, and Computation*, pages 4–4. Springer, 2011.

[3] Thomas Hales, Mark Adams, Gertrud Bauer, Dat Dang, John Harrison, Truong Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Nguyen, Truong Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, An Ta, Trân Trung, Diep Trieu, and Roland Zumkeller. A formal proof of the kepler conjecture. *Forum of Mathematics, Pi*, 5, 01 2017. doi: 10.1017/fmp.2017.1.

[4] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

[5] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[6] Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. Minif2f: a cross-system benchmark for formal olympiad-level mathematics. *arXiv preprint arXiv:2109.00110*, 2021.

[7] P. C. Gilmore. A proof method for quantification theory: Its justification and realization. *IBM J. Res. Dev.*, 4(1):28–35, jan 1960. ISSN 0018-8646. doi: 10.1147/rd.41.0028. URL `https://doi.org/10.1147/rd.41.0028`.

[8] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7 (3):201–215, jul 1960. ISSN 0004-5411. doi: 10.1145/321033.321034. URL `https://doi.org/10.1145/321033.321034`.

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

[10] Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward W Ayers, and Stanislas Polu. Proof artifact co-training for theorem proving with language models. *arXiv preprint arXiv:2102.06203*, 2021.

[11] Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. Formal mathematics statement curriculum learning. *arXiv preprint arXiv:2202.01344*, 2022.

[12] Daniel Whalen. Holophrasm: a neural automated theorem prover for higher-order logic. *arXiv preprint arXiv:1608.02644*, 2016.

[13] Minchao Wu, Michael Norrish, Christian Walder, and Amir Dezfouli. Tacticzero: Learning to prove theorems from scratch with deep reinforcement learning. *Advances in Neural Information Processing Systems*, 34:9330–9342, 2021.

[14] Maxwell Crouse, Ibrahim Abdelaziz, Bassem Makni, Spencer Whitehead, Cristina Cornelio, Pavan Kapanipathi, Kavitha Srinivas, Veronika Thost, Michael Witbrock, and Achille Fokoue. A deep reinforcement learning approach to first-order logic theorem proving. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 6279–6287, 2021.

[15] Bruce Abramson and Richard E Korf. A model of two-player evaluation functions. In *AAAI*, pages 90–94, 1987.

[16] Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Miroslav Olšák. Reinforcement learning of theorem proving. *Advances in Neural Information Processing Systems*, 31, 2018.

[17] L Victor Allis, Maarten van der Meulen, and H Jaap Van Den Herik. Proof-number search. *Artificial Intelligence*, 66(1):91–124, 1994.

[18] Ti-Rong Wu, Chung-Chin Shih, Ting Han Wei, Meng-Yu Tsai, Wei-Yuan Hsu, and I-Chen Wu. Alphazero-based proof cost network to aid game solving. In *International Conference on Learning Representations*, 2021.

[19] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[20] Jean-Bastien Grill, Florent Altché, Yunhao Tang, Thomas Hubert, Michal Valko, Ioannis Antonoglou, and Rémi Munos. Monte-carlo tree search as regularized policy optimization. In *International Conference on Machine Learning*, pages 3769–3778. PMLR, 2020.

[21] Guillaume MJ-B Chaslot, Mark HM Winands, and HJVD Herik. Parallel monte-carlo tree search. In *International Conference on Computers and Games*, pages 60–71. Springer, 2008.

[22] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014.

[23] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[24] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, et al. Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296*, 2015.

[25] Norman D. Megill and David A. Wheeler. *Metamath: A Computer Language for Mathematical Proofs*. Lulu Press, Morrisville, North Carolina, 2019. `http://us.metamath.org/downloads/metamath.pdf`.

[26] The mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, jan 2020. doi: 10.1145/3372885.3373824. URL `https://doi.org/10.1145%2F3372885.3373824`.

[27] Stephan Schulz. E—a brainiac theorem prover. *AI Commun.*, 15(2–3):111–126, 2002.

[28] Alexandre Riazanov and Andrei Voronkov. Vampire 1.1 (system description). In *IJCAR*, 2001.

[29] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[30] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[31] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[32] Kshitij Bansal, Sarah Loos, Markus Rabe, Christian Szegedy, and Stewart Wilcox. Holist: An environment for machine learning of higher order logic theorem proving. In *International Conference on Machine Learning*, pages 454–463. PMLR, 2019.

[33] John Harrison. Hol light: A tutorial introduction. In *International Conference on Formal Methods in Computer-Aided Design*, pages 265–269. Springer, 1996.

[34] Thibault Gauthier, Cezary Kaliszyk, Josef Urban, Ramana Kumar, and Michael Norrish. Tactictoe: learning to prove with tactics. *Journal of Automated Reasoning*, 65(2):257–286, 2021.

[35] Karel Chvalovskỳ, Jan Jakubův, Miroslav Olšák, and Josef Urban. Learning theorem proving components. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 266–278. Springer, 2021.

[36] Mingzhe Wang and Jia Deng. Learning to prove theorems by learning to generate theorems. *Advances in Neural Information Processing Systems*, 33:18146–18157, 2020.

[37] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511*, 2020.

[38] David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. Analysing mathematical reasoning abilities of neural models. In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=H1gR5iR5FX`.

[39] Guillaume Lample and François Charton. Deep learning for symbolic mathematics. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=S1eZYeHFDS`.

[40] Stéphane d'Ascoli, Pierre-Alexandre Kamienny, Guillaume Lample, and François Charton. Deep symbolic regression for recurrent sequences. *arXiv preprint arXiv:2201.04600*, 2022.

[41] Brenden K Petersen, Mikel Landajuela Larma, Terrell N. Mundhenk, Claudio Prata Santiago, Soo Kyung Kim, and Joanne Taery Kim. Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients. In *International Conference on Learning Representations*, 2021. URL `https://openreview.net/forum?id=m5Qsh0kBQG`.

[42] François Charton, Amaury Hayat, and Guillaume Lample. Learning advanced mathematical computations from examples. *arXiv preprint arXiv:2006.06462*, 2020.

[43] Yizao Wang and Sylvain Gelly. Modifications of uct and sequence-like simulations for monte-carlo go. In *2007 IEEE Symposium on Computational Intelligence and Games*, pages 175–182. IEEE, 2007.

[44] Mark HM Winands, Yngvi Björnsson, and Jahn-Takeshi Saito. Monte-carlo tree search solver. In *International Conference on Computers and Games*, pages 25–36. Springer, 2008.

[45] Jungo Kasai, Nikolaos Pappas, Hao Peng, James Cross, and Noah A Smith. Deep encoder, shallow decoder: Reevaluating non-autoregressive machine translation. *arXiv preprint arXiv:2006.10369*, 2020.

[46] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[47] Guillaume Lample and Alexis Conneau. Cross-lingual language model pretraining. *arXiv preprint arXiv:1901.07291*, 2019.

[48] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. Mass: Masked sequence to sequence pre-training for language generation. *arXiv preprint arXiv:1905.02450*, 2019.

[49] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[50] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[51] Angela Fan, Edouard Grave, and Armand Joulin. Reducing transformer depth on demand with structured dropout. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=SylO2yStDr`.

[52] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. *NIPS 2017 Autodiff Workshop*, 2017.

7 Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] Section 5.1 includes our results as well as their limitations.
   (c) Did you discuss any potential negative societal impacts of your work? [No] Beyond environmental cost of training models, we do not see any direct path from the research presented in this paper to negative applications.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] The code for the Equations environment will be open-sourced. We also plan to make our trained model publicly available to help people in the formal community. Code for the overall distributed training architecture is tied to our infrastructure and will be difficult to open-source.
   (b) Did you specify all the training details (e.g., data splits, hyper-parameters, how they were chosen)? [Yes] We provide all training details in Section D.2 of the Appendix.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [No] As stated in Section 4.2, we report the result of one training and one evaluation as running all evaluations multiple times would be too costly.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [No] License is available on the repositories associated with each asset.
   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A] No new assets in this work.
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

A Proving environments

In this section we present the three proving environments used in this paper briefly. For each environment, we show a proof hypertree representation of a theorem, and give an example of tokenized goal and tactic from the training set.

A.1 Metamath

Metamath's only rule is string substitution. Starting from a theorem to be proven, variables are substituted until we reach axioms. In our setup, we consider a tactic to be the label of a theorem in `set.mm`, along with the necessary substitutions. For instance, to show that $2 + 2 = 4$, we can use the rule `eqtr4i` which states that $A = B \wedge C = B \Rightarrow A = C$ with substitutions: $A = (2 + 2)$, $B = (2 + (1 + 1))$, and $C = 4$. We are then left with two subgoals to prove: $(2 + 2) = (2 + (1 + 1))$ and $4 = (2 + (1 + 1))$. The corresponding proof-tree can be found in Figure 4.

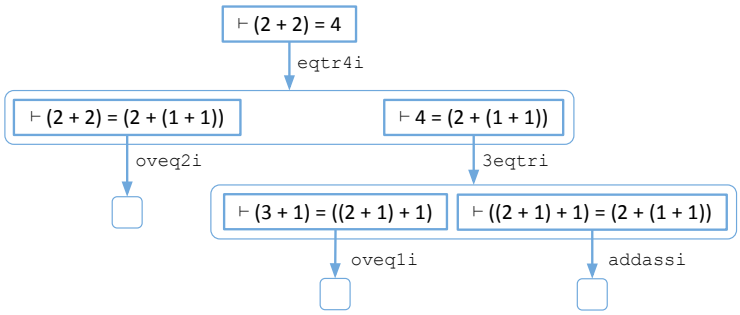

Figure 4: **A visualization of the proof-tree for $2 + 2 = 4$ in Metamath.**

The simplicity of Metamath makes it a great test bed for our algorithms. However, its lack of automation leads to larger proof sizes and its syntax and naming conventions make each step difficult to interpret for neophytes. Similar to GPT-f[9], we implement a parser for Metamath in order to automatically prove the syntactic correctness of statements. Moreover, we use this parser to allow generating only substitutions that cannot be inferred from the goal. The model is conditioned on a goal to prove, and is trained to output a sequence of the following format:

```
LABEL MANDATORY_SUBSTS <EOU> PREDICTABLE_SUBSTS
```

Below is a concrete example of tokenized goal along with its corresponding tactic. The applied rule is `ee10an`[4]. Since the values of `ph` and `th` can be directly inferred from the goal, we do not need generate them at training time, but we still use them during training to reduce overfitting.

```
<GOAL> |- ( A e. RR -> ( ( A - 2 ) + 2 ) = A ) </GOAL>

<TACTIC>
    ee10an
        <VAR> ps = A e. CC </VAR>
        <VAR> ch = 2 e. CC </VAR>
    <EOU>
        <VAR> ph = A e. RR </VAR>
        <VAR> th = ( ( A - 2 ) + 2 ) = A </VAR>
</TACTIC>
```

In order to speed-up decoding, we use a maximum decoding length of 512 tokens for Metamath which covers over 99% of the human tactics in the supervised dataset.

[4] `https://us.metamath.org/mpeuni/ee10an.html`

A.2 Lean

Lean is a full-fledged programming language and benefits from more powerful automation than Metamath, with tactics such as `ring` (able to prove goals using manipulations in semirings), `norm_num` (able to prove numerical goals) or `linarith` (able to find contradictions in a set of inequalities). Unlike Polu and Sutskever [9], our Lean API attempts to split tactic states into separate subgoals when no metavariable is shared. More details about our API and an example proof-tree can be found in Appendix D.4 and Figure 5.

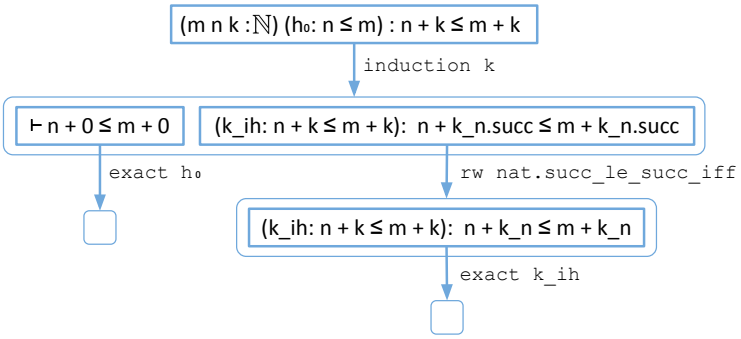

Figure 5: **A visualization of the proof-tree for the proof discussed in the introduction in Lean.**

Similar to Metamath, we use a maximum decoding length of 128 tokens which covers over 99% of the supervised human tactic dataset.

A.3 Equations

We developed the Equations environment as a simpler analogue to existing proving environments. Its expressivity is restricted to manipulating mathematical expressions (e.g. equalities or inequalities) with simple rules (e.g. $A + B = B + A$, or $A < B \Rightarrow -B < -A$). This reduced expressivity makes goals and tactics easy to understand, helping with interpretability and debugging: plotting the set of goals explored during a Metamath proof search does not give a lot of insights on whether it is on track to find a proof. In Section E, we give an in-depth presentation of this environment.

Unlike in Metamath or Lean, we do not have access to a training set of human annotated proofs for this environment. Instead, we create a training set composed of randomly generated synthetic theorems and their proofs (see Section E.5 for details), and manually create an out-of-domain set of non-trivial mathematical identities for which we do not provide proofs. We refer to this evaluation split as *Identities*, a set of 160 mathematical expressions. As synthetic theorems randomly generated are much simpler and significantly differ from statements in the *Identities* split, we can evaluate the ability of our model to generalize to complex and out of domains data. An example proof-tree in Equations is shown in Figure 6.

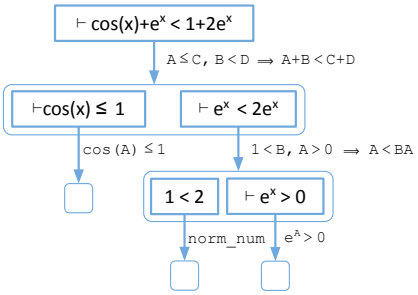

Figure 6: **A visualization of the proof-tree for $\cos(\mathbf{x}) + \mathbf{e^x} < \mathbf{1} + \mathbf{2e^x}$ in Equations.**

Below is an example tokenized goal with its tactic. The goal to prove is $((-(-((4 \times x4) + (x1 \times x4)))) <= (-(-((4 + x1) \times x4))))$, and is tokenized using the reverse Polish notation of the expression. The tactic factorizes the term in position 3, i.e. $(4 \times x4) + (x1 \times x4)$:

```
<GOAL> <= neg neg add mul + 4 x4 mul x1 x4 neg neg mul add + 4 x1 x4 <GOAL>
<TACTIC> ((A_*_C)_+_(B_*_C))|((A_+_B)_*_C) 3 <TACTIC>
```

Similar to Metamath and Lean, we limit the decoder size to 32 tokens to speed-up decoding.

B Related works

Early approaches focused on simpler logics, culminating in extremely efficient first-order provers such as E [27] or Vampire [28]. However, these approaches are insufficient when it comes to theorems written in modern proof assistants such as Isabelle [29], Coq [30], or Lean [4]. Recently, the rising success of deep language models [31] and model-guided search methods [5] has spurred a renewed interest in the problem of automated theorem proving.

Neural theorem provers DeepHOL [32] focuses on the HOL-Light environment [33]. Their model relies on a classifier that can select among a restricted set of tactics and arguments, while we rely on a seq2seq model that can generate arbitrary tactics. The suggested tactics are then used in a breadth-first search. TacticToe [34] uses an MCTS without learned components, using ranking on predefined features to guide the search. Machine learning has also been used to improve classical provers by re-ranking clauses [35]. Finally, [36] uses a neural theorem generator to add data for policy and value training in Holophrasm [12].

Reasoning abilities of language models. Impressive performance of large language models in one or few shot learning [31], machine translation [22] or more recently code generation [37] spurred interest into the reasoning capabilities of large transformers. These model perform quite well on formal tasks such as expression simplification [38], solving differential equations [39], symbolic regression [40, 41], or predicting complex properties of mathematical objects [42]. These studies suggest that deep neural networks are well adapted to complex tasks, especially when coupled with a formal system for verification.

C Hypertree Proof Search

C.1 Hypergraph and definitions

We begin with some useful notations and concepts for our hypergraphs.

Formally, let $\mathcal{G}$ be a set of nodes, and $\mathcal{T}$ a set of tactics. A hypergraph is a tuple $H = (G, r, T, U)$ with $G \subset \mathcal{G}$ the nodes, $r \in G$ the root, and $T \subset G \times \mathcal{T} \times \mathcal{P}(G)$ the admissible tactics. An element of T is written (g, t, c) where g is the start goal, t is the applied tactic and c is the potentially empty set of children that the tactic creates when applied to g in the proving environment.

A hypertree is a hypergraph without cycles, i.e, such that we cannot find a path $g_0, \ldots, g_\ell = g_0$ with $\ell > 0$ and with g_{i+1} in the children of g_i for all i's.

Let $S \subset G$ be the set of solved nodes. A node $g \in G \setminus U$ is solved if one of its tactic leads to no subgoals, or one of its tactics leads to only solved nodes. Formally: $\exists (g, t, \emptyset) \in T$ or $\exists (g, t, c) \in T$ such that $c \subset S$. We say that a tactic t is *solving* for g if all the children it leads to are solved. Conversely, let $U \subset G$ be the set of invalid nodes. A node $g \in G \setminus U$ is invalid if it has been expanded but has no tactics in the hypergraph, or all of its tactics have an invalid child. Formally: $\{(g, t, c) \in T\} = \emptyset$ or $\forall (g, t, c) \in T$, $c \cap I \neq \emptyset$.

These recursive definitions naturally lead to algorithms `MaintainSolved` and `MaintainStatus` to maintain sets S and I when elements are added to H.

A sub-hypertree H_T of H is a connected hypertree rooted at some goal of H. Its leaves leaves(H_T) are its subgoals without children (either elements of U or S). The set of *proofs* of g in H, Proofs(g, H) are all the hypertrees rooted at g that have all their leaves in S. Similarly, the *expandable* subtrees of H rooted in g, Expandable(g, H) are the subtrees with at least one leaf in U. A tactic is said to be *expandable* if it is part of an expandable subtree, this can be computed with a graph-search `ComputeExpandable`.

We can now reformulate the process of proof-search. Starting from a hypergraph that contains only the root theorem r, we produce a sequence of *expandable* subtrees. The unexpanded leaves of these subtrees are expanded in the hypergraph, then the new value estimates are backed-up. The hypergraph grows until we use all our expansion budget, or we find a proof of r.

C.2 Algorithm

C.3 Policies

When a goal g is added to the hypergraph, its visit count $N(g,t)$ and total action value $W(g,t)$ are initialized to zero. Its virtual visit count $VC(g,t)$ are updated during proof search. Let $C(g,t) = N(g,t) + VC(g,t)$ be the total counts. These values are used to define the value estimate with a constant *first play urgency* [43]:

$$Q(g,t) = \begin{cases} \frac{\max(1,N(g,t))}{\max(1,C(g,t))} & \text{if } t \text{ is solving for } g \\ \frac{0.5}{\max(1,C(g,t))} & \text{if } N(g,t) = 0 \\ \frac{W(g,t)}{C(g,t)} & \text{otherwise.} \end{cases}$$

Notice that the value of solving tactics decreases with virtual counts, allowing exploration of already solved subtrees.

Given the visit count N, the total counts C, value estimates Q, the model prior P_θ and an exploration constant c. The policy used in Alpha-Zero is PUCT:

$$\text{PUCT}(g) = \arg\max_{t \in \mathcal{A}} \left[Q(g,t) + c \cdot P_\theta(t|g) \cdot \frac{\sqrt{\sum N(g,\cdot)}}{1 + C(g,t)} \right]$$

Notice that more weight is given to the value estimate Q as N grows which decreases the second term. Another work [20] obtains good performances using as search policy the greedy policy regularized by the prior policy.

$$\pi_{RP}(g) = \arg\max_{y \in \mathcal{S}} \left[Q(g)^T y - c \cdot \frac{\sqrt{\sum C(g,\cdot)}}{\sum (C(g,\cdot) + 1)} KL(\pi_\theta, y) \right] \quad \text{with } \mathcal{S} \text{ the policy simplex at } g$$

Again, note that this policy balances the prior with the value estimates as the count grows, but does not account for individual disparities of visits of each tactics. In our experiments, we obtained better performances with π_{RP} on Equations, and better performances with $PUCT$ on Metamath and Lean.

C.4 Implementation details

Simulation During simulation, we only consider subtrees that could become proofs once expanded. This means we cannot consider any invalid nodes or consider subgraphs containing cycles. If we encounter a tactic that creates a cycle during a simulation, this tactic is removed from the hypergraph, virtual counts from this simulation are removed and we restart the search from the root. This may remove some valid proofs, but does not require a backup through the entire partial subtree which would lead to underestimating the value of all ancestors. Removing tactics from the hypergraph also invalidates computations of *expandable* tactics. This is dealt with by periodically calling `MaintainExpandable` if no valid simulation can be found. A full description of the algorithm that finds one expandable subtree is available in Algorithm 1. Selection of nodes to expand requires finding expandable subtrees until a maximum number of simulations is reached, or no expandable tactic exists at the root. In addition to W, N and v_T, we maintain a virtual loss counter VC following Chaslot et al. [21], Silver et al. [5], so that successive simulations select different leaf subsets. This counter is initialized to zero for all nodes.

Expansion The policy model produces tactics for an unexpanded node g. These tactics are evaluated in the proving environments. Valid tactics are filtered to keep a unique tactic (e.g. the fastest in Lean) among those leading to the same set of children. Finally, we add the filtered tactics and their children to the hypergraph. If no tactics are valid, the node is marked as invalid and we call `MaintainInvalid`. If a tactic solves g, the node is marked as solved and we call `MaintainSolved`.

Algorithm 1 Finding an expandable subtree

Input: A hypergraph H and its $root$
Output: A partial proof tree with unexpanded leaves
:start
T: hypertree($root$)
to_explore: list = [$root$]
while to_explore **do**
  g = to_explore.pop()
  if g is internal **then**
    if expandable$(g) \neq \emptyset$ **then**
      tactic = $\arg\max_t \pi\big|_{\text{expandable}(g)} \pi(g, t)$
    else
      continue { expandable nodes are in a sibling branch }
    end if
    if tactic leads to cycle **then**
      kill tactic
      remove virtual counts for elements of T
      goto *start*
    end if
    $VC(g, tactic) \mathrel{+}= 1$
    T.add(g, tactic, children(g, tactic))
    to_explore += [children(g, tactic)]
  end if
end while

Algorithm 2 Back-propagation of total action value W

Input: Partial proof-tree T and value estimates $c_\theta(g)$ of its leaves.
to_backup = []
for g in leaves of T **do**
  $v_T(g) = c_\theta(g)$
  to_backup.append(($\text{parent}_T(g)$, $\text{parent_tactic}_T(g)$))
end for
while to_backup **do**
  g, t = to_backup.pop()
  to_update = $\prod_{c \in \text{children}_T(g)} v_T(c)$
  $W(g, t) \mathrel{+}=$ to_update
  $N(g, t) \mathrel{+}= 1$
  $VC(g, t) \mathrel{-}= 1$
  $v_T(g) =$ to_update
  g.is_prop = true
  if all c.is_prop for c in $\text{siblings}_T(g)$ **then**
    to_backup.append(($\text{parent}_T(g)$, $\text{parent_tactic}_T(g)$))
  end if
end while

Backup The backup follows topological order from the leaves of a simulated partial proof-tree T, updates W and N, and removes the added virtual count. The algorithm is described in Algorithm 2

C.5 Comparison with other search algorithms

Best First Search Several best-first search variations have been suggested for theorem proving. Proof number search (PNS) [17] is a best-first search algorithm that maintains the minimum number of expansion required to prove or disprove a node. Recent extensions have been proposed, including using a model for estimating the remaining proof / disproof number of a newly expanded node [18]. Similarly, Polu et al. [11] prioritize estimated proof-size in their best-first search objective.

Our node selection heuristic differs slightly: our critic gives the probability of solving a leaf, but our updates to $W(g)$ sums these log-probabilities and thus includes the proof-number information. Moreover, the arities at AND and OR nodes in our cases are highly unbalanced (up to 32 tactics at OR nodes, but very few children per tactic), selecting all children at AND nodes is computationally feasible (which is not the case in games where this would lead to an exponential growth of states to expand). Thus, we depart from the standard best-first search by expanding full candidate proof-trees at once.

Monte Carlo Tree Search [15]. MCTS has been famously used as part of AlphaZero [19] to obtain great performances on two player games. This two player set-up can be mapped to theorem-proving by assigning one player to choosing the best tactics while the other player picks the most difficult goal to solve (a method explored in Holophrasm [12]). However, since we need to provide a proof of the root theorem, we need to ensure that we can solve *all* goals that a tactic leads to. This set-up has been studied for two player games when attempting to compute the game-theoretical value of positions. Using MCTS in this set-up is suboptimal [44], ignoring unlikely but critical moves from the opponent (in our case, a subgoal that looks easy but is impossible to solve). We decided to exploit the highly asymmetrical arities of our two players (most tactics lead to one or two goals) which makes simulating partial proof-trees computationally feasible. Thus, the values we back-propagate always take into account all possible moves from the opponent, while only requiring a few expansions per simulation.

Polu and Sutskever [9] This best-first search expands goals one at a time according to a priority-queue of either a value model or the cumulative log-prior from the language model. Since the priority is equal among siblings but strictly decreasing with depth, this means siblings will always be expanded together. However, nothing prevents the algorithm from jumping from one potential proof-tree to another, and potentially favoring breadth over depth. In comparison, depth does not appear in the value estimate we compute, but rather the remaining number of nodes to solve a particular proof-tree. Moreover, our algorithm leads to value estimates that can be used to train our critic, which performs better than 0-1 estimates provided by best-first search (c.f. Section 5.2.2).

D Training details

D.1 Full training pipeline

In order to bootstrap our online learning procedure we require a policy model P_θ that outputs coherent tactics. While the critic is left untrained, the policy model is fine-tuned on a pretrained transformer using a supervised dataset specific to the target environment. The full training pipeline can be summarized as follows:

- **Pretraining** of the encoder-decoder model on a large unsupervised corpus (c.f. Section D.2).
- **Fine-tuning** of the policy model on supervised datasets detailed in (c.f. Section 4.1).
- **Online training** of both the policy and critic models on data extracted from proof search (illustrated in Figure 7).

D.2 Model architecture and training

Model architecture. Our transformer architecture uses a 12-layer encoder and a 6-layer decoder in all experiments. We use an embedding dimension of 1600 in the encoder and 1024 in the decoder

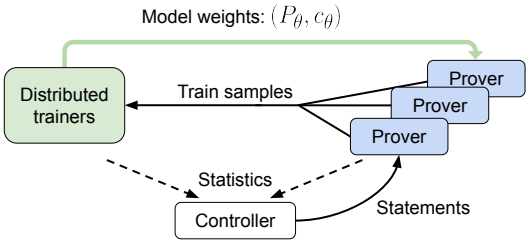

Figure 7: **An overview of our online training architecture.** The controller sends statements to asynchronous HTPS provers and gathers training and proving statistics. The provers send training samples to the distributed trainers and periodically synchronize their copy of the models.

for both Metamath and Lean. For Equations, where we expect the model to require less decoding capacity, the decoding dimension is lowered to 512. We found that reducing the decoder capacity increases the decoding speed without impacting the performance, as previously observed by Kasai et al. [45] in the context of machine translation. This observation led us to use "encoder-decoder" architecture and not a "decoder only" model (as in Polu and Sutskever [9]), in order to store most of the model capacity in the encoder and to sample tactics efficiently with a small decoder. Our models are composed of 440M parameters for Equations and 600M parameters for Metamath and Lean (for comparison, GPT-f uses a 770M parameter, 36-layer model).

Model pretraining. Model pretraining can be critical in low-resource scenarios where the amount of supervised data is limited [46, 47]. Thus, we do not immediately fine-tune our model but first pretrain it on a large dataset to reduce overfitting and improve generalization. In particular, we pretrain our model with a masked seq2seq objective (MASS [48]) on the LaTeX source code of papers from the mathematical section of arXiv. After tokenization, our filtered arXiv dataset contains around 6 billion tokens for 40GB of data. Similar to Polu and Sutskever [9], we observed large performance gains using pretraining. However, we found that arXiv alone provides a better pretraining than when it is combined with other sources of data (e.g. GitHub, Math StackExchange, or CommonCrawl).

Supervised fine-tuning. During fine-tuning, we train our models with the Adam optimizer [49] and an inverse square-root learning rate scheduler [23]. We use a dropout of 0.2 [50] to reduce the overfitting of our models. We also apply layer-dropout [51] with a dropout rate of 0.1 to further reduce overfitting and stabilize training. We implement our models in PyTorch [52] and use float16 operations to speed up training and to reduce the memory usage of our models.

Online training. During online training, we alternate between the *goal-tactic* objective, used during fine-tuning on the supervised dataset, and the *goal-tactic* and *goal-critic* objectives on data generated by the provers. As the model and the data generated by the provers are constantly evolving, we do not want the learning rate to decrease to 0, and we fix it to 3×10^{-5} after the warm-up phase. Unless mentioned otherwise (e.g. for large experiments), we run all Metamath and Equations experiments with 16 trainers and 32 provers for a total of 48 V100 GPUs.

D.3 Proof search hyper-parameters

HTPS depends on many hyper-parameters: the decoding hyper-parameters of the policy model and the search hyper-parameters. Selecting their optimal values would be difficult in practice, if not impractical, for several reasons. First, the model is constantly evolving over time, and the optimal parameters may evolve as well. For instance, if the model becomes too confident about its predictions, we may want to increase the decoding temperature to ensure a large diversity of tactics. Second, even for a fixed model, the ideal parameters may be goal-specific. If an input statement can only be proved with deep proofs, we should favor depth over breadth, and a small number of tactics per node. If the proof is expected to be shallow and to use rare tactics, we will want to penalize the exploration in depth and increase the number of tactics sampled per node. Finally, there are too many parameters to tune and running each experiment is expensive. Thus, we do not set HTPS hyper-parameters to a fixed value, but sample them from pre-defined ranges at the beginning of each proof. These pre-defined ranges were set *a priori* and were not tuned over the course of the experiments.

The decoding parameters and the chosen distribution are the following:

- **Number of samples:** the number of tactics sampled from the policy model when a node is expanded. Distribution: uniform on discrete values $[8, 16, 32, 48]$.
- **Temperature:** sampling temperature used during decoding. Distribution: uniform on range $[0.8, 2.0]$.
- **Length penalty:** penalty on the length of generated sequence. Distribution: uniform on range $[0, 1.2]$.

For the search parameters we have:

- **Number of expansions:** the search budget, i.e. the maximum number or nodes in the proof graph before we stop the search. Distribution: log-uniform with range $[1000, 10000]$.
- **Depth penalty:** an exponential value decay during the backup-phase, decaying with depth to favor breadth or depth. Distribution: uniform on discrete values $[0.8, 0.9, 0.95, 1]$.
- **Exploration:** the exploration constant c in the policy (PUCT or RT). Distribution: log-uniform with range $[0.01, 100]$.

When sampling proof search parameters during evaluation, we use the same distributions than at training time, with two differences: we fix the number of expansions to 5k in Lean and 10k in Metamath.

D.4 Details on our Lean API

States are more complex in Lean than in Metamath: metavariables can appear which are holes in the proof to be filled later. Subgoals sharing a metavariable cannot be solved in isolation. This is addressed in Polu and Sutskever [9] by using as input the entire tactic state. Instead, we inspect tactic states to detect dependencies between subgoals, and split the tactic state into different subgoals where possible in order to maximize state re-use and parallelization in the proof search algorithm. We only ever split the tactic state into contiguous lists of subgoals to make exporting the final proof easier.

Lean's kernel type checker has to be called after each tactic application as tactics sometimes generate incorrect proofs and rely on the kernel for correctness. For every goal in the previous tactic state, we type check the proof term inserted by the tactic. Since the kernel does not support metavariables, we replace every metavariable by a lambda abstraction.

D.5 Metamath Lean versions

To compare our models in the same setup while working on this project, we ran all our experiments with a fixed version of Metamath and Lean. In particular, all experiments were run with the following GitHub commits of `set.mm`, Lean, miniF2F, and Mathlib:

- `https://github.com/metamath/set.mm`: 861bd3552636dcdb9cbc8df59d01b14520c72f82
- `https://github.com/leanprover/lean/`: tag/v3.3.0
- `https://github.com/openai/miniF2F`: 21723db70bbd030e034ed374db74cea4be1bf681
- `https://github.com/openai/miniF2F/tree/statement_curriculum_learning`: c9d827c871aff2ab0f5ec64a0d72e61111a7f072
- `https://github.com/leanprover-community/mathlib`: 9a8dcb9be408e7ae8af9f6832c08c021007f40ec

E Equations environment

In this section, we give additional details about the environment Equations. First, we described its main elements, theorems (resp. tactics) in Section E.1 (resp. E.2). Then, we describe a proof in this environment in Section E.3, how numerical expressions are evaluated and what vulnerabilities this can lead to in Section E.4. Finally, we describe our random theorem generator in Section E.5 and how theorems and their proofs can be translated to Lean in Section E.6.

E.1 Theorems

Each theorem in Equations consists in proving mathematical expressions composed of functions of real numbers, by manipulating and rewriting expressions. A theorem to prove can be an inequality or an equality, conditioned to a set of (potentially empty) initial assumptions. For instance:

$$x^2 + 1 \geq 2x \quad \text{or} \quad x > y \implies e^{y-x} - 1 < 0$$

In the first example, the goal does not have any hypothesis and consists in proving that for every $x \in \mathbb{R}$, $x^2 + 1 \geq 2x$. In the second example, the goal consists in proving that $e^{y-x} - 1 < 0$ for every $x, y \in \mathbb{R}$ that satisfy the hypothesis $x > y$.

Equalities and inequalities are represented as trees with the three following elements:

- **Leaves:** represent a variable, an integer, or a constant (e.g. π).
- **Internal nodes:** represent unary or binary operators, e.g. $+$, $-$, $/$, $\times$, $\exp$, $\ln$, $\cos$, $\sin$, $\sinh$, $\cosh$, etc. More advanced operators such as $\gcd$, lcm, mod (the rest of an euclidean division) are possible when dealing with integers.
- **A root node:** represents a comparison operator, e.g. $=$, $\leq$, $<$, $\geq$, $>$, $\neq$. More advanced comparison operators such as $|$ (divides) are possible when dealing with integers.

E.2 Tactics

Equations allows to deduce equalities and inequalities from simpler subgoals, using elementary rules (i.e. tactics). The environment contains two types of rules: transformations, which consist in matching a pattern in an expression and replacing it by an equivalent expression; and assertions, which consist in asserting that an expression is true. Both types of rules can have assumptions.

Transformation rules A transformation rule (`TRule`) consists in a set of two expressions, L and R, equivalent under a set of assumptions S. For instance $\texttt{TRule}(A + B, B + A)$ is the transformation rule stating the commutativity of the addition, namely that $A + B = B + A$ for any expressions A and B. Note that in this case, the set of assumption S is empty as the equality always holds. Another example is $\texttt{TRule}(\sqrt{A^2}, A, [A \geq 0])$ that states that $\sqrt{A^2} = A$ provided that $A \geq 0$.

Applying such a rule to an existing equation works as follows:

- matching a term T in the expression that has the pattern of L
- identifying the matching variables and substituting them in R
- replacing T by R in the input equation
- return the resulting equation with the set of hypotheses required for the transformation

For instance, if the input goal is:

$$\sqrt{(e^x)^2} = e^x$$

Applying $\texttt{TRule}(\sqrt{A^2}, A, [A \geq 0])$ on this expression will result in two subgoals:

- The same expression, where $\sqrt{A^2}$ has been replaced by A: $e^x = e^x$
- The hypothesis required for the assumption to hold: $e^x \geq 0$

More generally, a transformation rule will result in $N + 1$ subgoals, where N is the number of hypotheses required by the rule.

Assertion rules An assertion rule (`ARule`) expresses the fact that an expression is true, provided some hypotheses. It is represented by a main expression, and a set of assumptions sufficient for the main expression to hold. For instance, the rule $\texttt{ARule}(A \leq C, [A \leq B, B \leq C])$ states the transitivity of the partial order $\leq$, i.e. $A \leq C$ provided that there exists an expression B such that $A \leq B$ and $B \leq C$.

Assertion rules do not always have hypotheses, for instance the reflexivity rule `ARule`$(A = A)$, or the rule `ARule`$(e^A > 0)$ stating that e^A is positive, for any real value A. Note that the two subgoals generated in the previous paragraph ($e^x = e^x$ and $e^x > 0$) can be respectively solved by these two assertion rules (i.e. by matching $A = e^x$ and $A = x$).

Unlike transformation rules that always result in at least one subgoal (the initial expression on which we applied the transformation), assertion rules will only generate N subgoals, where N is the number of hypotheses. As a result, being able to apply an assertion rule without hypotheses to an expression is enough to close (e.g. solve) the goal. Assertion rules are in fact very similar to rules in Metamath.

In Table 6, we provide the number of Equations rules in different categories. Some examples of transformation and assertion rules are given in Table 7.

Table 6: **Number of Equations rules in each category.**

Rule type	Basic	Exponential	Trigonometry	Hyperbolic	All
Transformation	74	18	9	8	109
Assertion	90	11	9	0	110
Total	171	29	18	11	219

Table 7: **Trigonometric rules** accessible by the model. The model only has access to these elementary rules when proving statements from *Identities*. In particular, it cannot use more involved theorems such as $\cos^2(x) + \sin^2(x) = 1$.

Transformation rules	Assertion rules
$\sin(0) = 0$	$\lvert\cos(A)\rvert \leq 1$
$\cos(0) = 1$	$\lvert\sin(A)\rvert \leq 1$
$\sin(\frac{\pi}{2}) = 1$	$\lvert\sin(A)\rvert \leq \lvert A\rvert$
$\cos(\frac{\pi}{2}) = 0$	$A = B \implies \sin(A) = \sin(B)$
$\sin(-A) = -\sin(A)$	$A = B \implies \cos(A) = \cos(B)$
$\cos(-A) = \cos(A)$	$\sin(A) \neq \sin(B) \implies A \neq B$
$\cos(A) \neq 0 \implies \tan(A) = \frac{\sin(A)}{\cos(A)}$	$\cos(A) \neq \cos(B) \implies A \neq B$
$\sin(A + B) = \sin(B)\cos(A) + \sin(A)\cos(B)$	$A = B, \cos(A) \neq 0 \implies \tan(A) = \tan(B)$
$\cos(A + B) = \cos(A)\cos(B) - \sin(A)\sin(B)$	$\tan(A) \neq \tan(B), \cos(A)\cos(B) \neq 0 \implies A \neq B$

E.3 Proving a statement with Equations

In order to prove a theorem with Equations, the user (or automated prover) has to apply tactics on the current expression. A tactic can correspond either to a transformation rule, or to an assertion rule.

For transformation rules, the model needs to provide:

- the rule (using a token identifier)
- the direction in which the rule is applied (a Boolean symbol, for forward or backward)
- an integer that represents the position where the rule is applied
- an optional list of variables to specify (c.f. paragraph below)

The direction of the rule indicates whether we want to transform L by R or R by L (e.g. replace A by $\sqrt{A^2}$, or the opposite). The position where the rule is applied is given by the prefix decomposition of the input expression. For instance, the prefix notation of $(x + y) + 1$ is given by `+ + x y 1`. Applying the commutativity rule $A + B = B + A$ to the expression in position 0 will result in $1 + (x + y)$. Applying it in position 1 will result in $(y + x) + 1$, since the rule was applied to $(x + y)$. Note that for the commutativity rule, the direction in which we apply the rule does not matter. The list of variables to specify is required when variables in the target patterns are absent from the source pattern. For instance, applying the transformation rule `TRule(A,A+B-B)` in the forward direction will require to provide the value of B.

For assertion rules, the format is simpler. We no longer need to specify a direction or a position (the position is always 0 as the assertion statement must match the expression to prove). We just need to provide:

- the rule (using a token identifier)
- an optional list of variables to specify

In this case, the list of variables to specify corresponds to variables that appear in hypotheses and cannot be inferred from the main expression. For instance, to apply the assertion rule $A \leq B, B \leq C \implies A \leq C$, we need to specify the value of B. We will then be left with two subgoals: $A \leq B$ and $B \leq C$.

Proving a statement in Equations requires to recursively apply tactics to unproved subgoals, until we are left with no subgoals to prove.

An example of proof-tree in Equations is shown in Figure 6. Figure 8 shows an example proof of the statement $(\mathbf{x} - \mathbf{y}) - (\mathbf{x} + \mathbf{y}) + \mathbf{2y} = \mathbf{0}$ using rules from the environment. Although simple, this statement requires 22 proof steps and highlights the depth required to prove complex mathematical identities when using elementary proof steps.

Statement to prove	Rule used
$(x-y)-(x+y)+2y=0$	$A-B=A+(-B)$
$(x-y)+(-(x+y))+2y=0$	$-(A+B)=(-A)+(-B)$
$(x-y)+((-x)+(-y))+2y=0$	$A+(B+C)=A+B+C$
$(x-y)+(-x)+(-y)+2y=0$	$A+(-B)=A-B$
$(x-y)+(-x)-y+2y=0$	$A+(-B)=A-B$
$(x-y)-x-y+2y=0$	$\text{int}(a+b)=\text{int}(a)+\text{int}(b)$
$(x-y)-x-y+(1+1)\times y=0$	$A\times B=B\times A$
$(x-y)-x-y+y\times(1+1)=0$	$A\times(B+C)=A\times B+A\times C$
$(x-y)-x-y+y\times 1+y\times 1=0$	$A\times 1=A$
$(x-y)-x-y+y+y\times 1=0$	$A-B=A+(-B)$
$(x-y)-x+(-y)+y+y\times 1=0$	$A+B=B+A$
$(x-y)-x+y+(-y)+y\times 1=0$	$A+(-B)=A-B$
$(x-y)-x+y-y+y\times 1=0$	$A-A=0$
$(x-y)-x+0+y\times 1=0$	$A+0=A$
$(x-y)-x+y\times 1=0$	$A-B=A+(-B)$
$x+(-y)-x+y\times 1=0$	$A+B=B+A$
$(-y)+x-x+y\times 1=0$	$A-A=0$
$(-y)+0+y\times 1=0$	$A+0=A$
$(-y)+y\times 1=0$	$A+B=B+A$
$y\times 1+(-y)=0$	$A+(-B)=A-B$
$y\times 1-y=0$	$A\times 1=A$
$y-y=0$	$A-A=0$
$0=0$	

Figure 8: **Proof of the identity $(\mathbf{x} - \mathbf{y}) - (\mathbf{x} + \mathbf{y}) + \mathbf{2y} = \mathbf{0}$ with elementary rules.** In this example we provide at each step the current goal and the rule that is used to obtain the next goal. This example shows how difficult it can be to prove even simple statements in Equations as they may require a significant number of proof steps (22 in that case). This explains that proving more involved statements from *Identities* such as $\cosh(3x) = 4\cosh(x)^3 - 3\cosh(x)$ can require to generate very large proof trees.

E.4 True expressions and numerical evaluation

Some theorems are trivial, either because their statements match the pattern of an assertion rule that has no assumptions (e.g. $x^2 \geq 0$ or $e^{y-x} \neq 0$), or because they do not contain any variable and an exact numerical evaluation can attest that they are true (e.g $(-1)/2 < 6$ or $1 - 7/4 = -6/8$).

To prevent the model from wasting budget in "uninteresting" branches, we automatically discard generated subgoals that can be trivially verified. However, we only perform numerical verification of expressions without variables when they exclusively involve rational numbers. For instance, we will automatically close subgoals such as $5 < (-3)^2$ or $\frac{1}{2} > \frac{1}{4}$, but not $e^1 < e^2$ or $\cos(3) \neq 0$. To prove that $e^1 < e^2$ the model will need to use, for instance, an assertion rule such as $A < B \implies e^A < e^B$ ($1 < 2$ will then be closed automatically).

In early implementations of the Equations environment, we found that the model was able to leverage vulnerabilities in the environment to reach a 100% accuracy and to prove any statement. These issues where coming from numerical approximations that were initially allowed during the numerical verification of constant expressions (c.f. Section E.4). To prevent these vulnerabilities, we restricted the numerical verification to rational expressions, in order to have an exact numerical evaluation and to avoid errors due to approximations. We give two examples of vulnerabilities found by the model when expressions were verified with an approximate numerical evaluation.

In Figure 9, the model manages to prove that $2 = 3$ by using the injectivity of the exponential function, and the fact that for NumPy, $\exp(-\exp(\exp(2))) = \exp(-\exp(\exp(3)))$. Evaluating the left and the right-hand side both numerically evaluate to 0.0, and the environment incorrectly considered the expression to be valid.

In Figure 10, the model manages to prove that $0 \neq 0$ by first proving that $\cos(\pi/2) \neq 0$, and combining this result with the fact that $\cos(\pi/2) = 0$. The imprecision came from the NumPy approximation of $\cos(\pi/2)$ in 6.123×10^{-17}, and in particular the fact that $(((\cos(\pi/2)^{0.5})^{0.5})^{0.5}) \approx 9.4 \times 10^{-3}$, which was considered large enough by our threshold to be considered non-zero. By using this approximation, and the assertion rule $\sqrt{A} \neq 0 \implies A \neq 0$, the model was able to conclude that $(((\cos(\pi/2)^{0.5})^{0.5})^{0.5}) \neq 0 \implies \cos(\pi/2) \neq 0 \implies 0 \neq 0$.

$$
\begin{array}{rcll}
 & 2 = 3 & \text{Statement to prove} \\
\iff & e^2 = e^3 & \text{Rule: } A = B \iff e^A = e^B, \\
\iff & e^{e^2} = e^{e^3} & \text{Rule: } A = B \iff e^A = e^B, \\
\iff & -e^{e^2} = -e^{e^3} & \text{Rule: } A = B \iff -A = -B, \\
\iff & e^{-e^{e^2}} = e^{-e^{e^3}} & \text{Rule: } A = B \iff e^A = e^B, \\
\iff & 0 = 0 & \text{Numerical evaluation}
\end{array}
$$

Figure 9: **False "proof" of $2 = 3$ found by the model when allowing numerical approximation to verify constant expressions.** The model noticed that $\exp(-e^{e^2})$ = $\exp(-e^{e^3})$ is considered true by NumPy (as the left and the right hand side are both approximated to 0.0) to conclude that $2 = 3$ using the injectivity of the exponential function.

E.5 Random theorem generator

While Metamath and Lean come with a collection of annotated theorems that can be used for training, Equations does not have an equivalent of manually proved statements. Instead, we generate a supervised training set of theorems to pretrain the model before we start the online training. We propose two simple generation procedures: a random walk, and a graph generation approach.

Random walk generation The random walk is the simplest way to generate a theorem. We start from an initial expression A_0 and a set of initial hypotheses, both randomly generated following the method of Lample and Charton [39]. Then, we randomly apply an admissible transformation rule on

$$0 \neq 0 \iff \cos\frac{\pi}{2} \neq 0 \iff \left(\cos\frac{\pi}{2}\right)^{0.5} \neq 0$$
$$\iff \left(\left(\cos\frac{\pi}{2}\right)^{0.5}\right)^{0.5} \neq 0 \iff \quad \dots \quad \iff \left(\left(\left(\cos\frac{\pi}{2}\right)^{0.5}\right)^{\cdots}\right)^{0.5} \neq 0$$

Figure 10: **False "proof" that $0 \neq 0$ found by the model when allowing numerical approximation to verify constant expressions.** Since $\cos(\frac{\pi}{2})$ evaluates to 6.123×10^{-17} in NumPy (and not exactly to 0), the model found that for any tolerance threshold applying the assertion rule $\sqrt{A} \neq 0 \implies A \neq 0$ enough times lead to an expression where the left-hand side evaluates numerically to a strictly positive value. In particular, $(((\cos(\frac{\pi}{2})^{2^{-3}}) \approx 9.4 \times 10^{-3}$, which was considered large enough by our threshold to be considered non-zero. After that, any expressions A and B can be shown to be equal using the assertion rule $(A \times C = B \times C \ \wedge \ C \neq 0) \implies A = B$ where C is chosen to be 0 since $0 \neq 0$.

A_0 to get an equivalent expression A_1. The process is repeated, to get a sequence $A_0, A_1, \dots, A_N$ of equivalent expressions. The final theorem consists in proving that $A_0 = A_N$, and the proof corresponds to the sequence of rules sequentially applied. To increase the diversity of generations, and to avoid sampling only rules without or with simple assumptions, we add a bias in the random sampling of rules to over-sample the underrepresented ones.

Graph generation Because of the simplicity of the random walk approach, the generated theorems tend to be easy to prove, and the model quickly reaches a perfect accuracy on the generated theorems. Moreover, proofs generated by the random walk are only composed of transformation rules. To generate a more diverse set of theorems, we also use a graph generation procedure, that creates a large acyclic graph of theorems, where each node is connected to its children by a rule in the environment. To create such a graph, we proceed as follows. We first generate a set of initial hypotheses, and initialize the graph with a node for each hypothesis. We then randomly apply a transformation or assertion rule on nodes already in the graph.

For instance, if $A \leq B$ and $B \leq C$ are two nodes in the graph, then we can add the node $A \leq C$ using the assertion rule $A \leq B \wedge B \leq C \implies A \leq C$. If $x = y \times (z - 1)$ is a node in the graph, we can use the transformation rule $B \neq 0 \implies A/B = C \iff A = B \times C$ to add the node $x/y = z - 1$, provided that the node $y \neq 0$ is also in the graph. Required hypotheses that are trivially verifiable (e.g. $2 > 0$ or $e^{-x} > 0$) are automatically added to the graph.

E.6 Translating Equations theorems to Lean

Exporting theorems to Lean. To enrich the existing Lean supervised dataset with synthetic data, we built a translator from Equations to Lean. Although Equations statements are easy to translate, proofs can only be translated if they involve rules that also exist in Lean. Since Equations is a modular environment where rules can be specified by the user, we created a collection of Equations rules from existing Mathlib statements. Synthetic theorems can then be generated using the random walk or random graph approaches described in Section E.5, and converted into Lean to augment the existing supervised dataset. Examples of randomly generated Lean proofs are provided in Figure 11.

```
theorem SYNTHETIC_0
  (x1 x3 x4 : ℝ) :
  ((0:ℝ) ≤ ((real.cos (real.cos ((-6:ℝ) / ((x1 - x4) / x3)))) / (2:ℝ))) :=
begin
  apply norm_num.nonneg_pos,
  apply half_pos,
  apply real.cos_pos_of_le_one,
  apply real.abs_cos_le_one,
end

theorem SYNTHETIC_1
  (x1 x4 : ℝ)
  (h0 : ((x4 * (real.exp x1)) < 10)) :
  ((-((abs ((x4 * (real.exp x1)) - 10)) / 2)) < ((abs (10 - (x4 * (real.exp x1)))) / 2)) :=
begin
  have h1 : ((0:ℝ) < ((abs ((x4 * (real.exp x1)) - 10)) / 2)),
  apply half_pos,
  apply abs_pos_of_neg,
  apply sub_neg_of_lt h0,
  apply norm_num.lt_neg_pos _ _ h1,
  rw ← abs_sub_comm,
  apply half_pos,
  apply abs_pos_of_neg,
  apply sub_neg_of_lt h0,
end
```

Figure 11: **Example of a randomly generated theorems in Lean**. The theorems were initially generated in the Equations environment using rules from the Mathlib library, and converted to Lean.

Importing rules from Mathlib. To allow interfacing Equations and Lean, we automatically parsed Mathlib statements from the Lean library, and extracted theorems with a statement compatible with the Equations environment. Compatible theorems are converted into Equations transformation or assertion rules. Overall, we converted 1702 theorems from the Lean Library into our Equations environment. Details about the number of converted theorems are provided in Table 8.

Table 8: **Number of Equations rules converted from Lean**. The converted Lean theorems can be used to generate synthetic theorems within the Equations environment. The generated theorems can then in turn be converted back to Lean, along with their proofs. Some theorems are generic and can be applied to different types of variables (e.g. `add_comm`), and will appear in different categories. Overall, we automatically converted 1702 different Lean rules in our Equations environment.

Rule type	Natural numbers	Integers	Real numbers
Transformation	304	452	799
Assertion	314	292	407
Total	618	744	1206

E.7 Examples of identities solved by the model on Equations

In Table 9, we give some examples of identities solved by the model. For each statement, we indicate the proof size and the proof depth, for the first proof found by the model, and for the optimal proof. We observe that the first proofs are sometimes very large, with more than 100 nodes, and that the model later manages to find shorter proofs as it improves.

Table 9: **Examples of identities solved.** Some of the 144 identities found by our model, in the order they were first solved. For each identity, we provide the size and the depth, both the for first proof, and for the minimal proof (i.e. the proof with the smaller number of steps) found during online training. The model found proofs with over 350 steps, some exceeding a depth of 100. After additional proof search, the model is often able to find shorter proofs. The proof of $\sin(2\pi + x) = \sin(x)$ requires a large number of steps, as the model can only use simple rules (e.g. the trigonometric rules provided in Table 7), and it does not have access to the value of $\sin(2\pi)$ or $\sin(\pi)$.

Identity	Proof size		Proof depth	
	First	Best	First	Best
$\exp(-x)\exp(x-y) = \exp(-y)$	6	6	6	6
$\cosh(-x) = \cosh(x)$	4	4	4	4
$\sin(\pi/2 + x) = \cos(x)$	8	8	8	7
$0 < x \implies 2\ln(\sqrt{x}) = \ln(x)$	16	3	7	3
$\cos(\pi/2 - x) = \sin(x)$	19	11	19	10
$\sin(\pi/2 - x) = \cos(x)$	14	10	14	10
$\cos(x)^2 + \sin(x)^2 = 1$	13	11	13	10
$\cos(x) = \cos(x/2)^2 - \sin(x/2)^2$	16	11	16	7
$\sin(x+y) - \sin(x-y) = 2\sin(y)\cos(x)$	24	14	23	14
$0 < x \implies 2x\cosh(\ln(x)) = x^2 + 1$	20	14	18	12
$\tanh(x) = (\exp(x) - \exp(-x))/(\exp(x) + \exp(-x))$	46	23	30	11
$\cos(x-y) + \cos(x+y) = 2\cos(x)\cos(y)$	33	19	33	13
$\cosh(x) - \sinh(x) = \exp(-x)$	27	20	27	19
$\cosh(x) - \sinh(x) = \frac{1}{\sinh(x)+\cosh(x)}$	55	38	40	20
$\sin(2x) = 2\sin(x)\cos(x)$	27	15	19	8
$\cos(2x) = 1 - 2\sin(x)^2$	130	27	118	21
$\cosh(x-y) + \cosh(x+y) = 2\cosh(x)\cosh(y)$	84	31	84	29
$\tanh(x) = (\exp(2x) - 1)/(\exp(2x) + 1)$	205	65	176	39
$\sin(x) = 2\sin(x/2)\cos(x/2)$	29	17	21	8
$\cos(2x) = 2\cos(x)^2 - 1$	72	26	68	19
$\cos(x)^2 = 1 + \cos(2x)/2$	71	30	61	16
$\sinh(x) = 2\sinh(x/2)\cosh(x/2)$	64	37	51	25
$\sinh(2x) = 2\sinh(x)\cosh(x)$	71	34	61	24
$\sinh(x+y) = \sinh(x)\cosh(y) + \cosh(x)\sinh(y)$	130	77	121	63
$\cosh(x-y) = \cosh(x)\cosh(y) - \sinh(x)\sinh(y)$	90	66	75	56
$\cos(x+y)\cos(x-y) = \cos(x)^2 - \sin(y)^2$	117	64	117	64
$\sin(x+y)\sin(y-x) = \cos(x)^2 - \cos(y)^2$	118	64	118	63
$\lvert\sinh(x/2)\rvert = \sqrt{(\cosh(x) - 1)/2}$	86	53	61	36
$\sin(x+y)\sin(x-y) = \sin(x)^2 - \sin(y)^2$	183	66	183	65
$\cosh(x)^2 = 1 + \cosh(2x)/2$	87	40	71	32
$\cosh(2x) = 2\cosh(x)^2 - 1$	78	42	62	33
$\cosh(2x) = \cosh(x)^2 + \sinh(x)^2$	97	72	80	64
$\tanh(x) - \tanh(y) = \sinh(x-y)/(\cosh(x)\cosh(y))$	154	135	85	81
$\tanh(x) + \tanh(y) = \sinh(x+y)/(\cosh(x)\cosh(y))$	162	144	95	91
$\sqrt{1 + \sinh(x)^2} = \cosh(x)$	82	70	76	62
$\sin(x)^3 = (3\sin(x) - \sin(3x))/4$	72	58	63	49
$\sin(3x) = 3\sin(x) - 4\sin(x)^3$	80	56	71	47
$\cosh(3x) = 4\cosh(x)^3 - 3\cosh(x)$	204	105	176	79
$\cosh(x)^3 = (3\cosh(x) + \cosh(3x))/4$	162	106	137	79
$\sin(4x) = \cos(x)(4\sin(x) - 8\sin(x)^3)$	73	73	60	60
$\cos(\pi + x) = -\cos(x)$	148	28	118	9
$\sin(\pi - x) = \sin(x)$	73	28	45	11
$\cos(\pi/3) = \sin(\pi/6)$	26	17	26	17
$\cos(\pi/4) = \sin(\pi/4)$	24	17	24	17
$\cos(\pi/6) = \sin(\pi/3)$	22	17	22	17
$\cos(2\pi + x) = \cos(x)$	125	70	37	18
$\sin(2\pi + x) = \sin(x)$	353	69	62	16

F Example Lean proofs

In this section, we show examples of proofs found by our model.

```
theorem imo_1964_p1_2 (n : ℕ) : ¬7|2^n + 1 :=
begin
  rw nat.dvd_iff_mod_eq_zero,
  rewrite [nat.add_mod, nat.mod_eq_of_lt],
  obviously,
  apply nat.strong_induction_on n,
  induction n,
  {
    intros n IH,
    cases n,
    norm_num,
    cases n,
    norm_num,
    rw [nat.succ_eq_add_one, pow_succ],
    rw [nat.succ_eq_add_one, pow_succ],
    induction n,
    norm_num,
    rw [nat.succ_eq_add_one, pow_succ],
    norm_num [nat.mul_mod, ←mul_assoc],
    contrapose! IH,
    refine ⟨n_n, nat.lt_succ_iff.mpr _, IH⟩,
    exact nat.le_succ_of_le (nat.le_succ _),
  },
  exact n_ih,
end
```

Figure 12: **A proof of the `imo_1964_p1_2` problem found by our model.** The model shows that for any value of $n \in \mathbb{N}$, $2^n + 1$ is not divisible by 7, by showing that $2^n \mod 7 + 1 \neq 0$, and $2^n \mod 7 + 1 < 7$. The second part of the proof uses strong induction and the fact that $2^n \equiv 2^{n+3} \mod 7$. We provide a version of the proof that was automatically cleaned by removing unnecessary tactics and tactic arguments.

```
theorem imo_2001_p6
  (a b c d : ℕ)
  (h₀ : 0 < a ∧ 0 < b ∧ 0 < c ∧ 0 < d)
  (h₁ : d < c)
  (h₂ : c < b)
  (h₃ : b < a)
  (h₄ : a * c + b * d = (b + d + a - c) * (b + d - a + c)) :
  ¬ nat.prime (a * b + c * d) :=
begin
  contrapose h₄,
  rw mul_comm,
  simp [nat.prime, not_le_of_gt h₀.1, not_forall, not_le_of_gt h₃,
        nat.mul_sub_right_distrib, nat.add_comm],
  contrapose! h₄,
  contrapose! h₄,
  apply has_lt.lt.ne,
  apply nat.lt_sub_right_of_add_lt,
  nlinarith,
end
```

Figure 13: **A proof found by our model of another IMO problem in miniF2F.** Although the proof is valid, the statement is erroneous. The hypothesis $h_4 : b + d - a + c$ actually represents $\max(b + d - a, 0) + c$. This is due to Lean's `nat` type behaviour where $(a : \mathbb{N}) - (b : \mathbb{N}) = (0 : \mathbb{N})$ if $b \geq a$. This makes the exercise easier than it should be, and the proof is no longer valid on the fixed statement.