AITP 2022

Seventh Conference on
Artificial Intelligence and Theorem Proving

Abstracts of the Talks

September 4–9, 2022, Aussois, France

Preface

This volume contains the abstracts of the talks presented at AITP 2022: Seventh Conference on Artificial Intelligence and Theorem Proving held September 4–9, 2022 in Aussois, France.

This year AITP has been co-located with a meeting of Working Group 5 of Cost Action European Research Network on Formal Proofs. We thank Frédéric Blanqui and the Cost Action CA20111 for covering the local organizer costs for the events, as well as supporting the travel and accommodation of 16 of the participants.

We are organizing AITP because we believe that large-scale semantic processing and strong computer assistance of mathematics and science is our inevitable future. New combinations of AI and reasoning methods and tools deployed over large mathematical and scientific corpora will be instrumental to this task. We hope that the AITP conference will become the forum for discussing how to get there as soon as possible, and the force driving the progress towards that. AITP 2022 consists of several sessions discussing connections between modern AI, ATP, ITP and (formal) mathematics. The sessions are discussion oriented and based on 30 contributed talks.

We would like to thank the CNRS conference center in Aussois for hosting AITP 2022. Many thanks also to Andrei Voronkov and his EasyChair for their support with paper reviewing and proceedings creation. The conference was partly funded from the European Research Council (ERC) under the EU-H2020 project SMART (no. 714034), and the Czech project AI&Reasoning CZ.02.1.01/0.0/0.0/15003/0000466 and the European Regional Development Fund. Finally, we are grateful to all the speakers, participants and PC members for their interest in discussing and pushing forward these exciting topics!

October 2022

Michael Douglas
Thomas Hales
Cezary Kaliszyk
Stephan Schulz
Josef Urban

Program Committee

Ulrich Furbach	University of Koblenz
Thibault Gauthier	Czech Technical University in Prague
Sean Holden	University of Cambridge
Mikoláš Janota	University of Lisbon
Michael Kinyon	University of Denver
Peter Koepke	University of Bonn
Michael Kohlhase	FAU Erlangen-Nürnberg
Konstantin Korovin	The University of Manchester
Ramana Kumar	DeepMind
Adam Pease	Articulate Software
Michael Rawson	TU Wien
Christian Szegedy	Google Research
Sean Welleck	University of Washington
Sarah Winkler	University of Bolzano
Zsolt Zombori	Alfréd Rényi Institute of Mathematics

Additional Reviewers

Jan Hula
Karol Pąk

Invited Talks

João Araújo
Forbidden Substructure Theorems

Johannes Brandstetter
How GNNs and Symmetries can help to solve PDEs

Kevin Buzzard
Formalizing Fermat

Walter Dean and Alberto Naibo
Mathematical difficulty, SAT solvers, and bounded arithmetic

Ben Goertzel
The Role of Automated Theorem-Proving in Neural-Symbolic Approaches to Artificial General Intelligence

Michael Rawson
ML4ATP: What I Wish I Had Known 5 Years Ago

Talia Ringer
Concrete Problems in Proof Automation

Stephen Wolfram
Theorem Proving in Metamathematics, the Universe and More

Table of Contents

Reinforcement Learning for Schedule Optimization*

Nikolai Antonov[13], Jan Hůla[12], Mikoláš Janota[1], and Přemysl Šůcha[1]

[1] Czech Technical University in Prague, Prague, Czech Republic
[2] University of Ostrava, Ostrava, Czech Republic
[3] Kazan Federal University, Kazan, Russia

Problem formulation and related work. In this paper we use machine learning to optimize a specific problem in integer linear algebra, which is practically motivated as job scheduling. Let us have a machine (system) capable of doing some work divided into a sequence of jobs. The machine follows three basic assumptions. First, it can do only one job a time. Secondly, a started job cannot be interrupted: a job must be completed before starting a next one. Third, the machine cannot idle: having finished one job, it immediately moves onto the next one until all the jobs assigned to the machine are finished. We are given a set of jobs $N = \{1, 2, ..., n\}$ with *processing times* p_i and *due dates* d_i for all $i \in N$. We assume that p_i and d_i are positive integers and $p_i \leq d_i$ for all $i \in N$. Additionally, each job has a *weight* (or *cost*), which is a positive integer w_i, $i \in N$ representing how valuable a particular job is. Assume that all the jobs are available form the very beginning (time moment 0) and executed one by one in the order specified by a permutation P of N. Let C_i^P denote the completion time of i-th job executed according to permutation P and define a set $S = \{i \in N \mid C_i^P \leq d_i\}$. The goal is to find a permutation P^* maximizing the weighted number of jobs that will be completed no later than the specified due date, i.e. maximize $f(P) = \sum_{i \in S} w_i$. We formulate the problem in satisfiability modulo theories (SMT). We want to find an integer vector

$$(s_1, s_2, ..., s_n) \geq \mathbf{0}$$

maximizing

$$\sum \sigma(C_i^P, d_i) w_i, \text{ where } \sigma(x, y) = 1 \text{ if } x \leq y \text{ else } 0$$

subject to

$$i < j \implies (s_i + p_i \leq s_j) \vee (s_j + p_j \leq s_i),\ i \in N,\ j \in N$$

The formulation does not explicitly specify that there must not be idling time, because at the post-processing stage any solution can be easily adjusted to eliminate any idling. In this work, we use reinforcement learning (RL) to solve this maximization problem, without guaranteeing optimality—approximate optima are also practically interesting. We remark that a decision version of the problem is obtained by bounding the objective function by some integer K.

The problem is proven to be NP-hard [4]—Knapsack is a special case when all jobs have the same due date. Due to its practical importance, the problem has been studied extensively in scheduling and OR communities: Potts and Van Wassenhove [6] gave a branch and bound algorithm for solving instances with up to 1,000 jobs; M'Hallah and Bulfin [5] propose an exact algorithm capable of handling instances with up to 2,500 jobs; Baptiste et al. [1] developed an algorithm solving up to 50000 jobs instances of particular type; Hejl et al. [3] investigated strongly-correlated instances and achieved a progress of solving 5000 jobs within one hour.

To the best of our knowledge, the considered problem was not studied in the ML community. However, number of combinatorial optimization problems tackled by reinforcement learning and other ML approaches is growing every year; we refer the reader to a survey by Bengio et al. [2].

*Results of this project No. LL1902 were supported by the Ministry of Education, Youth and Sports within the programme ERC CZ

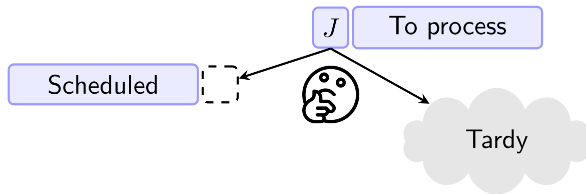


Figure 1: Decision-making of the RL agent for job J

Approach. We solve the considered problem using *deep reinforcement learning* [8, 7]. A sketch of the approach is provided in the Figure 1. Initially, all the jobs are sorted in ascending order by due date. At each moment of time, the agent observes a set of jobs that have not been completed yet and one of these jobs that the agent has to decide about (the unprocessed job with the earliest due date). The featurization of the unprocessed jobs is based on the distribution of their weights/proc. time/due date (represented as histograms). The agent has two possible actions: (1) schedule the first unprocessed job immediately and therefore it will be *on time*; (2) mark the job as *tardy* and move it to the end of schedule. One step of the agent corresponds to a decision regarding one job from a given instance. During the training phase, the agent receives a reward whenever the decision is right, i.e. if the job turned out to be the same (on time or late) as the agent predicted it to be in the optimal permutation that we have as a label. During the validation phase, the agent is only rewarded at the very end. The reward is equal to the ratio of the cost obtained by following the policy to the cost of the optimal solution.

Evaluation and conclusions. To make a fair comparison with actual results, we generate data according to the algorithm presented in [1] and [3]. Weights and durations are random integers from $[1; 100]$ and every due date is random integer from $[0.3S; 0.7S]$, where $S = \sum_{i \in N} p_i$. We compare to greedy heuristics MAX COST, MAX COST/DUR and MAX COST/DUE, which process the jobs in ascending order based on w_i, ratios $\frac{w_i}{p_i}$, and $\frac{w_i}{d_i}$, respectively. Terms $\mu(n)$ and $\sigma(n)$ stand for mean and standard deviation of optimality gap, obtained on the instances with n jobs. An optimality gap is defined as $\frac{v^* - v}{v^*}$, where v^* is the optimal value of the instance and v is the cost obtained by following the policy. The results show that our approach achieves much lower optimality gap than any of the greedy-heuristic approaches. This indicates that reinforcement learning is a viable approach to optimization of the linear integer algebra problems studied in this paper. We believe that this work would inspire further research on the use of reinforcement learning on more general problems or on probabilistic decision procedures.

	$\mu(100)$	$\sigma(100)$	$\mu(250)$	$\sigma(250)$
MAX COST	0.14	0.03	0.14	0.02
MAX COST/DUR	0.09	0.02	0.09	0.01
MAX COST/DUE	0.09	0.02	0.09	0.01
DRL model	0.065	0.027	0.015	0.01

	$\mu(500)$	$\sigma(500)$	$\mu(1000)$	$\sigma(1000)$
MAX COST	0.14	0.01	0.14	0.01
MAX COST/DUR	0.08	0.01	0.08	0.01
MAX COST/DUE	0.09	0.01	0.09	0.01
DRL model	0.017	0.01	0.009	0.006

References

[1] Philippe Baptiste, Federico Della Croce, Andrea Grosso, and Vincent T'kindt. Sequencing a single machine with due dates and deadlines: an ILP-based approach to solve very large instances. *J. Sched.*, 13(1):39–47, 2010.

[2] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: A methodological tour d'horizon. *Eur. J. Oper. Res.*, 290(2):405–421, 2021.

[3] Lukás Hejl, Premysl Sucha, Antonín Novák, and Zdenek Hanzálek. Minimizing the weighted number of tardy jobs on a single machine: Strongly correlated instances. *Eur. J. Oper. Res.*, 298(2):413–424, 2022.

[4] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

[5] Rym M'Hallah and R. L. Bulfin. Minimizing the weighted number of tardy jobs on a single machine. *Eur. J. Oper. Res.*, 145(1):45–56, 2003.

[6] Chris N. Potts and Luk N. Van Wassenhove. A branch and bound algorithm for the total weighted tardiness problem. *Oper. Res.*, 33(2):363–377, 1985.

[7] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[8] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning.* Adaptive Computation and Machine Learning series. Bradford Books, Cambridge, MA, 2 edition, November 2018.

Proving theorems using Incremental Learning and Hindsight Experience Replay

1 Introduction

The highest performing ATP systems (e.g., [7, 18]) in first order logic have been evolving for decades and have grown to use an increasing number of manually designed heuristics mixed with some machine learning, to obtain a large number of search strategies that are tried sequentially or in parallel. Some recent works [5, 13, 19] build on top of these provers, using modern machine learning techniques to augment, select or prioritize their already existing heuristics, with some success. Other recent works do not build on top of other provers, but still require existing proof examples as input (e.g., [9, 23]). Such machine-learning-based ATP systems can struggle to solve difficult problems when the training dataset does not provide problems of sufficiently diverse difficulties.

In this paper, we propose an approach which can build a strong theorem prover without relying on existing domain-specific heuristics or on prior input data (in the form of proofs) to prime the learning. We strive to design a learning methodology for ATP that allows a system to improve even when there are large gaps in the difficulty of given set of theorems. In particular, given a set of conjectures without proofs, our system trains itself, based on its own attempts and (dis)proves an increasing number of conjectures, an approach which can be viewed as a form of incremental learning. Additionally, all the previous approaches [19, 1, 13] learn exclusively on *successful* proof attempts. When no new theorem can be proven, the learner may not be able to improve anymore and thus the system may not be able to obtain more training data. This could in principle happen even at the very start of training, if all the theorems available are too hard. To tackle this challenge, we adapt the idea of hindsight experience replay (HER) [3] to ATP: Clauses reached during proof attempts (whether successful or not) are turned into goals in hindsight, producing a large amount of 'auxiliary' theorems with proofs of varied difficulties for the learner, even in principle when no theorem from the original set can be proven initially. This leads to a smoother learning regime and a constantly improving learner.

We evaluate our approach on two popular benchmarks: MPTP2078 [2] and M2k [17] and compare it both with TRAIL [1], a recent machine learning prover as well as with E prover [24, 7], one of the leading heuristic provers. Our proposed approach substantially outperforms TRAIL [1] on both datasets, surpasses E in the *auto* configuration with a 100s time limit, and is competitive with E in the *autoschedule* configuration with a 7 days time limit. In addition, our approach almost always (99.5% of cases) finds shorter proofs than E.

2 Methodology

We describe the two key components of our approach: how we adapt hindsight experience replay in an incremental learning pipeline, and how clauses are represented for the learner.

Incremental Learning and Hindsight Experience Replay Similar to previous approaches [19, 13, 1], we use the given clause algorithm [21] where the clause scoring heuristics are replaced with a neural network. We start with no proof data to start, and train a simple binary classifier to determine if a particular clause appears in a proof of a conjecture or not. The classifier is

Table 1: Number of conjectures proven on MPTP2078 and M2k.

Domain	Conjectures	Heuristic Approaches				ML Approaches		
		E-basic (100s)	E-auto (100s)	E auto-schedule (100s)	E-best (7 days)	TRAIL	IL w/o HER	IL w/HER
MPTP2078	2078	555	1139	1289	**1369**	1213	1056	**1353**
M2k	2003	1451	1845	**1911**	**1923**	1808	1688	**1861**

trained in an incremental manner where the new proof data obtained by the proof attempts is used to feed the classifier in a continuous manner. The key issue in such an approach arises if the complete set of conjectures are either very difficult or there are big gaps in difficulty of given conjectures such that no training data can be generated by proof attempts to train the classifier. To counter this, we adapt the idea of hindsight experience replay in ATP where any proof attempt whether successful or failure would generate new data for classifier. The core idea of HER is to take any "unsuccessful" trajectory in a goal-based task and convert it into a successful one by treating the final state that happened to be reached as if it were the goal state, in hindsight. Inspired by HER, we use the clauses generated during *any* proof attempt as additional conjectures, which we call *hindsight goals*, leading to a supply of positive and negative examples. Let D be any non-input clause generated during the refutation attempt of C_s. We call D a *hindsight goal*.[1] Then, the set $C_s \cup \{\neg D\}$ can be refuted. Further, we can use the ancestors of D as positive examples for the negated conjecture and axioms $C_s \cup \{\neg D\}$. This generates a very large number of examples, allowing us to effectively train the neural network, even with only a few conjectures at hand.

Representation Our clause scoring network receives as input the clause to score, x, the hindsight goal clause, g, and a sequence of negated conjecture clauses C_s. Individual clauses are transformed into a heterogeneous directed acyclic graphs, called *clause graph* similar to [4]. We use a Transformer encoder architecture [25] for the clause-scoring network, whose input is composed of the set of node embeddings in the current clause x, goal clause g and conjecture clauses C_s, up to 128 nodes. For each node, we compute a spectral encoding vector representing its position in the clause graph [8]; this is given by the eigenvectors of the Laplacian matrix of the graph. This replaces the traditional positional encoding in transformers.

3 Experiments and Results

We implement our approach on top of E prover but disable all clause scoring heuristics of E. We use a maximum time of 100s for each proof attempt. We evaluate on two popular benchmarks: MPTP2078[2] and M2k[17] which are widely used in literature. Further, we compare our results with E in different configurations as well as incremental learning without hindsight (IL w/o HER) and TRAIL[1], a recent ML based prover. Table 1 shows the number of proved conjectures by all provers. IL w/HER not only outperforms TRAIL, IL w/o HER and E (100s) but achieves a competing performance when E is run for the *whole* duration of training time. We refer the reader to the appendix for further details of methodology, experimental setup and additional results.

[1]Note that, while the original version of HER [3] only uses the last reached state as a single hindsight goal, we use all intermediate clauses, providing many more data points.

References

[1] I. Abdelaziz, M. Crouse, B. Makni, V. Austel, C. Cornelio, S. Ikbal, P. Kapanipathi, N. Makondo, K. Srinivas, M. Witbrock, and A. Fokoue. Learning to guide a saturation-based theorem prover. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, 2022.

[2] Jesse Alama, Tom Heskes, Daniel Kühlwein, Evgeni Tsivtsivadze, and Josef Urban. Premise selection for mathematics by corpus analysis and kernel methods. *Journal of Automated Reasoning*, 52(2):191–213, 2014.

[3] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[4] Eser Aygün, Zafarali Ahmed, Ankit Anand, Vlad Firoiu, Xavier Glorot, Laurent Orseau, Doina Precup, and Shibl Mourad. Learning to prove from synthetic theorems. *arXiv preprint arXiv:2006.11259*, 2020.

[5] Karel Chvalovskỳ, Jan Jakubův, Martin Suda, and Josef Urban. Enigma-ng: efficient neural and gradient-boosted inference guidance for e. In *International Conference on Automated Deduction*, pages 197–215. Springer, 2019.

[6] 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. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(7):6279–6287, 2021.

[7] Simon Cruanes, Stephan Schulz, and Petar Vukmirović. Faster, Higher, Stronger: E 2.3. In *TACAS 2019*, volume 11716 of *LNAI*, pages 495–507, April 2019.

[8] Vijay Prakash Dwivedi and Xavier Bresson. A generalization of transformer networks to graphs. *CoRR*, abs/2012.09699, 2020.

[9] Zarathustra Amadeus Goertzel. Make E smart again (short paper). In *Automated Reasoning*, pages 408–415. Springer International Publishing, 2020.

[10] Adam Grabowski, Artur Korniłowicz, and Adam Naumowicz. Four decades of mizar. *Journal of Automated Reasoning*, 55(3):191–198, 2015.

[11] Malte Helmert, Tor Lattimore, Levi H. S. Lelis, Laurent Orseau, and Nathan R. Sturtevant. Iterative budgeted exponential search. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, page 1249–1257. AAAI Press, 2019.

[12] S. Jabbari Arfaee, S. Zilles, and R. C. Holte. Learning heuristic functions for large state spaces. *Artificial Intelligence*, 175(16-17):2075–2098, 2011.

[13] Jan Jakubův, Karel Chvalovskỳ, Miroslav Olšák, Bartosz Piotrowski, Martin Suda, and Josef Urban. Enigma anonymous: Symbol-independent inference guiding machine (system description). In *International Joint Conference on Automated Reasoning*, pages 448–463. Springer, 2020.

[14] Jan Jakubův and Josef Urban. Enigma: efficient learning-based inference guiding machine. In *International Conference on Intelligent Computer Mathematics*, pages 292–302. Springer, 2017.

[15] Jan Jakubův and Josef Urban. Hammering Mizar by Learning Clause Guidance (Short Paper). In *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141, pages 34:1–34:8. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.

[16] Cezary Kaliszyk and Josef Urban. Learning-assisted theorem proving with millions of lemmas. *Journal of symbolic computation*, 69:109–128, 2015.

[17] Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Miroslav Olšák. Reinforcement learning of theorem proving. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[18] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In *International Conference on Computer Aided Verification*, pages 1–35. Springer, 2013.

[19] Sarah Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. Deep network guided proof search. In *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 46 of *EPiC Series in Computing*, pages 85–105, 2017.

[20] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of Las Vegas algorithms. *Inf. Process. Lett.*, 47(4):173–180, September 1993.

[21] William McCune and Larry Wos. Otter - the CADE-13 competition incarnations. *Journal of Automated Reasoning*, 18(2):211–220, 1997.

[22] Laurent Orseau and Levi H. S. Lelis. Policy-guided heuristic search with guarantees. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(14):12382–12390, May 2021.

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

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

[25] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

Appendix

Methodology

we describe the basic search algorithm used by most of the traditional first-order automated theorem provers, explain how we integrate our method into one of these provers and finally, provide a detailed description of our overall incremental learning system.

Given-clause algorithm. Almost all of the powerful automated theorem provers for first-order logic, including E, use some variation of a *given-clause* search algorithm [18, 7, 21]. This type of algorithm works by continuously choosing a new *given clause* to expand, with the help of one or more priority queues, until an empty clause (*i.e.* contradiction) is reached. The given clause is combined according to various logical operations (like resolution, factoring, etc.;) with previously chosen *active clauses* to generate more clauses, which are consequently added to the priority queues. Each priority queue depends on a scoring function for sorting the clauses. At every step, a queue is selected based on a schedule, which usually consists of a simple cycle through all queues and each queue occurs for a fixed number of pre-determined steps within in each cycle. For example, the simplest schedule could be round robin sampling of all queues where each cycle consists of a single occurrence of each queue.

The two most basic types of queues are *the FIFO queue* and *the clause weight queue*. The former keeps the clauses sorted from oldest to youngest, guaranteeing that every clause will be visited after some finite amount of time. The latter uses a simple linear function that combines the numbers of various elements in the clauses (such as literals, atoms, variables) to obtain a "weight" and sorts the clauses from lightest to heaviest. The idea is to prioritize lighter or smaller clauses which, empirically, helps in reaching the empty clause faster.

Using machine learning to improve provers that depend on the given-clause algorithm. There are many ways to incorporate machine learning into a prover that is based on the given-clause algorithm. One option is to replace the queues with a policy over clauses that has full control over the search [6, 1]. Another option is to train a clause scoring function which merely provides an additional queue that can be added to any existing set of queues [19, 5].

Integrating our method into E. We take the latter approach in this work. We train a classifier that predicts the probability of a clause appearing in the proof given a set of initial clauses and use the predictions of this classifier to construct a "learned queue". We integrate this queue into the popular open-source first-order prover E using remote procedure calls (in a fashion similar to Enigma [14]). This allows us to take advantage of the sophisticated logic engine in E.

E, however, is more than its logic engine. It comes preloaded with hundreds of thousands of lines of code for heuristics (optimized for certain datasets) which help E pick the right set of queues with the right set of ratios for the given problem. As our goal is to replace these complicated heuristics with a single machine learning system, when we evaluate our method, we use a simple, fixed queue structure: a FIFO queue for completeness, a basic clause weight queue for greedy search and a 'learned' queue for guided search.

Clause-scoring and hindsight experience replay

In order to perform clause-scoring, we use deep neural networks, which can be trained in many ways so as to find proofs faster. A method utilized by [19] and [15] turns the scoring task into a classification task: a network is trained to predict whether the clause to be scored will appear in the proof or not. In other words, the probability predicted by an 'in-proofness' classifier is used as the score. To train, once a proof is found, the clauses that participate in the proof (i.e., the ancestors of the empty clause) are considered to be positive examples, while all other generated clauses are taken as negative examples.[2] Then, given as input one such generated clause x along with the input clauses C_s, the network must learn to predict whether x is part of the (found) proof.

There are two main drawbacks to this approach. First, if all conjectures are too hard for the initially unoptimized prover, no proof is found and no positive examples are available, making supervised learning impossible. Second, since proofs are often small (often a few dozen steps), only few positive examples are generated. As the number of available conjectures is often small too, there is far too little data to train a modern high-capacity neural network. Moreover, for supervised learning to be successful, the conjectures that can be proven must be sufficiently diverse, so the learner can steadily improve. Unfortunately, there is no guarantee that such a curriculum is available. If the difficulty suddenly jumps, the learner may be unable to improve further. These shortcomings arise because the learner only uses successful proofs, and all the unsuccessful proof attempts are discarded. In particular, the overwhelming majority of the generated clauses become negative examples, and need to be discarded to maintain a good balance with the positive examples.

To leverage the data generated in unsuccessful proof attempts, we adapt the concept of hindsight experience replay (HER) [3] from goal-conditioned reinforcement learning to theorem proving. The core idea of HER is to take any "unsuccessful" trajectory in a goal-based task and convert it into a successful one by treating the final state that happened to be reached as if it were the goal state, in hindsight. A deep network is then trained with this trajectory, by contextualizing the policy with this state instead of the original goal. This way, even in the absence of positive feedback, the network is still able to adapt to the *dataset*, if not to the goal, thus having a better chance to reach the goal on future tries.

Inspired by HER, we use the clauses generated during *any* proof attempt as additional conjectures, which we call *hindsight goals*, leading to a supply of positive and negative examples. Let D be any non-input clause generated during the refutation attempt of C_s. We call D a *hindsight goal*.[3] Then, the set $C_s \cup \{\neg D\}$ can be refuted. Furthermore, once the prover reaches D starting from $C_s \cup \{\neg D\}$, only a few more resolution steps are necessary to reach the empty clause; that is, there exists a refutation proof of $C_s \cup \{\neg D\}$ where D is an ancestor of the empty clause. Hence, we can use the ancestors of D as positive examples for the negated conjecture and axioms $C_s \cup \{\neg D\}$. This generates a very large number of examples, allowing us to effectively train the neural network, even with only a few conjectures at hand.

Furthermore, to keep the network small, axioms are not provided as input to the scoring network Although the set of active clauses is an important factor in determining the usefulness of a clause, we ignore it in the network input to keep the network size smaller.

[2] These examples are technically not necessarily negative, as they may be part of another proof. But avoiding these examples during the search still helps the system to attribute more significance to the positive examples.

[3] Note that, while the original version of HER [3] only uses the last reached state as a single hindsight goal, we use all intermediate clauses, providing many more data points.

Algorithm 1 Distributed incremental learning. `launch` starts a new process in parallel. For each conjecture an instance of UBS decides the sequence of time limits for solving attempts.

```
def main(conjectures):
 # Launch and connect learners, actors and manager with example buffer &  task queue
  example_buffer = create_example_buffer()
  task_queue = create_task_queue()
  learners = [for i = 1..10:
      launch learner(example_buffer)]
  for i = 1..1000: launch actor(task_queue,
      learners, example_buffer)
  actor_manager = launch actor_manager(conjectures, task_queue)
  wait for actor_manager to finish

def learner(example_buffer):
  repeat forever:
    # Sample a batch of examples and train the network.
    batch = sample_batch_uniformly(example_buffer)
    minimize_classification_loss(batch)  # we use cross-entropy

def actor(task_queue, learners, example_buffer)
  repeat forever:
    # Fetch a task and attempt to prove the conjecture.
    conjecture, time_limit = get_task(task_queue)
    learner = sample_uniformly(learners)
    run E on conjecture
        for at most time_limit seconds;
        obtain generated_clauses
    examples = sample_examples(generated_clauses) # see Alg. (*\ref{alg:sample_examples
        }*)
    put_examples(example_buffer, examples)

def actor_manager(conjectures, task_queue):
  schedulers = []
  for conjecture in conjectures:
    schedulers[conjecture] = initialize_UBS() # see Section (*\ref{sec:ubs}*)
  repeat until all conjectures have been proven:
    # Choose a random conjecture and enqueue it.
    conjecture = sample_uniformly(conjectures)
    scheduler = schedulers[conjecture]
    time_limit = get_next_time_limit(scheduler)
    put_task(task_queue, (conjecture, time_limit))
```

Incremental learning algorithm

Typical supervised learning ATP systems require a set of proofs (provided by other provers) to optimize their model (e.g., [19, 13, 4]). Success is assessed by cross-validation. In contrast, we formulate ATP as an incremental learning problem—see in particular [22, 12]. Given a pool of unproven conjectures, the objective is to prove as many as possible, even using multiple attempts, and ideally as quickly as possible. Hence, the learning system must bootstrap directly from initially-unproven conjectures, without any initial supervised training data. Success is

Algorithm 2 Example sampling algorithm.

```
def sample_examples(generated_clauses):
  # Estimate the number of examples that can be consumed by the learner
  target_num_examples =
    time_elapsed_since_last_attempt $\times$ target_num_examples_per_second

  # Remove the input clauses
  hindsight_goals =
    generated_clauses \ input_clauses

  # Subsample the goals and the examples
  examples = []
  sizes = {tree_size(c) : c $\in$ hindsight_goals}
  for size in sizes:
    size_goals = {c $\in$ hindsight_goals :
        tree_size(c) == size}
    w_size = 1 / ln(size + e) - 1 / ln(size + e + 1)
    num_examples = ceil(target_num_examples $\times$ w_size)
    for _ in range(num_examples):
      goal = uniform_sample(size_goals) # pick hindsight goal of this size
      anc = ancestors(goal)
      examples += [positive_example(uniform_sample(anc), goal)]
      examples += [negative_example(uniform_sample(hindsight_goals \ anc), goal)]
  return examples
```

assessed by the number of proven conjectures, and the time spent solving them. Hence, we do not need to split the set of conjectures into train/test/validate sets because, if the system overfits to the proofs of a subset of conjectures, it will not be able to prove more conjectures.

Our incremental learning system is described in Algorithm 1. Initially, all conjectures are unproven and the clause-scoring network is initialized randomly. At this stage, we have no information on how long it takes to prove a certain conjecture, or whether it can be proven at all. The prover attempts to prove all conjectures provided using a scheduler (described below), so as to vary time limits for each conjecture. This ensures that proofs for easy conjectures are obtained early, and the resulting positive and negative examples are then used to train the clause-scoring network. As the network learns, more conjectures can be proven, providing in turn more data, and so on. This incremental learning algorithm thus allows us to automatically build a capable prover for a given domain, starting from a basic prover that may not even be able to prove a single conjecture in the given set.

Time scheduling. All conjectures are attempted in parallel, each on a CPU. For each conjecture, we use the uniform budgeted scheduler (UBS) algorithm [11, section 7] to further simulate running in (pseudo-)parallel the solver with varying time budgets, and restarting each time the budget is exhausted. In the terminology of UBS, we take $T(k, r) = 3r2^{k-1}$ in seconds, but we cap $k \leq k_{\max} = 10$. A UBS instance simulates on a single CPU running $k_{\max}$ restarting programs, by interleaving them: On a 'virtual' CPU of index $k \in \{1, \ldots, k_{\max}\}$, a program corresponds to running the prover for a budget of $3 \cdot 2^{k-1}$ seconds before restarting it for the same budget of time and so on; r is the number of restarts. Hence, as the network learns, each conjecture is incrementally attempted with time budgets of varying sizes (3s, 6s, 12s, ..., 3072s), using no more than one hour, while carefully balancing the cumulative time spent within each budget [20, 11]. Once a proof has been found for a conjecture, the scheduler is not stopped, so as to continue searching for more (often shorter) proofs.

Distributed implementation. Our implementation consists of multiple actors running in parallel, a manager that distributes tasks to the actors using the time scheduling algorithm, and a task queue that handles manager-actors communication. We used ten learners training ten separate models to increase the diversity of the search without having to increase the number of actors. These learners are fed with training examples from the actors and use them to update their parameters of their clause-scoring networks. Note that during the first 1 000 updates, the actors do not use the clause-scoring network as its outputs are mostly random.[4]

Subsampling hindsight goals and examples. With HER, the number of available examples is actually far too large: if, after a proof attempt, n clauses have been generated (n may be in the thousands), not only can each clause be used as a hindsight goal, but there are about n^2 pairs of the form (positive example, hindsight goal), and far more negative examples. This suddenly puts us in a very data-rich regime, which contrasts with the data scarcity of learning only from complete proofs of the given conjecture. Hence, we need to *subsample* the examples in order to prevent overwhelming the learner. To this end, we first estimate the number of examples the learner can consume per second before sampling. But there is an additional difficulty: the number of possible clauses is exponentially large in the `tree_size` (number of nodes in the clause tree) of the clause, while small clauses are likely more relevant since the empty clause (which is the true target) has size 0. Moreover, clauses can be rather large: a `tree_size` over 300 is quite common, and we observed some `tree_size` values over 6 000. To correct for this, we fix the proportion of positive and negative examples for each hindsight goal clause size, ensuring that small hindsight goal clauses are favoured, while allowing a diverse sample of large clauses, using a heavy-tail distribution w_s. Finally, all the positive and negative examples thus sampled are added to the training pool for the learners.

Representation

Our clause scoring network receives as input the clause to score, x, the hindsight goal clause, g, and a sequence of negated conjecture clauses C_s. Individual clauses are transformed into directed acyclic graphs (an example is depicted in Figure 1) with five different node types : clause, literal, atomic-term, variable-term or variable. First, there is a clause node, whose children are literal nodes, corresponding to all literals of the clause (each one is associated with a predicate). The children of literal nodes represent the arguments of the predicate; they are either variable-term nodes if the argument is a variable, or atomic-term nodes otherwise[5]. Children of atomic-term nodes follow the same description. Finally, each variable-term node is linked to a variable node, which has as many parents as there are instances of the corresponding variable in the clause.

To each node, we associate a feature vector composed of the following five components: (i) A one-hot vector of length 3, encoding if the node belongs to x, g or a member of C_s. (ii) A one-hot vector of length 5 encoding the node type: clause, literal, atomic-term, variable-term or variable. (iii) A one-hot vector of length 2 encoding if the node belongs to a positive or negative literal (null vector for clause and variable nodes). (iv) A hash vector representing the predicate name or the function/constant name respectively for predicate or atomic-term nodes (null vector for other nodes). (v) A hash vector representing the predicate/function argument slot in which the term is present (null vector for clause, literal and variable nodes). Hash vectors

[4] We picked 1000 as it appeared to be approximately the number of steps required for the learner to reach the base prover performance on a few experiments.

[5] A constant argument is equivalent with a function of arity 0.

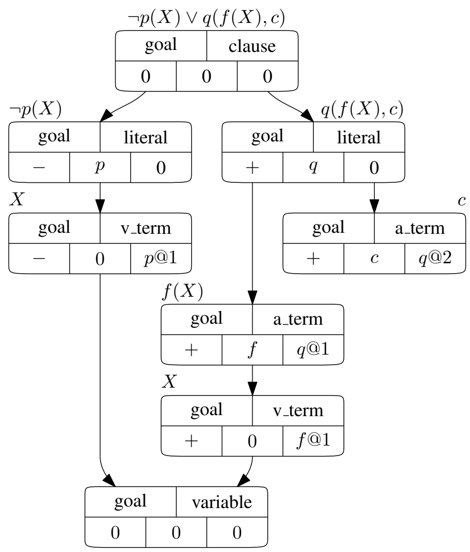


Figure 1: Clause graph of a goal clause. Each node has five features: clause type, node type, literal polarity, symbol hash and argument slot hash. The parts of formula corresponding to each node are shown outside of the nodes.

are randomly sampled uniformly on the 64 dimensional unit hyper-sphere, using the name of the predicate, function or constant (and the argument position for slots) as seed.

The node feature vectors are projected into a 64-dimensional node embedding space using a linear layer that trains during learning. We use a Transformer encoder architecture [25] for the clause-scoring network, whose input is composed of the set of node embeddings in the current clause x, goal clause g and conjecture clauses C_s, up to 128 nodes. For each node, we compute a spectral encoding vector representing its position in the clause graph [8]; this is given by the eigenvectors of the Laplacian matrix of the graph from which we keep only the 64 first dimensions, corresponding to the low frequency components. It replaces the traditional positional encoding of Transformers. Note that if there are more than 128 nodes in the set of clause graphs, we prioritize x, then g and C_s. Within each graph, we order the nodes from top to bottom then left to right (e.g. the first nodes to be filtered out would be variable- or atomic-term nodes of the last conjecture clause). We only keep the transformer encoder output corresponding to the root node of the target clause and project it, using a linear layer, into a single logit, representing the probability that x will be used to reach g starting from C_s.

Additional Results

To evaluate our approach, we use two popular benchmarks created out of the larger Mizar Mathematical Library [10] and used in previous works [6, 17, 16]: MPTP2078 [2] is a sample of larger Mizar datasets which is a good mixture of hard and easy theorems; M2k [17] is a relatively easier benchmark which contains theorems that have already been proven by at least one of the automated theorem provers in the past. The relative hardness of these datasets is also illustrated by the fact that the state-of-the-art E prover proves less than 70% theorems in MPTP2078 while it achieves proof rate greater than 95% on M2k theorems. We ignore five problems in MPTP2078 and 13 problems in M2k due to E failing to generate a conjunctive

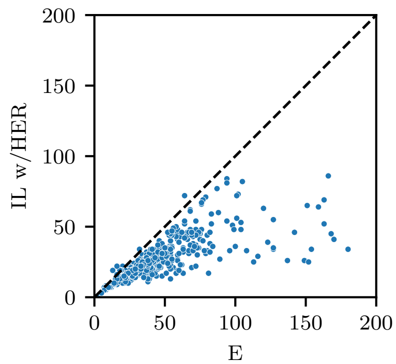


Figure 2: Scatter plot of the shortest proof lengths achieved by E vs. incremental learning with hindsight experience replay on the conjectures that can be proven by both.

normal form (first step in proving) of the problem.

We evaluate and compare our approach with both machine learning and heuristic based approaches on these two datasets. We compare our approach with E, considered a state-of-the-art heuristic based prover, in four configurations: (i) E in its default mode (without any sophisticated heuristics and scheduling) for 100s (referred to as E-basic), (ii) E in *auto* mode for 100s (the mode that was used by [6] and [1] [6]), (iii) E in *auto-schedule* mode for 100s (we observed that the auto-schedule mode significantly outperforms the auto mode), (iv) the best of different runs of E in auto-schedule mode with time limits of 100s, 1 hour, 1 day and 7 days (referred to as E-best). We used E prover version 2.5 [7] in each of these configurations with a memory limit of 8192 GB.

We ran our incremental learning algorithm with hindsight experience replay (IL w/HER) for seven days on each dataset, using 1000 actors where each attempt was allowed a maximum duration of 100s. Every successful attempt that leads to a proof during training is logged, along with the time elapsed, the number of clauses generated, the length of the proof, and the proof itself. In order to show the importance of HER in achieving the results above, we ran the same experiments with incremental learning but without HER (IL w/o HER), by training the clause-scoring network using solely the data extracted from proofs found for the input problems. As another point of comparison, we include the results of TRAIL, which is a top performing learning method built on top of E prover, as reported in [1]. Like our approach, TRAIL does not rely on E's heuristics and does not use additional input data from which to bootstrap, so it is directly comparable. [1] also reported numbers for other learning provers that are similar in spirit, but since their performance is inferior to TRAIL, we do not include their reported numbers. We note that there are other machine-learning based theorem provers, such as ENIGMA [13] and its variants, and [19]; but these provers rely heavily either on E's heuristics on or input proof data to bootstrap from, and thus fall in a different category from ours, where the machine learning system based on a basic prover should bootstrap on its own.

Conjectures proven. Table 1 shows the number of conjectures proven by each of these approaches as well as the actual number of conjectures in each dataset. According to these results, IL w/HER significantly outperforms TRAIL on both datasets. Interestingly, since using HER is orthogonal to the methods used by TRAIL, one could hope that combining both

[6]The exact results reported by [1] for E prover are significantly lower than what we obtained in our experiment. This could be attributed to a difference in the version of E prover, memory allocated or processor speed—the exact configuration details are not reported in their paper.

Table 2: Problems uniquely solved by one method but not the other (E-best or IL w/HER) on both datasets.

Domain	Only E-best	Only IL w/HER
MPTP2078	94	78
M2k	79	17

approaches could lead to even better results—but we leave this as future work. IL w/HER proved 2.5 times as many problems as the E-basic on MPTP2078 and 1.28 times as many as E basic on M2k, improving its performance substantially through the use of a learned clause-scoring network. IL w/HER also outperforms E-auto as well as E auto-schedule on the MPTP2078 dataset. We do not see a similar improvement on the M2k dataset. This can be due to the fact that M2k is a subset of theorems already proven by ATPs and hence, by construction, it consists of the sub-sample of theorems on which E already performs well. Lastly, as IL w/HER ran for seven days, attempting each conjecture multiple times (though each attempt was allowed a maximum of 100s) , in order to give E a fair chance, we also ran E for multiple time durations (100s, 1h, 1d, 7d) and we report the maximum number of conjectures proved in of all these runs as E-best. Our approach comes very close (less than 1% difference) to the performance of E-best on the MPTP2078 dataset.

Unique theorems proved by our approach. Additionally, Table 2 shows the number of theorems proven only by our approach and not E-best, and the other way around.. IL w/HER manages to prove 78 theorems on MPTP-2078 and 17 theorems on M2k which are not proven by E-best. This suggests that IL w/HER can find strategies that are absent from E.

Table 3: Comparison of different neural network architectures in IL w/HER on MPTP-2078 and M2k.

Domain	Conjectures	MLP	GNN	Sequential transformer	Spectral transformer
MPTP2078	2078	1049	1221	1076	**1353**
M2k	2003	1772	1756	1704	**1861**

Without hindsight. In order to evaluate specifically the impact of using HER, we also report the performance of incremental learning alone which does not use any data from unsuccessful proof attempts. As seen in Table 1, IL w/o HER performed significantly worse, failing to prove 297 (14.3%) of the conjectures on MPTP2078 and 173(8.6%) conjectures on M2k that can be proven by IL w/HER. Without enough proofs of hard theorems from which to learn, IL w/o HER underperformed significantly on these domains compared to IL w/HER.

Quality of proofs. We also looked at the individual proofs discovered by both systems. Incremental learning combined with the revisiting of previously proven conjectures allowed our system to discover shorter proofs continually. Figure 2 shows a scatter plot of the lengths of the shortest proofs found by E vs. found by IL w/HER for each theorem. The shortest proofs found by our system were consistently shorter than those found by E. Out of the 3119 conjectures proven by both systems, our proofs were shorter for 3106 conjectures (99.5%) whereas E's proofs were shorter for only 8 conjectures, with 5 proofs being of the same length.

Speed of search. E was able to search 13.6 times faster than our provers, in terms of clauses generated per second. We believe that the only way for our system to compete with E under these conditions is to find scoring functions that are much stronger than the numerous heuristics that have been built into E over time.

Comparison between different representations: In order to understand the impact of the choice of network architecture on the results, we compared different neural networks trained with the proposed approach. We compared the spectral transformer representation described in Sec. 2.3 with MLP (based on manually defined features), Graph Neural Networks (GNNs) and a sequential text-based representation of the logical formulae which is used in a standard sequential transformer. For GNNs, we used the same graph structure as the spectral transformer described in Sec. 3. An additional root node is added at the top to connect the target clause with the negated conjecture clauses, in order to allow message passing between different clauses. Table 3 shows the conjectures solved by using different representations trained with IL w/HER using 1000 actors. We observe that GNNs outperform MLPs but fall short of the spectral transformer in our implementation on the MPTP2078 dataset. It should be noted that there are multiple ways to represent logical formulae as graphs, but we confine ourselves within the representation which is closest to spectral transformers. A detailed investigation of other graph representations proposed in the literature in combination with IL w/HER is left for future work. Also, we observe that spectral transformers outperform sequential transformers significantly in all our experiments. This can be attributed to the fact that spectral transformers capture graphical structure, and hence exploit logical invariances in formulae, in contrast to sequential transformers which treat these formulae as text.

Project Proposal: Efficient Neural Clause Selection by Weight*

Filip Bártek and Martin Suda

Czech Technical University in Prague
filip.bartek@cvut.cz

1 Introduction

Saturation-based automated theorem provers (ATPs) that use the given clause algorithm [1] maintain two sets of clauses during the proof search: processed and unprocessed. In each iteration of the saturation loop, the prover selects a clause, called the given clause by convention, from the unprocessed set. All possible inferences are then made between the given clause and the processed clauses. The newly inferred clauses are added to the unprocessed set, while the given clause is moved from the unprocessed set to the processed set. When the proof search successfully concludes (by inferring the empty clause, a trivial contradiction), a subset of processed clauses constitutes the proof.

The basic clause selection heuristics choose the clause that is the oldest according to the time it was inferred, or the smallest in the symbol count, that is, the number of symbol occurrences [11]. The symbol count heuristic can be generalized by assigning each symbol a weight (a positive real number); the clause weight is then calculated by summing the weights of the symbol occurrences. In addition to that, clauses can be penalized, for example, for each positive literal, maximal literal, or unorientable equation [10].

The goal of this project is to improve the standard weight-based clause selection heuristic by machine learning from proof searches. Specifically, our aim is to find problem-specific values for the coefficients of the clause weight function (that is, the weights of symbols, variables, maximal literals, unorientable equations, etc.). These parameter values are to be output from a neural network (NN) trained on successful proof searches.

In section 2 we describe in detail our proposed solution in its basic form – a system that automatically configures the parameters of the clause weight function in a problem-specific fashion. In section 3 we outline possible generalizations and modifications.

2 GNN-based clause selection

Prediction When a new problem is to be solved, the problem is processed with a graph neural network (GNN) [13, 8, 4, 9]. The GNN produces a weight w_f for each symbol f and a common weight w_X for all variables. The weights are then used to instantiate a weight-based clause selection heuristic: The weight of a clause is computed as the sum of the weights of all the symbol and variable occurrences in the clause.

Note that our GNN only needs to be evaluated once at the beginning of the proof search: Once the symbol and variable weights are calculated, they can be used throughout the proof search without any additional input from the GNN. This can be interpreted as a conservative

*This work is supported by the Czech Science Foundation project no. 20-06390Y (JUNIOR grant), and the Grant Agency of the Czech Technical University in Prague, grant no. SGS20/215/OHK3/3T/37.

modification of the standard clause selection heuristic: We introduce neural guidance into the proof search without increasing the cost of computing the weight of a newly inferred clause. This contrasts with neural proof guidance systems that process each clause considered for selection with a NN, such as ENIGMA Anonymous [5] or the prototype extension of E by Loos et al. [7].

Training The GNN is trained on successful proof searches. Similarly to ENIGMA [5], our GNN is trained so that the clause selection heuristic favors clauses that have been observed to contribute to a proof. Using a GNN that does not have access to symbol names, we train a signature-agnostic system.

Each training example consists of an input problem, a proof clause c^+, and a non-proof clause c^-. The loss function ℓ is defined so that it is monotone with respect to the weight of the proof clause $w(c^+)$ and anti-monotone with respect to the weight of the non-proof clause $w(c^-)$. Additional penalty terms, scaled by hyperparameters λ_f and λ_X, ensure that the symbol weights w_{f_i} and the variable weight w_X do not drift far from the standard value 1. The loss of the proof clause c^+ and the non-proof clause c^- in a problem with symbols $f_1, \dots, f_n$ is:

$$\ell(c^+, c^-) = -\log \operatorname{sigmoid}(w(c^-) - w(c^+)) + \lambda_f \frac{1}{n} \sum_{i=1}^{n} (\log w_{f_i})^2 + \lambda_X (\log w_X)^2 \tag{1}$$

This loss design is inspired by the approach that we successfully applied to symbol precedence recommendation [2].

Training examples will be collected by running the target ATP Vampire [6] on problems randomly sampled from the TPTP problem library [12].

3 Possible modifications

Additional input features The clause weight function may include additional clause features available in the prover, namely clause derivation depth and size. These clause features may be multiplied by coefficients predicted by the GNN to make the influence of the features trainable. In a similar fashion, the weights of terms, atoms, and literals may be augmented by term-ordering-aware features [11], namely literal maximality in the clause, term maximality in an equation, and orientability of an equation.

Binary cross-entropy loss A straightforward alternative to the loss function defined above (1) is provided by training a binary clause classifier that predicts whether an input clause is non-proof. If we allow the symbol and variable weights to span all real numbers, we can train the NN using the standard binary cross-entropy loss, which allows extracting a prediction of probability of the input clause classifying as non-proof. Ranking the clauses by this probability prioritizes the inference of clauses that are likely to contribute to the proof. Although allowing negative term weights forfeits fairness of the clause selection, this can be salvaged (as is anyway done in the typical clause selection setup) by alternating weight-based and age-based clause section under some ratio.

Recursive neural network Without increasing the asymptotic computational cost of the evaluation of the clause weight, we can train a recursive neural network (RNN) for the clause syntax directed acyclic graph (dag) [3]. Depending on the complexity of the RNN, this can significantly increase the computational cost of evaluating the weight of a clause. The RNN may use a GNN to calculate the initial embeddings of the symbols and variables.

References

[1] Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 19–99. Elsevier and MIT Press, 2001. ISBN 0-444-50813-9. doi:10.1016/b978-044450813-3/50004-7.

[2] Filip Bártek and Martin Suda. Neural precedence recommender. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 525–542. Springer, 2021. ISBN 978-3-030-79875-8. doi:10.1007/978-3-030-79876-5_30.

[3] Karel Chvalovský. Top-down neural model for formulae. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL `https://openreview.net/forum?id=Byg5QhR5FQ`.

[4] Xavier Glorot, Ankit Anand, Eser Aygün, Shibl Mourad, Pushmeet Kohli, and Doina Precup. Learning representations of logical formulae using graph neural networks. In *Workshop on Graph Representation Learning at 33rd Neural Information Processing Systems (NeurIPS 2019)*, 2019. URL `https://grlearning.github.io/papers/58.pdf`.

[5] Jan Jakubuv, Karel Chvalovský, Miroslav Olšák, Bartosz Piotrowski, Martin Suda, and Josef Urban. ENIGMA Anonymous: Symbol-independent inference guiding machine (system description). In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 448–463. Springer, 2020. ISBN 978-3-030-51053-4. doi:10.1007/978-3-030-51054-1_29.

[6] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 1–35. Springer, 2013. ISBN 978-3-642-39798-1. doi:10.1007/978-3-642-39799-8_1.

[7] Sarah M. Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. Deep network guided proof search. In Thomas Eiter and David Sands, editors, *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*, volume 46 of *EPiC Series in Computing*, pages 85–105. EasyChair, 2017. doi:10.29007/8mwc.

[8] Miroslav Olšák, Cezary Kaliszyk, and Josef Urban. Property invariant embedding for automated reasoning. In Giuseppe De Giacomo, Alejandro Catalá, Bistra Dilkina, Michela Milano, Senén Barro, Alberto Bugarín, and Jérôme Lang, editors, *ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020)*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 1395–1402. IOS Press, 2020. ISBN 978-1-64368-100-9. doi:10.3233/FAIA200244.

[9] Michael Rawson and Giles Reger. Directed graph networks for logical reasoning (extended abstract). In Pascal Fontaine, Konstantin Korovin, Ilias S. Kotsireas, Philipp Rümmer,

and Sophie Tourret, editors, *Joint Proceedings of the 7th Workshop on Practical Aspects of Automated Reasoning (PAAR) and the 5th Satisfiability Checking and Symbolic Computation Workshop (SC-Square) Workshop, 2020 co-located with the 10th International Joint Conference on Automated Reasoning (IJCAR 2020), Paris, France, June-July, 2020 (Virtual)*, volume 2752 of *CEUR Workshop Proceedings*, pages 109–119. CEUR-WS.org, 2020. URL http://ceur-ws.org/Vol-2752/paper8.pdf.

[10] Stephan Schulz. *E 2.4 User Manual*, 2020. URL https://easychair.org/publications/preprint/8dss.

[11] Stephan Schulz and Martin Möhrmann. Performance of clause selection heuristics for saturation-based theorem proving. In Nicola Olivetti and Ashish Tiwari, editors, *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*, volume 9706 of *Lecture Notes in Computer Science*, pages 330–345. Springer, 2016. ISBN 978-3-319-40228-4. doi:10.1007/978-3-319-40229-1_23.

[12] Geoff Sutcliffe. The TPTP problem library and associated infrastructure. *Journal of Automated Reasoning*, 59(4), December 2017. ISSN 0168-7433. doi:10.1007/s10817-017-9407-7.

[13] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020. doi:10.1016/j.aiopen.2021.01.001.

A Parallel Corpus of Natural Language and Isabelle Artefacts

Anthony Bordg*, Yiannos A. Stathopoulos†, and Lawrence C. Paulson‡

Department of Computer Science and Technology, University of Cambridge, UK
[apdb3,yas23,lp15]@cam.ac.uk

Parallel corpora are key resources for machine translation in natural language processing (NLP). A parallel corpus maps textual scripts in one language (e.g., French) to their equivalents in another language (e.g., English). The language-paired scripts in a parallel corpus are data points used to train language models that learn how to translate text from one language to the other.

Recently, the theorem proving community explored *autoformalisation* – the task of generating formal proofs that can be recognised by a theorem prover from their counterparts expressed in informal natural language – as an instance of machine translation [1, 2]. Large transformer models, such as Codex [3], have demonstrated that machines can learn to generate code from natural language text through the use of large (parallel) corpora.

We introduce the *Isabelle Parallel Corpus* (IPC) of natural language and Isabelle/HOL proofs. Natural language proofs in our corpus are expressed using sentences in the natural language of mathematics, with mathematical expressions transcribed using LaTeX. The aforementioned textual proofs have been extracted from textbooks, International Olympiad of Mathematics solution sheets and other real-world mathematics resources.

In this presentation we will describe our multi-stage approach for constructing our corpus, showcase our annotation tools and discuss the challenges involved in designing the annotation scheme of a parallel corpus linking natural language to formal proofs.

We developed an annotation tool that allows us to (a) record information about artefacts in the corpus, (b) collect parallel natural language and Isabelle/Isar scripts and (c) implement the annotation scheme for the IPC. Our tool is built on top of a special instance of the SErAPIS search engine for Isabelle and supports multi-user annotation.

In the first phase of building our corpus we have sourced over 500 Isabelle artefacts, including theorems, definitions, lemmata and proof scripts. For each artefact we record information that includes a statement of each artefact in the natural language of mathematics typeset in LaTeX, a BibTeX citation to the source material (textbooks, journal etc), the page and number (e.g., Theorem 4.1) as they appear in the source material. The second phase, which is ongoing, involves attaching informal and formal Isabelle/Isar proofs to the recorded statements. At the time of writing, we have paired Isabelle/Isar proofs with corresponding informal proofs for 18 statements.

The consensus in NLP is that machine translation models benefit from word and sentence alignments [4, 5]. A sentence alignment for two parallel text scripts in different languages is a pairing that links sentences in one language to sentences in the other language. Similarly, a word alignment links tokens from a script in one language to the tokens of its equivalent script in the other language. The parallel corpus designers are responsible for including annotations

*Anthony Bordg thanks Manuel Eberl for sharing his knowledge of the Archive of Formal Proofs' contents.

†Yiannos Stathopoulos thanks Sean Holden and Albert Qiaochu Jiang for discussing the corpus with the authors.

‡The authors were supported by the ERC Advanced Grant ALEXANDRIA (Project 742178) funded by the European Research Council and led by Professor Lawrence Paulson at the University of Cambridge, UK.

for sentence and word alignments if this information is required by the intended use of the corpus.

However, without answering questions like "Does every sentence in a natural language proof correspond to a statement in Isabelle/Isar?" and "Can one Isabelle/Isar statement account for multiple natural language sentences?", the nature of sentence and word alignments for a parallel corpus like the IPC is unclear.

Therefore, the **first challenge** in designing the IPC is to identify the annotation requirements of the corpus for aligning natural language sentences to Isabelle/Isar statements. We conducted a pilot study to determine the requirements of such an annotation and made some observations, including:

1. there are sentences in the natural language that do not correspond to any statement in Isabelle/Isar and vice-versa and this occurs, for example, when the source text and the Isabelle formalisation assume different prerequisites;

2. there is a many-to-many mapping (*i.e.* not a perfect one-to-one correspondence) between facts within the textual proof of a statement and facts within the corresponding Isabelle/Isar proof script;

3. it sometimes happens that results embedded in Isabelle proofs are not factorised as lemmata, which could be possibly useful results on their own, but this phenomenon does not occur in natural language proofs since one can always refer to a result even if it is not explicitly factorised;

4. both textual and formal proofs may import dependencies in their argumentation. Dependencies in Isabelle/Isar proofs may span multiple theory files.

Our observations give rise to the **second challenge** in designing IPC: how should dependencies in parallel textual and Isabelle/Isar proofs be incorporated in the corpus? One solution would be to integrate dependencies in the corpus and include data about the reference graph between artefacts [6].

Unlike general-purpose natural language, the language of mathematics follows its own conventions and is interspersed with mathematical expressions [7]. Similarly, proofs in the Isabelle/Isar language are structured and include statements with terms representing assumptions and symbolic reasoning. Therefore, the **third challenge** is designing a suitable annotation scheme for (a) representing Isabelle/Isar terms and mathematical expressions in textual proofs and (b) establishing an alignment between them. Attractive solutions come from Mathematical Knowledge Management (MKM) and code understanding and generation. For instance, mathematical expressions and terms can be encoded using Presentation and Content MathML [8]. Furthermore, we can overlay our corpus with token type and other information, such as identifier tagging (IT), that will allow researchers to implement masked span/identifier prediction[9] and skip-tree training [10] in models trained with our corpus. We envisage that the third phase of our process will address these challenges and introduce sentence and token alignments to the IPC.

The IPC will be made public on GitHub prior to our presentation in the hope that phase 2 material (data linking natural language proofs to their Isabelle/Isar counterparts) will be useful to researchers in machine learning for theorem proving. We intend to continuously update the corpus (*e.g.* with sentence and token alignments in phase 3) and this strategy reflects our vision that the IPC is a *living* corpus with standardised releases to facilitate comparative analysis of machine learning models. We also intend to open our annotation tools to the wider community and we invite all Isabelle users to join in the annotation effort to continuously expand the IPC.

References

[1] Christian Szegedy. A promising path towards autoformalization and general artificial intelligence. In Christoph Benzmüller and Bruce Miller, editors, *Intelligent Computer Mathematics*, pages 3–20, Cham, 2020. Springer International Publishing.

[2] Qingxiang Wang, Chad Brown, Cezary Kaliszyk, and Josef Urban. Exploration of neural machine translation in autoformalization of mathematics in mizar. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, jan 2020.

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. 2021.

[4] Philipp Koehn. *Statistical Machine Translation.* Cambridge University Press, USA, 1st edition, 2010.

[5] Dan Jurafsky and James H. Martin. *Speech and language processing : an introduction to natural language processing, computational linguistics, and speech recognition.* Pearson Prentice Hall, Upper Saddle River, N.J., 2009.

[6] Sean Welleck, Jiacheng Liu, Ronan Le Bras, Hannaneh Hajishirzi, Yejin Choi, and Kyunghyun Cho. Naturalproofs: Mathematical theorem proving in natural language, 2021.

[7] Mohan Ganesalingam. *The Language of Mathematics.* PhD thesis, Cambridge University Computer Laboratory, 2008.

[8] Minh-Quoc Nghiem, Giovanni Yoko Kristianto, Goran Topić, and Akiko Aizawa. Which one is better: Presentation-based or content-based math search? In Stephen M. Watt, James H. Davenport, Alan P. Sexton, Petr Sojka, and Josef Urban, editors, *Intelligent Computer Mathematics*, pages 200–212, Cham, 2014. Springer International Publishing.

[9] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.

[10] Markus Norman Rabe, Dennis Lee, Kshitij Bansal, and Christian Szegedy. Mathematical reasoning via self-supervised skip-tree training. In *International Conference on Learning Representations*, 2021.

Strategies and Machine Learning for Lash

Chad E. Brown[1], Jan Jakubuv[1,2], and Cezary Kaliszyk[2]

[1] Czech Technical University in Prague, Czechia
[2] University of Innsbruck, Austria

Lash [BK22] is an automated theorem prover for higher-order logic. Lash is a fork of the theorem prover Satallax [Bro12; FB16]. Lash replaces Satallax's Ocaml representation of terms with an efficient representation of normal terms (with unique integer ids) in C. The representation in C also stores useful information such as which de Bruijn indices are free in the term. Knowing the free de Bruijn indices of terms makes recognizing potential η-redexes possible without traversing the λ-abstraction. Likewise it is possible to determine when shifting and substitution of de Bruijn indices would not affect a term, avoiding the need to traverse the term. Computations such as substitutions and shifting de Bruijn indices are cached to prevent recomputation.

In addition to the low-level C term reimplementation, we have also provided a number of other low-level functionalities replacing the slower parts of the Ocaml code. This includes low-level priority queues, as well as C code used to associate the integers representing normal propositions with integers that are used to communicate with MiniSat.

As with Satallax, the nature of Lash's search is highly dependent on the settings of boolean and integer flags. A *mode* (also called a *strategy*) is a collection of flag settings. A *schedule* is a sequence of modes, along with a timeout. A few of the important flags include:

- INITIAL_SUBTERMS_AS_INSTANTIATIONS is a boolean flag. If set to true, Lash uses closed subterms of the initial problem as instantiations.

- EAGERLY_PROCESS_INSTANTIATIONS is a boolean flag. If set to true, new instantiations are processed immediately instead of being put onto the priority queue.

- SPLIT_GLOBAL_DISJUNCTIONS is a boolean flag. If set to true, Lash tries to split the problem into several subproblems to be proven independently before beginning the search.

- MINISAT_SEARCH_PERIOD is an integer flag that controls how often Minisat is asked to search for a model of the current set of propositional clauses. Sometimes this is set to a low number, e.g., 10, so that Minisat checks for unsatisfiability after every tenth step. Often it is useful to set the flag to a very high number, e.g., a billion, so that Lash will effectively not ask Minisat to check for unsatisfiability, unless all other options are exhausted. (In many problems Minisat determines unsatisfiability via unit propagation without decisions.)

- ENUM_START is an integer flag that determines how long to delay beginning enumeration of instantiations for quantifiers at function types. If a problem has a higher-order quantifier that is irrelevant to the problem, then a high value is helpful (since no higher-order instantiation is needed). If the problem requires instantiating a higher-order quantifier, then a lower value is more likely to lead to success.

- AXIOM_DELAY is an integer flag that (if nonzero) nudges Lash to work on the negated conjecture before working on the axioms, with larger integers corresponding to longer delays.

Following Satallax, Lash reads a problem and determines if it exceeds a certain size threshold, classifying problems as "small" or "large." For large problems an implementation of SInE [HV11] reduces the size of the problem and then searches using a schedule appropriate for large problems. For small problems a search proceeds using a schedule appropriate for small problems. Which modes and schedules are appropriate for which kinds of problems is determined experimentally.

Starting with several manually designed strategies (i.e., modes) and we use the strategy invention system GRACKLE[1] to invent more strategies targeted to randomly selected 1800 TPTP THF problems. GRACKLE is a generalization of strategy invention system BLISTR [Urb13]. Both systems are based an evolutionary algorithm, where strategies are considered "animals" and problems to be solved their "food". Only animals that consume enough food, that is, solve enough problems, survive to the next generation and are given a chance to conceive an offspring. This algorithm favors animals that consume food not consumed by others. This leads to diversity and complementarity of invented strategies.

While BLISTR is a strategy invention system for E Prover, GRACKLE can be instantiated to invent strategies for any solver. A requisite of this instantiation is a parametrization of the solver configuration space, in our case, the collection of Lash mode flags. We start by extracting flag names and possible option values from the collection of manually designed modes. This gives us 130 flag names. Out of them, 43 correspond to boolean options. The rest are integer options, with domains ranging in size from 2 to 26. Altogether, the space contains around 10^{60} possible configurations.

Once the configurations space is parametrized, we can launch GRACKLE with 10 initial configurations, selected from the manually designed strategies as the most complementary and strongest ones. During the strategy invention, configurations are evaluated with a short runtime limit of 1 second. The best of the initial strategies solves 285 problems (out of 1800 TPTP problems) in 1 second. Together, the 10 initial strategies solve 358 problems.

We limit GRACKLE runs to 24 hours to be able to evaluate several strategy invention options. We try different settings for maximal strategy generation size, and several variations of strategy specialization used to produce offspring. Together we run around 14 GRACKLE runs, each running on 8 CPU cores. All the runs produce more than 3000 different modes, the best of them solves 355 problems in 1 second. The greedy collection of 10 best new modes solves together 449 problems, and the total coverage of all the modes is 489.

We construct a Lash *schedule* by evaluating best 110 strategies on the training problems in 30 seconds. From this evaluation, we construct a greedy cover of 10 best strategies. This greedy cover gives us an order in which to run the modes, provided the overall time limit is evenly distributed among the modes. Additionally, we construct a schedule with 20 modes, extending the first 10 with another greedy cover constructed without the previously selected modes. Furthermore, we can also split the results of the evaluation into results on "small" and "large" problems, and we can construct two different schedules for them.

We are investigating the possible ways how machine learning can be included in Lash. In particular, this could involve the extension of the shared C representation by precomputing various features useful for prioritizing the available actions [FB16]. Other ways to apply machine learning include the selection or even generation of interesting instances for a given problem, as well as machine-learning guided interaction with the SAT-solver.

Acknowledgements The results were supported by the Ministry of Education, Youth and Sports within the dedicated program ERC CZ under the project POSTMAN no. LL1902 and the ERC starting grant no. 714034 SMART.

[1] https://github.com/ai4reason/grackle

References

[BK22] C. E. Brown and C. Kaliszyk. "Lash 1.0 (System Description)". In: accepted for publication at IJCAR 2022. 2022.

[Bro12] C. E. Brown. "Satallax: An Automatic Higher-Order Prover". In: *IJCAR*. Ed. by B. Gramlich, D. Miller, and U. Sattler. Vol. 7364. LNCS. Springer, 2012, pp. 111–117.

[FB16] M. Färber and C. E. Brown. "Internal Guidance for Satallax". In: *Automated Reasoning - 8th International Joint Conference, IJCAR 2016*. Ed. by N. Olivetti and A. Tiwari. Vol. 9706. LNCS. Springer, 2016, pp. 349–361. URL: https://doi.org/10.1007/978-3-319-40229-1_24.

[HV11] K. Hoder and A. Voronkov. "Sine Qua Non for Large Theory Reasoning". In: *23rd International Conference Automated Deduction - CADE-23*. Ed. by N. S. Bjørner and V. Sofronie-Stokkermans. Vol. 6803. LNCS. Springer, 2011, pp. 299–314. DOI: 10.1007/978-3-642-22438-6_23. URL: https://doi.org/10.1007/978-3-642-22438-6_23.

[Urb13] J. Urban. "BliStr: The Blind Strategymaker". In: *CoRR* abs/1301.2683 (2013). arXiv: 1301.2683. URL: http://arxiv.org/abs/1301.2683.

Embedding SUMO into Set Theory *

Chad Brown[1], Adam Pease[2], and Josef Urban[1]

[1] Czech Technical University in Prague, Czech Republic
[2] Articulate Software, San Jose, CA, USA

The Suggested Upper Merged Ontology (SUMO) [13, 14] is a comprehensive ontology of around 20,000 concepts and 80,000 hand-authored logical statements in a higher-order logic, that has an associated integrated development environment called Sigma [16][1] that interfaces to leading theorem provers such as Eprover [19] and Vampire [12]. In previous work on translating SUMO to THF [2] a syntactic translation to THF was created but did not resolve many aspects of the intended higher order semantics of SUMO. It is our objective in our current efforts to lay the groundwork for a new translation to TH0, based on expressing SUMO in set theory. We believe this will attach to SUMO a stronger set-theoretical interpretation that will allow deciding more queries and provide better intuition for avoiding contradictory formalizations. Once this is done, our plan is to train ENIGMA-style [5–8] query answering and contradiction-finding [20] AITP systems on such SUMO problems and develop autoformalization [9–11, 23, 24] methods targeting common-sense reasoning based on SUMO.

In earlier work, we described [16] how to translate SUMO to the strictly first order language of TPTP-FOF [18] and TF0 [15, 22]. SUMO has an extensive type structure and all relations have type restrictions on their arguments. Translation to TPTP FOF involved implementing a sorted (typed) logic axiomatically in TPTP by altering all implications in SUMO to contain type restrictions on any variables that appear.

We give a grammar for the fragment of the SUMO language in the domain of our translation of SUMO. There are some aspects of SUMO that do not fall into this grammar – namely formulas with modal or probabilistic operators. We have ordinary variables (x), row variables (ρ) and constants (c). We mutually define the sets of terms t, spines s and formulas ψ as follows:

$$t ::= x|c|x\ s|c\ s|(\kappa x.\psi)$$

$$s ::= t\ s|\cdot|\rho$$

$$\psi ::= \bot|\top|(\neg\psi)|(\psi \to \psi)|(\psi \wedge \psi)|(\psi \vee \psi)|(\psi \leftrightarrow \psi)|(\forall x.\psi)|(\exists x.\psi)|(t = t)|c\ s$$

The definition is mutually recursive since the term $\kappa x.\psi$ depends on the formula ψ. Of course, κ, $\forall$ and $\exists$ are binders.

Properly parsing SUMO terms and formulas requires mechanisms for infering implicit type guards for variables (interpreted conjunctively for κ and $\exists$ and via implication for $\forall$). Free variables in SUMO assertions are implicitly universally quantified. For simplicity, we assume all type guards and implicit quantifiers have already been inferred (as in [16]) before beginning the translation.

Our translation maps terms t and spines s to sets and formulas ψ to set theoretic propositions. The particular set theory we use is higher-order Tarski-Grothendieck as described in [4]. For simplicity we assume SUMO variables (both ordinary and row) are also set theoretic variables ranging over sets. We likewise assume all SUMO constants are also set theoretic constants (with the exception of instance and subclass, described below). To translate spines, we need

*Supported by by the Czech project AI&Reasoning CZ.02.1.01/0.0/0.0/15_003/ 0000466 and the European Regional Development Fund.

[1] https://www.ontologyportal.org

a way to form lists as sets. We do this generically by simply assuming a set nil, an operator cons taking two sets to a set (meant to be the cons pair) and an operator $\mathsf{listprod}$ taking a two sets to a set (where $\mathsf{listprod}\ A\ B$ is meant to be the set of cons pairs $\mathsf{cons}\ a\ b$ where $a \in A$ and $b \in B$). We also assume a special set U, which will act as the universe of elements that may be members of classes. The universe is assumed to be closed under set theoretic function application and the list operators mentioned above.

A major commitment of our translation is that, if t_i are SUMO terms translated to sets t'_i, then SUMO formulas of the form $\mathsf{instance}\ t_1\ t_2$ will translate to $t'_1 \in t'_2$ and SUMO formulas of the form $\mathsf{subclass}\ t_1\ t_2$ will translate to $t'_1 \subseteq t'_2$.

We isolate a few special cases in SUMO: Class, SetOrClass, Abstract and Entity. These are considered *classes* in SUMO, but will be considered *superclasses* in our translation. In particular, the interpretation of Class will be the $\wp\mathsf{U}$, power set of U. Hence every member of the interpretation of Class will be a subset of U. Since SUMO declares SetOrClass, Abstract and Entity to contain Class, none of these four superclasses can be a member of $\wp\mathsf{U}$. Whenever SUMO globally declares t_1 to be a subclass of t_2, we also declare the corresponding sets t'_1 and t'_2 to be members of $\wp\mathsf{U}$, except in the four special superclass cases.

The translation of spines is the obvious one: we use nil for the empty spine, cons for a term followed by a spine, and the same variable ρ for row variables. For terms, we translate x and c directly, since we assumed these are variables and constants in the set theory. We translate $x\ s$ and $c\ s$ by using set theoretic function application (where the translation of the spine s is the argument). We translate $\kappa x.\psi$ as $\{x \in \mathsf{U} | \psi'\}$ where ψ' is the translation of ψ. Translation of formulas proceeds in the obvious way, with only the $c\ s$ case being noteworthy. Note that $c\ s$ is both a term and a formula. Interpreting $c\ s$ as a term gives a set b and we translate $c\ s$ to the formula $0 \in b$. The idea is that b will be either 0 or 1, with $0 \in b$ being false if b is 0 and true if b is 1.

Future work While we have an incomplete translation of all the elements of SUMO that are beyond syntactic first order expressions, we now have a basis for mapping SUMO into set theory. The current translation addresses the κ binder, a term level binder one cannot represent in standard first-order logic. In addition, we can translate row variables directly as sets, although we expect more work is needed to create proper bounds for quantifiers of row variables. In addition a future translation should account for temporal `holdsDuring` and modal operators (including `modalAttribute`, `confersRight` etc). We expect this to be the next stage of our efforts.

An initial use the mapping will be to have a type-checker that gives immediate feedback to SUMO developers on one aspect of the correctness of their higher-order axioms, in the same way that the TPTP FOF and TF0 translations provide feedback on the correct use of types in the FOF portion of SUMO.

We have some inference tests for SUMO[2] for TPTP FOF and TF0. We will expand this corpus to include tests expressible in the new TH0 translation. This will serve as a regression suite for SUMO's higher-order content, as well as a source for testing the performance of HOL provers, such as Satallax [3], Zipperposition [1] and LEO-III [21].

It is possible that set theory provides a model for (some part or all of) SUMO after the translation. Proving that at least portions of SUMO have a model would be a much stronger statement than current method that only allow us to say that no contradictions have been found with first order theorem proving within a large set of tests and a generous time bound [17].

[2] https://github.com/ontologyportal/sumo/tree/master/tests/TPTP

References

[1] Alexander Bentkamp, Jasmin Christian Blanchette, Simon Cruanes, and Uwe Waldmann. Superposition for lambda-free higher-order logic. *Logical Methods in Computer Science*, 17(2):1:1–1:38, April 2021.

[2] Christoph Benzmüller and Adam Pease. Higher-Order Aspects and Context in SUMO. In Ivan José Varzinczak Jos Lehmann and Alan Bundy, editors, *Special issue on Reasoning with context in the Semantic Web*, volume 12-13. Science, Services and Agents on the World Wide Web, 2012.

[3] Chad E. Brown. Satallax: An automatic higher-order prover. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *IJCAR*, volume 7364 of *LNCS*, pages 111–117. Springer, 2012.

[4] Chad E. Brown and Karol Pąk. A tale of two set theories. In Cezary Kaliszyk, Edwin C. Brady, Andrea Kohlhase, and Claudio Sacerdoti Coen, editors, *Intelligent Computer Mathematics - 12th International Conference, CICM 2019, Prague, Czech Republic, July 8-12, 2019, Proceedings*, volume 11617 of *Lecture Notes in Computer Science*, pages 44–60. Springer, 2019.

[5] Karel Chvalovský, Jan Jakubuv, Martin Suda, and Josef Urban. ENIGMA-NG: efficient neural and gradient-boosted inference guidance for E. In Pascal Fontaine, editor, *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*, volume 11716 of *Lecture Notes in Computer Science*, pages 197–215. Springer, 2019.

[6] Jan Jakubuv, Karel Chvalovský, Miroslav Olsák, Bartosz Piotrowski, Martin Suda, and Josef Urban. ENIGMA anonymous: Symbol-independent inference guiding machine (system description). In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 448–463. Springer, 2020.

[7] Jan Jakubuv and Josef Urban. ENIGMA: efficient learning-based inference guiding machine. In Herman Geuvers, Matthew England, Osman Hasan, Florian Rabe, and Olaf Teschke, editors, *Intelligent Computer Mathematics - 10th International Conference, CICM 2017, Edinburgh, UK, July 17-21, 2017, Proceedings*, volume 10383 of *Lecture Notes in Computer Science*, pages 292–302. Springer, 2017.

[8] Jan Jakubuv and Josef Urban. Hammering Mizar by learning clause guidance. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*, volume 141 of *LIPIcs*, pages 34:1–34:8. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[9] Cezary Kaliszyk, Josef Urban, and Jirí Vyskocil. Automating formalization by statistical and semantic parsing of mathematics. In *ITP*, volume 10499 of *Lecture Notes in Computer Science*, pages 12–27. Springer, 2017.

[10] Cezary Kaliszyk, Josef Urban, and Jiří Vyskočil. Learning to parse on aligned corpora (rough diamond). In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, volume 9236 of *Lecture Notes in Computer Science*, pages 227–233. Springer, 2015.

[11] Cezary Kaliszyk, Josef Urban, Jiří Vyskočil, and Herman Geuvers. Developing corpus-based translation methods between informal and formal mathematics: Project description. In Stephen M. Watt, James H. Davenport, Alan P. Sexton, Petr Sojka, and Josef Urban, editors, *Intelligent Computer Mathematics - International Conference, CICM 2014, Coimbra, Portugal, July 7-11, 2014. Proceedings*, volume 8543 of *LNCS*, pages 435–439. Springer, 2014.

[12] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In *Proceedings of the 25th International Conference on Computer Aided Verification*, volume 8044 of *CAV 2013*, pages 1–35, New York, NY, USA, 2013. Springer-Verlag New York, Inc.

[13] Ian Niles and Adam Pease. Toward a Standard Upper Ontology. In Chris Welty and Barry Smith, editors, *Proceedings of the 2nd International Conference on Formal Ontology in Information Systems (FOIS-2001)*, pages 2–9, 2001.

[14] Adam Pease. *Ontology: A Practical Guide*. Articulate Software Press, Angwin, CA, 2011.

[15] Adam Pease. Arithmetic and inference in a large theory. In *AI in Theorem Proving*, 2019.

[16] Adam Pease and Stephan Schulz. Knowledge Engineering for Large Ontologies with Sigma KEE 3.0. In *The International Joint Conference on Automated Reasoning*, 2014.

[17] Adam Pease and Stephan Schulz. Contradiction detection and repair in a large theory. In *The International FLAIRS Conference Proceedings*, 2022.

[18] Adam Pease, Geoff Sutcliffe, Nick Siegel, and Steven Trac. Large Theory Reasoning with SUMO at CASC. *AI Communications, Special issue on Practical Aspects of Automated Reasoning*, 23(2-3):137–144, 2010.

[19] Stephan Schulz. E - A Brainiac Theorem Prover. *AI Commun.*, 15(2-3):111–126, 2002.

[20] Stephan Schulz, Geoff Sutcliffe, Josef Urban, and Adam Pease. Detecting inconsistencies in large first-order knowledge bases. In *Proceedings of CADE 26*, pages 310–325. Springer, 2017.

[21] Alexander Steen and Christoph Benzmüller. The higher-order prover leo-iii. *CoRR*, abs/1802.02732, 2018.

[22] Geoff Sutcliffe, Stephan Schulz, Koen Claessen, and Peter Baumgartner. The TPTP Typed First-order Form with Arithmetic. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR 2012)*, pages 406–419, 2012.

[23] Qingxiang Wang, Chad E. Brown, Cezary Kaliszyk, and Josef Urban. Exploration of neural machine translation in autoformalization of mathematics in mizar. In *CPP*, pages 85–98. ACM, 2020.

[24] Qingxiang Wang, Cezary Kaliszyk, and Josef Urban. First experiments with neural translation of informal to formal mathematics. In *CICM*, volume 11006 of *Lecture Notes in Computer Science*, pages 255–270. Springer, 2018.

1 Appendix

Examples We briefly consider two first-order example queries and three queries involving κ. Queries differ from assertions in that their free variables are implicitly existentially quantified, with implicit type guards added conjunctively. We prove the translated query in the Megalodon interactive prover (the successor to the Egal system [4]).

Oure first example is given by the SUMO query:

```
(instance Org1-1 Organization)
(query (member ?MEMBER Org1-1))
```

This translates into Megalodon as follows:

```
Variable s_ORG1_x2D1:set.
Hypothesis p5315: (s_ORG1_x2D1 :e s_ORGANIZATION).
Theorem p5316: exists v_MEMBER, v_MEMBER :e s_PHYSICAL
    /\ (bp (s_MEMBER (cons v_MEMBER (cons s_ORG1_x2D1 nil)))).
```

The (interactively constructed) proof makes use of (translated) SUMO assertions that all collections have a physical object member and that organizations are collections.

Our second example is given by the SUMO query:

```
(=>
  (and
    (instance ?A Animal)
    (not
      (exists (?PART)
        (and
          (instance ?PART SpinalColumn)
          (part ?PART ?A)))))
  (not
    (instance ?A Vertebrate)))

(not
  (exists (?SPINE)
    (and
      (instance ?SPINE SpinalColumn)
      (part ?SPINE BananaSlug10-1))))

(instance BananaSlug10-1 Animal)

(and
  (instance BodyPart10-1 BodyPart)
  (component BodyPart10-1 BananaSlug10-1))

(query (instance BananaSlug10-1 Invertebrate))
```

This translates into the following Megalodon formalization:

```
Variable s_SPINALCOLUMN:set.
Hypothesis p5320: forall v_A, v_A :e s_ENTITY -> v_A :e s_OBJECT ->
 ((v_A :e s_ANIMAL)
   /\ (~ (exists v_PART, v_PART :e s_ENTITY /\ v_PART :e s_OBJECT
            /\ (v_PART :e s_SPINALCOLUMN) /\ (bp (s_PART (cons v_PART (cons v_A nil))))))
```

```
   -> (~ (v_A :e s_VERTEBRATE))).
Variable s_BANANASLUG10_x2D1:set.
Hypothesis p5321: (~ (exists v_SPINE, v_SPINE :e s_ENTITY /\ v_SPINE :e s_OBJECT
     /\ (v_SPINE :e s_SPINALCOLUMN) /\ (bp (s_PART (cons v_SPINE (cons s_BANANASLUG10_x2D1 nil)))))).
Hypothesis p5322: (s_BANANASLUG10_x2D1 :e s_ANIMAL).
Variable s_BODYPART10_x2D1:set.
Hypothesis p5323: (s_BODYPART10_x2D1 :e s_BODYPART)
  /\ (bp (s_COMPONENT (cons s_BODYPART10_x2D1 (cons s_BANANASLUG10_x2D1 nil)))).
Theorem p5324: (s_BANANASLUG10_x2D1 :e s_INVERTEBRATE).
```

The proof uses the translation of the SUMO assertion that the classes of vertebrates and invertebrates form a partition of the class of animals.

Our κ examples are all variants of the same idea, and all are easily provable. The first κ example query is given in SUMO as follows:

```
(query (forall (?V) (=> (instance ?V Atom)
        (forall (?E) (=> (instance ?E Electron)
  (=> (part ?E ?V)
      (instance ?E (KappaFn ?x (and (part ?x ?V) (instance ?x Electron)))))))))
```

This translates to the following Megalodon formalization:

```
Theorem p5326: (forall v_V, v_V :e s_ENTITY -> v_V :e s_OBJECT -> ((v_V :e s_ATOM)
 -> (forall v_E, v_E :e s_ENTITY -> v_E :e s_OBJECT -> ((v_E :e s_ELECTRON)
 -> ((bp (s_PART (cons v_E (cons v_V nil))))
 -> (v_E :e {v_X :e Univ1 | v_X :e s_OBJECT /\ v_X :e s_ENTITY
      /\ (bp (s_PART (cons v_X (cons v_V nil)))) /\ (v_X :e s_ELECTRON)}))))))).
```

The second κ example query is given in SUMO as follows:

```
(query (forall (?V) (=> (instance ?V Atom)
        (forall (?E) (=> (instance ?E Electron)
  (=> (instance ?E (KappaFn ?x (and (part ?x ?V) (instance ?x Electron))))
      (part ?E ?V)))))))
```

This translates to the following Megalodon formalization:

```
Theorem p5327: (forall v_V, v_V :e s_ENTITY -> v_V :e s_OBJECT -> ((v_V :e s_ATOM) ->
 (forall v_E, v_E :e s_ENTITY -> v_E :e s_OBJECT -> ((v_E :e s_ELECTRON)
 -> ((v_E :e {v_X :e Univ1 | v_X :e s_OBJECT /\ v_X :e s_ENTITY
         /\ (bp (s_PART (cons v_X (cons v_V nil)))) /\ (v_X :e s_ELECTRON)})
 -> (bp (s_PART (cons v_E (cons v_V nil))))))))).
```

The final κ example does not use κ in the statement, though κ is vital to proving the translated theorem. In SUMO the example is given as follows:

```
(query (forall (?V) (=> (instance ?V Atom)
        (forall (?E) (=> (instance ?E Electron)
  (exists (?C)
    (and (instance ?C Class)
      (<=> (part ?E ?V)
           (instance ?E ?C)))))))))
```

This translates to the following Megalodon formalization:

```
Theorem p5328: (forall v_V, v_V :e s_ENTITY -> v_V :e s_OBJECT -> ((v_V :e s_ATOM) ->
 (forall v_E, v_E :e s_ENTITY -> v_E :e s_OBJECT -> ((v_E :e s_ELECTRON)
 -> (exists v_C, v_C :e s_ENTITY /\ v_C :e s_CLASS /\ (v_C :e s_CLASS)
      /\ ((bp (s_PART (cons v_E (cons v_V nil)))) <-> (v_E :e v_C))))))).
```

Converting SUMO to First Order All the strictly higher-order content in SUMO was previously lost in translation to first-order, whether TPTP or TF0. The translation steps include:

- expanding "row variables" which allow for stating axioms without commitment to the number of arguments a relation has, similar to Lisp's @REST construct
- instantiating "predicate variables" with all possible values. This is needed for any axiom that has a variable in place of a relation.
- expanding the arity of all variable arity relations as set of relations with different names depending upon their fixed number of arguments
- renaming any relations given as arguments to other relations

SUMO has no native implementation in a theorem prover, and has no formal semantics beyond that of standard first order logic, so the process of translating SUMO into a language with a fully specified semantics, such as TPTP_FOF, TF0 or THF gives SUMO its semantics.

Type Mechanisms All relations (including functions) in SUMO have a type signature. As a consequence, we don't need an explicit syntax for types/sorts of variables, and can deduce them automatically. We can have classes as well as instances as arguments. The `domain` and `range` relations are meta-predicates that direct the Sigma translators to state that arguments to a given relation (or the return type of a function, respectively) are instances of a given type. The `domainSubclass`, and `rangeSubclass` relations state that arguments to a given relation (or the return type of a function, respectively) are a given class or one of its subclasses. For example

```
(domain DensityFn 1 MassMeasure)
(domain DensityFn 2 VolumeMeasure)
(instance DensityFn BinaryFunction)
(range DensityFn FunctionQuantity)
```

`DensityFn` is a `BinaryFunction` that takes an instance of a `MassMeasure` and a `VolumeMeasure`, respectively, as its first and second arguments. In

```
(domainSubclass typicalPart 1 Object)
(domainSubclass typicalPart 2 Object)
(instance typicalPart BinaryPredicate)
```

the first and second arguments to the `typicalPart` relation are of the class `Object` or one of its subclasses.

THF Translator Below by *SUMO objects (SOs)* we mean arbitrary SUMO classes and instances.

Our translator maps all SUMO objects to sets in HO TG set theory. The subclass relation is translated as inclusion and the instance relation as membership. SOs that are potentially large such as abstract, mathematical and related SOs thus become sets that may live in higher TG universes.

SOs can be applied to other SOs and variables, creating terms and formulas. Such SOs will be sets that encode relations and functions. Their application to other SOs is the corresponding application of the set theoretical functional and relational sets to other sets. To handle variable arities and row variables, arguments are always appended together into lists.

SUMO quantifiers and logical connectives are mapped directly to their FOL counterparts. Applications that are at predicate positions in formulas are casted by a special *bp* predicate into propositions.

To illustrate a significant higher order construct in SUMO, consider the following problem that uses an axiom with `KappaFn`, which defines a class on the fly, without the need to reify it.

```
(<=>
    (totalFacilityTypeInArea ?AREA ?TYPE ?COUNT)
    (cardinality
        (KappaFn ?ITEM
            (and
                (instance ?ITEM ?TYPE)
                (located ?ITEM ?AREA))) ?COUNT))

(instance DejvickaStation TrainStation)
(located DejvickaStation PragueCzechRepublic)
(instance HradCanskaStation TrainStation)
(located HradCanskaStation PragueCzechRepublic)

Q: (totalFacilityTypeInArea ?AREA ?TYPE ?COUNT)
A: [?AREA=PragueCzechRepublic,?TYPE=TrainStation,?COUNT=2]
```

The first axiom states that for the ternary relation of `totalFacilityTypeInArea`, which related an area, a class of `Object` and a count of those objects within that area, it is equivalent to the cardinality of the instances of the class that are defined to be instances of the same type, and present within a particular `?AREA`.

We should be able to ask what relations are deducable for `totalFacilityTypeInArea` and get the answer that, for this knowledge base, there are two instances of `TrainStation` that are known to be in the `CzechRepublic`.

```
(SLEEPING c= PSYCHOLOGICALPROCESS).
(ASLEEP :e CONSCIOUSNESSATTRIBUTE).
((bp (ATTRIBUTE (cons v_AGENT (cons ASLEEP nil))))
\/ (bp (ATTRIBUTE (cons v_AGENT (cons AWAKE nil))))
-> (bp (ATTRIBUTE (cons v_AGENT (cons LIVING nil))))).
```

are the translations of:

```
(subclass Sleeping PsychologicalProcess)
(instance Asleep ConsciousnessAttribute)
(=>
    (or
        (attribute ?AGENT Asleep)
        (attribute ?AGENT Awake))
    (attribute ?AGENT Living))
```

Analyzing Proof Components*

Karel Chvalovský and Josef Urban

Czech Technical University in Prague, Czech republic,
karel@chvalovsky.cz and josef.urban@gmail.com

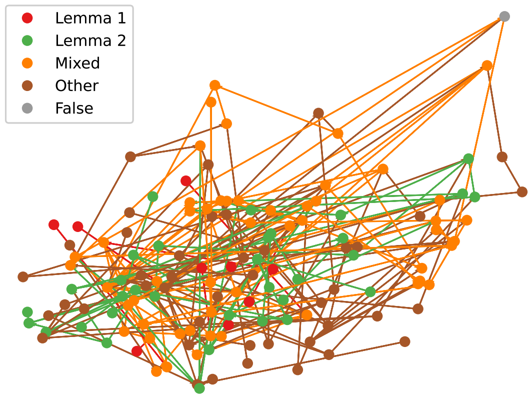
Modern automated theorem provers (ATPs) for first-order logic, such as E [6] or Vampire [5], usually start a proof search by trying to classify the input problem so that they can use strategies that fit the problem best. However, not only are there different problems, each problem also commonly consists of various (semi)independent parts that should be treated differently. The provers are able to treat the most common special cases, like equalities or arithmetic, specifically, but no general approach exists. Although a guidance based on various machine learning models provides a partial solution to these problems, because such models can, in principle, dynamically adapt their guidance as the proof search evolves, it has only been of limited success so far. Therefore, it seems natural to study these issues directly to better understand them.

In [2], we made some initial steps in this direction; we try to detect components in unsuccessful proof attempts, run these components individually, and then use the best obtained clauses from these individual runs to complete the proof. This approach seems promising, but it is hard to analyze what is actually happening inside. A key issue being that detecting components in our setting is an unsupervised task with no ground truth available. For that reason, we decided to create a dataset that allows us to better analyze this behavior with the available training and testing examples.

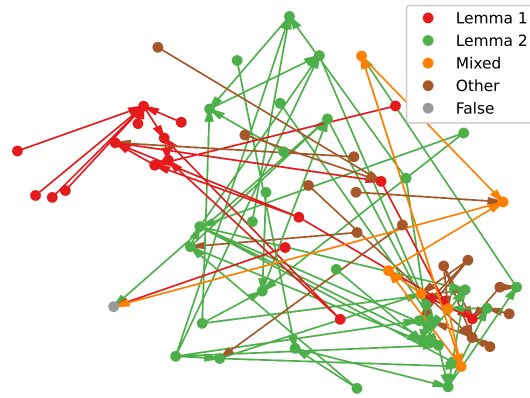
Although there are various mathematically well-defined ways to combine two (or more) components into one problem, we take advantage of already available mathematical formal libraries and created a natural dataset with possibly many types of components. We take the Mizar Mathematical Library, which can be exported to first-order logic, and hence have a long list of problems (theorems or lemmata) with their dependencies. Many of these problems can be proved by ATPs (various versions of E, ENIGMA, Vampire,...) using a reasonable premise selection. Moreover, these proofs can be analyzed, and we obtain for each problem a minimized set (or more sets) of dependencies required for the proof. For example, we have a theorem T (exported from Mizar) that is automatically provable from the lemmata $L_1, \ldots, L_n$ (also exported from Mizar) by an ATP. Assume that L_1 and L_2 are two lemmata from different mathematical libraries. Furthermore, L_1 is provable from $L_1^1, \ldots, L_1^m$ and L_2 is provable from $L_2^1, \ldots, L_2^k$, where $L_1^1, \ldots, L_1^m$ and $L_2^1, \ldots, L_2^k$ have no overlap or insignificant overlap. Then we can expand L_1 and L_2 in T, so a new expanded problem is to prove T from $L_1^1, \ldots, L_1^m, L_2^1, \ldots, L_2^k, L_3, \ldots, L_n$. This new problem has well-defined components (the expansions of L_1 and L_2) as we wanted. In fact, we can produce many such datasets depending on requirements like number of components, overlap of components, size of expansions, ...

It is possible to understand exported Mizar problems as a large graph in which the nodes are first-order formulae and the edges show dependencies; an edge from L to T means that L was used in a proof of T. In fact, we usually have more types of edges because we have more minimized solutions to a problem. With such a graph, we can easily produce the expansions described previously. In our particular case, we obtain a graph with 67795 nodes, where 29687 nodes are leaves (having no dependencies) and 19334 nodes are roots (not used in other proofs). Although the number of roots may seem a bit high, it is because we are only interested in

*Supported by the Czech project AI&Reasoning CZ.02.1.01/0.0/0.0/15_003/0000466 and the European Regional Development Fund.



(a) A common situation.



(b) A good situation.

Figure 1: Processed clauses (nodes) and their derivations (arrows) in two successful runs are shown. Red (green) nodes correspond to clauses obtained from expanded Lemma 1 (2) including derived clauses that use only expanded Lemma 1 (2) as their dependencies. Brown clauses depend only on clauses different from Lemma 1 and 2 expansions in their derivations. Orange clauses depend on a mixture of previous types of clauses. An arrow from c_1 to c_2 means that c_1 was used in the derivation of c_2. The final contradiction is the gray node.

problems that can be proved by ATPs, and hence many real dependencies may be lost. An advantage of such a graph is that train/test splits are easy to define. One way is that we, for example, randomly split root nodes into two parts—train and test roots. All nodes having a path to a train root (or are train roots themselves) are now called train nodes. On the other hand, remaining nodes, which have no path to a train node (and hence they are test roots or have a path to a test root), are test nodes. Clearly, there are more train nodes than test nodes. We obtain a fair split of train and test datasets by expanding only train and test nodes, respectively. In our example, we may obtain 275K possible expansions for the train dataset and 5K possible expansions for the test dataset; however, they correspond to approximately 3K unique problems on the train dataset and roughly 100 unique problems on the test dataset, because there are many possible ways to expand a problem.

Having such a dataset enables further analysis of algorithms used to obtain components in [2] and compare them with intended components (our expansions). A clause selection guidance based on GNNs[1], see [4], allows one to extract representations of individual clauses in a latent space and then identify components there, moreover, we can visualize them and also display intended components, see Figure 1. In Figure 1a we see a common situation where it is hard to distinguish both expansions. Sometimes, like in 1b, they are reasonably separated. However, the former situation is much more common than the latter. This improves when we retrain our algorithms using the new dataset, however, preliminary results show that further analysis of our pipeline is still necessary.

Clearly, our dataset is useful for many other experiments, including conjecturing (reconstructing L_1 and L_2), and we will present some preliminary results in this direction.

[1]Note that GNNs are useful for similar problems that involve components, as was recently observed [7] that GNNs align with dynamic programming. Instead of learning algorithms, for example, in combinatorial optimization, from scratch, it is beneficial to learn their individual subroutines (modules) separately, cf. [1, 3].

References

[1] Quentin Cappart, Didier Chételat, Elias B. Khalil, Andrea Lodi, Christopher Morris, and Petar Veličković. Combinatorial optimization and reasoning with graph neural networks. In Zhi-Hua Zhou, editor, *IJCAI-21*, pages 4348–4355. International Joint Conferences on Artificial Intelligence Organization, 8 2021. Survey Track.

[2] Karel Chvalovský, Jan Jakubův, Miroslav Olšák, and Josef Urban. Learning theorem proving components. In Anupam Das and Sara Negri, editors, *TABLEAUX 2021*, pages 266–278, Cham, 2021. Springer International Publishing.

[3] Andrew Dudzik and Petar Velickovic. Graph neural networks are dynamic programmers. *CoRR*, abs/2203.15544, 2022.

[4] Jan Jakubův, Karel Chvalovský, Miroslav Olsák, Bartosz Piotrowski, Martin Suda, and Josef Urban. ENIGMA anonymous: Symbol-independent inference guiding machine (system description). In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 448–463. Springer, 2020.

[5] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *CAV*, volume 8044 of *LNCS*, pages 1–35. Springer, 2013.

[6] Stephan Schulz. System description: E 1.8. In Kenneth L. McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *LPAR*, volume 8312 of *LNCS*, pages 735–743. Springer, 2013.

[7] Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S. Du, K. Kawarabayashi, and Stefanie Jegelka. What can neural networks reason about? In *ICLR*, 2020.

Scaling ℕaproche

Adrian De Lon and Peter Koepke

University of Bonn, Germany, https://www.math.uni-bonn.de/ag/logik/

The ℕaproche [1] (Natural proof checking) theorem prover demonstrates that it is possible to write non-trivial fully natural machine-verified proofs in a controlled natural language for mathematics: every statement is a statement of the common language of mathematics, and the argument uses familiar declarative proof structures. ℕaproche formalizations can be written in a LaTeX format which allows immediate mathematical typesetting. The system is available within the Isabelle prover platform [3], including some example formalizations which partly resemble undergraduate mathematical material.

Natural mathematical texts assume that the reader could in principle fill in lots of technical detail and implicit proof steps. Likewise ℕaproche's aspiration to process similar texts requires a high degree of "machine intelligence" which is achieved through careful organization of proof tasks which are given to strong automated provers. Modeling the softly typed natural language is carried out with first-order defined types so that subtypes and partial functions with guards on definedness are available. Unfortunately type-checking of complex terms presently spawns a number of general first-order proof tasks which in general can only be discharged at run-time, i.e., during the proof process. "Human-sized" proof steps also stretch the abilities of ATPs. At the moment basically all previous statements are made into premisses of proof tasks. Often reformulations of proofs are necessary to help the ATP whilst keeping the naturalness and readability of a text.

As improvements to the ℕaproche system are allowing longer and interlinked formalizations, the above-mentioned difficulties are mounting up. ℕaproche's predecessor SAD was only able to process and check mathematical "miniatures" which each came along with purpose-built ontological preliminaries. Small text sizes resulted in small proof tasks which did not overwhelm the ATPs. With the incorporation of SAD into ℕaproche we have gone from miniatures to chapter sized texts and recently to small libraries to be re-used in other formalizations. Obviously this led to a steep increase in the number of possible premisses which resulted in several types of prover problems:

Check times. Many ℕaproche formalizations require several minutes checking time on standard laptops but some texts need even some hours. Usually more time is spent on type checking (which is called ontological checking) than on the logical checking of proof steps. These problems got worse after replacing a crude and potentially contradictory underlying set theory by a Kelley–Morse-style ontology which distinguishes between sets and classes. This, however, leads to many additional proof in ontological checking: when unions $A \cup B$ have been originally defined for classes, then the union $a \cup b$ of two sets requires proofs that a and b are also classes.

Erratic prover behaviour. Extensive texts usually contain many function symbols which an ATP can use for unification-based proof searches. Automatic proof steps can be confused by adding symbols which in principle are not connected with the actual prover task.

Ontological checking. Although type-checking should normally be a mild proving task, the use of general purpose ATPs in a large number of checks increases the probability that an ATP will get on a wrong track and miss obvious arguments. Sometimes we could only get ontological correctness by putting some type information into a statement immediately before the statement with the problematic term - something one would hardly find in natural mathematical texts.

Stability of formalizations. We use E [7, 8] as the main external prover. Many Naproche formalizations are written against a specific version of E, with proof steps chosen accordingly.

Replacing that version of E with a different version (or a different ATP entirely) typically results in some proof tasks failing, even when the different version performs better overall. This results in a significant maintenance burden, particularly for larger formalizations.

In our talk we shall report on ongoing work aiming to avoid or mitigate these scaling issues.

1. Tracing prover behaviour: when do provers latch on to obviously hopeless search path? What can we learn from concrete examples?

2. Experimenting with multiple ATPs: where lie the relative strengths of individual ATPs in the context of Naproche? For example we have seen proof tasks where the addition of a single irrelevant hypothesis makes the task impossible for some provers, while others can solve the problem within roughly one second. So far we have used E, iProver [4, 2], SPASS [9, 11], and Vampire [10, 5] for our experiments.

3. Adding premise selection (e.g. starting with a MePo-like [6] filter) to Naproche.

4. Simplifying or reducing ontological checking. Most ontological checks should not amount to full first-order proof tasks.

5. Experiments with different ontologies (Kelley–Morse vs. Zermelo–Fraenkel) show that there are trade-offs between richer ontologies (adding classes and/or urelements) on the one hand and burdening users with proof obligations as well as cluttering exported proof tasks with type guards on the other.

6. Reducing checking times through improvements to the architecture of Naproche: increasing parallelism, caching, and more.

We shall also talk more generally about the potential of the Naproche approach with respect to article-sized and textbook-sized formalizations.

References

[1] De Lon, A., Koepke, P., Lorenzen, A., Marti, A., Schütz, M., Wenzel, M.: The Isabelle/Naproche natural language proof assistant. In: Platzer, A., Sutcliffe, G. (eds.) Automated Deduction – CADE 28. pp. 614–624. Springer International Publishing, Cham (2021)

[2] Duarte, A., Korovin, K.: Implementing superposition in iProver (system description). In: Peltier, N., Sofronie-Stokkermans, V. (eds.) Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12167, pp. 388–397. Springer (2020). https://doi.org/10.1007/978-3-030-51054-1_24, https://doi.org/10.1007/978-3-030-51054-1_24

[3] Isabelle contributors: The Isabelle2021 release (February 2021), https://isabelle.in.tum.de

[4] Korovin, K.: iProver – a theorem prover for first-order logic with support for arithmetic reasoning, http://www.cs.man.ac.uk/~korovink/iprover/

[5] Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: CAV 2013. pp. 1–35 (2013)

[6] Meng, J., Paulson, L.C.: Lightweight relevance filtering for machine-generated resolution problems. J. Appl. Log. **7**, 41–57 (2009)

[7] Schulz, S.: The E theorem prover, https://eprover.org

[8] Schulz, S., Cruanes, S., Vukmirović, P.: Faster, higher, stronger: E 2.3. In: Fontaine, P. (ed.) Proc. of the 27th CADE, Natal, Brasil. pp. 495–507. No. 11716 in LNAI, Springer (2019)

[9] SPASS contributors: SPASS workbench, https://www.mpi-inf.mpg.de/departments/automation-of-logic/software/spass-workbench

[10] Voronkov, A., Kovács, L., Reger, G., Suda, M., Rawson, M., Bhayat, A., Schoisswohl, J., Rath, J., Hozzova, P.: The Vampire prover, https://vprover.github.io/

[11] Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: SPASS version 3.5. In: Schmidt, R.A. (ed.) Automated Deduction – CADE-22. pp. 140–145 (2009)

Towards neuro-symbolic conjecturing

Sólrún Halla Einarsdóttir, Moa Johansson, and Nicholas Smallbone

Chalmers University of Technology, Gothenburg, Sweden.
{slrn, moa.johansson, nicsma}@chalmers.se

Abstract

Theory exploration systems automatically generate mathematical conjectures, by exploring a set of terms of interest. This search is expensive for large theories, as the set of terms becomes large. We describe ongoing work in combining data-driven and symbolic methods for automated conjecturing, where the data-driven part should identify which kinds of conjectures are likely to be useful and restrict the symbolic search to those ones. As a first step, we have extracted a dataset of lemma templates from Isabelle's Archive of Formal Proofs[1].

1 Introduction

Theory exploration is a symbolic technique for automated conjecturing based on testing [7]. It has been used to successfully discover, for example, lemmas needed in automated (co)-inductive provers [5, 1]. Our theory exploration system QuickSpec [7], takes as input a number of functions and datatypes, and builds terms of increasing size. The search space is managed by using already discovered properties to avoid larger terms that could be reduced or subsumed by something already known. While this works well for smaller signatures (up to around 10–20 functions) and terms up to about 10 symbols, it eventually runs into exponential blow up. To address this, we developed a variant of QuickSpec called RoughSpec [2], which restricts the search space to properties of specific shapes using *templates*. For example, the template $?F(?F(X,Y),Z) = ?F(X,?F(Y,Z))$ describes an associative binary function $?F$. Currently, the human user decides which templates to use.

We plan to instead select templates automatically using a data-driven approach. As a first step towards this goal, we have collected and started to analyze a large dataset of lemmas from Isabelle's Archive of Formal Proofs. The long term aim is to build a neuro-symbolic system for conjecturing, where given a theory, a machine learning system selects the most promising templates, and a symbolic system fills in the templates to produce conjectures, discarding any conjecture which is trivial, trivially false, or already known. Our hypothesis is that this approach combines the best of the machine learning and symbolic approaches: machine learning to learn which parts of the search space to focus on, and symbolic methods to reason about and evaluate specific conjectures.

2 A Library of Lemma Templates

In our previous work [2], our theory exploration system RoughSpec required the user to provide templates for the properties they were looking for. We also provided a small collection of "default" templates describing some properties we guessed might be useful for theory exploration based on our intuitions and experience. This collection included templates for properties such as commutativity and distributivity. In order to gain a more robust empirical understanding of

[1] https://www.isa-afp.org/index.html

what kinds of templates are useful and to provide a dataset for data-driven experiments we have mined equational lemma templates from the Archive of Formal Proofs(AFP). The AFP contains 676 entries from 425 authors, containing almost 200,000 lemmas and more than 3 million lines of code. The entries consist of proof formalizations from a variety of areas of Computer Science, Logic and Mathematics and we believe they are a good source of interesting and useful lemmas, as the lemmas we find there are handwritten as part of proofs that Isabelle users have seen a reason to formalize.

2.1 Preliminary results

We have collected and performed preliminary analysis on a dataset containing 22,767 equational lemmas, extracted from 2169 different theory files from 611 AFP entries. For each extracted lemma we generated a template representation of the lemma statement, showing the statement's term structure with function and variable names abstracted away but using integer labels to keep track of function symbols and variables that occur more than once. The dataset along with the code used to generate it is available at: `https://github.com/solrun/LibraryOfLemmas`.

These 22,767 lemmas are captured by 6567 different templates. In Figure 1 we can see that a small number of templates occur very frequently while the majority occur very seldom with 4099 templates occurring only once. The 10 most frequent templates together describe 3057 lemmas or 13.5% of the lemmas in our set, while more than 50% of the lemmas can be described using only 266 of the 6567 templates. This supports our hypothesis that only a smaller number of templates is needed to discover many lemmas using template-based conjecturing.

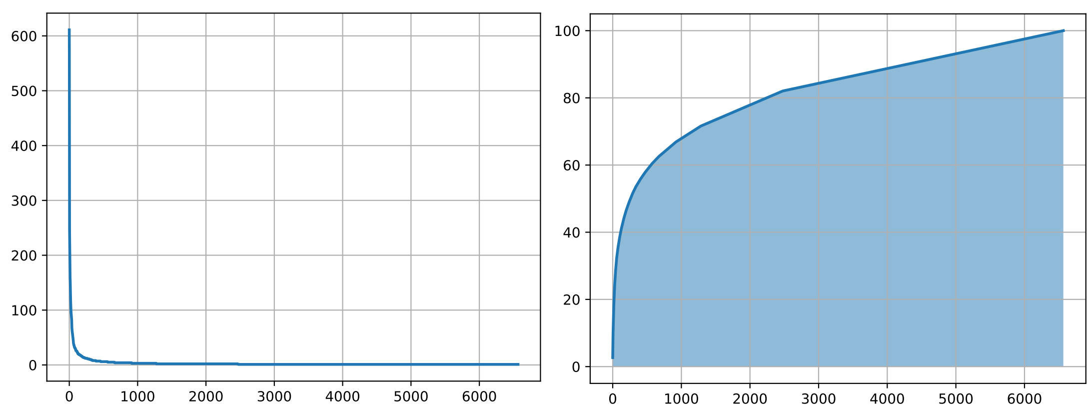


Figure 1: Left: Number of lemmas per template, sorted by frequency. Right: Cumulative percentage of lemmas in the dataset covered by most frequent templates.

Table 1 shows the top 10 most frequently occurring templates in our dataset, where # lemmas represents the number of lemmas matching the template, # thys is the number of different theory files it occurs in and # sessions is the number of different AFP sessions it occurs in. Template holes are represented by a question mark followed by a capitalized name, while variables are represented by a capitalized name. Among these common templates, we see a lot of similarity: many of the templates in Table 1 resemble each other, differing only in the number of variables or the order of application. Our hypothesis is that we can define a smaller set of "supertemplates" describing families of similar templates that can be generalized to describe the different family members, where using those templates and their generalizations we can

	Template	# lemmas	# thys	# sessions
1	?F (?G X Y) = ?H (?F X) (?F Y)	611	261	172
2	?F X = ?G (?H X)	566	265	169
3	X = ?F (?G X)	340	191	139
4	?F X = ?F (?G X)	280	149	118
5	X = ?F ?G X	247	136	98
6	?F (?G X Y) Z = ?H (?F X Z) (?F Y Z)	233	90	70
7	X = ?F X ?G	210	132	103
8	?F X (?G Y Z) = ?H (?F X Y) (?F X Z)	194	90	74
9	?F = ?G (?H X)	192	65	56
10	?F = ?G ?H X	184	110	85

Table 1: Top 10 most frequently occurring templates in the dataset.

generate a large proportion of the lemmas we need. For example (1) (6) and (8), (3) (5) and (7), and (9) and (10) should be grouped together and described by common "supertemplates". This would further reduce the space of templates to be searched over by RoughSpec.

In our previous work [2], we defined a set of 10 default templates capturing very common properties which we found useful in our case studies. Comparing these to the most frequent templates as shown above, we see that 4 out of our 10 default templates are also in the 10 most frequently occurring templates in our dataset. Of the remaining six default templates one did not show up at all, and the other occur in places 20–388. The second most commonly occurring template in the dataset, ?F X = ?G (?H X), is in a style we had previously disregarded as being too general to be suited to template-based theory exploration, but seeing how common this exact form of equivalence template seems to be we will definitely try out using it in future experiments. The differences between our collection of default templates and the most common templates in the dataset show the value of collecting a dataset for empirical evaluation.

Extending the dataset. We are currently expanding this dataset to also contain non-equational lemmas, such as conditionals, inequalities, and predicates. We also plan to extend the template language to cover e.g. lambda abstractions and quantifiers. With these extensions we should be able to cover all the lemmas in the AFP. Adding more data concerning for example the topic of the theory where the lemma in question is defined and used or the function definitions involved may also prove necessary in order to use this dataset to learn what templates are useful in various theorem proving contexts.

3 Future steps and related work

Having compiled a library of lemma templates, our next steps will be to analyze the data available there and apply it in the context of theory exploration and theorem proving. Our aim is to develop machine-learning based methods to make helpful template suggestions for a given set of functions. We envisage this implemented as a machine learning model trained to predict likely useful templates, given a suitable representation of a theory (i.e. a set of definitions of datatypes, functions and perhaps already known properties). These templates are then passed to a symbolic theory exploration systems (RoughSpec), which instantiates, tests and possibly proves properties before presented to the user. This has some similarities to neuro-symbolic program synthesis systems like DreamCoder [3], which use a neural network to predict a symbolic program, given a set of input-output examples. We could also use clustering methods to group together templates that are often seen together in the same theory formalization and then suggest templates based on lemmas that have already been defined by the user or found by exploration.

There have been recent attempts to use large language models for conjecturing tasks [8, 6]. A problem here is that the output typically contains a mixture of interesting theorems, non-theorems that "look like" theorems as well as many copies and alpha-renamings of lemmas occurring in the training data. Symbolic theory exploration methods are usually better at targeting more specifically novel conjectures, but struggle with large scale theories instead.

We believe that the most promising way forward is a combination of neural and symbolic methods, where the neural part makes suggestions of potential analogies to similar theories seen before, while they symbolic part fills in the details in such a way that redundant conjectures are avoided. Heras et al. demonstrated a prototype system similar to what we propose, for suggesting lemmas by analogy via templates [4]. Here, the user is supposed to be wanting to prove a particular conjecture, and asks the system for analogous prior theorems. If such a similar theorem exists (determined by data-driven methods), and its proof uses a lemma, then the lemma was generalized into a template. This template was then instantiated using symbols relevant to the new proof attempt at hand. Our proposed system differs in that we do not want to rely on a particular proof attempt, but rather suggest interesting conjectures based on the functions and datatypes in scope, and how they are defined.

References

[1] S. H. Einarsdóttir, M. Johansson, and J. Å. Pohjola. Into the infinite - theory exploration for coinduction. In *Proceedings of AISC 2018*, pages 70–86, 01 2018.

[2] S. H. Einarsdóttir, N. Smallbone, and M. Johansson. Template-based theory exploration: Discovering properties of functional programs by testing. In *IFL 2020: Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages*, IFL 2020, page 67–78, New York, NY, USA, 2020. Association for Computing Machinery.

[3] K. Ellis, C. Wong, M. Nye, M. Sablé-Meyer, L. Morales, L. Hewitt, L. Cary, A. Solar-Lezama, and J. B. Tenenbaum. DreamCoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, page 835–850, New York, NY, USA, 2021. Association for Computing Machinery.

[4] J. Heras, E. Komendantskaya, M. Johansson, and E. Maclean. Proof-pattern recognition and lemma discovery in ACL2. In *Proceedings of LPAR*, 2013.

[5] M. Johansson, D. Rosén, N. Smallbone, and K. Claessen. Hipster: Integrating theory exploration in a proof assistant. In *Proceedings of CICM*, pages 108–122. Springer, 2014.

[6] M. N. Rabe, D. Lee, K. Bansal, and C. Szegedy. Mathematical reasoning via self-supervised skip-tree training. In *Proceedings of ICLR*, 2021.

[7] N. Smallbone, M. Johansson, K. Claessen, and M. Algehed. Quick specifications for the busy programmer. *Journal of Functional Programming*, 27, 2017.

[8] J. Urban and J. Jakubův. First neural conjecturing datasets and experiments. In *Proceedings of CICM*, 2020.

Program Synthesis from Integer Sequences: Initial Self-Learning Run on the OEIS *

Thibault Gauthier

Czech Technical University in Prague, Prague, Czech Republic
email@thibaultgauthier.fr

Abstract

Through self-learning, our system discovers in three weeks programs that generate the first 16 numbers of more than 50000 OEIS sequences

1 Introduction

In this work, we propose to rely on a "self-learning" system (a system that learns from its own searches) to create programs generating sequences from the On-Line Encyclopedia of Integer Sequences (OEIS) [6] and beyond. Program synthesis for different domains (e.g. operations on lists) has been attempted by inductive logic programming systems (such as Popper [5]) and reinforcement learning systems (such as DeepCoder [1]). Within the theorem proving community, the development of methods for term synthesis has been explored in inductive theorem proving [4] and in counterexample generators [2, 3].

Here is how our self-learning approach creates programs for OEIS sequences. Its self-learning loop consists of two alternating phases: a search phase and a learning phase. Initially, our search discovers some solutions by randomly building programs and checking if they generate OEIS sequences. From those solutions, a tree neural network is trained to predict what the right building action is, given a target sequence and a partially built program. The next search is then guided by the statistical correlations learned by the network, usually producing even more solutions. One iteration of the self-learning loop is called a generation. Note, as it is usual for reinforcement learning systems, that our approach is completely unsupervised. That is to say, the system is never told the corresponding program for a particular sequence but has to discover it through guided search.

2 Programming Language

Our language contains the tokens $0, 1, 2, +, -, x, i, \times, div, mod, cond, \lambda, loop, compr$ which follow the semantics of Standard ML except for $cond, loop, compr$ defined by:

$$
\begin{aligned}
cond(a,b,c) &:= \text{if } a \leq 0 \text{ then } b \text{ else } c \\
loop(f,a,b) &:= b && \text{if } a \leq 0 \\
& \quad\; f(loop(f,a-1,b),a) && \text{otherwise} \\
compr(f,a) &:= \text{failure} && \text{if } a < 0 \\
& \quad\; min\{y \mid y \geq 0 \wedge f(y,0) \leq 0\} && \text{if } a = 0 \\
& \quad\; min\{y \mid y > compr(f,a-1) \wedge f(y,0) \leq 0\} && \text{otherwise}
\end{aligned}
$$

*This work was supported by the Czech Science Foundation project 20-06390Y.

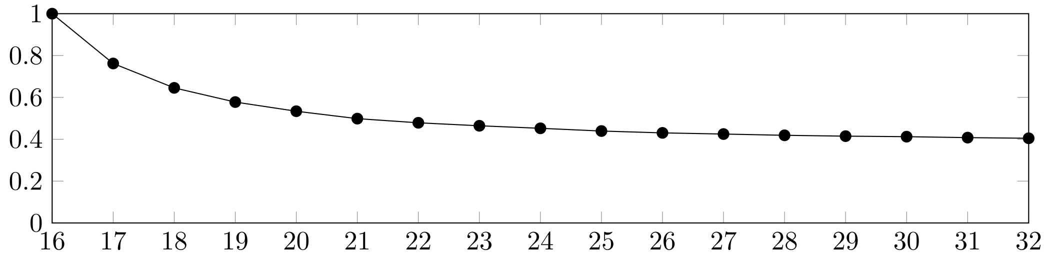


Figure 1: Percentage of sequences n-covered among 16-covered sequences with at least 32 elements

Table 1: Solutions for famous OEIS sequences

Sequence	Program
Catalan numbers	loop(f,x,1) where $f(x',i') = ((i' \times 2 - 1) \times x' \times 2)\ \mathit{div}\ (i'+1)$
Pseudo-prime numbers	$\mathit{compr}(\{x' \geq 0 \mid (2^{x'+2} - 2)\ \mathit{mod}\ (x'+2) = 0\},\ x) + 2$ where $2^{x'+2} - 2 := \mathit{loop}(\lambda(x'',i'').\ (x''+1) \times 2,\ x',\ 2)$
Prime characteristic function	$(\mathit{loop}(\lambda(x,i).i \times x, x, x)\ \mathit{mod}\ (x+1))\ \mathit{mod}\ 2$

3 Results

The code for our project is available in this repository [7]. A user can also test our system using the web interface `http://grid01.ciirc.cvut.cz/~thibault/qsynt.html`.

We report on the number of solutions found during self-learning. At generation 0, the search finds 5247 16-solutions (covering the first 16 elements of an OEIS sequence) using a tree neural network initialized with random weights. After generation 11, we get 37400 16-solutions. After 100 generations, more than 50000 OEIS sequences had their first 16 elements generated by at least one program. Figure 1 measures our programs' ability to cover sequences for increasing value of n. Extrapolating the plot, we can conjecture that the percentage of solutions that generalize to arbitrary inputs converges towards 40%. In Table 1, solutions for famous sequences are presented. Each of these three solutions was manually proven to match the description given by OEIS editors for the corresponding sequence.

In the future, our priority will be to increase the number of tested inputs before declaring a program to be a solution. With this change, we should observe a decrease in the number of solutions but those solutions will be more likely to generalize to larger inputs.

References

[1] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[2] Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, pages 131–146, 2010.

[3] Lukas Bulwahn. The new Quickcheck for Isabelle - random, exhaustive and symbolic testing under one roof. In *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*, pages 92–108, 2012.

[4] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Automating inductive proofs using theory exploration. In Maria Paola Bonacina, editor, *Conference on Automated Deduction (CADE)*, volume 7898 of *LNCS*, pages 392–406. Springer, 2013.

[5] Bruce Nielson and Daniel C. Elton. Induction, popper, and machine learning. *CoRR*, abs/2110.00840, 2021.

[6] Neil J. A. Sloane. The on-line encyclopedia of integer sequences. In Manuel Kauers, Manfred Kerber, Robert Miner, and Wolfgang Windsteiger, editors, *Towards Mechanized Mathematical Assistants, 14th Symposium, Calculemus 2007, 6th International Conference, MKM 2007, Hagenberg, Austria, June 27-30, 2007, Proceedings*, volume 4573 of *Lecture Notes in Computer Science*, page 130. Springer, 2007.

[7] Gauthier Thibault. Software accompanying the paper "Program Synthesis for the OEIS". `https://github.com/barakeel/oeis-synthesis`, 2022.

Project Proposal: Formal Ethics Ontology in SUMO*

Zarathustra Amadeus Goertzel[1], Adam Pease[2], and Josef Urban[1]

[1] Czech Technical University in Prague, Czech Republic
[2] Articulate Software, San Jose, CA, USA

We propose a project to formalize a portion of ethical theory. AI ethics and AI safety are growing fields that aim to study how AI can be used in safe and beneficial manners for humans. There are arguments for shifting the focus from AI ethics to computational ethics (see Segun [14]), which is the field of studying how to make ethical decisions computationally. Ethics can be seen as encompassing many approaches to the "human safety problem" and porting the lessons learned in the field should help clarify the domain of AI and computational ethics. In this proposal, the three primary ethical paradigms, utilitarianism, deontology, and virtue ethics, will be expressed in a multi-agent reinforcement learning (RL) model.

The other goal is to formally define ethics and these paradigms in SUMO. The Suggested Upper Merged Ontology (SUMO) [9,10] is a comprehensive ontology of around 20,000 concepts and 80,000 hand-authored logical statements in a higher-order logic that has an associated integrated development environment called Sigma [11][1] that interfaces to leading theorem provers such as E [13] and Vampire [5]. Previous work in logical formalizations of ethical theories [4] has been limited to work strictly on ethics itself without support from a larger formalization of objects, human actions, and events that form the situations in which ethical decisions take place. SUMO provides that context and allows us the potential to create a more practical formalization that is situated in the real world, with its complexity of choices and influences.

Summary of Ethical Paradigms: Ethics is "the normative science of the conduct of human beings living in society, which judges this conduct to be right or wrong, to be good or bad, or in some similar way" [7].

The paradigm of virtue ethics specifies the psychological traits of an agent such that the agent's behavior will be good, deontology seeks to develop rules by which to judge behavior, and utilitarianism asserts that the goal is to maximize well-being and minimize suffering among all involved in a society; any effective action to this end is judged to be good [2].

There have been many attempts to justify the particular paradigms and to argue from first principles that such-and-such a way is the "correct" or "rational" way to judge conduct. Virtues and deontological rules are often implicitly justified as necessary for humans to harmoniously and cooperatively live in a society. There have been many debates on whether ethical judgments are objectively universal or simply subjective assertions. The Stanford Encyclopedia of Philosophy (SEP) article on *The Definition of Morarality* states that normative claims on how agents 'ought' to act are usually justified as codes of conduct that "would be put forth by all rational people" [3]. Kant distinguished *hypothetical* and *categorical* imperatives and tried to argue for the truth of some categorical imperatives that apply for all subjective goals, which involved the claim that all people by 'natural necessity' desire their own happiness. Happiness as an axiomatic goal also appears in Aristotle's virtue ethics as '"eudaimonia" (a state of well-being) [6] and Mill, the author of Utilitarianism [8], considers the fact that "happiness is good" to be self-evident and without further proof [2].

*Supported by the ERC Consolidator grant no. 649043 AI4REASON and by the Czech project AI&Reasoning CZ.02.1.01/0.0/0.0/15_003/ 0000466 and the European Regional Development Fund.

[1] https://www.ontologyportal.org

[2] E.g., "be an honest person", "do not lie", and "say whatever will bring about the best consequences."

Grounding ethical theory in a formal ontology should help to make clear what can and cannot be said about objective norms and subjective assessments in multi-agent settings, such as human society, thus this proposal focuses on developing a common framework that is agnostic to conjectures about the nature of 'value' by which meta-ethical or axiological axioms may be proposed, their consequences explored (with automation), and proofs attempted.

Multi-agent reinforcement learning model: The multi-agent RL model includes a set of states (S) of the environment and agents ($N = \{1, \ldots, n\}$); a set of actions for each agent ($A(s) = A_1(s) \times A_n(s)$); a stochastic transition structure from actions to probability measures over the states ($T(s, a)$); a reward function for each agent that depends on the old state, the actions, and the new state ($r_i(s, a, s')$); and the agent's policy that outputs an action for each state ($\pi_i(s)$) [1, 16]. The goal is for agents to maximize their expected reward (contingent on what other agents do), which may be technically enough to represent diverse value landscapes [15].

The ethical paradigms can be expressed in this model by the additional of a societal coherence constraint, v, that must be (approximately) satisfied by each agent i while maximizing the expected reward r_i.

1. Utilitarianism holds that for every agent, i, $v(s, a, s') = \sum_i r_i(s, a, s')$ should be maximized in each step.

2. Deontology encodes ethical rules into an evaluation of actions, $v(s, a, s') \rightarrow \{good, bad\}$, and holds that agents should always take '*good*' actions.

3. Virtue ethics judges psychological processes with $v(s, a, s') \rightarrow \{virtous, vicious\}$ and holds that should maintain 'virtuous' inner states and implementations.

Additionally, utilitarianism under the term 'consequentialism' often stipulates that only the consequences, s', matter for v and r_i. Deontology emphasizes the actions, a, and virtues emphasize the state from which action is taken, s, suggesting that the paradigms are complementary. Specifying the interrelationships between these approaches may mirror the Curry–Howard correspondence between logic and programming languages.

Relation to AI Ethics and Safety: Humans have developed much common sense about ethical behavior, and AI safety research often focuses on the RL paradigm, so formalizing the main paradigms in the RL model should help leverage this knowledge when studying how to design and cooperate with AI. For AI ethics applied to military uses, conditional reasoning is valuable, such as, "if one believes the use of remotely piloted drones to be ethically justified, then I present an argument one should also support the use of automated weapon systems" [12]. The combination of international codes of ethics with automated reasoning and councils of ethicists is also mentioned. There is room for neuro-symbolic integration as AI systems learn to behave ethically. This suggests that for some purposes, a common ontology should be helpful.

Initial work in SUMO focuses on formalizing a standard ethical dilemma about organ transplants. Suppose there is a surgeon, a healthy patient, and a patient who will die without a kidney transplant. The dilemma is that utilitarianism superficially recommends the surgeon to perform the kidney transplant from the healthy patient without regard to consent, for by most metrics, two non-terminally ill people will result in greater happiness and less pain than one. Deontologically, there are ethical codes such as "first, do no harm" and to require "informed consent" before operating. The formalization should allow for computer-assisted exploration of what outcomes different axiomatic principles and definitions will result in, advancing the field of computational ethics.

References

[1] Peter Albrecht, Stefano; Stone. Multiagent learning: Foundations and recent trends. tutorial. https://www.cs.utexas.edu/~larg/ijcai17_tutorial/multiagent_learning.pdf. IJCAI-17 conference.

[2] Julia Driver. The History of Utilitarianism. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Winter 2014 edition, 2014.

[3] Bernard Gert and Joshua Gert. The Definition of Morality. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Fall 2020 edition, 2020.

[4] Ganascia J.-G. Ethical system formalization using non-monotonic logics. In *Proc. of the Cognitive Science conference (CogSci2007)*, 2007.

[5] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In *Proceedings of the 25th International Conference on Computer Aided Verification*, volume 8044 of *CAV 2013*, pages 1–35, New York, NY, USA, 2013. Springer-Verlag New York, Inc.

[6] Richard Kraut. Aristotle's Ethics. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Summer 2022 edition, 2022.

[7] William Lillie. *An Introduction to Ethics*. New York: Barnes & Noble, 1948.

[8] J. S. Mill. *Utilitarianism*. Oxford University Press UK, 1861.

[9] Ian Niles and Adam Pease. Toward a Standard Upper Ontology. In Chris Welty and Barry Smith, editors, *Proceedings of the 2nd International Conference on Formal Ontology in Information Systems (FOIS-2001)*, pages 2–9, 2001.

[10] Adam Pease. *Ontology: A Practical Guide*. Articulate Software Press, Angwin, CA, 2011.

[11] Adam Pease and Stephan Schulz. Knowledge Engineering for Large Ontologies with Sigma KEE 3.0. In *The International Joint Conference on Automated Reasoning*, 2014.

[12] Eric Reisen. The moral case for the development of autonomous weapon systems, Feb 2022.

[13] Stephan Schulz. E - A Brainiac Theorem Prover. *AI Commun.*, 15(2-3):111–126, 2002.

[14] Samuel T. Segun. From machine ethics to computational ethics. *AI and Society*, 36(1):263–276, 2021.

[15] David Silver, Satinder Singh, Doina Precup, and Richard S. Sutton. Reward is enough. *Artificial Intelligence*, 299:103535, 2021.

[16] Koundinya Vajjha, Avraham Shinnar, Barry M. Trager, Vasily Pestun, and Nathan Fulton. Certrl: formalizing convergence proofs for value and policy iteration in coq. In Catalin Hritcu and Andrei Popescu, editors, *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, pages 18–31. ACM, 2021.

A First-order transplant scenario in SUMO

The following SUMO formulas specify the situation of the organ transplant dilemma and the inference that should take place if there is informed consent (under a deontological paradigm with this standard ethical code). The formulas are expressed in first-order logic without modal operators because their translation from SUMO to TPTP is a work in progress and this is easier as a proof-of-concept.
The setting is declared that there is a HospitalBuilding where three non-equal humans are located and two of them are patients of the other. One human is healthy and the other human is terminally ill.

```
(instance Hospital HospitalBuilding)
(instance Surgeon0 Human)
(instance Human1 Human)
(instance HealthyHuman Human)

(not (equal Surgeon0 Human1))
(not (equal Surgeon0 HealthyHuman))
(not (equal HealthyHuman Human1))

(located Surgeon0 Hospital)
(located Human1 Hospital)
(located HealthyHuman Hospital)

(patientMedical Human1 Surgeon0)
(patientMedical HealthyHuman Surgeon0)

(attribute HealthyHuman Healthy)
(attribute Human1 FatalDisease)
```

A Definition of terminally ill as meaning there's a greater than 99% chance the patient will die. As well as a specification that healthy primates have two kidneys.

```
(=>
    (instance ?DISEASE FatalDisease)
    (and
        (diseaseMortality ?DISEASE ?RATE)
        (greaterThan ?RATE 0.99)))

(=>
    (and
        (instance ?H Primate)
        (instance ?D DiseaseOrSyndrome)
        (not
            (attribute ?H ?D)))
    (exists (?K1 ?K2)
        (and
            (instance ?K1 Kidney)
            (instance ?K2 Kidney)
            (not
                (equal ?K1 ?K2))
            (part ?K1 ?H)
            (part ?K2 ?H))))
```

A specification that the dying patient has an impaired kidney and needs a kidney that is not impaired. Moreover, the healthy patient has two healthy kidneys.

```
(attribute Human1 (ImpairedBodyPartFn Kidney))
(needs Human1 K1)
(instance K1 Kidney)
(not (attribute K1 (ImpairedBodyPartFn Kidney)))

(instance HealthyKidney1 Kidney)
(instance HealthyKidney2 Kidney)
(part HealthyKidney1 HealthyHuman)
(part HealthyKidney2 HealthyHuman)
(not (equal HealthyKidney1 HealthyKidney2))
(not (attribute HealthyKidney1 (ImpairedBodyPartFn Kidney)))
(not (attribute HealthyKidney2 (ImpairedBodyPartFn Kidney)))
```

A definition of an organ transplant as a subclass of the Surgery class and the Substitution class.

```
(subclass OrganTransplant Surgery)
(subclass OrganTransplant Substitution)

(=>
    (instance ?Trans OrganTransplant)
    (exists (?Sur ?Org ?Pat ?Don)
        (and
            (attribute ?Sur Surgeon)
            (instance ?Don Human)
            (instance ?Pat Human)
            (instance ?Org Organ)
            (agent ?Trans ?Sur)
            (origin ?Trans ?Don)
            (patient ?Trans ?Org)
            (destination ?Trans ?Pat))))
```

The statement that there is the capacity for the organ transplant to take place.

```
(capability OrganTransplant destination Human1)
(capability OrganTransplant patient HealthyKidney1)
(capability OrganTransplant origin HealthyHuman)
(capability OrganTransplant agent Surgeon0)
```

A first-order instance of the inference neded to declare that the surgeon can perform the surgery if informed consent is provided.

```
(=>
    (attribute Surgeon0 InformedConsent)
    (and
        (instance Transplant1 OrganTransplant)
        (destination Transplant1 Human1)
        (patient Transplant1 HealthyKidney1)
        (origin Transplant1 HealthyHuman)
        (agent Transplant1 Surgeon0)))
```

The inference in “deontological style” is tested via loading the transplant.kif file into Sigma and querying Vampire.

First, the following assertion is needed:

```
(attribute Surgeon0 InformedConsent)
```

Next the following query may be posed:

```
(agent Transplant1 ?X)

Answer ?X = Surgeon0
```

An ethical conjecture is that one with the virtue of practical wisdom will still require consent before performing an organ transplant surgery (even if there is no formal ethical code). With modal operators, this reasoning will be more distinct from the deontological case above.

```
(=>
    (attribute Surgeon0 PracticalWisdom)
    (and
        (=>
            (attribute Surgeon0 Consent)
            (and
                (instance Transplant1 OrganTransplant)
                (destination Transplant1 Human1)
                (patient Transplant1 HealthyKidney1)
                (origin Transplant1 HealthyHuman)
                (agent Transplant1 Surgeon0)))
        (=>
            (not (attribute Surgeon0 Consent))
            (not
                (exists (?Transplant)
                    (and
                        (instance ?Transplant OrganTransplant)
                        (destination ?Transplant Human1)
                        (patient ?Transplant HealthyKidney1)
                        (origin ?Transplant HealthyHuman)
                        (agent ?Transplant Surgeon0)))))))
```

The following assertsions are needed:

```
(attribute Surgeon0 PracticalWisdom)
(attribute Surgeon0 Consent)
```

One can also query whether an organ transplant took place:

```
(instance ?X OrganTransplant)

Answer ?X = Transplant1
```

The proofs of these queries as well as additional work will be hosted on the public github repo: https://github.com/zariuq/Formalization-of-Ethical-Theory---AITP.

LightGBM Hyperparameter Optimization for Clause Classification in Theorem Proving *

Zarathustra Goertzel[1], Jan Jakubův[1,2], Mikoláš Janota[1], and Cezary Kaliszyk[2]

[1] Czech Technical University in Prague, Prague, Czech Republic
[2] University of Innsbruck, Innsbruck, Austria

1 Motivation: Machine Learning in Theorem Proving

Applications of machine learning (ML) in automated theorem proving (ATP) often involve training of models from large datasets. This training is usually performed by some machine learning framework, and there are many frameworks to choose from. However, many machine learning methods are parametrized and quite often using the right hyperparameter values is essential to achieve good prediction results. In our experiments with the ML theorem prover ENIGMA [6, 7, 3, 5], we have selected hyperparameter values based on our experience, or we've performed a grid search over some set of possible values. Additionally, as a part of our recent experiments with ENIGMA on Isabelle [4], we have implemented an automated hyperparameter selection tool `lgbtune`.[1] While this tool has been successfully used during the experiments, its performance has never been evaluated in detail, and this is the main topic of this work.

ENIGMA is a machine learning guidance system for automated theorem prover E [9]. E's input is a first-order logic problem consisting of axioms and a conjecture to be proved. This problem is translated to first order *clauses*, and a proof search based on a *given clause loop* is launched in the space of clauses. In each step of this proof search, one unprocessed clause, called *given*, is selected for processing. The given clause selection is one of the most important choice points performed during the proof search, and this is where ENIGMA guides the prover.

ENIGMA experiments are done using the *training/evaluation loop*. Given a set of benchmark problems $\mathcal{P}$, we run E over all problems $\mathcal{P}$. We analyze *successful* proof searches, and clauses selected during the search are classified as *useful* and *useless*. The clauses that participate in the final proof are considered useful, while the others are useless because their processing might have been avoided. On thusly labeled clauses, we train a machine learning model $\mathcal{M}$ to distinguish useful clauses from useless ones. Model $\mathcal{M}$ is then used to guide next E search over problems $\mathcal{P}$. This results in new successful proof searches used to construct new training data for the next iteration of the training/evaluation loop.

As the underlying machine learning method, ENIGMA can use decision trees or graph neural networks. In this work we concentrate on hyperparameter setting for training of decision tree models. ENIGMA support two decision tree frameworks: XGBoost [2] and LightGBM [8]. Recently, we favor LightGBM because it is faster and more stable on large training data. The data input for the LightGBM trainer are labeled clauses (as useful/useless) represented by numeric feature vectors. The training data we encounter with ENIGMA are quite specific and consist of a large amount (millions of clauses) of long (single vector length over 60k) but sparse vectors (around 1% of non-zero values). Apart from the training vectors, the LightGBM trainer take other *hyperparameters* as an input. We target this topic in the rest of this work.

*Supported by the ERC Project *SMART* Starting Grant no. 714034, and by the Czech MEYS under the ERC CZ project *POSTMAN* no. LL1902.

[1] https://github.com/ai4reason/enigmatic/blob/master/enigmatic/lgbtune.py

2 LightGBM Hyperparameter Tuning

LightGBM supports few dozens of hyperparameters that influence the model training process. While many of them might be safely used with their default values, setting of some hyperparameters is crucial for success and to prevent overfitting. While there exist basic guidelines for setting of hyperparameters, setting them in practice often requires experience and understanding of the training process. There are, however, automated tools to search for suitable parameter values like Optuna [1] or FLAML [10] which support LightGBM. For our experiments with ENIGMA, we have developed our tool `lgbtune` to search for suitable values of LightGBM hyperparameters targeted to our specific ATP needs.

Our tool `lgbtune` is implemented in Optuna. Given the training data, it keeps a small amount of the training data (5%) for a later independent evaluation, and it trains the models only on the rest. Then it proceeds in phases when it tries to search optimal values for selected hyperparameters, different ones in each phase. In each phase, several values of tuned hyperparameters are tried, and the resulting models are evaluated on the part of input data not used for training. The best hyperparameter values are then fixed in the phases to follow.

Some of the hyperparameters are dependent, that is, the optimal value of one parameter might depend on the value of another one. For example, the optimal number of tree leaves might depend on the maximal tree depth. We tune dependent parameters together in one phase. In phase (1) we tune probably the most influential parameter, that is, the number of leaves (`leaves`) in a decision tree. We set the tree depth (`max_depth`) to unlimited and thus we eliminate one dependent parameter. In other phases we then tune (2) randomized feature sampling (`bagging_fraction`, `bagging_freq`), (3) the minimal number of data in leaves (`min_data`), and (4) L1/L2 regularization terms (`lambda_l1`, `lambda_l2`). Value selection mechanism and distribution are derived from Optuna.

3 Machine Learning and ATP Evaluation

In `lgbtune`, the quality of a model is estimated based on its accuracy on the shelved training data. This *ML evaluation* should correlate with the actual performance of the prover guided by the model, that is, with *ATP evaluation*. However, this is not always the case, because the training data are typically unbalanced, and more than 90% are typically negative training samples (clauses classified as *useless*). Hence it is crucial to compute separately accuracies on positive and negative testing samples. Moreover, it seems that the positive accuracy is even more important for ATP evaluation. This can be explained by the behavior of E, where accidental processing of a single useless clause does not need to do much harm, while postponing of the processing of a useful clause can effectively block any path to success. Hence we measure the quality of a model as $2pos + neg$, where *pos* and *neg* are positive and negative accuracies.

We have successfully used `lgbtune` during our recent ENIGMA experiments [4], where it helped us to increase the accuracy of models by more than 5%. In this presentation, we would like to present our tool, and, additionally, perform an extended evaluation and testing of our tool (the extended evaluation is currently in progress). We will evaluate the ATP performance of trained models and test our assumptions about the correlation of ML and ATP performance. Furthermore, we would like to evaluate the impact of a different phase orders during the tuning, and the importance of tuned parameters. This could help us to decide which phases should be given more time, and which phases should be removed. Finally, `lgbtune` is motivated by LightGBM plugins from Optuna and FLAML, and we would like to compare directly with their performance. All the experiments will be targeted to data from our ATP experiments.

References

[1] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

[2] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *KDD*, pages 785–794. ACM, 2016.

[3] Karel Chvalovský, Jan Jakubův, Martin Suda, and Josef Urban. ENIGMA-NG: efficient neural and gradient-boosted inference guidance for E. In *CADE*, volume 11716 of *Lecture Notes in Computer Science*, pages 197–215. Springer, 2019.

[4] Zarathustra Amadeus Goertzel, Jan Jakubuv, Cezary Kaliszyk, Miroslav Olsák, Jelle Piepenbrock, and Josef Urban. The isabelle ENIGMA. *CoRR*, abs/2205.01981, 2022.

[5] Jan Jakubův, Karel Chvalovský, Miroslav Olsák, Bartosz Piotrowski, Martin Suda, and Josef Urban. ENIGMA anonymous: Symbol-independent inference guiding machine (system description). In *IJCAR (2)*, volume 12167 of *Lecture Notes in Computer Science*, pages 448–463. Springer, 2020.

[6] Jan Jakubův and Josef Urban. ENIGMA: efficient learning-based inference guiding machine. In *CICM*, volume 10383 of *Lecture Notes in Computer Science*, pages 292–302. Springer, 2017.

[7] Jan Jakubův and Josef Urban. Enhancing ENIGMA given clause guidance. In *CICM*, volume 11006 of *Lecture Notes in Computer Science*, pages 118–124. Springer, 2018.

[8] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 3146–3154, 2017.

[9] Stephan Schulz. E - A Brainiac Theorem Prover. *AI Commun.*, 15(2-3):111–126, 2002.

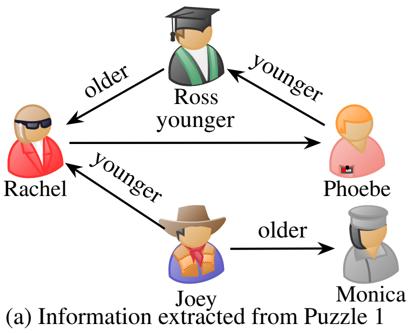
[10] Chi Wang, Qingyun Wu, Markus Weimer, and Erkang Zhu. Flaml: A fast and lightweight automl library. In *MLSys*, 2021.

A Corpus for Precise Natural Language Inference

We propose a corpus for tasks related to natural language inference. The corpus contains logical puzzles in natural language from two domains: comparing puzzles and truth-telling puzzles. For instance:

Example 1 (Comparison puzzle) *Ross is older than Rachel who is younger than Phoebe. Joey is older than Monica but younger than Rachel. Phoebe is younger than Ross. Who is the tallest? Is Phoebe older than Monica? Is Ross younger than Joey? (Figure 1)*

First, note that the text in the queries does not explicitly appear within the text of the puzzle. Since reasoning is mandatory to answer, approaches based only on machine learning may not suffice. Solutions based on reasoning or hybrid approaches will be required by this puzzle-based corpus. Second, note that some background knowledge is required: *older* and *younger* are transitive relations, or the definition of concept *tallest*. Third, good puzzles have two properties: (i) each piece of information is necessary and (ii) no unnecessary information is provided. These properties make puzzles interesting candidates for machine comprehension tasks. Fourth, since the solution of the puzzle is clear, one can void the cases of mislabelling the text due to subjective annotation or human biases. Recall the many troublesome annotations with the SICK dataset identified by Kalouli et al. [Kalouli et al.2017], in which 611 pairs out 9,840 does not make sense. Fifth, there is wide range of logical puzzles with available solutions[1] that can be collected and adapted for building such puzzle based corpus, starting with simple ones (e.g. comparison puzzles) and continuing with more complex ones (e.g. zebra puzzles). Sixth, the existing work on automatic puzzle generation combined with the work on natural language generation can be used to automatically create such puzzle-based benchmarks for precise inference tasks. One example are the 382 knights and knaves puzzles popularised by Raymond Smullyan and automatically generated by Lau and Chan[2]. The complexity of these puzzles depends on the number of the individuals ranging from 2 to 9 individuals. The above aspects can be discussed during the workshop to clarify the best way for delivering a puzzle-based benchmark for question answering.



(a) Information extracted from Puzzle 1

Question	Theorem in FOL	Answer
Is Ross the tallest ?	$tallest(Ross)$	Entailment
Is Monica the shortest ?	$shortest(Monica)$	Entailment
Is Phoebe older than Monica ?	$older(Phoebe, Monica)$	Entailment
Is Monica younger than Joey ?	$younger(Monica, Joey)$	Entailment
Is Rachel the tallest ?	$tallest(Rachel)$	Contradiction
Is Phoebe the shortest ?	$shortest(Phoebe)$	Contradiction
Is Monica older than Phoebe ?	$older(Monica, Phoebe)$	Contradiction
Is Ross younger than Joey?	$younger(Ross, Joey)$	Contradiction

(b) Sample of atomic question with 4 predicates and 5 individuals

Figure 1: Question answering for unambiguous comparison puzzles

Automatically solving such text-based puzzles requires several technical challenges. Consider the following puzzle with two friends:

[1] Consider for instance many sites like `https://www.brainzilla.com/logic/zebra/`, `https://www.ahapuzzles.com/logic/logic-puzzles`, `https://www.mathsisfun.com/puzzles/logic-puzzles-index.html` to name a few

[2] `https://philosophy.hku.hk/think/logic/knights.php`

Figure 2: A truth-telling puzzle with two characters

Example 2 (Truth telling puzzle) *In the Central Perk cafe there is this particular behaviour: married people don't lie, while single people always lie. While Joey and Chandler were sitting on the sofa a woman is approaching them and asked: "Are you married?" Joey promptly replied: "We are both single!" Can the woman figure out weather the two friends are married or not? (Figure 2)*

When translating the puzzle into some logical formalism (e.g. First Order Logic [Groza and Nitu2022] using NLTK [Perkins2014] and Prover9 [McCune2005] or Description Logics using for instance FRED [Gangemi et al.2017]), a machine comprehension tool should handle various technical challenges including: (i) recognising the named entities (e.g. Joey, Chandler); (ii) coreference resolution (e.g. "We are both single"); (iii) automatically computing the domain size for model finders (e.g. Mace4), (iv) reducing the interpretation models to a single one (e.g. adding the unique name assumption, adding relevant background knowledge, closing the world, removing isomorphic models).

For each puzzle one can generate a large set of questions, as exemplified in the right part of Figure 1. Each puzzle can be associated with the entire set of atomic questions that can be generated based on the relations and individuals occurring in the text [Szomiu and Groza2021]. If the puzzle has a unique solution, there should be only pairs labelled as "entailment" and "contradiction". To obtain unknown pairs, one can remove some clues from the puzzle. The resulted ambiguous puzzle will have several interpretation models, in which the statements can be proved as true (entailment), false (contradiction), or unknown (if they appear true or false in the computed models.

In line with the work of Pease et al. [Pease et al.2020], we propose a corpus for reasoning tasks. The puzzle-based corpus may benefit from the huge number of puzzle available that provide unique solutions. Apart from the comparison and truth-telling puzzles exemplified here, the corpus can be extended with various types of logical puzzles [Groza2021].

References

Aldo Gangemi, Valentina Presutti, Diego Reforgiato Recupero, Andrea Giovanni Nuzzolese, Francesco Draicchio, and Misael Mongiovì. 2017. Semantic web machine reading with FRED. *Semantic Web*, 8(6):873–893.

Adrian Groza and Cristian Nitu. 2022. Question answering over logic puzzles using theorem proving. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, pages 871–874.

Adrian Groza. 2021. *Modelling Puzzles in First Order Logic*. Springer International Publishing.

Aikaterini-Lida Kalouli, Livy Real, and Valeria De Paiva. 2017. Textual inference: getting logic from humans. In *IWCS 2017-12th International Conference on Computational Semantics—Short papers*.

William McCune. 2005. Prover9. *University of New México*.

Adam Pease, Paulo Santos, and Alexandre Rademaker. 2020. A corpus of spatial reasoning problems. In *5th Conf. on Artificial Intelligence and Theorem Proving, September 13-19, 2020, Aussois, France*.

Jacob Perkins. 2014. *Python 3 text processing with NLTK 3 cookbook*. Packt Publishing Ltd.

Roxana Szomiu and Adrian Groza. 2021. A puzzle-based dataset for natural language inference. *arXiv preprint arXiv:2112.05742*.

LISA: Towards a Foundational Theorem Prover

Simon Guilloud , Florian Cassayre, Viktor Kunčak

EPFL IC LARA, Station 14, CH-1015 Lausanne, Switzerland
{Simon.Guilloud,Viktor.Kuncak}@epfl.ch

We present the foundations and initial implementation of a new interactive theorem prover, named LISA. In a slight contrast to most popular type-theoretic frameworks, and much like Mizar [9], LISA aims to use classical mainstream foundations of mathematics, taking a hint, among others from the talk of John Harrison in this very venue in 2018 [12]. LISA uses (single-sorted) first order logic (with schematic variables) as the syntactic framework and set theory axiom schemas as the semantic framework. On top of these foundations we can construct numbers and other mathematical theories and models of computation without introducing new axioms. As the target use of LISA we envision mathematical statements as well as formal proofs of computer programs and systems, possibly with probabilistic and distributed behavior. For automation in LISA we expect to employ newly developed algorithms for equivalence checking of formulas and proofs [4], existing high-performance superposition-based theorem provers such as Vampire [7], SPASS [13], E [10] and Zipperposition, as well as SMT solvers such as Z3 [8], CVC5 [**?**], and veriT [11], and OpenSMT [2]. An important aim of LISA is interoperability with other proof assistants. We hope that the design of LISA with small, fresh code base, simple foundation and explicit proof objects will encourage building bridges with other tools. We also expect that the system will serve as a good vehicle to explore machine-learning guided superposition proof search, with learned heuristics complementing hand-tuned ones. We also plan to explore high-level translation of proofs from other systems into LISA, inspired by the success of deep neural networks in natural language translation.

Development LISA is separated into a trusted logical core called the kernel that we keep as small as possible, and a front part adding arbitrarily complex or powerful layers of abstractions and features for the user, but which need not be trusted. Our language of choice for LISA is Scala (version 3). On one hand, Scala offers very powerful syntax and features, such as dependent types, string interpolation and implicits, which allows (for example) to use Scala compiler to check that terms are well formed, or to define very nice syntax for proofs or objects we want to manipulate. On the other hand, the kernel is developed only with a restricted subset of the Scala language, paving the door for future verification of the kernel using program verification tools such as Stainless [5].

We strived to keep the core of LISA as close as possible to well-known and uncontroversial mathematical theory, but nonetheless includes some improvements, in particular relating to space and time complexity. Hence, our base language is first order logic, to which we add schematic functions and predicates. Those schematic symbols give some flavour of second-order logic, the possibility to define a single syntactic object representing an axiom schema or theorem schema. Our experience already suggests that schematic variables make many further developpements and proofs simpler, but do not increase the proof strength of FOL.

LISA kernel's proof checker is based on an implementation of Gentzen's sequent calculus, with the addition of some deduced proof steps such as the substitution of equals for equals and the possibility to enclose subproofs to reduce the length of proofs.

Proofs in LISA are represented as linearized directed acyclic graphs (i.e. lists of proof steps), which can verified by the proof checker. Proofs can either exists in isolation, or be part of *theories*. Theories in LISA form the LCF part of the kernel. They use the proof checker to verify theorems, and to then accept such theorems as assumptions of future proofs without needing rechecking. Theories also enable introduction of (set theoretic) axioms and definitional extensions of new symbols.

Equivalence Checker A unique feature of our kernel is the inclusion of a quasilinear-time *sufficient-equivalence checker* for FOL formulas that takes into account associativity, commutativity, and other laws and has a clear algebraic completeness characterization [4]. The aim of implementing such equivalence checker in the kernel is to shorten the proofs (by a constant factor), making the system more efficient to use by humans and tactics alike.

The equivalence checker takes into account properties of FOL such as alpha equivalence and symmetry of equality, but most of its complexity take place at the level of propositional logic. Since equivalence of propositional formulas is coNP complete, we can only aim for an approximation of it. Nonetheless, it is important to note that our algorithm is not an arbitrary heuristic; it is a complete decision procedure for a well-defined subalgebra of boolean algebra called orthocomplemented bisemilattice, which is essentially Boolean algebra without the distributivity law. We expect that this aspect helps make the equivalence checker predictable. Moreover, it runs in quasilinear time, meaning it can replace syntactic equality checking everywhere in the proof checking procedure with only a logarithmic cost. Equivalence checking algorithm is implemented using a combination of techniques for tree isomorphism with memoization and term rewriting. Our anecodotal experience suggests that manual construction of proofs greatly benefited from the implementation of this formula equivalence.

High Level Proofs We also succesfully implemented a semi-interactive interface to the logical kernel allowing forward and backward reasoning, creation of new tactics and combination of existing ones, high syntax insurance through Scala type checking, higher order matching and more. Works is still ongoing, but it demonstrates the feasibility of designing a powerfull interface for the system.

Why set theory? Even though set theory is by far the most studied, accepted and well-known foundation of mathematics, this choice is uncommon among interactive theorem provers and the Computer Science community in general. Indeed, most modern ITPs such as Coq [1], Lean [3], Isabelle [14] or the HOL-family [6] are based either on some form of Type Theory or on Higher Order Logic. While those theory have the advantage to look closer from the start to the mathematical formalism, we believe there are several reasons to prefer set theory. It may need higher upfront work, but we strongly believe that through the use of powerful tactics and a soft type system (where types are represented by set or class membership with meta-information for polymorphism) apposed on top of set theory, arbitrarily familiarity in the writing of proofs can be achieved with set theory, with ultimately greater flexibility. More arguments in favour of set theoretic foundations have been proposed by John Harrison [12].

Verification of Programs One of LISA's goals is to contribute to verificaton of reliability of our computing infrastructure. Medium term, we aim for two different paths for using LISA inside program verifiers, such as Stainless, in which it can be cumbersome to reason about quantified propositions. On the one hand, we hope to develop semantics of transition systems and programming languages and use such semantics to provide high-confidence proofs about behavior of programs and embedded systems.

Future work and conclusion LISA is in early stage of development, which has focused on proof checking for FOL with equality and schematic variables, as well as the support for theories and LCF-style tracking of theoremhood. Most of the development dependent on set theoretic axiom schemas remains in the future. Medium term goals include further development of abstraction layers and proof tactics, development of a core library of results, implementation of the program verification side of LISA and more exploratory work such as the tradeoff between , or the extension of our equivalence checker to orthocomplemented lattices. We also had some exploratory work on the interoperability of proofs, and we plan to continue to work on that aspect.

References

[1] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, Berlin Heidelberg, 2004.

[2] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The OpenSMT Solver. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Javier Esparza, and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015, pages 150–153. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[3] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, volume 9195, pages 378–388. Springer International Publishing, Cham, 2015.

[4] Simon Guilloud and Viktor Kuncak, editors. *Equivalence Checking for Orthocomplemented Bisemilattices in Log-Linear Time*. Springer, 2022.

[5] Jad Hamza, Nicolas Voirol, and Viktor Kunčak. System FR: Formalized foundations for the Stainless verifier. *Proc. ACM Program. Lang*, 3, November 2019.

[6] John Harrison. HOL Light: An Overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674, pages 60–66. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[7] Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 1–35, Berlin, Heidelberg, 2013. Springer.

[8] Stephan Merz and Hernán Vanzetto. Automatic Verification of TLA + Proof Obligations with SMT Solvers. In Nikolaj Bjørner and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, Lecture Notes in Computer Science, pages 289–303, Berlin, Heidelberg, 2012. Springer.

[9] Adam Naumowicz and Artur Kornilowicz. A Brief Overview of Mizar. volume 5674, pages 67–72, August 2009.

[10] Stephan Schulz. System Description: E 1.8. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, Lecture Notes in Computer Science, pages 735–743, Berlin, Heidelberg, 2013. Springer.

[11] Hans-Jörg Schurr, Mathias Fleury, and Martin Desharnais. Reliable Reconstruction of Fine-grained Proofs in a Proof Assistant. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, Lecture Notes in Computer Science, pages 450–467, Cham, 2021. Springer International Publishing.

[12] John HarrisonAmazon Web Services. Let's make set theory great again! *axiomatic set theory*, page 46.

[13] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. SPASS Version 3.5. In Renate A. Schmidt, editor, *Automated Deduction – CADE-22*, Lecture Notes in Computer Science, pages 140–145, Berlin, Heidelberg, 2009. Springer.

[14] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle Framework. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science, pages 33–38, Berlin, Heidelberg, 2008. Springer.

Model Discovery for Efficient Search*

Jan Hůla[1 2] and Mikoláš Janota[1]

[1] Czech Technical University in Prague, Prague, Czech Republic
[2] University of Ostrava, Ostrava, Czech Republic

Motivation and overview. In this contribution, we focus on exploration strategies inspired by human cognition. In comparison to AI search and learning, humans show greater efficiency in terms of the number of states explored. This efficiency is mostly due to the fact that humans have aversion towards prolonged periods of random exploration and most of the time are following goal-oriented behavior. The goals often form a hierarchy and are proposed by the person themselves. Moreover, the person is constantly tracking the expected difficulty of achieving the given goal and, based on the ratio between this difficulty and the value of this goal, they can decide to abandon the goal and instead pursue another.

Goals are proposed and their difficulty is tracked using knowledge about a given domain. In completely novel domains, in which it seems that random exploration will not lead to success, the person typically starts by trying to create a mental model of the domain and then uses this mental model to explore more effectively. For example, if they have already learned that keys unlock doors and then find themselves in a locked room, they can temporarily set a goal to find a key.

Inspired by these observations, we design an agent which explores the state space by proposing various sub-goals to itself. First, these sub-goals are curiosity-driven and their aim is to incrementally learn a mental model of the domain. These sub-goals are later interleaved with the original goals and their sub-goals. Each goal and sub-goal is being searched for with a planner that leverages the mental model discovered thus far.

Related Work. Our work is related to a current trend of learning *world models* in model-based reinforcement learning [6, 5, 12]. Such world models are represented as a neural network which predicts state dynamics and rewards conditioned on the actions of the agent. The learned world model can later be used by the agent to decide how to act in an online fashion or even to learn the entire policy in an offline fashion. Our approach differs by the fact that in our case the model is represented as a set of concepts and declarative statements which can be used by the planner and learned incrementally. This different kind of learning [10, 1] overcomes many issues inherent to learned statistical models such as catastrophic forgetting [9], failure to generalize due to spurious correlations [8], inability to recompose old knowledge in novel ways [13], etc. Another trend related to our work is the *learnig to propose sub-goals* research direction where the goal is to train a neural network to propose sub-goals that are useful during the search for the original goal. [14, 3]. In our case, the sub-goal proposal is manually engineered. Our approach is also related to symbolic regression and ontology learning, where the goal is to discover a symbolic model that explains or summarizes the observed data [4, 2].

Problem Domain. We study our approach in the domain of logical puzzles and simple games. We chose this domain to avoid the complexity of real-world problems that are hard to model precisely. Our aim is to learn and understand basic principles of efficient exploration in

*The results were supported by the Ministry of Education, Youth and Sports within the program ERC CZ under the project POSTMAN no. LL1902.

the simplified domain with the hope that later we would be able to include these principles in a more complex approach scaling to realistic domains.

Concretely, we study exploration strategies in games written in the video game description language (VGDL) [11]. VGDL enables one to define games in a few lines of declarative statements. These declarative statements are divided into three categories: The first category corresponds to the definitions of entity types and their default behavior, the second category describes the interaction events between each pair of entity types, and the third category describes termination conditions. An example of a simple game defined in VGDL is visible in Figure 1. To instantiate a concrete game, a rectangular grid of symbols representing different entities is used. The VGDL engine parses this description into an instantly playable game.

We propose an agent that learns the mechanistic model of the game to explore more effectively. At each time step, our agent receives the current state of the board as input and outputs one of the possible actions. The agent has the VGDL engine together with a planner in his "head" but does not have access to the definition of the current game. Therefore, the initial exploration aims to recover this definition by discovering the types of entities present in the game and the effects of interaction between different pairs of entities. The recovered definition is constantly updated and used by the VGDL engine to produce a "mental simulation" of a possible evolution of the game. The planner is used to propose sub-goals which are either aimed to observe the result of an interaction between a pair of entity types or to get the agent closer to the original goal. After finding a "mental plan" for the currently schedule goal, the agent executes the plan in the real game. The discrepancy between the imagined dynamics and the real dynamics is used to update the mental model of the agent.

As mentioned at the beginning of this section, we consider this simplistic domain as an exploration playground for prototyping efficient search algorithms. We believe that the ability to discover abstractions (in our case entities) and relations between them (in our case interaction events) is a generally useful and interesting area of research, which is also related to theorem proving where one can see an analogy in concept and lemma discovery [7]. We hope that this work will inspire further research on model discovery for the purpose of efficient search and reasoning.

```
SpriteSet
    hole   > Immovable color=DARKBLUE
    avatar > MovingAvatar #cooldown=4
    box    > Passive
InteractionSet
    avatar wall > stepBack
    box avatar  > bounceForward
    box wall    > undoAll
    box box     > undoAll
    box hole    > killSprite
TerminationSet
    SpriteCounter stype=box    limit=0 win=True
```

Figure 1: An example of a VGDL definition of a game (Sokoban). The concrete game also requires a grid of symbols which defines the initial conditions of the game. SpriteSet contains definitions of entity types and their default behavior, InteractionSet describes what happens when two entities end up in a same position and TerminationSet describes termination conditions.

References

[1] Susan Carey. The origin of concepts. *Journal of Cognition and Development*, 1(1):37–41, 2000.

[2] Xinlei Chen, Abhinav Shrivastava, and Abhinav Kumar Gupta. Neil: Extracting visual knowledge from web data. *2013 IEEE International Conference on Computer Vision*, pages 1409–1416, 2013.

[3] Konrad Czechowski, Tomasz Odrzygóźdź, Marek Zbysiński, Michał Zawalski, Krzysztof Olejnik, Yuhuai Wu, Łukasz Kuciński, and Piotr Miłoś. Subgoal search for complex reasoning tasks. *Advances in Neural Information Processing Systems*, 34:624–638, 2021.

[4] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *arXiv preprint arXiv:2006.08381*, 2020.

[5] Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning latent dynamics for planning from pixels. In *International conference on machine learning*, pages 2555–2565. PMLR, 2019.

[6] Danijar Hafner, Timothy Lillicrap, Mohammad Norouzi, and Jimmy Ba. Mastering atari with discrete world models. *arXiv preprint arXiv:2010.02193*, 2020.

[7] Jónathan Heras, Ekaterina Komendantskaya, Moa Johansson, and Ewen Maclean. Proof-pattern recognition and lemma discovery in acl2. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 389–406. Springer, 2013.

[8] Robert Kirk, Amy Zhang, Edward Grefenstette, and Tim Rocktäschel. A survey of generalisation in deep reinforcement learning. *arXiv preprint arXiv:2111.09794*, 2021.

[9] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017.

[10] Dharshan Kumaran, Demis Hassabis, and James L McClelland. What learning systems do intelligent agents need? complementary learning systems theory updated. *Trends in cognitive sciences*, 20(7):512–534, 2016.

[11] Tom Schaul. A video game description language for model-based or interactive learning. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pages 1–8. IEEE, 2013.

[12] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering Atari, Go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

[13] Paul Smolensky, R Thomas McCoy, Roland Fernandez, Matthew Goldrick, and Jianfeng Gao. Neurocompositional computing in human and machine intelligence: A tutorial. 2022.

[14] Josef Urban and Jan Jakubův. First neural conjecturing datasets and experiments. In *International Conference on Intelligent Computer Mathematics*, pages 315–323. Springer, 2020.

Selecting Quantifiers for Instantiation in SMT *

Mikoláš Janota[1], Jelle Piepenbrock[1,2], and Bartosz Piotrowski[1,3]

[1] Czech Technical University, Prague
[2] Radboud University Nijmegen, The Netherlands
[3] University of Warsaw, Poland

Introduction In satisfiability modulo theories (SMT), quantifiers are treated opaquely: they are seen as sources of possible instantiations. A quantified sub-formula $\forall \vec{x} \phi$ is abstracted as a Boolean variable Q and instantiations are registered in the form of implications $Q \Rightarrow \phi[\vec{x} \mapsto \vec{t}]$ for some ground terms $\vec{t}$. Consider the following toy formula:

$$f(3) < 0 \wedge ((\forall x f(x) > 0) \vee (\forall x f(x) > x)),$$

where $f\colon \text{int} \to \text{int}$. To obtain a refutation, we abstract and instantiate as follows:

abstraction:	$Q_1 \equiv (\forall x f(x) > 0), Q_2 \equiv (\forall x f(x) > x)$
abstracted formula:	$f(3) < 0 \wedge (Q_1 \vee Q_2)$
instantiation 1:	$Q_1 \Rightarrow f(3) > 0$
instantiation 2:	$Q_2 \Rightarrow f(3) > 3$

Most approaches instantiate quantifiers *gradually*, meaning, the new instantiations are added after testing that the old ones do not already yield a contradiction in the ground solver. In this work, we propose to use machine learning to decide which quantifiers should be instantiated during solving. Figure 1 schematically illustrates the considered setup.

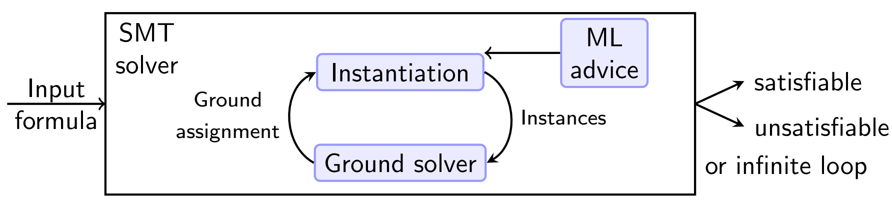


Figure 1: Schematic of the SMT solver with ML guidance for quantifier instantiation.

There are a number of methods for devising new instantiations [2, 3]. Here we consider a simple setup where instantiation terms are taken from the set of terms present in the current ground formula—this includes the instantiations already performed and therefore this set grows throughout the life of the solver. Without theories, the set of candidate ground terms grows towards the Herbrand universe. This approach is referred to as *enumerative instantiation* [8, 5].

In standard setting, the SMT solver would instantiate *all* quantifiers that are true in the current ground model. Here, we propose to instantiate only a subset, based on a pre-trained ML-predictor. Selecting the right quantifier for instantiation puts less burden on the ground solver but also facilitates the search for the right ground terms to be used for instantiations.

*The results were supported by the Ministry of Education, Youth and Sports within the dedicated program ERC CZ under the project POSTMAN no. LL1902.

Approach Consider quantified sub-formulas $Q_1, \ldots, Q_n$ true in the current model of the ground solver. Consider some pre-trained ML-predictor $\mathcal{V}$ from formulas to the interval $[0, 1]$, where $\mathcal{V}(Q) = 1$ means that the predictor believes that Q is important for the solution of the whole SMT problem. A straightforward approach would be to instantiate Q with the probability $V(Q)$. This brings about the risk of instantiating too seldom if the ML predictor has low overall confidence. Therefore, we rescale to $\max \mathcal{V}(Q_i)$, i.e., the quantifier $\operatorname{argmax}_i \mathcal{V}(Q_i)$ is instantiated with probability 1. But even with this policy, we are running the risk that a bad judgment of the ML prediction will cause some important quantifiers never to be instantiated. To that end, with probability $\epsilon \in [0, 1]$ a quantifier is always instantiated. In essence, this corresponds to the well-known ϵ-greedy policy [9] and a quantifier Q is instantiated with probability:

$$\epsilon + (1 - \epsilon)\frac{\mathcal{V}(Q)}{\max_i \mathcal{V}(Q_i)}.$$

Implementation and experiments The approach was implemented in the SMT-solver cvc5 [1] with LightGBM [6] as the ML-predictor. Currently, the ML-predictor only gets as features the bag of words representation of the quantified sub-formula, i.e., ignoring the overall context. For evaluation we use SMT-LIB problems from the UFLIA/sledgehammer category, while filtering out problems solvable without instantiation. Looping-style evaluation is run (similar to [4, 7, 10]). In the first iteration, an unguided SMT-solver is run on the benchmark and data extracted from the solved problems is used to train the ML model. Then, the model is used to guide the solver in the next iteration of solving the benchmark. The success rate is recorded and examples extracted from the newly solved problems are added to the training set. This solving-training procedure is repeated 6 times with time limit 60 s for the solver. Figure 2 shows results for $\epsilon \in \{0, 0.1, 0.5\}$ and a version with no ML-guidance.

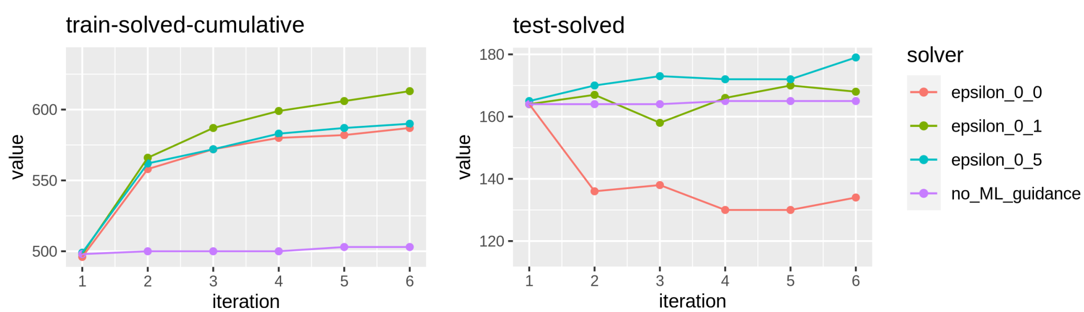


Figure 2: Looping evaluation on UFLIA/sledgehammer dataset. Left: number of training problems solved cumulatively. Right: number of testing problems solved in individual iterations.

Conclusions and future work The paper shows that that ML guidance can effectively guide quantifier instantiation in SMT. The evaluation points to a significant impact of learning on the number of solved instances but we only observe improvements on the training set, not on testing. This suggests that there is some version of over-fitting, even though not in the standard sense because the training examples constantly change. We plan to improve featurization by including the context of quantifiers (looking at the whole SMT problem) and using more advanced featurization approaches such as graph neural networks.

References

[1] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[2] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[3] Yeting Ge and Leonardo Mendonça de Moura. Complete instantiation for quantified formulas in satisfiability modulo theories. In *Computer Aided Verification, 21st International Conference, CAV*, pages 306–320, 2009.

[4] Mikoláš Janota, Jelle Piepenbrock, and Bartosz Piotrowski. Towards learning quantifier instantiation in SMT. In *Theory and Applications of Satisfiability Testing – SAT 2022*. LIPIcs, 2022.

[5] Mikoláš Janota, Haniel Barbosa, Pascal Fontaine, and Andrew Reynolds. Fair and adventurous enumeration of quantifier instantiations. In *Formal Methods in Computer-Aided Design*, 2021.

[6] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[7] Bartosz Piotrowski and Josef Urban. ATPboost: Learning premise selection in binary setting with ATP feedback. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings*, volume 10900 of *Lecture Notes in Computer Science*, pages 566–574. Springer, 2018.

[8] Andrew Reynolds, Haniel Barbosa, and Pascal Fontaine. Revisiting enumerative instantiation. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 10806, pages 112–131, 2018.

[9] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning.* Adaptive Computation and Machine Learning series. Bradford Books, Cambridge, MA, 2 edition, November 2018.

[10] Josef Urban, Geoff Sutcliffe, Petr Pudlák, and Jirí Vyskocil. MaLARea SG1 – machine learner for automated reasoning with semantic guidance. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of *Lecture Notes in Computer Science*, pages 441–456. Springer, 2008.

Learning plausible and useful conjectures

Albert Q. Jiang, Wenda Li, and Mateja Jamnik

University of Cambridge
{qj213, wl302, mj201}@cam.ac.uk

Abstract

Conjecturing is an important activity in mathematics. In this paper, we look at the why and the how of using machine learning to generate mathematical conjectures. We argue that (1) conjecturing is beneficial, both practically and theoretically; (2) conjecture learning should make use of available premises and goals in theorems. We also deliver some design considerations for building an automated conjecturer.

1 Conjecturing as an essential mathematical activity

Consider lemma construction in theorem proving: in the course of proving a theorem, one might realise that a particular conjecture, if true, makes it easier to prove the original theorem. Once the conjecture is proved, it becomes a lemma. A well-known example of this is the proof of the Taniyama–Shimura–Weil conjecture, which implied Fermat's last theorem [Wil95]. Mathematical discovery is interleaved with finding interesting conjectures, and proving, refuting, or revising them [Lak15].

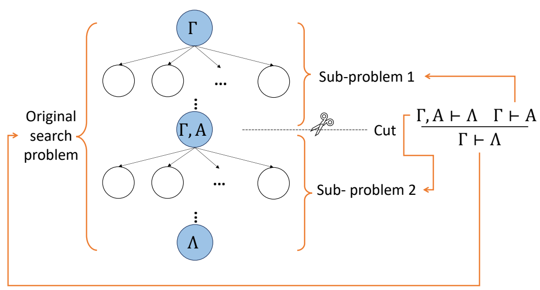


Figure 1: Theorem proving as search. The cut transforms the original search problem into two sub-problems by creating intermediate goal A.

From a computational perspective, formal theorem proving can be viewed as a search problem of finding the goal Λ given the premise Γ, as illustrated in Figure 1. A conjecture is then analogous to an intermediate goal (a *"cut"*). Cuts reduce the search space size exponentially w.r.t. their depth and therefore simplify the search [Boo84, CS97].

Conjecture/lemma construction has received relatively little attention from the AI community, considering its essential role in mathematics (see Appendix for a review). This is partially due to the difficulty of conjuring them: the space of possible conjectures is infinite. If we limit their size, the space is still combinatorially large and a universal heuristic for good conjectures is hard to find. Language models [BHA+21] sample from combinatorial spaces and have shown promising reasoning capabilities like solving maths competition problems [PHZ+22]. They can be easily configured to learn a general heuristic from human data. They are also not restricted to generating conjectures syntactically close to the premises or the goals. Thus, they complement symbolic reasoners, and are an obvious candidate for the task of conjecturing.

2 A quantitative metric for conjecturers

Before embarking on the specifics of a conjecturer, a metric for measuring the quality of conjectures is needed. We can examine conjectures qualitatively but that can be costly, hence not suited to large-scale experiments. For a quantitative metric, we should consider that ineffective conjectures can be true but not easily provable, untrue but not easily refutable, or trivially true and useless for our goal. Meanwhile, effective conjectures usually unlock multiple theorems. In summary, the metric should prefer conjectures that are more provable, useful, and general. [CBW00] also used these criteria to measure how interesting mathematical discoveries are.

Suppose we have a conjecturer $\mathcal{C}$ and a base prover $\mathcal{B}$ (which could take the form of Sledgehammer [PB10] or a learning-assisted prover like LISA [JLHW21]). We can use $\mathcal{C}$ to propose new subgoals, and $\mathcal{B}$ to close them. The performance of conjecturers is measured by *the proportion of theorems proven this way, subtracting the proportion of theorems proven with* $\mathcal{B}$ *alone*. We can calculate a vector of this value indexed by the choice of $\mathcal{B}$. The vector is then a quantitative measure of the conjecturer's performance.

In this paper we argue that given the proposed metric, **conjecture learning should make use of available premises and goals of theorems.** Premises constrain the variable space in ways that are of interest for mathematics (e.g., the premise p *is prime* limits us to a small but interesting set of natural numbers). Focusing on these special variable spaces, one is more likely to make conjectures of relevance to human mathematics. This can increase the generality of conjectures: conjectures syntactically or semantically related to goals have a better chance of helping to prove them, instead of being trivially true (e.g., $0 = 0$). Conditioning on goals can improve the utility of conjectures. In the next section we detail some design considerations.

3 Building an automated conjecturer

The data and the environment Plenty of mathematical corpora and interactive environments are available: lean-gym [PHZ+22] for Lean, PISA [JLHW21] for Isabelle, coq-gym [YD19] for Coq, mlCoP [KUMO18] for Mizar, etc. Inside these proof corpora there are examples of conjecturing (Isabelle and Mizar have more due to their declarative proof style). Behaviour cloning can be deployed on these human conjecturing examples to bootstrap the conjecturer. For each datapoint, we should have the input-output pair to be in the following format: (input) the premises of the current problem; the goals of the current problem; the proof so far; (output) the conjecture written by the human.

The training loop Behaviour cloning alone does not guarantee a hugely useful conjecturer [LYWP21]. Since the conjecturer is bootstrapped from human conjecture examples, it is not aware of the ability of the base prover $\mathcal{B}$ and might propose conjectures that are too hard for it (we presume that the base prover $\mathcal{B}$ is weaker than a human). To deal with this problem, we should adapt our system to come up with conjectures that are both useful and *can be proven by* $\mathcal{B}$. For a set of problems, run the following procedure until convergence: use $\mathcal{C}$ and $\mathcal{B}$ to prove them; for failed proofs, filter out refutable conjectures with counter-example finding tools like `quickcheck` and `nitpick` (filtering out refutable conjectures with Isabelle tools was experimented by [NP18]); re-run $\mathcal{C}$ to refine the goals until they are either proved by $\mathcal{B}$ or a recursion depth limit is reached; collect all successful proofs as the gold-standard data; fine-tune both the conjecturer and the base prover on the gold-standard data. This procedure ensures that the conjecturer is not over-ambitious by only including provable conjectures in the gold-standard data. As the base prover improves via expert iteration [PHZ+22], we should expect the theorems proven and the conjectures created to become more and more advanced.

The network architecture We want to leverage the recent advances in learning-assisted theorem proving. As language model (LM)-based systems have demonstrated their potentials on multiple theorem provers [PS20, PHZ+22, JLHW21], textual information as input is preferred for generality. It is clear that the conjecturer $\mathcal{C}$ and the base prover $\mathcal{B}$ may share many common capabilities, therefore a network architecture that reflects this can be advantageous.

References

[BHA+21] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri S. Chatterji, Annie S. Chen, Kathleen Creel, Jared Quincy Davis, Dorottya Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren Gillespie, Karan Goel, Noah D. Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, Omar Khattab, Pang Wei Koh, Mark S. Krass, Ranjay Krishna, Rohith Kuditipudi, and et al. On the opportunities and risks of foundation models. *CoRR*, abs/2108.07258, 2021.

[Boo84] George Boolos. Don't eliminate cut. *Journal of Philosophical Logic*, pages 373–378, 1984.

[BSŻC93] Rajiv Bagai, Vasant Shanbhogue, JM Żytkow, and Shang-Ching Chou. Automatic theorem generation in plane geometry. In *International Symposium on Methodologies for Intelligent Systems*, pages 415–424. Springer, 1993.

[Bun88] Alan Bundy. The use of explicit plans to guide inductive proofs. In *International conference on automated deduction*, pages 111–120. Springer, 1988.

[CBW99] Simon Colton, Alan Bundy, and Toby Walsh. Automatic concept formation in pure mathematics. In *The 16th international joint conference on Artificial Intelligence - IJCAI '99*, 1999.

[CBW00] Simon Colton, Alan Bundy, and Toby Walsh. On the notion of interestingness in automated mathematical discovery. *International Journal of Human-Computer Studies*, 53(3):351–375, 2000.

[CS97] Alessandra Carbone and S Semmes. Making proofs without modus ponens: An introduction to the combinatorics and complexity of cut elimination. *Bulletin of the American Mathematical Society*, 34(2):131–159, 1997.

[DL82] Randall Davis and Douglas B Lenat. *Knowledge-Based Systems in Artificial Intelligence: 2 Case Studies*. McGraw-Hill, Inc., 1982.

[DVB+21] Alex Davies, Petar Veličković, Lars Buesing, Sam Blackwell, Daniel Zheng, Nenad Tomašev, Richard Tanburn, Peter Battaglia, Charles Blundell, András Juhász, et al. Advancing mathematics by guiding human intuition with ai. *Nature*, 600(7887):70–74, 2021.

[Eps87] Susan L Epstein. On the discovery of mathematical theorems. In *IJCAI*, pages 194–197, 1987.

[Eps88] Susan L Epstein. Learning and discovery: One system's search for mathematical knowledge. *Computational Intelligence*, 4(1):42–53, 1988.

[Faj88] Siemion Fajtlowicz. On conjectures of graffiti. *Discret. Math.*, 72(1-3):113–118, 1988.

[IB96] Andrew Ireland and Alan Bundy. Productive use of failure in inductive proof. In *Automated Mathematical Induction*, pages 79–111. Springer, 1996.

[JLHW21] Albert Qiaochu Jiang, Wenda Li, Jesse Michael Han, and Yuhuai Wu. Lisa: Language models of isabelle proofs. In *Proceedings of the 6th Conference on Artificial Intelligence and Theorem Proving, AITP 2021, 5-11 September 2021, Aussois, France*, 2021.

[KUMO18] Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Miroslav Olsák. Reinforcement learning of theorem proving. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 8836–8847, 2018.

[Lak15] Imre Lakatos. *Proofs and refutations: The logic of mathematical discovery*. Cambridge university press, 2015.

[Len76] Douglas Bruce Lenat. *AM: an artificial intelligence approach to discovery in mathematics as heuristic search.* Stanford University, 1976.

[LYWP21] Wenda Li, Lei Yu, Yuhuai Wu, and Lawrence C. Paulson. Isarstep: a benchmark for high-level mathematical reasoning. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021.* OpenReview.net, 2021.

[NP18] Yutaka Nagashima and Julian Parsert. Goal-oriented conjecturing for isabelle/hol. *CoRR*, abs/1806.04774, 2018.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[PB10] Lawrence C. Paulson and Jasmin Christian Blanchette. Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. In Geoff Sutcliffe, Stephan Schulz, and Eugenia Ternovska, editors, *The 8th International Workshop on the Implementation of Logics, IWIL 2010, Yogyakarta, Indonesia, October 9, 2011*, volume 2 of *EPiC Series in Computing*, pages 1–11. EasyChair, 2010.

[PHZ+22] 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.

[PS20] Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393*, 2020.

[RGM+21] Gal Raayoni, Shahar Gottlieb, Yahel Manor, George Pisha, Yoav Harris, Uri Mendlovic, Doron Haviv, Yaron Hadad, and Ido Kaminer. Generating conjectures on fundamental constants with the ramanujan machine. *Nature*, 590(7844):67–73, 2021.

[Rud92] Piotr Rudnicki. An overview of the mizar project. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, pages 311–330, 1992.

[RWC+19] 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.

[UJ20] Josef Urban and Jan Jakubův. First neural conjecturing datasets and experiments. In *International Conference on Intelligent Computer Mathematics*, pages 315–323. Springer, 2020.

[Wan60] Hao Wang. Toward mechanical mathematics. *IBM J. Res. Dev.*, 4(1):2–22, 1960.

[Wil95] Andrew Wiles. Modular elliptic curves and fermat's last theorem. *Annals of mathematics*, 141(3):443–551, 1995.

[YD19] Kaiyu Yang and Jia Deng. Learning to prove theorems via interacting with proof assistants. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 6984–6994. PMLR, 2019.

Appendix: a review of mathematical conjecturers

In a 1960 piece, Wang pointed out that making interesting conjectures is less easily mechanisable than formalising proofs [Wan60]. Indeed, compared with the research on automated theorem proving, conjecturing has received much less attention. Here we look at some prior works on mechanising mathematical conjecturing, including both symbolic and learning-based methods.

[Len76, DL82] described the **AM program**, which can reinvent important concepts in set theory and number theory, given basic facts such as sets and bags. AM is able to conjecture generalisations of existing concepts, among the discovery of other concepts, based on 242 heuristics. [Eps87, Eps88] detailed the **GT program** , which does concept forming, conjecture making, and theorem proving in graph theory. Graphs are very carefully represented such that efficient automation of these activities can be done. **The Graffiti program** [Faj88] maked numerical conjecutres on graph theory. Whenever a conjecture is made, the program tries to refute them using a database of graphs. Those that were not refuted were left as the final conjectures. **Bagai et. al's system** [BSŻC93] made and proved conjectures in plane geometry of the form that certain diagrams cannot be constructed. **The HR program** [CBW99] applied to many finite algebras, as well as number theory and graph theory. HR used seven production rules to find new concepts from old ones.

One common feature of these programs is that their domains of applications are relatively narrow. This is due to that representations of mathematical concepts are very different across different domains. It is thus difficult to design a symbolic algorithm to find new concepts that have a wide range of application areas. **The Ramanujan machine** [RGM+21] conjectures polynomial continued fractions that equate to fundamental constants, and **Davies et. al's system** [DVB+21] hints mathematicians about important relations in knot theory and representation theory. Although using learning, their representations are also very hard to extend.

Works that relate the most to our paper are **PGT** [NP18], **proof planners** [Bun88] and **critics** [IB96]. PGT [NP18] generates conjectures by mutating the goals and uses multiple filters to make sure that they were useful and not easily refutable. This approach requires the conjecture to lead directly to the goal (the gap can be closed by fastforce). Proof planners [Bun88] specify the high-level structure of a proof and proof critics [IB96] try to come up with useful conjectures from failed proofs. Both utilise the proof premises and goals extensively. These three methods all have a formal logic backend and thus are potentially very general. But they all require the conjectures to be similar to the premises or goals of the theorem, while the conjecture that is half way between them is the most effective at reducing the size of the search space. Proof planners and critics also need explicit instructions in the meta-logic of the proof assistant and can be costly to deploy.

A refreshing attempt was **Urban and Jakubův's system** [UJ20], where they fed theorems in Mizar [Rud92] in textual form to a GPT-2 style transformer [RWC+19] and directly sampled new theorem statements from it. However, the sampling was purely unconditional, so the generated statements could be seen as random extrapolations of other theorems in the latent space. Unsurprisingly, most generated theorems ended up quite trivial; how they related to other theorems, if at all, was entirely opaque. The **IsarStep** dataset [LYWP21] consists of intermediate conjectures in the Isabelle proof assistant [NPW02], but it suffers from requiring conjectures to be equivalent to the ground truth, when multiple equally valid proofs may have non-equivalent conjectures.

Exploring Representation of Horn Clauses using GNNs

Chencheng Liang[1], Philipp Rümmer[1,2], and Marc Brockschmidt[3]

[1] Uppsala University, Uppsala, Sweden
[2] University of Regensburg, Regensburg, Germany
[3] Microsoft Research, Cambridge, United Kingdom

Automatic program verification has been used in safety-critical industrial software for decades. Constrained Horn Clauses (CHCs) [7] as an intermediate verification language is a standard representation of program verification problems. The program is safe if and only if the CHCs are satisfied. In practice, it is essential to extract information from program features (e.g., loops, control flow, or data flow) to guide the CHC solvers. For instance, the authors of [9] and [4] perform static analysis systematically to extract semantic program features (e.g., loop variables) to guide refinement process in the counterexample-guided abstraction refinement (CEGAR) [3] based solver. In recent years, along with breakthrough practices in deep learning [10, 8, 6, 20], many studies [19, 2, 14, 15, 19] have introduced deep learning methods to guide program verification and produce promising results. In particular, since graphs can represent highly structured relations naturally, some closely related fields, such as automatic reasoning, theorem proving, and SAT solving, begin to use the graph to represent logic formulas and apply graph neural networks (GNNs) [1] to learn the features to guide the solving process. Works such as FormulaNet [21], LERNA [13], NeuroSAT [17, 18], [12], and [11] have used this graph-based framework to improve their results by various learning tasks, e.g., premise selection and unsat-core prediction. However, to the best of our knowledge, we did not see any study which encodes CHCs to graph representations and use GNNs to learn the program features.

We believe GNNs can learn useful program features from graph represented CHCs to guide CHC solvers. In this work, to evaluate our assumption, we first answer two preliminary questions: (1) What kind of graph representation is suitable for CHCs? (2) Which kind of GNN is suitable for learning CHC graph representations?

To answer the first question, we have designed two graph representations (see Figure 1) of CHCs. Our *constraint graph* (CG) representation emphasizes the syntactic information of CHCs by constructing abstract syntax trees for constraints and building binary connections for relation symbols and their arguments. Our *control- and data-flow hypergraph* (CDHG) emphasizes semantic information of programs by using (ternary) hyperedges to represent the flow of control and data. To better express control- and data-flow, we construct CDHG from normalized CHCs. The normalization adding extra clauses to the original CHC but retains logical meaning.

For the second question, we introduce a new Relational Hypergraph Neural Network (R-HyGNN) architecture which is an extension of a message-passing GNN, namely, Relational Graph Convolutional Networks (R-GCN) [16]. In R-HyGNN, messages exchanged between nodes are computed from the representations of all nodes connected by typed edges. Then, the messages from all typed edges are aggregated to update the node representations.

To evaluate our framework, we introduce five proxy tasks (see Table 1) with increasing difficulties. Task 1 is a trivial sanity check, evaluating whether models can recover information from the initial node features. Task 2 evaluates the ability of models to handle counting problems in the overall graph. Task 3 requires the models to answer basic questions about the wider graph structure. Task 4 is significantly harder than the previous task, requiring the model to infer if a program variable is bounded from below or above. Finally, Task 5 is much harder, as it requires implicitly identifying counter-examples (CEs) traces. Moreover, we hope

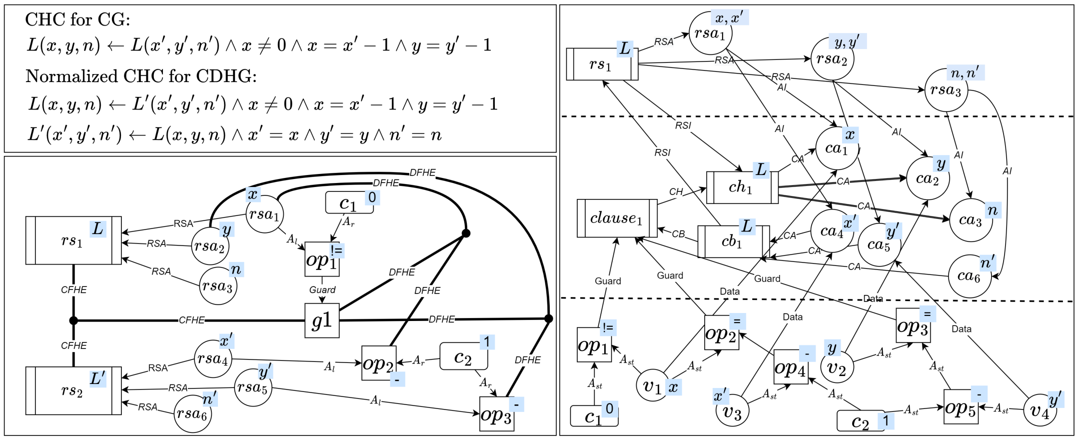


Figure 1: A CHC and the corresponding normalized CHCs are in the left upper corner. The CDHG constructed from the normalized CHCs is in the left lower corner. The CG for the CHC is on the right side. The texts on nodes and edges indicate the types of nodes and edges. To better illustrate the graphs, we add the blue boxes with text on nodes to relate the corresponding concrete symbol names in CHCs.

Task description	CG		CDHG	
1. If a node is an argument of a relation symbol	100% (95%)		99% (73%)	
2. How many times a relation symbol occurs in all clauses	1.0		4.2	
3. If a typed node is in a cycle	96% (70%)		99% (51%)	
4. If a relation symbol argument has **upper** and **lower** bound	**upper** 91% (80%)	**lower** 91% (75%)	**upper** 94% (75%)	**lower** 94% (68%)
5. If a clause occurs in **some** or **all** minimum CEs	**some** 95% (85%)	**all** 84% (53%)	**some** 96% (86%)	**all** 90% (55%)

Table 1: Description and experimental results for five proxy tasks. Task 2 performs regression task on nodes and is measured by mean square error, while other tasks perform binary classification task on nodes and are measured by accuracy. Both the fourth and fifth task consists of two independent binary classification tasks. The values in parentheses are the ratios of the dominant labels in the binary data distribution. Note that the label distribution differs for the two graph representations, as CDHGs are constructed from normalized CHCs.

that learning models on the five representative proxy tasks can reduce the bias from adapting to a particular application.

The test data is extracted from 8705 linear and 8425 non-linear Linear Integer Arithmetic (LIA) problems in CHC-COMP repository (see Table 1 in the competition report [5]). We divide the extracted dataset to train, valid, and test set by 60%, 20%, and 20%, respectively. The experimental results on the test set are shown in Table 1. As expected, for both graph representations, the performance of GNN models decreases along with the increasing difficulty of the tasks. However, even for the hardest (fifth) task, the accuracy is far higher than predicting the data distribution (values in the parentheses in Table 1), indicating that the models learn more than trivial patterns. In particular, we see a slight advantage of using the hypergraph representation (CDHG) comparing with binary graph representation (CG). We plan to use this framework to support predicate selection of CEGAR-based program verification.

References

[1] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinícius Flores Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Çaglar Gülçehre, H. Francis Song, Andrew J. Ballard, Justin Gilmer, George E. Dahl, Ashish Vaswani, Kelsey R. Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matthew Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks. *CoRR*, abs/1806.01261, 2018.

[2] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural code comprehension: A learnable representation of code semantics. *CoRR*, abs/1806.07336, 2018.

[3] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[4] Yulia Demyanova, Philipp Rümmer, and Florian Zuleger. Systematic predicate abstraction using variable roles. In *NASA Formal Methods*, pages 265–281, Cham, 2017. Springer International Publishing.

[5] Grigory Fedyukovich and Philipp Rümmer. Competition report: CHC-COMP-21, 2021.

[6] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014.

[7] Alfred Horn. On sentences which are true of direct unions of algebras. *The Journal of Symbolic Logic*, 16(1):14–21, 1951.

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[9] Jérôme Leroux, Philipp Rümmer, and Pavle Subotić. Guiding Craig interpolation with domain-specific abstractions. *Acta Informatica*, 53(4):387–424, Jun 2016.

[10] Tomas Mikolov, Martin Karafiát, Lukás Burget, Jan Honza Cernocký, and Sanjeev Khudanpur. Recurrent neural network based language model. In *INTERSPEECH*, 2010.

[11] Miroslav Olsak and Josef Urban Cezary Kaliszyk. Property invariant embedding for automated reasoning. *arXiv*, 2019.

[12] Aditya Paliwal, Sarah M. Loos, Markus N. Rabe, Kshitij Bansal, and Christian Szegedy. Graph representations for higher-order logic and theorem proving. *CoRR*, abs/1905.10006, 2019.

[13] Michael Rawson and Giles Reger. *A Neurally-Guided, Parallel Theorem Prover*, pages 40–56. 08 2019.

[14] Cedric Richter, Eyke Hüllermeier, Marie-Christine Jakobs, and Heike Wehrheim. Algorithm selection for software validation based on graph kernels. *Automated Software Engineering*, 27(1):153–186, June 2020.

[15] Cedric Richter and Heike Wehrheim. Attend and represent: A novel view on algorithm selection for software verification. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1016–1028, 2020.

[16] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks, 2017.

[17] Daniel Selsam and Nikolaj Bjørner. Neurocore: Guiding high-performance SAT solvers with unsat-core predictions. *CoRR*, abs/1903.04671, 2019.

[18] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT solver from single-bit supervision. *CoRR*, abs/1802.03685, 2018.

[19] Xujie Si, Aaditya Naik, Hanjun Dai, Mayur Naik, and Le Song. Code2inv: A deep learning framework for program verification. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II*, page 151–164, Berlin, Heidelberg, 2020. Springer-Verlag.

[20] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, Jan 2016.

[21] Mingzhe Wang, Yihe Tang, Jian Wang, and Jia Deng. Premise selection for theorem proving by deep graph embedding. In *Advances in Neural Information Processing Systems 30*, pages 2786–2796. Curran Associates, Inc., 2017.

Using machine learning to detect non-triviality of knots via colorability of knot diagrams

Alexei Lisitsa[1] and Alexei Vernitski[2]

[1] University of Liverpool, Liverpool, UK
a.lisitsa@liverpool.ac.uk
[2] University of Essex, UK
asvern@essex.ac.uk

Abstract

We apply machine learning to combinatorial knot theory, specifically, we consider a classical problem of deciding if a knot diagram represents the trivial knot as a classification problem. As a part of this process, we use a reformulation of this problem expressed via so-called Fox coloring of knot diagrams or, more generally, coloring knot diagrams with elements of algebraic structures called quandles.

Introduction

Knot theory is a branch of mathematics in which being assisted by machine learning feels especially attractive and promising, since small and numerous illuminating examples and counterexamples can be built successfully; let us discuss recent examples of such studies. In [24] the authors consider the problem of classification of 5 types of simple knots in the polymers where polymers are encoded by sequences of monomers, and train feed-forward neural networks and (with much better results) recurrent neural networks for this classification task. In [15] encoding of knots by rectangular diagrams was used and bidirectional LSTM networks were trained to recognize 36 knots types. In [12] reinforcement learning was used to untangle knot diagrams presented in braid encoding. In [16] and in our ongoing research we used reinforcement learning (multi-agent Q-learning and deep learning) to untangle braids. In [18] we compared performance of machine learning in testing realizability of Gauss diagrams with that of humans. In [4, 5, 6] machine learning is applied to studying various knot invariants.

A *quandle* is an algebraic structure whose binary operation is a generalization of the operation of conjugation in a group; see, for example, [7]. Quandles were introduced in [14, 23] as a powerful knot invariant. To be precise, the fact whether the arcs of a knot diagram can be colored by elements a given quandle (with certain conditions satisfied at the crossings) is a knot invariant. In [11, 10, 9] this approach was combined with automated reasoning and SAT solving to detect trivial knots and, more generally, to recognize knots; see also [3]. In this study we use machine learning to recognize colorability of knot diagrams with quandles and, therefore, to detect non-trivial knots.

In general, the efficient detection of non-trivial knots remains a challenge. The problem belongs to a complexity class NP $\cap$ co-NP [21, 13] and polynomial time algorithms for it are unknown. Very recently quasi-polynomial time algorithm for unknot detection was proposed in [22]. The recent work on machine learning applied to unknot detection [24, 15, 12] has shown encouraging performance of learned classifiers for this algorithmically difficult problem. The research reported in this paper continues the work in this direction and has the following novel features. We use the most traditional encoding of knots by realiziable Gauss codes/diagrams [1] and by more recent petal diagrams [2]. We apply classical machine learning algorithms, such as multilayered perceptrons/feed-forward neural networks. We use approximations of unknotedness by quandle colorability.

Methodology and details of implementation

In our experiments in this study we used two approaches to representing knot diagrams in the computer. In one approach, we used classical Gauss codes/diagrams [1]. To produce a dataset, a pre-defined amount of random Gauss diagrams[1] is generated using our tool [17], then diagrams are checked for realisability using the algorithm for signed realiziability from [20], and then, the variants are produced by varying at each crossing, which arc goes above or below the crossing. In another approach, we used petal diagrams of knots [8, 2], and to produce a dataset, we chose random permutations indicating in which order arcs pass behind each other at the crossing. Whereas standard permutation matrices were successful, we were more successful when we represented permutations by new ternary matrices, inspired by an encoding of permutations as a certain list of numbers called the Lehmer code or the inversion table [19]. Namely, we represent a permutation p by a matrix in which the entry at i, j is equal to 1 (or -1, or 0) if p does not swap the order of i and j (if p swaps the order of i and j, if $i = j$). The second step in creating the training set and the test set was finding out, for these randomly generated knot diagrams, whether they represent the trivial knot or a non-trivial knot. In this study, instead of attempting to untangle the knot, we replace this question by the question of colorability by certain quandles. At this step, we used two approaches. One approach was coloring by quandles of small sizes. Another approach was coloring by quandles induced by cyclic groups, which is equivalent to finding the number called the *determinant* of the knot diagram. Why do we consider the question of quandle colorability instead of the question of being the trivial knot? There are several reasons for this. Firstly, quandle colorability is an interesting research area in its own right [11, 10, 9, 3]. Secondly, it is known that for small sizes of diagrams, a diagram represents the trivial knot if and only if it cannot be colored by one of several small quandles [11, 3] or has a particular value of the determinant. Thirdly, even for larger diagrams, colorability by quandles of small size is a good approximation to being a non-trivial knot.

Experiment results

Table 1 presents some of the results of ongoing work in the first approach. G and EG in the names of datasets are referring to Gauss and Extended Gauss notation, respectively, in a sense of [1]; SQ-N is referring to the the initial segment of N quandles from a sequence SC of all simple quandles of small size used in [10]. #Frames are referring to the number of different unsigned diagrams used in the generation of datasets of signed diagrams.

Dataset	Size of dataset	Size of diagrams	#Frames	Quandle set	Accuracy
1-SQ-EG-8all	3072	8	6	SQ-1	75.3%
2-SQ-EG-8all	3072	8	6	SQ-2	65.2%
5-SQ-EG-8all	3072	8	6	SQ-5	62.3%
25-SQ-EG-8all	3072	8	6	SQ-25	55.2%
3Q-11-G-1x1024	2048	11	1	SQ-1	86.5%
3Q-11-G-4x250	2000	11	4	SQ-1	65.3%
3Q-11-G-20x200	8000	11	20	SQ-1	59.2%

Table 1: The accuracy of MLP (Multi-Layered Perceptron) of recognition of colorability of knot diagrams by sets of quandles (by any in a set); diagrams are encoded by “one hot” encoding from [18]; 70% training/30% testing split; WEKA Workbench [25] is used with default settings for MLP

Our initial results shows that the classical machine learning model of perceptron demon-

[1] random permutations and encoding of diagrams by permutations[17] are used here

strates good performance for the recognition of quandle colorability of knot diagrams, especially for the cases of colorability by a single quandle (SQ-1 set consists of single 3-element quandle) and for the datasets with small number of frames. Increasing the number of quandles and the number of frames leads to some degradation of the accuracy of learned models.

In the second approach we considered petal diagrams[2] of size 7 (there are $7! = 5040$ petal diagrams of this size, in total) and trained a binary classifier to distinguish between the trivial knot and non-trivial knots. We used the training set consisting of an equal number ($500+500 = 1000$) of petal diagrams whose determinant is 1 (they represent the trivial knot) and petal diagrams whose determinant is not 1 (they represent non-trivial knots 3_1, 4_1, 5_1, or 5_2 [2]). If permutations are presented by their permutation matrices, some learning occurs successfully, with the accuracy on the training set 100% and the accuracy on the test set around 80%. We also introduced a new way of presenting permutations by ternary matrices (see the definition above), instead of permutation matrices, and the accuracy on the test set increased to around 96%. For these experiments, we use Keras and TensorFlow in Python, and the binary classifier is a feed-forward neural network with one hidden layer of size 100, with an input layer of size $7 \times 7 = 49$ and a softmax output layer. Our results for this approach indicate that indeed the recognition of diagrams with determinant 1 is learnable and the accuracy is dramataically increased by using novel encoding by ternary matrices.

References

[1] Gauss notation. https://knotinfo.math.indiana.edu/descriptions/gauss_notation.html. Accessed: 2022-05-10.

[2] Colin Adams, Thomas Crawford, Benjamin DeMeo, Michael Landry, Alex Tong Lin, MurphyKate Montee, Seojung Park, Saraswathi Venkatesh, and Farrah Yhee. Knot projections with a single multi-crossing. *Journal of Knot Theory and Its Ramifications*, 24(03):1550011, 2015.

[3] W Edwin Clark, Mohamed Elhamdadi, Masahico Saito, and Timothy Yeatman. Quandle colorings of knots and applications. *Journal of knot theory and its ramifications*, 23(06):1450035, 2014.

[4] Jessica Craven, Mark Hughes, Vishnu Jejjala, and Arjun Kar. Learning knot invariants across dimensions. arXiv:2112.00016, 2021.

[5] Alex Davies, András Juhász, Marc Lackenby, and Nenad Tomasev. The signature and cusp geometry of hyperbolic knots. arXiv:2111.15323, 2021.

[6] Alex Davies, Petar Veličković, Lars Buesing, Sam Blackwell, Daniel Zheng, Nenad Tomašev, Richard Tanburn, Peter Battaglia, Charles Blundell, András Juhász, Marc Lackenby, Geordie Williamson, Demis Hassabis, and Pushmeet Kohli. Advancing mathematics by guiding human intuition with AI. *Nature*, 600:70–74, 2021.

[7] Mohamed Elhamdadi and Sam Nelson. *Quandles*, volume 74. American Mathematical Soc., 2015.

[8] Chaim Even-Zohar, Joel Hass, Nati Linial, and Tahl Nowik. Invariants of random knots and links. *Discrete & Computational Geometry*, 56(2):274–314, 2016.

[9] Andrew Fish and Alexei Lisitsa. Detecting unknots via equational reasoning, I: exploration. In Stephen M. Watt, James H. Davenport, Alan P. Sexton, Petr Sojka, and Josef Urban, editors, *Intelligent Computer Mathematics - International Conference, CICM 2014, Coimbra, Portugal, July 7-11, 2014. Proceedings*, volume 8543 of *Lecture Notes in Computer Science*, pages 76–91. Springer, 2014.

[10] Andrew Fish, Alexei Lisitsa, and David Stanovský. A combinatorial approach to knot recognition. In Ross Horne, editor, *Embracing Global Computing in Emerging Economies - First Workshop,*

[2] We are grateful to Chaim Even-Zohar for sharing some of his code with us.

EGC 2015, Almaty, Kazakhstan, February 26-28, 2015. Proceedings, volume 514 of *Communications in Computer and Information Science*, pages 64–78. Springer, 2015.

[11] Andrew Fish, Alexei Lisitsa, David Stanovský, and Sarah Swartwood. Efficient knot discrimination via quandle coloring with SAT and #-SAT. In Gert-Martin Greuel, Thorsten Koch, Peter Paule, and Andrew J. Sommese, editors, *Mathematical Software - ICMS 2016 - 5th International Conference, Berlin, Germany, July 11-14, 2016, Proceedings*, volume 9725 of *Lecture Notes in Computer Science*, pages 51–58. Springer, 2016.

[12] Sergei Gukov, James Halverson, Fabian Ruehle, and Piotr Sułkowski. Learning to unknot. arxiv:2010.16263, 2020.

[13] Joel Hass, Jeffrey C. Lagarias, and Nicholas Pippenger. The computational complexity of knot and link problems. *J. ACM*, 46(2):185–211, mar 1999.

[14] D. Joyce. Simple quandles. *J. Algebra*, 79:307–318, 1982.

[15] L. H. Kauffman, N. E. Russkikh, and I. A. Taimanov. Rectangular knot diagrams classification with deep learning. arxiv:2011.03498, 2020.

[16] A. Khan, A. Vernitski, and A. Lisitsa. Untangling braids with multi-agent q-learning. In *2021 23rd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 135–139, Los Alamitos, CA, USA, dec 2021. IEEE Computer Society.

[17] Abdullah Khan, Alexei Lisitsa, and Alexei Vernitski. Gauss-lintel, an algorithm suite for exploring chord diagrams. In Fairouz Kamareddine and Claudio Sacerdoti Coen, editors, *Intelligent Computer Mathematics*, pages 197–202, Cham, 2021. Springer International Publishing.

[18] Abdullah Khan, Alexei Lisitsa, and Alexei Vernitski. Training ai to recognize realizable gauss diagrams: The same instances confound ai and human mathematicians. In *Proceedings of the 14th International Conference on Agents and Artificial Intelligence - Volume 3: ICAART,*, pages 990–995. INSTICC, SciTePress, 2022.

[19] D.E. Knuth. *The Art of Computer Programming: Volume 3: Sorting and Searching*. Pearson Education, 1998.

[20] Vitaliy Kurlin. Gauss paragraphs of classical links and a characterization of virtual link groups. *Mathematical Proceedings of the Cambridge Philosophical Society*, 145(1):129–140, 2008.

[21] Marc Lackenby. The efficient certification of knottedness and Thurston norm. arxiv:1604.00290, 2016.

[22] Marc Lackenby. Unknot recognition in quasi-polynomial time. https://video.ucdavis.edu/media/quasipolynomial-unknot/1_w3i5jvqi, 2021. record of the talk, Accessed: 2022-08-18.

[23] S. V. Matveev. Distributive groupoids in knot theory. *Math. USSR - Sbornik*, 47(1):73–83, 1984.

[24] Olafs Vandans, Kaiyuan Yang, Zhongtao Wu, and Liang Dai. Identifying knot types of polymer conformations by machine learning. *Phys. Rev. E*, 101:022502, Feb 2020.

[25] Ian H. Witten, Eibe Frank, Mark A. Hall, and Christopher J. Pal. *The WEKA Workbench. Online Appendix for Data Mining, Fourth Edition: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2016.

Reinforcement Learning in E

Jack McKeown and Geoff Sutcliffe

University of Miami, Miami, Florida, U.S.A.
jam771@miami.edu, geoff@cs.miami.edu

1 Introduction

Modern saturation based theorem provers such as E [8] and Vampire [5] rely on heuristics to guide their search. Historically, manually engineered heuristics have been the norm. More recently, advances in machine learning have inspired attempts to leverage that power for guiding search. While most efforts have used supervised learning, reinforcement learning (RL) has also been successfully applied [3, 1, 6]. This work aims to incorporate RL into E in a way that generalizes the approach taken by E's `--auto` mode. In the words of E's documentation:

> Clause selection is determined by a heuristic evaluation function, which conceptually sets up a set of priority queues and a weighted round robin scheme that determines from which queue the next clause is to be picked.

The order of each priority queue is dictated by its corresponding Clause Evaluation Function (CEF). When E is invoked on a problem in `--auto` mode, E analyzes the problem and selects a set of CEFs and a weighted round-robin *CEF-schedule*. The CEF-schedule is used for all given clause selections throughout the proof attempt. The approach taken in this work is to replace the CEF-schedule with an external mapping from the state of E to a CEF to use for the next given clause selection. When this mapping is allowed to be stochastic, it can be thought of as mapping the state of E to a probability distribution over CEFs.

2 Reinforcement Learning Framing

The *policy* of an RL agent is a mapping from the *state* of the RL *environment* to a categorical probability distribution over the available *action*s. When queried with a state, s, the agent samples an action, a, from the distribution $\pi_\theta(s)$, and receives a *reward* from the environment.

Previous approaches to RL for ATP have mostly been *tableaux-based* where the state features are derived from the tableaux and actions are tableaux extension steps [3]. The latest *saturation-based* approaches to RL for ATP essentially represent state as the clauses in the processed set and represent actions as the potential given clauses from the unprocessed set [1, 6]. These sets of clauses are encoded via neural networks such as Graph Neural Networks. Rewards are typically given for completing proofs.

In this work, reinforcement learning is incorporated into E as follows: E is invoked on a problem with a fixed set of CEFs. The state of E is sent to the agent each time E needs to select a given clause. At the time of writing, the policy being used, π_θ, is implemented by a shallow neural network with ReLU activation on the hidden layers and a softmax activation function on the output layer. The state consists of 4 features: the number of clauses and average clause weight within the unprocessed and processed sets, but many possible features exist. The agent responds with one of the CEFs as its action. E chooses a given clause using the corresponding CEF. If the selection completes a proof then the reward is one, otherwise the reward is zero. The

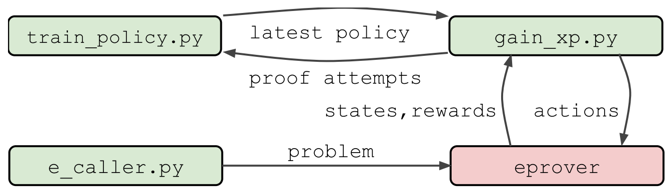


Figure 1: Experimental Setup and Training Architecture

parameters of the policy are learned via Monte-Carlo policy gradients (aka REINFORCE [9]). Originally, an epsilon-greedy strategy with Q-learning was considered, but was rejected due to concerns that the agent would heavily favor one CEF in underexplored regions of state-space.

3 Architecture

The architecture shown in Figure 1 is used to train π_θ:

1. A python script `e_caller.py` repeatedly invokes E on problems from the "bushy" problems of the MPTP2078 dataset [2]. This script allows for control of the sampling distribution over problems and can be used to focus training on easy or hard problems.

2. A python script `gain_xp.py` receives states and rewards from E and sends actions to E. It chooses actions via a saved policy and saves completed proof attempts to disk. All communication between E and `gain_xp.py` is performed using named pipes.

3. A python script `train_policy.py` continually updates the policy used by `gain_xp.py` using the saved proof attempts.

Multiple instances of `e_caller` and `gain_xp` are run in parallel to speed up learning.

4 Initial Results

For the results shown in Table 1, a constant list of 75 CEFs was extracted from the CEF-schedules employed by E's `--auto` mode. `e_caller.py` was configured to try all MPTP2078 bushy problems, and E is invoked with a timeout of 20 seconds. To see if any learning occurs, a uniform distribution over the 75 CEFs (ignoring state) is used as a baseline policy. The simplest extension of this policy is to learn constant probabilities with which to choose each CEF (still ignoring state). Next, a simple neural network that takes state into account is considered. Finally, these policies are compared to E's `--auto` mode.

Approach	**Solved**
Uniform Distribution	1105
Learned Distribution	1105
Simple Neural Network	1110
E –auto	1156

Table 1: Number of MPTP2078 Bushy Problems solved by different approaches

5 Concerns and Future Work

There are various reasons why E's auto mode still has an advantage over the approaches presented here. The most apparent reasons are that E's auto mode has a much richer space of features, and that the neural network used in these experiments is very simple. A more insidious reason is that the CEF-schedules used in E's `--auto` mode were explicitly evolved to solve more problems, whereas the policy gradient training directly favors finding proofs more quickly. From an RL perspective, it seems that the goal of quickly solving easy problems may be in conflict with the goal of solving hard problems. Perhaps oversampling hard problems in `e_caller.py` could help with this. Alternatively, the policy gradient loss could be scaled by some notion of problem difficulty. Future work will also explore Actor-Critic [4] models, which typically outperform REINFORCE.

It is unclear whether CEF choices are an expressive enough action space for improving guidance using RL. *This* work opted for CEFs as actions because it is convenient, and perhaps more sample efficient, to have a consistent set of available actions. Another reason for using CEFs is that it side-steps the issue of having to represent clauses as input for neural networks. (Graph Neural Networks, Recurrent Neural Networks, and Recursive Neural Networks have all been used with varying degrees of success for this [7].) Despite these theoretical benefits to using CEFs as actions, with the exponential growth of the unprocessed set during a proof attempt, the CEFs can only represent a small fraction of the available given clause selections. If the best clause to select is not preferred by any of the CEFs, then it is impossible to select. One might also be concerned that 75 CEFs is too many for a first attempt at learning a CEF-schedule. As a response to this, a smaller list of 7 CEFs was established. These 7 CEFs were chosen to be very different from one-another in order to retain an expressive choice of action. Despite this, the results were qualitatively similar to the results in Table 1. The only appreciable difference was that with 7 CEFs, the models peaked at a much worse performance of around 43% problems proved.

References

[1] I. Abdelaziz, M. Crouse, et al. Learning to Guide a Saturation-Based Theorem Prover. *CoRR*, abs/2106.03906, 2021.

[2] J. Alama, D. Kühlwein, E. Tsivtsivadze, J. Urban, and T. Heskes. Premise Selection for Mathematics by Corpus Analysis and Kernel Methods. *CoRR*, abs/1108.3446, 2011.

[3] C. Kaliszyk, J. Urban, H. Michalewski, and M. Olsák. Reinforcement Learning of Theorem Proving. *CoRR*, abs/1805.07563, 2018.

[4] V. Konda and J. Tsitsiklis. Actor-Critic Algorithms. In S. Solla, T. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999.

[5] L. Kovacs and A. Voronkov. First-Order Theorem Proving and Vampire. In *Proceedings of the 25th International Conference on Computer Aided Verification*, number 8044 in Lecture Notes in Artificial Intelligence, pages 1–35. Springer-Verlag, 2013.

[6] G. Reger M. Rawson, A. Bhayat. Reinforced External Guidance for Theorem Provers. 2020.

[7] J. McKeown. Clause Representation for Proof Guidance using Neural Networks, 2021.

[8] S. Schulz, S. Cruanes, and P. Vukmirovic. Faster, Higher, Stronger: E 2.3. In *Proceedings of the 27th International Conference on Automated Deduction*, number 11716 in Lecture Notes in Computer Science, pages 495–507. Springer-Verlag, 2019.

[9] R. J. Williams. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Learning*, 8:229–256, 2004.

Evolutionary Computation for Program Synthesis in SuSLik

Yutaka Nagashima[1]

Independent Researcher, Cambridge, the United Kingdom
united.reasoning@gmail.com

Abstract

A deductive program synthesis tool takes a specification as input and derives a program that satisfies the specification. The drawback of this approach is that search spaces for such correct programs tend to be enormous, making it difficult to derive correct programs within a realistic timeout. To speed up such program derivation, we improve the search strategy of a deductive program synthesis tool, SuSLik, using evolutionary computation. Our cross-validation shows that the improvement brought by evolutionary computation generalises well to unforeseen problems.

1 Deductive Program Synthesis

A far-fetched goal of artificial intelligence (AI) research is to build a system that writes computer programs for humans. To achieve this goal, researchers take two distinct approaches for program synthesis: deductive program synthesis and inductive program synthesis.

Both approaches attempt to produce programs requested by human users. The difference lies how they produce programs and the guarantee of the resulting programs: deductive synthesis tries to *deduce* programs that satisfy specifications, while inductive program synthesis tries to *induce* programs from examples. A notable example of inductive program synthesis is the automated spreadsheet data manipulation implemented as an add-in for Microsoft Excel spreadsheet system [1].

While such inductive synthesis alleviates the burden of implementation by guessing programs from given input-output examples, in inductive synthesis the resulting programs are never trustworthy: there is always a risk that incorrect generalisation results in programs that are correct for the present examples but not for future cases.

Deductive synthesis overcomes this limitation with formal specifications: it allows users to formalise *what* they want as specifications, whereas inductive synthesis tools guess *how* programs should behave from examples provided by users. Thus, in deductive synthesis providing formal specifications remains as users' responsibility. The upside of deductive synthesis is, however, users can obtain *correct* programs automatically upon success. This correctness assurance is particularly useful when it comes to synthesising imperative programs with pointers, as manually developing heap-manipulating programs is known to be error-prone.

SuSLik [4], for example, is one of such deductive synthesis tools. It takes a specification provided by humans and attempts to produce heap-manipulating programs satisfying the specification in a language that resembles the C language. Internally, this derivation process is formulated as proof search: SuSLik composes a heap-manipulating program by conducting a best-first search for a proof goal presented as specification. The drawback is that the search algorithm often fails to find a proof within a realistic timeout. That is, even we pass a specification to SuSLik, SuSLik may not produce a program satisfying the specification. According to Itzhaky *et al.* [2],

experiment	gen-0	gen-20	gen-40
1st (32)	18	16	15
2nd (41)	21	21	15
3rd (31)	18	16	15
4th (31)	16	13	13

(a) Unsolved problems in the training set

experiment	gen-0	gen-20	gen-40
1st (33)	22	16	16
2nd (24)	17	16	15
3rd (34)	22	18	16
4th (34)	24	21	18

(b) Unsolved problems in the validation set

> different synthesis tasks benefit from different search parameters, and that we might need a mechanism to tune SuSLik 's search strategy for a given synthesis task.

2 Evolutionary Computation for Better Search Strategies

To address this issue, we built an evolutionary framework that improves SuSLik's synthesis strategy. Basically, this framework tries to identify suitable search parameters for SuSLik's proof search strategy. These parameters include the weights associated with each step of search. Our artefact is publicly available at GitHub [3].

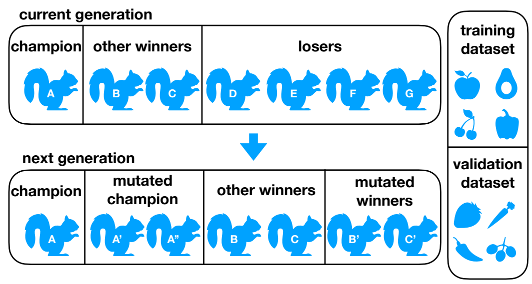


Figure 1: Evolution of SuSLik instances

In this framework, we firstly create a pair of specification sets: one for training and the other for validation. Secondly, we produce the initial population consisting of 40 instances of SuSLik by mutating the original search parameters. In each generation, we assign the specifications in the training set to each SuSLik instance. Then, we count how many specifications each SuSLik instance manages to solve within 2.5 seconds. We take 20 best performing instances and produce new mutants from them. Then, we pass these winners and their mutants to the next generation and repeat this process 40 times. To accelerate evolution, we allow the champion of each generation to produce two instances of mutants as shown in Figure 1.

We experimented our framework four times. Table 1a shows the results of training. For example, the second row in Table 1a reads as follows: in the first experiment 32 specifications fell into the set for training, and 18 specifications were left unsolved by the best SuSLik instance in the zeroth generation. This number decreased to 16 and 15 for the 20th and 40th generation, respectively.

In our experiments, we conducted cross-validation for each generation. Their results are shown in Table 1b. Note that we used a fixed pair of training set and validation set throughout the evolution of each experiment to maintain the distinction between the two sets. All these four experiments showed that improvements from training sets translates to improvements on validation sets despite the small size of dataset. That is, we found that

> there are strategies that tend to perform better for unforeseen problems, and we can find such strategies using genetic algorithms.

References

[1] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 317–330. ACM, 2011.

[2] Shachar Itzhaky, Hila Peleg, Nadia Polikarpova, Reuben N. S. Rowe, and Ilya Sergey. Deductive synthesis of programs with pointers: Techniques, challenges, opportunities - (invited paper). In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, volume 12759 of *Lecture Notes in Computer Science*, pages 110–134. Springer, 2021.

[3] Yutaka Nagashima. `https://github.com/yutakang/suslik/tree/evolutionary`, 2021.

[4] Nadia Polikarpova and Ilya Sergey. Structuring the synthesis of heap-manipulating programs. *Proc. ACM Program. Lang.*, 3(POPL):72:1–72:30, 2019.

Project Proposal: Learning Variable Mappings to Repair Programs

Pedro Orvalho[1]*, Jelle Piepenbrock[23], Mikoláš Janota[2], and Vasco Manquinho[1]

[1] INESC-ID, IST - U. Lisboa, Portugal
[2] CIIRC, Czech Technical University in Prague, Czechia
[3] Radboud University Nijmegen, The Netherlands

Abstract

The increasing demand for programming education has given rise to all kinds of online evaluations, such as Massive Open Online Courses (MOOCs) focused on introductory programming assignments (IPAs), especially over the last few years due to the coronavirus outbreak. As a consequence of a large number of enrolled students, one of the main challenges in these courses is to provide valuable and personalized feedback to students. This personalized feedback can be provided as a list of possible repairs to a student's program. Typically semantic program repair tools repair an incorrect program using a correct implementation for the same IPA. In order to compare both programs, a relation between both programs' sets of variables is required. Thus, in this work, we propose to learn how to map the set of variables between different small imperative programs based on both programs' abstract syntax trees (ASTs) using graph neural networks (GNNs).

Introduction. Program Synthesis, the task to automatically generate programs and mathematical objects that satisfy a given high-level specification [3, 12], is a well-studied problem in Theorem Proving [4, 5], and it has even been considered the Holy Grail of Computer Science [6, 9]. Program Repair can be seen as a special case of Program Synthesis, where a given program has a faulty region that needs to be repaired by synthesizing a correct patch or by reusing code snippets from other correct programs. Automated program repair [1, 7, 8, 13] has become crucial to provide feedback to each novice programmer by checking their introductory programming assignments (IPAs) submissions using a pre-defined test suite. Semantic program repair frameworks use a correct implementation, provided by the lecturer or submitted by a previously enrolled student, to repair a new incorrect student's submission. These tools need to compare both programs, i.e., the correct and the faulty implementation. In order to compare both programs, a relation between both programs' sets of variables is required. For example, consider both programs presented in Listings 1, where having a mapping between both programs' variables lets us reason about which repairs one should perform to fix the faulty program. In this position paper, we propose to take advantage of the structural information of the *abstract syntax trees (ASTs)* of small imperative programs to learn how to map the set of variables between a correct program and a faulty one using *graph neural networks (GNNs).*

IPAs Dataset. We used the C-Pack-IPAs [10] benchmark to evaluate this work. This benchmark is composed by student programs developed during an introductory programming course in C language were collected at Instituto Superior Técnico. First, we selected only submissions that compiled without any error and satisfied a set of input-output test cases for each IPA. Afterwards, we used MultIPAs [11], a program transformation tool that can augment IPAs benchmarks by performing program mutations and introducing bugs to the programs.

*This work was done while this author was visiting CIIRC, CTU in Prague.

Listing 1: Function that finds and returns the maximum number among `n1, n2` and `n3`.

```c
int max(int n1, int n2, int n3)
{
  int m = n1 > n2 ? n1 : n2;
  return n3 > m ? n3 : m;
}
```

Listing 2: Function that finds and returns the maximum number among `x, y` and `z`.

```c
int max(int x, int y, int z){
  int m = 0;
  m = x > m ? x : m;
  m = y > m ? y : m;
  return z > m ? z : m;
}
```

Listings 1: Both functions find and return the maximum number among their parameters' values. However, the function in Listing 2 is only correct for positive numbers, if we consider negative numbers the function is incorrect since it assigns the variable `m` to 0. The mapping between these programs' sets of variables is {`m` : `m`; `n1` : `x`; `n2` : `y`; `n3` : `z`}.

MultIPAs can perform simple mutations to each program (e.g. swapping comparison operators, swapping the if's then-block with the else-block and negating the test condition) to generate semantically equivalent programs with the same variables. Hence, we gathered a dataset of programs and the mappings between their sets of variables. For example, we have 94 correct submissions for the first IPA. By just swapping comparison operators (e.g. $\geq, \leq, ==, \neq$), we are able to compute a dataset of 27261 pairs of programs and the mappings between their sets of variables. We plan to perform more complex mutations to the set of IPAs.

Program Representations. We represent programs based on their abstract syntax trees (ASTs). An AST is described by a set of nodes that correspond to non-terminal symbols in the programming language's grammar and a set of tokens that correspond to terminal symbols. Then, we create a unique node in the graph for each distinct variable in the program and connect all the variable occurrences in the program to the same unique node. Regarding the edges of the program representation, we consider two types of edges in our representation: child and sibling edges. Child edges correspond to the typical edges in the AST representation that connect each parent node to its children. Child edges are bidirectional. Sibling edges connect each child to its sibling successor. These edges denote the order of the arguments for a given node [2]. Sibling edges allow the program representation to differentiate between different arguments when the order of the arguments is important (e.g. binary operation such as $\leq$). For example, consider the node that corresponds to the operation $\sigma(A_1, A_2, \ldots, A_m)$. The parent node σ is connected to each one of its children by a child edge e.g. $\sigma \leftrightarrow A_1, \sigma \leftrightarrow A_2, \ldots, \sigma \leftrightarrow A_m$. Additionally, each child its connected to its successor by a sibling edge e.g. $A_1 \rightarrow A_2, A_2 \rightarrow A_3, \ldots, A_{m-1} \rightarrow A_m$. The interested reader is referred to appendix A for a graphical representation of a small example.

GNNs. Graph Neural Networks are a subclass of neural networks designed to operate on graph-structured data, which may be citation networks, first-order logic or representations of computer code. Here, we use a pair of ASTs, representing two programs for which we want to match variables, as the input. The main operative mechanism is to perform *message passing* between the nodes, so that information about the global problem can be passed between the local constituents. The content of these messages and the final representation of the nodes is parameterized by neural network operations (matrix multiplications composed with a non-linear function). For the variable matching task, we do the following to train the parameters of the network. After several message passing rounds, through the edges defined by the program representations above, we obtain numerical vectors corresponding to each variable node in the two programs. We compute scalar products between each possible combination of variable nodes in the two programs, followed by a softmax function. As the correct mapping of variables is known because the samples are obtained by program mutation, we can compute a cross-entropy loss and minimize it so that the network output corresponds to the labeled variable matching.

Acknowledgments. This research was supported by Fundação para a Ciência e Tecnologia (FCT) through grant SFRH/BD/07724/2020, and projects UIDB/50021/2020, PTDC/CCI-COM/32378/2017; by European funds through COST Action CA2011; and by the Ministry of Education, Youth and Sports within the program ERC CZ under the project POSTMAN no. LL1902.

References

[1] Umair Z. Ahmed, Zhiyu Fan, Jooyong Yi, Omar I. Al-Bataineh, and Abhik Roychoudhury. "Verifix: Verified Repair of Programming Assignments". In: *ACM Trans. Softw. Eng. Methodol.* (2022). ISSN: 1049-331X. DOI: 10.1145/3510418. URL: https://doi.org/10.1145/3510418.

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. "Learning to Represent Programs with Graphs". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. 2018.

[3] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. "DeepCoder: Learning to Write Programs". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. 2017.

[4] Chad E. Brown and Thibault Gauthier. "Self-Learned Formula Synthesis in Set Theory". In: *CoRR* abs/1912.01525 (2019).

[5] Thibault Gauthier. "Synthesis of Recursive Functions from Sequences of Natural Numbers1". In: *6th Conference on Artificial Intelligence and Theorem Proving, AITP* (2021).

[6] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. "Program Synthesis". In: *Foundations and Trends in Programming Languages* 4.1-2 (2017), pp. 1–119.

[7] Sumit Gulwani, Ivan Radicek, and Florian Zuleger. "Automated clustering and program repair for introductory programming assignments". In: *PLDI 2018*. ACM, 2018, pp. 465–480.

[8] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade. "DeepFix: Fixing Common C Language Errors by Deep Learning". In: *AAAI 2017*. Ed. by Satinder P. Singh and Shaul Markovitch. AAAI Press, 2017, pp. 1345–1351.

[9] A Solar Lezama. "Program synthesis by sketching". PhD thesis. UC Berkeley, 2008.

[10] Pedro Orvalho, Mikoláš Janota, and Vasco Manquinho. *C-Pack of IPAs: A C90 Program Benchmark of Introductory Programming Assignments*. 2022. DOI: 10.48550/arXiv.2206.08768. URL: https://doi.org/10.48550/arXiv.2206.08768.

[11] Pedro Orvalho, Mikoláš Janota, and Vasco Manquinho. “MultIPAs: Applying Program Transformations to Introductory Programming Assignments for Data Augmentation”. In: *ESEC/FSE ’22: 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Singapore, November 14-18, 2022.* ACM, 2022. DOI: 10.1145/3540250.3558931. URL: https://doi.org/10.1145/3540250.3558931.

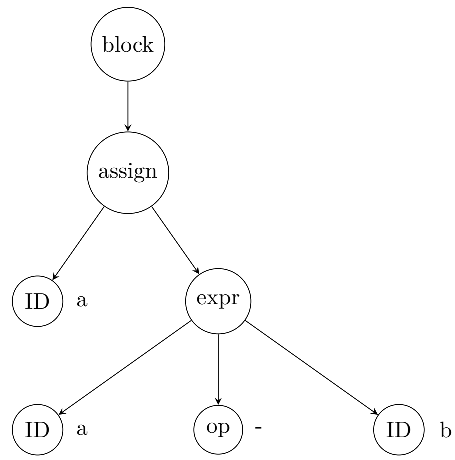
[12] Pedro Orvalho, Miguel Terra-Neves, Miguel Ventura, Ruben Martins, and Vasco M. Manquinho. “Encodings for Enumeration-Based Program Synthesis”. In: *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings.* 2019, pp. 583–599.

[13] Michihiro Yasunaga and Percy Liang. “Graph-based, Self-Supervised Program Repair from Diagnostic Feedback”. In: *ICML 2020.* Vol. 119. Proceedings of Machine Learning Research. PMLR, 2020, pp. 10799–10808.

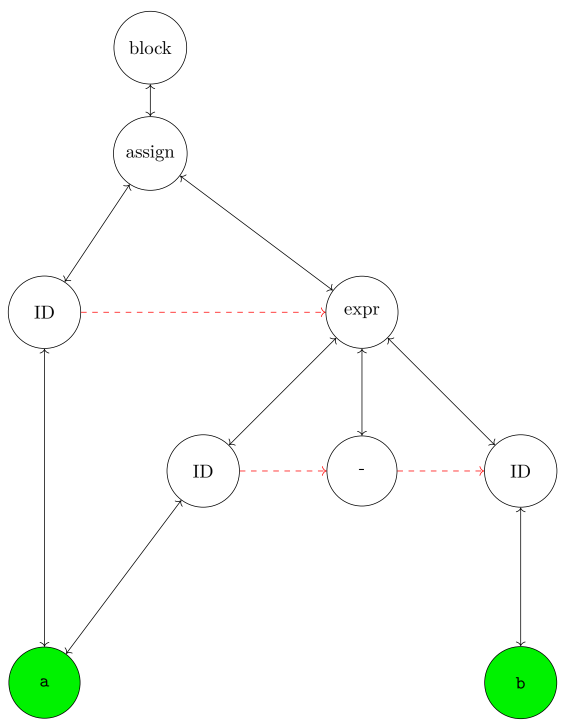
A Appendix

Listing 3: Small example of a C code block with an expression that uses int variables `a` and `b`, previously declared in the program.

```
{
  // a and b are ints
  a = a - b;
}
```



(a) Part of the AST representation of Listing 3.



(b) Our program representation for the program presented in Listing 3. We add additional variable nodes (green nodes), new sibiling edges (red dashed edges) and we also make the AST edges (black edges) bidirectional.

Figure 1: AST and our program representation for the small code snippet presented in Listing 3.

Synthetic Proof Term Data Augmentation for Theorem Proving with Language Models

Joseph Palermo[1], Johnny Ye[1], and Jesse Michael Han[2]

[1] Cash App Labs
[2] University of Pittsburgh*

Introduction

Imitation learning for the task of theorem proving is bottlenecked by the limited size of existing libraries of formalized mathematics (e.g. mathlib [1]). Prior work utilizing language models for theorem proving indicates that training data limitations are causing performance saturation [2, 3]. Prior work has also demonstrated the utility of synthetic data for improving language models [4, 5, 6, 7] and for learning theorem proving [8, 9, 10, 11, 12]. We propose using samples from trained language models in conjunction with the Lean kernel [13] to generate novel training examples. In particular, we train language models to generate Lean proof terms unconditioned by any theorem statement and we sample from these models to generate collections of proof term candidates. We then apply the Lean kernel to identify type-correct proof term candidates and infer corresponding types. From this synthetic data we construct training examples for proof term language modelling. Augmenting training sets by adding synthetic examples is shown to improve the performance of proof term language modeling on a held-out test set.

Bootstrap Datasets

In order to use trained language models to generate new candidate training examples, we first create a "bootstrap" dataset from which a model can be trained. Our examples for unconditioned proof term language modelling (which we call unconditioned examples) have the form: PROOF <proof> EOT. They are unconditioned in the sense that they are not conditioned on a theorem statement. We call the dataset comprising these examples: **unconditioned bootstrap**. This dataset is used to train the model that is sampled to produce synthetic examples. It is also used to train a baseline model and to provide examples for a held-out test set.

By contrast, our examples for theorem-conditioned proof term language modelling (which we call conditioned examples) have the form: THEOREM <theorem> PROOF <proof> EOT. We call the dataset comprising these examples: **conditioned bootstrap**. This dataset is used only to train a baseline model and to provide examples for a held-out test set.

As a data augmentation strategy, we perform tree traversal on each expression in mathlib to identify unique sub-expressions which are also valid proofs. We convert these filtered sub-expressions into pretty-printed text format and filter for length less than 2048 characters. We parse and type-check these proofs to ensure that pretty-printing has not rendered them invalid and to obtain the corresponding theorem statement.

*work for this project was completed while at OpenAI

Splitting the Bootstrap Datasets

We split our bootstrap datasets into train, validation and test sets, firstly by splitting on mathlib declaration names. We also apply a further filter to reduce the maximum similarity of training examples to validation and test examples. Using TF-IDF [14] embeddings of our examples we remove any validation or test example (x_{test}) for which:

$$\frac{levenstein\_distance(x_{test}, \arg\max_{x_{train}} cosine\_similarity(x_{test}, x_{train}))}{length(x_{test})} < 0.15 \tag{1}$$

The threshold value of 0.15 was determined after we found empirically that it enabled us to overfit our training data. Our initial split of declaration names produced 207,194 train examples, 55,470 validation examples and 54,964 test examples. After the additional filtering step, 11,145 validation examples and 10,233 test examples remain.

Creating Synthetic Examples

To generate proof candidates, we sample language models trained on the unconditioned bootstrap dataset. We parse and type-check proof candidates, and if type-check is successful we serialize the corresponding type. This process produces synthetic training examples of both the conditioned and unconditioned variety. We also generate an additional example from each proof candidate regardless of whether or not it has passed type-checking: NON_TYPE_CHECKED_PROOF <non_type_checked_proof> EOT. These examples are useful in assessing the effect type-check filtering has on data quality.

Bootstrap Training Sets vs. Augmented Training Sets

For these experiments, we train language models using Fairseq [15]. We utilize Fairseq's implementation of GPT-2 [16] with approximately 2 billion parameters (the so-called "big" size). We also utilize Fairseq's implementation of the "gpt2" byte pair encoder. We set max-tokens to 1536, use SGD with a fixed learning rate of 0.01, set early-stopping patience to 100 epochs, and set dropout to 0.1.

After training a model on the unconditioned bootstrap dataset, we use the trained model to sample 20 million proof candidates using beam search. We set the beam search temperature to 1.3 and beam width to 5. From the set of candidates, 1.57 % or 352,469 unique proofs passed type-check.

We create augmented datasets by randomly sampling synthetic examples without replacement and adding them to the bootstrap datasets. Samples are added until the augmented dataset in question is 100% larger than the corresponding bootstrap dataset as measured by the number of examples in the conditioned case and by the number of characters in the unconditioned case. We weight the additional unconditioned examples by counting characters because the synthetic unconditioned examples can include both type-checked and non-type-checked proofs, and the average length of non-type-checked proofs tends to be longer (162 characters vs 275 characters on average).

We create 4 distinct augmented datasets by utilizing different combinations of synthetic examples:

- **conditioned augmented**: conditioned bootstrap +100% synthetic conditioned (weighted by # of additional examples)
- **unconditioned augmented (non-type-checked)**: unconditioned bootstrap +100% synthetic unconditioned non-type-checked (weighted by # of additional characters)
- **unconditioned augmented (50/50 type-correct & non-type-checked)**: unconditioned bootstrap +50% synthetic unconditioned non-type-checked and +50% synthetic unconditioned type-correct (weighted by # of additional characters)
- **unconditioned augmented (fully type-correct)**: unconditioned bootstrap +100% synthetic unconditioned type-correct (weighted by # of additional characters)

We use each of the 2 bootstrap and the 4 augmented datasets to train language models. Then we evaluate each of these 6 models on our held-out bootstrap test sets, matching models trained on conditioned or unconditioned examples to the conditioned or unconditioned test sets respectively. When evaluating models trained on conditioned examples we prompt the models with theorem statements.

Training Dataset	Test Loss	Test Ppl.	Test Accuracy
conditioned bootstrap	1.25	2.38	9.72%
conditioned augmented	**1.12**	**2.18**	**16.92**%
unconditioned bootstrap	1.74	3.35	N/A
unconditioned augmented (non-type-checked)	1.72	3.30	N/A
unconditioned augmented (50/50 type-correct & non-...)	1.71	3.28	N/A
unconditioned augmented (fully type-correct)	**1.70**	**3.25**	N/A

Table 1: Test loss, test perplexity, and test accuracy of the models trained on each dataset. Test accuracy measures the % of test examples for which the generated proof matches ground truth.

We find that training on the augmented datasets results in superior metrics on our test sets. In the unconditioned case we also find that better metrics are achieved by using training sets in which a higher percentage of the synthetic data is type-correct. This demonstrates the improvement in data quality afforded by using the Lean kernel as a filter. However, since only a small percentage of synthetic proofs pass type-check, in practice we can likely expect the best possible unconditioned language modelling metrics to be achieved by simply training on all generated examples, as such a dataset would be much larger.

Can Increased Regularization Explain the Performance Boost?

We investigate how much of the improvement in loss associated with training on an augmented dataset is accounted for by an increase in regularization that can be achieved with dropout. To do this we train models on the conditioned bootstrap dataset and the conditioned augmented dataset with successively higher levels of dropout (incrementing by 0.1), until increasing dropout no longer improves the best achieved validation loss.

Training Dataset	Metric	Dropout: 0.1	Dropout: 0.2	Dropout: 0.3	Dropout: 0.4
conditioned bootstrap	Loss	1.25	1.15	1.09	1.09
conditioned augmented	Loss	1.12	1.04	**1.01**	1.01
conditioned bootstrap	Perplexity	2.38	2.21	2.13	2.13
conditioned augmented	Perplexity	2.18	2.06	**2.01**	2.02
conditioned bootstrap	Accuracy	9.72%	11.47%	12.97%	14.2%
conditioned augmented	Accuracy	16.92%	18.29%	**20.82**%	19.37%

Table 2: Test loss, test perplexity, and test accuracy of the models trained on each dataset with varying dropout.

We find that optimal test loss is achieved at a dropout value of 0.3. Notably, even with optimized dropout we observe a significant performance advantage from training on the augmented dataset.

Code. Source code is available at: https://github.com/joepalermo/synthetic-proof-term-data-augmentation

Acknowledgements. The authors would like to thank Jason Rute, Alok Singh, Alex Krizhevsky, Ragavan Thurairatnam, Hashiam Kadhim, Marc Tyndel, Rayhane Mama, and Louay Hazami for helpful discussions.

References

[1] T. mathlib Community, "The lean mathematical library," in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, (New York, NY, USA), p. 367–381, Association for Computing Machinery, 2020.

[2] S. Polu and I. Sutskever, "Generative language modeling for automated theorem proving," *CoRR*, vol. abs/2009.03393, 2020.

[3] J. M. Han, J. Rute, Y. Wu, E. W. Ayers, and S. Polu, "Proof artifact co-training for theorem proving with language models," *CoRR*, vol. abs/2102.06203, 2021.

[4] A. Anaby-Tavor, B. Carmeli, E. Goldbraich, A. Kantor, G. Kour, S. Shlomov, N. Tepper, and N. Zwerdling, "Not enough data? deep learning to the rescue!," 2019.

[5] T. Schick and H. Schütze, "Generating datasets with pretrained language models," 2021.

[6] V. Kumar, A. Choudhary, and E. Cho, "Data augmentation using pre-trained transformer models," 2020.

[7] Z. Wang, A. W. Yu, O. Firat, and Y. Cao, "Towards zero-label language learning," 2021.

[8] M. Wang and J. Deng, "Learning to prove theorems by learning to generate theorems," *CoRR*, vol. abs/2002.07019, 2020.

[9] E. Aygün, Z. Ahmed, A. Anand, V. Firoiu, X. Glorot, L. Orseau, D. Precup, and S. Mourad, "Learning to prove from synthetic theorems," *CoRR*, vol. abs/2006.11259, 2020.

[10] Y. Wu, A. Jiang, J. Ba, and R. B. Grosse, "INT: an inequality benchmark for evaluating generalization in theorem proving," *CoRR*, vol. abs/2007.02924, 2020.

[11] Y. Wu, M. N. Rabe, W. Li, J. Ba, R. B. Grosse, and C. Szegedy, "LIME: learning inductive bias for primitives of mathematical reasoning," *CoRR*, vol. abs/2101.06223, 2021.

[12] V. Firoiu, E. Aygün, A. Anand, Z. Ahmed, X. Glorot, L. Orseau, L. Zhang, D. Precup, and S. Mourad, "Training a first-order theorem prover from synthetic data," *CoRR*, vol. abs/2103.03798, 2021.

[13] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer, "The Lean Theorem Prover (System Description)," in *Automated Deduction - CADE-25* (A. P. Felty and A. Middeldorp, eds.), (Cham), pp. 378–388, Springer International Publishing, 2015.

[14] J. Ramos *et al.*, "Using tf-idf to determine word relevance in document queries," in *Proceedings of the first instructional conference on machine learning*, vol. 242, pp. 29–48, Citeseer, 2003.

[15] M. Ott, S. Edunov, A. Baevski, A. Fan, S. Gross, N. Ng, D. Grangier, and M. Auli, "fairseq: A fast, extensible toolkit for sequence modeling," *CoRR*, vol. abs/1904.01038, 2019.

[16] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

Learning Instantiation in First-Order Logic

Jelle Piepenbrock[1,2], Josef Urban[1], Konstantin Korovin[3], Miroslav Olšák[4], Tom Heskes[2], and Mikoláš Janota[1]

[1] CIIRC, Czech Technical University in Prague, Czechia
[2] Radboud University Nijmegen, The Netherlands
[3] University of Manchester, UK
[4] Institut des Hautes Études Scientifiques, France

Introduction The appearance of strong CDCL-based propositional (SAT) solvers has greatly advanced several areas of automated reasoning (AR). One of the directions in AR is thus to apply SAT solvers to expressive formalisms such as first-order logic, for which large corpora of general mathematical problems exist today. This is possible due to Herbrand's theorem, which allows reduction of first-order problems to propositional problems by instantiating variables. The core challenge is choosing the right instances from the typically infinite Herbrand universe. Instantiation is a powerful tool for formal reasoning with quantifiers.

In this work, we develop the first machine learning system targeting this task, addressing its combinatorial and invariance properties. In particular, we develop a new GNN2RNN architecture based on an invariant graph neural network (GNN) that learns from problems and their solutions independently of symbol names (addressing the abundance of skolems), combined with a recurrent neural network (RNN) that proposes for each clause its instantiations. The architecture is then trained on a corpus of mathematical problems and their instances produced by the iProver system, and its performance is evaluated in several ways. We show that the system can achieve high accuracy in predicting the right instances, and that it is capable of solving a large number of problems by educated guessing when combined with a SAT solver.

The power of instantiation is formalized by *Herbrand's theorem* [5], which states, roughly speaking, that within first-order logic (FOL), quantifiers can always be eliminated by the right instantiations. Herbrand's theorem further states that it is sufficient to consider instantiations from the *Herbrand universe*, which consists of terms with no variables (*ground terms*) constructed from the symbols appearing in the problem. This fundamental result has been explored in automated reasoning (AR) systems since the 1950s [2]. It means that once the right instantiations are discovered, we end up with a problem without quantifiers, which is typically easy to solve by state-of-the-art SAT solvers [12].

Methods Our starting point for instantiation in first-order logic is iProver. At the core of iProver is the Inst-Gen [4, 9] instantiation calculus, which can be combined with resolution and superposition calculi [3]. At a high level, the procedure works as follows. Given a set of first-order clauses S its propositional abstraction $S\bot$ is obtained by mapping all variables to a designated ground term $\bot$. A propositional solver is applied to $S\bot$ and it either proves that $S\bot$ is unsatisfiable and in this case the set of first-order clauses S is also unsatisfiable or shows that $S\bot$ is satisfiable and in this case returns a propositional model of the abstraction $S\bot$. This propositional model is analyzed if it can be extended to a full first-order model. If it cannot be extended then it is possible to show that there must be complementary literals in the model that are unifiable.

A major bottleneck is however the large number of generated instances, with only a few typically needed for the final proof. This motivates our work here: a trained predictor that proposes the most relevant instantiations can significantly help and complement the complete search procedures used by systems like iProver.

We construct a large corpus of instantiations by running iProver on 113 332 first-order ATP problems created by the AI4REASON project. They originate from the Mizar Mathematical Library (MML) [8] and are exported to first-order logic by the MPTP system [14]. All these problems have an ATP proof (in general in a high time limit) found by either the E/ENIGMA [11, 6] or Vampire/Deepire [10, 13] systems. Additionally, the problems' premises have been *pseudo-minimized* [7] by iterated Vampire runs. We use the pseudo-minimized versions because our focus here is on guidance rather than on premise selection.

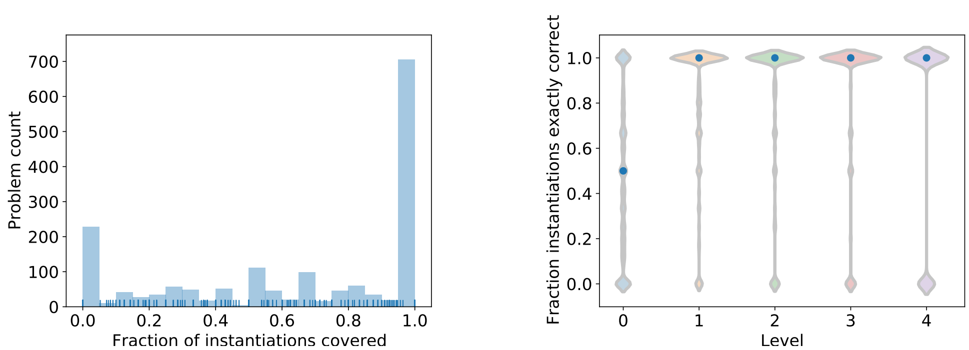
We reimplement and modify the GNN architecture used in [6] to allow the network to produce partial instantiations for each clause by using a recurrent neural network (RNN) after running the GNN. The method computes instantiations level-wise, meaning that one head symbol is picked for each variable (if needed) in each clause, after which we add fresh variables and again ask for head symbols (see Figure 1).

$$\forall\ x\ z\ P(f(x, z))$$
$$\forall\ x_1\ x_2\ z_1\ P(f(\overbrace{t(x_1, x_2)}^{t/2}, \overbrace{g(z_1)}^{g/1}))$$
$$P(f(t(\underset{c/0\ \uparrow}{c}, \underset{c/0\ \uparrow}{c}), g(\underset{e/0\ \uparrow}{e})))$$

(1) instantiate x by head symbol t with arity 2 and z by g of arity 1 (going from $level_0$ to $level_1$)
(2) instantiate x_1, x_2, z_1 by constants c, c, and e, respectively (going from $level_1$ to $level_2$)

Figure 1: Term instantiation through incremental deepening. In the figure, there are two instantiation steps, one after the other.

Results We first evaluate the trained GNN2RNN by measuring the overlap of the predicted instantiations on the unseen test problems at each level. The system manages to predict correct instantiations for a large part of the set, see Figure 2a. In particular, about for 700 out of 1682 problems, the predictions include the exact instances used in the iProver proof. Figure 2b



(a) (b)

Figure 2: **a**: Histogram of the fraction of needed instantiations predicted for unseen test set problems. **b**: Violin plot of the fraction of instantiations correctly predicted, split by how many symbol levels from the base problem the problem was. Blue dot is the median of each group.

shows the results per level, which reveals an interesting pattern: the system is much better at predicting the instances for levels 1–4 (almost fully correct), when the first head symbol of each term is already determined by the proof instance. Next, we combine GNN2RNN with EGround and PicoSAT [1] to see if the proposed ground instances are already propositionally unsatisfiable. The fraction of problems that PicoSAT finds unsatisfiable after one top-down GNN2RNN step at level_i is 21%, 80%, 80%, 83% and 80% respectively. Again, we see that picking the first head symbol for each variable is the hardest, but the system performs well for the subsequent symbol choices.

Acknowledgements This work was partially supported by the European Regional Development Fund under the Czech project AI&Reasoning no. CZ.02.1.01/0.0/0.0/15_003/ 0000466 (JP, JU), Amazon Research Awards (JP, JU) and by the Czech MEYS under the ERC CZ project *POSTMAN* no. LL1902 (JP, MJ).

References

[1] Armin Biere. Picosat essentials. *J. Satisf. Boolean Model. Comput.*, 4(2-4):75–97, 2008.

[2] Martin Davis. The early history of automated deduction. In *Handbook of Automated Reasoning*, pages 3–15. Elsevier and MIT Press, 2001.

[3] André Duarte and Konstantin Korovin. Implementing superposition in iProver (system description). In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 2020.

[4] Harald Ganzinger and Konstantin Korovin. New directions in instantiation-based theorem proving. In *18th IEEE Symposium on Logic in Computer Science (LICS 2003), 22-25 June 2003, Ottawa, Canada, Proceedings*, pages 55–64. IEEE Computer Society, 2003.

[5] Jacques Herbrand. *Recherches sur la théorie de la démonstration.* Doctorat d'état, La Faculté des Sciences de Paris, 1930.

[6] Jan Jakubuv, Karel Chvalovský, Miroslav Olsák, Bartosz Piotrowski, Martin Suda, and Josef Urban. ENIGMA anonymous: Symbol-independent inference guiding machine (system description). In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 448–463. Springer, 2020.

[7] Cezary Kaliszyk and Josef Urban. Learning-assisted automated reasoning with Flyspeck. *J. Autom. Reasoning*, 53(2):173–213, 2014.

[8] Cezary Kaliszyk and Josef Urban. MizAR 40 for Mizar 40. *J. Autom. Reasoning*, 55(3):245–256, 2015.

[9] Konstantin Korovin. Inst-Gen - A modular approach to instantiation-based automated reasoning. In Andrei Voronkov and Christoph Weidenbach, editors, *Programming Logics - Essays in Memory of Harald Ganzinger*, volume 7797 of *Lecture Notes in Computer Science*, pages 239–270. Springer, 2013.

[10] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *CAV*, volume 8044 of *LNCS*, pages 1–35. Springer, 2013.

[11] Stephan Schulz. System description: E 1.8. In Kenneth L. McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *LPAR*, volume 8312 of *LNCS*, pages 735–743. Springer, 2013.

[12] João P. Marques Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 131–153. IOS Press, 2009.

[13] Martin Suda. Improving ENIGMA-style clause selection while learning from history. In *CADE*, volume 12699 of *Lecture Notes in Computer Science*, pages 543–561. Springer, 2021.

[14] Josef Urban. MPTP 0.2: Design, implementation, and initial experiments. *J. Autom. Reasoning*, 37(1-2):21–43, 2006.

A small survey of mathematical abilities of modern transformer architectures*

Bartosz Piotrowski

University of Warsaw, Poland, and Czech Technical University, Prague

Introduction Neural networks (NNs) are versatile tools which established state-of-the-art in multiple domains. In particular, one of the spectacular advances achieved with use of NNs has been in natural language processing (NLP). Today, the dominating kind of a neural model used in this domain is based on the transformer architecture [10]. It was also observed that neural architectures designed for NLP have ability to deal with tasks of symbolic (or algorithmic) nature. These include: recognizing propositional entailment [2], computing integrals [4], solving differential equations [1], normalizing polynomials [6], autoformalization [11], premise selection [5], differentiation, solving linear equations, number base conversion, and many others [9].

It is not well understood how neural models are able to perform algorithmic tasks well. It is also unclear what features of a neural architecture make it more suitable for such tasks. In this work, we make a step towards understanding this. We compare two different architectures – encoder-decoder *versus* decoder-only – and two different modes of training – starting from scratch *versus* fine-tuning a model pre-trained on a natural language dataset. We also want to see what is performance of a modern transformer model trained in a practical, limited setting: training for no more than two days on a single GPU.

Data We took 8 different datasets representing mathematical tasks of varied difficulty: addition, multiplication, differentiation, integration, solving linear equations, division, number base conversion, and normalizing polynomials. The first two were created for the purpose of this work and the remaining six were taken from other works [9, 4, 6]. Each dataset consists of *input-output* examples, where *input* is a query to the model and *output* in an answer that the model is trained to produce. For each of the datasets a hold-out testing set of 10000 examples was drawn. Below there are examples of *input-output* pairs for the linear equations dataset:

input	output
`Solve - 3 8 * h - 6 * h + 4 7 8 + 4 0 2 = 0 for h .`	`9`
`Solve 2 9 * i + 1 3 0 0 = - 3 * i + 4 1 * i - 7 4 * i for i .`	`- 2 0`
`Solve 1 0 4 9 * d = 4 3 1 2 + 5 1 2 9 for d .`	`- 4 5`

We experimentally established that treating single digits as tokens is better then taking whole numbers as tokens, and we preprocessed all the datasets accordingly.

Transformer models We compare two different state-of-the-art transformer architectures:

1. **GPT2** [7]: a decoder-only architecture with 124 million of trainable parameters.
2. **T5** [8]: an encoder-decoder architecture (closely following the original transformer model described in [10]). We use the `T5-small` version of this model with 60 million parameters.

Both GPT2 and T5 proved to perform very well on a range of NLP tasks. For both of them there are available high-quality pre-trained checkpoints released by the authors of the models.[1]

*The author was supported by the grant of National Science Center, Poland, no. 2018/29/N/ST6/02903.

[1] They are available in Huggingface: https://huggingface.co/gpt2, https://huggingface.co/t5-small

dataset	T5		GPT2	
	pretrained	untrained	pretrained	untrained
addition	86.74%	96.95%	98.60%	99.26%
multiplication	24.10%	47.58%	46.54%	68.00%
division	67.23%	70.98%	72.62%	77.16%
number base conversion	0.03%	2.58%	1.63%	3.52%
solving linear equations	37.56%	17.62%	45.57%	47.40%
differentiation	98.84%	95.05%	99.80%	99.75%
integration	26.65%	35.88%	79.70%	81.80%
polynomial normalization	58.13%	90.83%	89.35%	92.93%

Table 1: Final testing accuracy of neural language models tested on the eight datasets.

Experimental setup We perform the experiments using the Huggingface framework [12]. In each experiment we train with the Adam optimizer [3] with parameters: learning rate = 1e-5, $\beta_1 = 0.9$, $\beta_2 = 0.999$, ϵ =1e−8, weight_decay = 0. When we fine-tune a pre-trained model, we must use a tokenizer that comes along with the model – in cases of both GPT2 and T5 these are pre-trained byte pair encoding tokenizers. When training from scratch we use a simple tokenizer splitting on whitespaces. All trainings were performed using GeForce GTX 2080 Ti GPUs. We limit all the trainings to passing through a model 64 million training examples.[2] All data and scripts required to reproduce the results presented here are available at https://github.com/BartoszPiotrowski/transformers-for-mathematics

Results and conclusions Figure 1 shows training curves for one of the datasets – linear equations. Table 1 shows the final testing accuracy for all the tasks. There are two conclusions:

1. In almost all cases, the pre-trained versions of models performed worse than the models trained from scratch. It likely means that the data on which the models were pre-trained does not contain much information relevant for dealing with mathematical problems. There are, however, two exceptions: for T5 and datasets on differentiation and solving linear equations. Especially for the latter the difference is much in favour of the pre-trained version of the model. As for now, we do not have explanation for this.
2. GPT2 performed better than T5 for all the datasets. It means that decoder-only architectures are capable of learning mathematical tasks, despite the fact that in most of the cited related works encoder-decoder architectures were used. However, it is unclear whether the superior performance of GPT2 was due to the different architecture, or possibly because of larger number of trainable parameters. Further experiments would be needed.

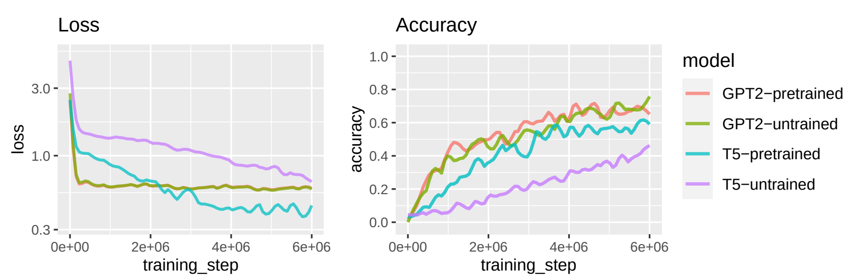


Figure 1: Training loss and accuracy on the linear equations dataset.

[2]This is a practical limit – full training takes then, depending on a dataset, between 4 and 50 hours.

References

[1] François Charton, Amaury Hayat, and Guillaume Lample. Learning advanced mathematical computations from examples. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[2] Richard Evans, David Saxton, David Amos, Pushmeet Kohli, and Edward Grefenstette. Can neural networks understand logical entailment? In *International Conference on Learning Representations*, 2018.

[3] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[4] Guillaume Lample and François Charton. Deep learning for symbolic mathematics. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[5] Bartosz Piotrowski and Josef Urban. Stateful premise selection by recurrent neural networks. In Elvira Albert and Laura Kovács, editors, *LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain, May 22-27, 2020*, volume 73 of *EPiC Series in Computing*, pages 409–422. EasyChair, 2020.

[6] Bartosz Piotrowski, Josef Urban, Chad E. Brown, and Cezary Kaliszyk. Can neural networks learn symbolic rewriting? *CoRR*, abs/1911.04873, 2019.

[7] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[8] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2020.

[9] David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. Analysing mathematical reasoning abilities of neural models. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[10] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[11] Qingxiang Wang, Cezary Kaliszyk, and Josef Urban. First experiments with neural translation of informal to formal mathematics. In Florian Rabe, William M. Farmer, Grant O. Passmore, and Abdou Youssef, editors, *11th International Conference on Intelligent Computer Mathematics (CICM 2018)*, volume 11006 of *LNCS*, pages 255–270. Springer, 2018.

[12] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, and Jamie Brew. Huggingface's transformers: State-of-the-art natural language processing. *CoRR*, abs/1910.03771, 2019.

Sifting through a large hypothesis space: Revisiting differentiable *learning through satisfiability* *

Stanisław J. Purgał[1], David M. Cerna[2,3], and Cezary Kaliszyk[1]

[1] University of Innsbruck, Innsbruck, Austria
{stanislaw.purgal,cezary.kaliszyk}@uibk.ac.at
[2] Czech Academy of Sciences Institute of Computer Science (CAS ICS), Prague, Czechia
dcerna@cs.cas.cz
[3] Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria
dcerna@risc.jku.at

Abstract

A difficultly which must be addressed by inductive logical programming (ILP) systems is how to deal with the enormous space of plausible solutions. The majority of modern ILP systems approach this problem through the *meta-learning paradigm*, that is, only consider plausible solutions which are constructable from a set of *clause templates*. This approach has been adopted by investigations into neuro-symbolic ILP. Our investigation uses clause templates together with a variant of δILP, to expand the hypothesis space, rather than contract it. Our experiments support the following hypothesis: providing gradient descent with a larger solution space aids the discovery of explanatory hypotheses.

Inductive Logic Programming (ILP) [5] is a form symbolic machine learning approach which learns explanatory hypothesis from positive and negative evidence together with fixed background knowledge. These explanatory hypotheses take the form of a logic program. In contrast to statistical approaches to machine learning, ILP systems are data-efficient in that a complex hypothesis can be learned from only a few examples; in some cases, even a single example is sufficient. Additionally, these hypotheses tend to be human-readable and provide a route towards explainable AI. While there are many positive aspects of the approach, ILP systems ability to generalize is typically, negatively impacted by noisy input, and is limited to certain problem domains [1, 4].

Attempts to combine the flexibility and agnosticism to noise of statistical learning with the benefits of a firm logical foundation, forms the bedrock of the current investigations into *neuro-symbolic AI* [6]. In this abstract, we discuss our modification of a prominent approach to neuro-symbolic ILP, δILP [3]. This system is based on the *learning from satisfiability* ILP paradigm. In the case of δILP, the plausible hypothesis space is turned into a SAT problem where a model denotes a hypothesis. The hypothesis space is finite as a fixed program template is provided and the background knowledge is assumed to be ground and finite. This classical SAT problem can be transformed into a *soft* SAT problem by replacing the classical operators by *differentiable* ones. For example, Classical conjunction is replaced by the product *T-norm* [2], $X \wedge Y \equiv x * y$.

To understand our investigation we need to briefly introduce the structure of the *program templates* used by δILP. Clauses are assumed to be at most length 2, predicate definitions contain at most 2 clauses, and predicates may take at most 2 arguments. Each auxiliary predicate definition (including the predicate being learned) is associated with at most two *rule templates* defining the structure of its clauses. These rule templates state how many existential

*Supported by the ERC starting grant no. 714034 SMART, the Math$_{LP}$ project (LIT-2019-7-YOU-213) of the Linz Institute of Technology and the state of Upper Austria, Cost action CA20111 EuroProofNet.

variables occur within the clause and whether the predicate symbols occurring therein may be *extensional* (defined in the background) or *intensional* (derived during learning).

Each of the auxiliary predicate definition is associated with a matrix of weights where each entry denotes how strongly the system believes that a pair of clauses (respecting the rule templates) is the correct definition for the given predicate. This design choice is prohibitively expensive and significantly limits uses of the system due to the significant memory requirements. An alternative would be to assign a weight to each instantiation of the associated rule templates (so called *splitting* the definition), however, as discussed in Appendix F of [3], this approach is less effective for ILP.

In our investigation, we take definition splitting one step further and split not only the definitions (as was discussed in Appendix F of [3]), but also the individual rule templates. This entails that for each auxiliary predicate definition entries in the weight vector denote how strongly the system believes an instance of a predicate (i.e. $father(X,Y)$) is the correct choice for a particular position in a particular clause. This significantly reduces the memory requirement, but also goes far beyond the relaxations made by the Evans and Grefenstette [3] which they claim are less effective for ILP. To deal with this issue, instead of providing a program template which roughly matches the structure of the program we expect the system to find, We provide our modified δILP with many more auxiliary predicate then needed to construct the goal program. This is possible giving the memory saving resulting from splitting the weight matrix twice.

Let us consider the example $fizz \equiv \{X | X \in \mathbb{N} \wedge 0 = X(mod\ 3)\}$ from [3]. As background knowledge the authors provided the zero predicate and instances of the successor predicate up to 6 (i.e. $succ(0,1)$,....). The positive examples are $0, 3$ and 6 while the negative examples are all other natural numbers less than 6. This example posed a challenge for δILP and only 10% of the runs resulted in a mean squared error less than $1e-4$. On the contrary, Up to 95% of our runs passed a validation phase regardless the mean squared error at the time of halting; The percentage is dependent on how many auxiliary predicates we allowed (see Figure 1).

This experiment together with a few others seem to contradict the exposition in Appendix F of [3]. However, it is not clear if these results can be further expended, nor how this can be generalized to handle more complex ILP task. An alternative approach to a more efficient search within the large search space would be the inclusion of supervised machine learning in the proposed framework. We leave these questions to future investigation.

```
 Possible explanatory Hypothesis
 for fizz example

fizz(X):- zero(X).
fizz(X):- p1(X,Y),p2(Y).
p1(X,Y):- succ(Y,Z),succ(Z,X).
p2(X):- succ(Y,X),fizz(Y).
```

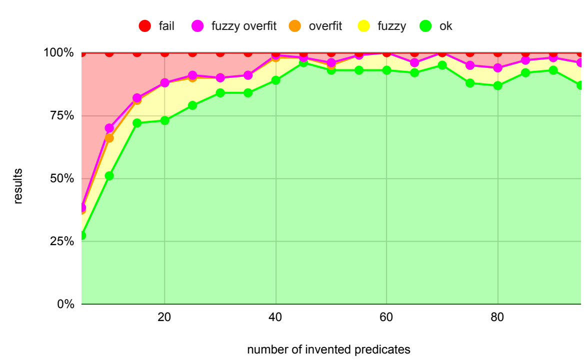


Figure 1: Percentage of runs finding a crisp solution which passes the validation phase.

References

[1] Andrew Cropper, Sebastijan Dumančić, Richard Evans, and Stephen H. Muggleton. Inductive logic programming at 30. *Machine Learning*, 111(1):147–172, Jan 2022.

[2] Francesc Esteva and Lluıs Godo. Monoidal t-norm based logic:towards a logic for left-continuous t-norms. *Fuzzy Sets and Systems*, 124(3):271–288, 2001. Fuzzy Logic.

[3] Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research*, 61:1–64, January 2018.

[4] Mark Law, Alessandra Russo, and Krysia Broda. Inductive learning of answer set programs. In *Proceedings of Logics in Artificial Intelligence*, pages 311–325. Springer, August 2014.

[5] Stephen Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–318, February 1991.

[6] Luc De Raedt, Sebastijan Dumancic, Robin Manhaeve, and Giuseppe Marra. From statistical relational to neuro-symbolic artificial intelligence. In *IJCAI 2020*, pages 4943–4950, 2020.

Project proposal: A modular reinforcement learning based automated theorem prover *

Boris Shminke[1]

Université Côte d'Azur, CNRS, LJAD, France
boris.shminke@univ-cotedazur.fr

Abstract

We propose to build a reinforcement learning prover of independent components: a deductive system (an environment), the proof state representation (how an agent sees the environment), and an agent training algorithm. To that purpose, we contribute an additional Vampire-based environment to `gym-saturation` package of OpenAI Gym environments for saturation provers. We demonstrate a prototype of using `gym-saturation` together with a popular reinforcement learning framework (Ray `RLlib`). Finally, we discuss our plans for completing this work in progress to a competitive automated theorem prover.

1 Introduction and related work

Reinforcement learning (RL) is applied widely in the automated reasoning domain. There are RL-related (including iterating supervised learning algorithms without applying recent RL advances) projects for interactive theorem provers (ITPs) (e.g. HOList [2] for HOL Light [8], ASTactic [33] for Coq [31], or TacticZero [32] for HOL4 [27]) as well as for automated theorem provers (ATPs) (e.g. Deepire [28] for Vampire [16], ENIGMA [25] for E [12], or rlCoP [13] for leanCoP [18]). Despite the variety of solutions and ideas, we are not aware of cases of significant code reuse between such projects.

We envision a prover capable of learning from its experience and composed of pluggable building blocks. We hope that such architecture could promote faster experimentation and easier flow of ideas between different projects for everyone's progress. For an RL-based prover, we identify at least three types of modules. They are a deductive system (an environment), a proof state representation (how an agent sees it), and an agent training algorithm.

When choosing whether to learn to guide an ITP or an ATP, we prefer the latter since ATPs can be relatively easy compared as black boxes [30] in contrast to RL guided ITPs, which often come with their distinctive benchmarks.

Among ATPs, one can consider saturation provers less suitable for the RL (e.g., see design considerations from [22]), but several existing projects (like ENIGMA, Deepire or TRAIL [5]) show encouraging results. Keeping that in mind, we decided to concentrate on guiding clause selection in the saturation algorithm by RL.

Inspired by HOList, CoqGym (from ASTactic) and `lean-gym` [20], we have created `gym-saturation` [26] — an OpenAI Gym [4] environment for training RL agents to prove theorems in clausal normal form (CNF) of the Thousands of Problems for Theorem Provers (TPTP) library [29] language.

*This work has been supported by the French government, through the 3IA Côte d'Azur Investments in the Future project managed by the National Research Agency (ANR) with the reference number ANR-19-P3IA-0002

2 Recent work in progress

Contemporary RL training algorithms are notorious for the number of details that can differ from one implementation to another [9]. To eliminate the risk of abandoning an RL algorithm as unsuitable for guiding an ATP only because of flaws in our implementation of it, we plan to rely on existing RL frameworks containing tested implementations of well-known baselines. As a starting point, we have chosen Ray `RLlib` [17] as a library claiming both deep learning (DL) framework independence and extendability. Similar solutions like Tensorflow Agents [7] or Catalyst.RL [15] tend to support only one DL framework, which we wanted to avoid for greater generality.

In contrast to CoqGym and others, `gym-saturation` is not only a 'gym' in some general sense, but it implements the standard OpenAI Gym API. It makes it easier to integrate with libraries like Ray `RLlib`. We contribute[1] a prototype of such integration. Even together with some domain related patches, the prototype remains a lightweight collection of wrappers around standard `RLlib` classes, taking only around 300 lines of Python code.

Since we postulated interchangeability of modules, we added a Vampire-based environment to `gym-saturation` (see the project page[2] for more details) in addition to the already existing naïve implementation of a saturation loop. Despite a different backend, one can plug a new environment into the prover prototype without additional edits of RL related code.

Similar systems for connection tableaux There exists a `FLoP` (Finding Longer Proofs) project [34] which implements a `ProofEnv` OpenAI Gym environment for a connection tableaux calculus, which can guide two different provers (`leanCoP` and its `OCaml` reimplementation `fCoP` [14]). `FLoP` shares many architectural features with our work, and we plan to test its approaches in saturation provers setting.

3 Prototype implementation details

Since this research is still in an early stage, we don't report any conclusive results of its performance, only describing the architecture. A prototype prover has two main parts: `gym-saturation` as an environment and a patched DQN [10] implementation from Ray `RLlib` training an agent. **An episode** starts with the environment reset. On environment reset, a random TPTP problem from a training subset is loaded, transformed to the CNF, and becomes a proof state. After an agent makes an **action** (selects a clause), the episode can stop for three reasons: a given clause is empty (refutation proof found, the **reward** is 1.0; in other cases, it's 0.0), we reach the step limit (a soft timeout), we reach the maximal number of clauses in the proof state (a soft memory limit). Only episodes with a positive final reward go to the **memory buffer**. Before storing in the buffer, the reward is spread evenly between the clauses from the proof (others remain zero). A memory buffer can contain the same proof for the same problem twice or different proofs (maybe of different lengths) for one problem. A **training** batch can contain steps from different episodes (and thus different initial environment states). We sample the memory buffer with higher weights for more recent episodes.

[1] https://github.com/inpefess/basic-rl-prover
[2] https://pypi.org/project/gym-saturation/

4 Future plans and discussion

In the prototype, we represent each clause in a proof state only by its size and order number, applying a logistic regression as a Q-value function. We will need an elaborate feature extraction procedure to complete this oversimplified model to a competitive ATP. We plan to use graph neural networks similar to those used for lazyCoP [23] and then compare and combine them with the graph representation of clause lineage pioneered by Deepire. We also plan to test training algorithm interchangeability by using IMPALA [6] and Ape-X [11] in addition to DQN.

A finished project will have to address many different problems. Here we list several obvious ones.

Delayed reward One of the well-known peculiarities of an ATP is the fact that a reward can be assigned only after proof is found, which can take a large number of steps in an RL episode. To make an agent learn to discern good steps, one has to spread the final reward to all the steps in a finished trajectory. A typical solution is to post-process a trajectory by assigning positive advantage values only to the steps encountered in a proof, and negative (or zero) values to all the rest. Here one can argue in favour of both higher values for longer proofs (since the ability to produce longer proofs is desirable) and higher values for shorter ones (since more concise proofs for simpler problems are preferable to verbose ones which in turn could help to find longer proofs otherwise unreachable because of time and memory constraints). A contrarian approach is to assign positive advantage values for all the steps in a trajectory on which proof was found, and non-positive to all the steps from trajectories finished because of the resource limitations. Such an approach works well, for example, in the Atari Pong game, where it's practically impossible to judge which action led to a goal.

Sparse positive reward Another well-known problem of applying RL to ATPs is related to the fact that even sub-human performance still seems out of reach. The majority of proof attempts finish without proof found. Discarding failed episodes seems too wasteful, although obvious as a first attempt. An opposite solution (assigning non-positive advantage values to all failed episodes) makes the training dataset too imbalanced. One possible solution to this is to use replay buffers and sample from them balanced train batches. This explains why we decided not to neglect DQN despite its known limitations when compared to on-policy algorithms like PPO [24].

Multiple proofs Many problems have multiple possible proofs, equivalent in some sense or not. An agent will have to decide which proofs are preferable to replicate. Again, replay buffers can be used for that. Ranking proofs can be based on their length or other important properties (reuse of previously proved lemmata, using only a selected subset of deduction rules or tactics etc)

High environment's inhomogeneity Some problems are inherently harder than others and can belong to areas of mathematics not connected in a given formalization. Curriculum learning [3] or at least limiting the training scope to a reasonable subset of the TPTP library will be needed.

State representation Usually, contemporary RL algorithms expect the observed state to have a form of a vector. Representing logic formulae as such is an active domain of research.

We plan to try both logic-specific approaches like [21] and general abstract syntax tree encoding models like `code2vec` [1] or `ast2vec` [19].

References

[1] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29, January 2019.

[2] Kshitij Bansal, Sarah M. Loos, Markus N. Rabe, Christian Szegedy, and Stewart Wilcox. Holist: An environment for machine learning of higher order logic theorem proving. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 454–463. PMLR, 2019.

[3] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, page 41–48, New York, NY, USA, 2009. Association for Computing Machinery.

[4] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.

[5] 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. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(7):6279–6287, May 2021.

[6] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. IMPALA: Scalable distributed deep-RL with importance weighted actor-learner architectures. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1407–1416. PMLR, 10–15 Jul 2018.

[7] Danijar Hafner, James Davidson, and Vincent Vanhoucke. Tensorflow agents: Efficient batched reinforcement learning in tensorflow. *CoRR*, abs/1709.02878, 2017.

[8] John Harrison. Hol light: An overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 60–66, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[9] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), Apr. 2018.

[10] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), Apr. 2018.

[11] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. Distributed prioritized experience replay. In *International Conference on Learning Representations*, 2018.

[12] Jan Jakubův and Josef Urban. Enigma: Efficient learning-based inference guiding machine. In Herman Geuvers, Matthew England, Osman Hasan, Florian Rabe, and Olaf Teschke, editors, *Intelligent Computer Mathematics*, pages 292–302, Cham, 2017. Springer International Publishing.

[13] Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Miroslav Olšák. Reinforcement learning of theorem proving. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[14] Cezary Kaliszyk, Josef Urban, and Jiři Vyskočil. Certified connection tableaux proofs for hol light and tptp. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, CPP '15, page 59–66, New York, NY, USA, 2015. Association for Computing Machinery.

[15] Sergey Kolesnikov and Oleksii Hrinchuk. Catalyst.rl: A distributed framework for reproducible RL research. *CoRR*, abs/1903.00027, 2019.

[16] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 1–35, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[17] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Joseph Gonzalez, Ken Goldberg, and Ion Stoica. Ray rllib: A composable and scalable reinforcement learning library. *CoRR*, abs/1712.09381, 2017.

[18] Jens Otten and Wolfgang Bibel. leancop: lean connection-based theorem proving. *Journal of Symbolic Computation*, 36(1):139–161, 2003. First Order Theorem Proving.

[19] Benjamin Paaßen, Irena Koprinska, and Kalina Yacef. Recursive tree grammar autoencoders. *Machine Learning*, Aug 2022.

[20] Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. Formal mathematics statement curriculum learning. *CoRR*, abs/2202.01344, 2022.

[21] StanisŁaw PurgaŁ, Julian Parsert, and Cezary Kaliszyk. A study of continuous vector representations for theorem proving. *Journal of Logic and Computation*, 31(8):2057–2083, 02 2021.

[22] Michael Rawson and Giles Reger. A neurally-guided, parallel theorem prover. In Andreas Herzig and Andrei Popescu, editors, *Frontiers of Combining Systems*, pages 40–56, Cham, 2019. Springer International Publishing.

[23] Michael Rawson and Giles Reger. lazycop: Lazy paramodulation meets neurally guided search. In Anupam Das and Sara Negri, editors, *Automated Reasoning with Analytic Tableaux and Related Methods - 30th International Conference, TABLEAUX 2021, Birmingham, UK, September 6-9, 2021, Proceedings*, volume 12842 of *Lecture Notes in Computer Science*, pages 187–199. Springer, 2021.

[24] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *ArXiv*, abs/1707.06347, 2017.

[25] Stephan Schulz, Simon Cruanes, and Petar Vukmirović. Faster, higher, stronger: E 2.3. In Pascal Fontaine, editor, *Automated Deduction – CADE 27*, pages 495–507, Cham, 2019. Springer International Publishing.

[26] Boris Shminke. gym-saturation: an openai gym environment for saturation provers. *Journal of Open Source Software*, 7(71):3849, 2022.

[27] Konrad Slind and Michael Norrish. A brief overview of hol4. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 28–32, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[28] Martin Suda. Improving enigma-style clause selection while learning from history. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 543–561, Cham, 2021. Springer International Publishing.

[29] Geoff Sutcliffe. The TPTP problem library and associated infrastructure - from CNF to th0, TPTP v6.4.0. *J. Autom. Reason.*, 59(4):483–502, 2017.

[30] Geoff Sutcliffe. The 10th IJCAR automated theorem proving system competition - CASC-J10. *AI Commun.*, 34(2):163–177, 2021.

[31] The Coq Development Team. The coq proof assistant, January 2022.

[32] Minchao Wu, Michael Norrish, Christian Walder, and Amir Dezfouli. Tacticzero: Learning to prove theorems from scratch with deep reinforcement learning. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 9330–9342. Curran Associates, Inc., 2021.

[33] Kaiyu Yang and Jia Deng. Learning to prove theorems via interacting with proof assistants. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 6984–6994. PMLR, 2019.

[34] Zsolt Zombori, Adrián Csiszárik, Henryk Michalewski, Cezary Kaliszyk, and Josef Urban. Towards finding longer proofs. In Anupam Das and Sara Negri, editors, *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 167–186, Cham, 2021. Springer International Publishing.

Elements of Reinforcement Learning in Saturation-based Theorem Proving*

Martin Suda

Czech Technical University in Prague, Czech Republic

The Promise and the Hype

Reinforcement learning (RL) [18], especially its *deep* variant relying on modern neural networks, is probably the most fashionable method for attacking problems in our machine learning (ML) era. The impressive successes in board games [13] or on the ATARI benchmark [3] justify the excitement. Moreover, it is very appealing to have the machine look for a solution unbiased by our preconceptions, since this intuitively increases the chances of discovering brand new strategies. However, we should also be aware of the various shortcomings of the approach [4].

In automatic theorem proving (ATP), we have seen the Monte-Carlo tree search paradigm [8] extend a connection tableaux prover [7] or, more recently, a saturation-based setup called TRAIL [1], featuring an interesting idea of multiplicative attention for expressing a dependence on prover's state. Despite the partial successes, we still seem to be far from getting a system that could challenge a state-of-the-art prover in a real-time evaluation (Kaliszyk et al. [7] use abstract time, TRAIL falls short of improving over plain E [11]), let alone on a versatile benchmark such as the TPTP library [17] (both mentioned works target a more uniform Mizar benchmark).

Ancient Lore and its Contemporary Extensions

It is instructive to recall the basic RL ingredients and project them to the state-of-the-art (SotA) saturation-based ATP technology and its recent improvements by ML. In this light, we can think of a prover as being guided by an *agent*, who monitors the prover's *state* and chooses appropriate *actions* to reach the goal of deriving the empty clause, ideally in the shortest time possible. A learning feedback for the agent should come in the form of a *reward*, received after executing each individual action or at the end of a proof attempt.

In saturation-based ATP (setting aside the role of proving strategies) the guiding agent is most fittingly identified with the clause selection heuristic [see, e.g., 12]. A proving state naturally decomposes into two conceptual parts: a *static* one, the formula subject to proving, and an *evolving* one, any information influencing what should be done next in order to prove it. Finally, the available actions correspond simply to the passive (unprocessed) clauses.

The author finds it noteworthy, that SotA provers, backed by decades of research in the field, mostly ignore the state for clause selection. Except possibly for a few bits to remember which queue to select the next clause from, the effective state is blank[1] and each selection aims greedily at the best available clause. Could this indicate there is actually little hope for meaningful proof planning in general purpose ATP?

The situation is different with the recent improvements by ML. Information about the conjecture (i.e., a static state) has been included since the second version of ENIGMA [5]

*Supported by the Czech Science Foundation project 20-06390Y and the project RICAIP no. 857306 under the EU-H2020 programme.

[1] Conjecture clauses sometimes get a different status for some heuristics [e.g., 14], but only uniformly, not depending on what the conjecture actually is.

and, before that, by the work of Loos et al. [10]. While the latter paper does not perform a corresponding ablation, ENIGMA is reported to moderately improve thanks to the conjecture features.[2] As mentioned, an evolving state is proudly included in TRAIL [1] and also in, e.g., ENIGMAWatch [2]. In both cases, the papers report on an improvement thanks to the evolving state feature. Although this is only shown for Mizar, maybe there is hope after all!

Let us close this section by returning to the concept of reward. It seems unrealistic to ever learn useful guidance for ATP by only rewarding the final proving step.[3] All the mentioned systems agree and retrospectively reward (or mark as positive) not just the final, but all the actions that contributed to the found proof. An ambiguity in the terminology seems to arise: can we have RL without an (explicit) reward? In the light of the just explained, does TRAIL really differ that much from looping in ENIGMA [6], which also iteratively improves the learned knowledge, generating training data for the next iteration using the current knowledge?

Back to the Drawing Board

In this project, we want to attack the ATP+RL target from a new angle. Rather than immediately aiming at designing an (end-to-end trainable) agent with access to the complete state (that could, in principle, solve the whole formula before the search even begins and would, effectively, only use the prover as a verifier), we want to start as close as possible to the SotA design and use RL as a research tool to further our understanding of proof search dynamics.

One possible setup, which is—at first sight—so glaringly impractical that it probably has not been tried yet, is training an agent *on a single problem only.* Yes, with a complete state description the agent can just memorize a proof (once it finds one, maybe after a long initial search) and then just keep replaying it afterwards. However, there are at least two aspects which make already this simple setup interesting.

First, in a typical proof search a complete state description very soon becomes intractably large to be processed by the agent efficiently (we talk about thousands of clauses generated in a few seconds) and thus *cheaply computable abstractions* have to come to rescue. Going back to the SotA agent, we often find it happy with representing each clause by just two numbers, its age and weight. Guiding towards a previously seen proof becomes an interesting challenge for an agent when "partially blindfolded" by simple abstractions.

The second aspect is the inherent *fragility of proof search*, on which the author recently shed light using randomization [16]. It turns out that even very small changes in a concrete run, introduced at the level of "don't care non-determinism" such as the exact order of literals in a newly generated clause, can have a tremendous impact on how long it takes to find a proof. Further investigation is needed to pinpoint what exactly causes so much chaos in our provers.[4]

In this project, we plan to undertake such investigation with the tools of RL, making use of the randomization code from our previous work [16] to turn theorem proving into a stochastic environment. This will create a second challenge for our agent, forcing it to seek robust strategies. Ultimately, we would like the agent to be able to recognize situations that are particularly unstable, so that it could respond particularly carefully. By examining the used features, we, as prover developers, will then hopefully learn how to build more robust provers ourselves.

[2] There is, however, also a meta-point: Deepire's guidance [15] does not depend on the conjecture in this sense, yet the system achieves a comparable, if not better, performance to ENIGMA on Mizar [6].

[3] And letting the prover figure out which actions were actually useful for the success by trial and error.

[4] Although a major part is probably caused by the eager simplifications and their interactions with clause selection (generating inferences on their own would stay nicely confluent), there is also the possibility that a sudden selection of what we could call a "highly explosive clause" dramatically changes the content of the weight-sorted queue, rendering the previously observed proof out of reach.

References

[1] M. Crouse, I. Abdelaziz, B. Makni, S. Whitehead, C. Cornelio, P. Kapanipathi, K. Srinivas, V. Thost, M. Witbrock, and A. Fokoue. A deep reinforcement learning approach to first-order logic theorem proving. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pp. 6279–6287. AAAI Press, 2021.

[2] Z. A. Goertzel, J. Jakubuv, and J. Urban. ENIGMAWatch: ProofWatch meets ENIGMA. In *Automated Reasoning with Analytic Tableaux and Related Methods - 28th International Conference, TABLEAUX 2019, London, UK, September 3-5, 2019, Proceedings*, vol. 11714 of *Lecture Notes in Computer Science*, pp. 374–388. Springer, 2019.

[3] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. G. Azar, and D. Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pp. 3215–3222. AAAI Press, 2018.

[4] A. Irpan. Deep reinforcement learning doesn't work yet. `https://www.alexirpan.com/2018/02/14/rl-hard.html`, 2018.

[5] J. Jakubuv and J. Urban. Enhancing ENIGMA given clause guidance. In *Intelligent Computer Mathematics - 11th International Conference, CICM 2018, Hagenberg, Austria, August 13-17, 2018, Proceedings*, vol. 11006 of *Lecture Notes in Computer Science*, pp. 118–124. Springer, 2018.

[6] J. Jakubuv and J. Urban. Hammering mizar by learning clause guidance (short paper). In *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*, vol. 141 of *LIPIcs*, pp. 34:1–34:8. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[7] C. Kaliszyk, J. Urban, H. Michalewski, and M. Olšák. Reinforcement learning of theorem proving. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pp. 8836–8847, 2018.

[8] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In *ECML 2006, Proceedings*, vol. 4212 of *Lecture Notes in Computer Science*, pp. 282–293. Springer, 2006.

[9] L. Kovács and A. Voronkov. First-order theorem proving and vampire. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, vol. 8044 of *Lecture Notes in Computer Science*, pp. 1–35. Springer, 2013.

[10] S. M. Loos, G. Irving, C. Szegedy, and C. Kaliszyk. Deep network guided proof search. In *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*, vol. 46 of *EPiC Series in Computing*, pp. 85–105. EasyChair, 2017.

[11] S. Schulz, S. Cruanes, and P. Vukmirović. Faster, higher, stronger: E 2.3. In *Proc. of the 27th CADE, Natal, Brasil*, number 11716 in LNAI, pp. 495–507. Springer, 2019.

[12] S. Schulz and M. Möhrmann. Performance of clause selection heuristics for saturation-based theorem proving. In *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*, vol. 9706 of *Lecture Notes in Computer Science*, pp. 330–345. Springer, 2016.

[13] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018.

[14] M. Suda. Aiming for the goal with sine. In *Vampire 2018 and Vampire 2019. The 5th and 6th Vampire Workshops*, vol. 71 of *EPiC Series in Computing*, pp. 38–44. EasyChair, 2019.

[15] M. Suda. Vampire with a brain is a good ITP hammer. In *Frontiers of Combining Systems - 13th International Symposium, FroCoS 2021, Birmingham, UK, September 8-10, 2021, Proceedings*, vol. 12941 of *Lecture Notes in Computer Science*, pp. 192–209. Springer, 2021.

[16] M. Suda. Vampire getting noisy: Will random bits help conquer chaos? (system description). EasyChair Preprint no. 7719, EasyChair, 2022.

[17] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.

[18] R. S. Sutton and A. G. Barto. *Reinforcement learning - an introduction.* Adaptive computation and machine learning. MIT Press, 1998.

Formal Premise Selection With Language Models

Szymon Tworkowski*[1], Maciej Mikuła*[1], Tomasz Odrzygóźdź*[5], Konrad Czechowski*[1], Szymon Antoniak*[1], Albert Q. Jiang[3], Christian Szegedy[2], Łukasz Kuciński[4], Piotr Miłoś[4], Yuhuai Wu[2]

[1] University of Warsaw, [2] Google Research
[3] University of Cambridge, [4] Polish Academy of Sciences, [5] IDEAS NCBR

Abstract

Premise selection, the problem of selecting a useful premise to prove a new theorem, is an essential part of theorem proving. Existing language models cannot access knowledge beyond a small context window, and therefore are unsatisfactory at retrieving useful premises (i.e., premise selection) from large databases for theorem proving. In this work, we provide a solution to this problem, by combining a premise selection model with a language model. We first select a handful (e.g., 8) of premises from a large theorem database consisting of $100K$ premises, and present them in the context along with proof states. The language model then utilizes these premises to construct a proof step. We show that this retrieval-augmented prover achieves significant improvements in proof rates compared to the language model alone.

1 Introduction

Language models have been recently applied to theorem proving [17, 25, 12, 14, 24] and program synthesis [7, 2, 20], achieving impressive results. **Premise selection** is a fundamental aspect of formal mathematics [28, 1, 4]. Early works in this domain often relied on symbolic [19, 5] or hybrid [15] approaches. Classical ML algorithms [30, 29, 9] have also proven effective, frequently outperforming symbolic methods by significant margins. More recently, graph neural networks mimicking the symbolic structure of mathematical expressions have shown promising results [22, 33, 10, 18].

Effective retrieval of premises from large databases is still an open challenge. In this work, we propose to approach it with a two-stage procedure, which, to the best of our knowledge, is the first method to do the selection process globally over the whole corpus. Firstly, a premise selection model (PSM) picks a handful (e.g. 8) of premises from a database. These are then presented, along with a proof state, to a premise selection guided language model (PGLM) responsible for generating a proof step. Importantly, our PSM can efficiently query large databases; in our case, we use over 100K lemmas from the entire Isabelle corpus. By providing a relatively small number of premises in context, we allow the PGLM to efficiently retrieve the correct ones, aiming to leverage its in-context learning capabilities [16].

2 Method

Premise selection model (PSM) is based on a batch contrastive learning approach similar to [1, 3, 26, 11]. It encodes proof state and premise text into embeddings. The cosine similarity of a given premise embedding and a proof state embedding estimates their mutual relevance. Premise embeddings can be precomputed and cached, allowing for the use of large databases.

*Equal Contribution

Premise-guided language model (PGLM) is a model for proof step generation. It takes as an input the *current proof state* s and *premises* (names and statements) selected by PSM. These are e.g. $k = 8$ premises from the whole database with the highest relevance to s.

We first train the PSM, then freeze the weights and use it in the training process of PGLM. The PGLM is designed to perform *premise-aware* proof step generation. By design, given the (small) context of k premises, the model selects the relevant ones to be applied in the generated proof step. This setup is motivated by recent findings [16] showing that LMs can grasp dependencies in the text within the same input much better than ones occurring across different training examples. The latter is how the state-only (our baseline model, described below) approach works. We hope that in-context learning helps the model focus on premise selection instead of memorization of frequently-occurring premises (as we hypothesize the state-only models do).

The **State-only model** is a language model that, given a proof state (goal), predicts the proof step. This is the most common setup found in prior work [14, 12, 24], used here as a baseline.

3 Experiments

We conduct our interactive theorem proving experiments on a dataset collected in Isabelle [23] which is one of the largest corpora of formal proofs. To interact with the formal environment, we use PISA [14]. The proof rates are presented in the table below.

Method	Proof rate, full	Proof rate, ≥ 1 premise	Proof rate, 0 premises
Sledgehammer [5] (baseline)	22.4%	17.7%	27.5%
State-only (baseline)	39.8%	14.7%	67.1%
PGLM+PSM (ours)	**43.1%**	**19.6%**	**68.6%**
PGLM+PSM $\cup$ *State-only*	**47.2%**	**22.6%**	**73.9%**

Table 1: Proof rate is evaluated using a best-first search solver, similar to the one mentioned in [14], on a test set of 1000 theorems. We split the test dataset into proofs originally using and not using premises; denoted ≥ 1 premise and 0 premises, respectively. For the sledgehammer baseline we use 50s timeout per proof.

Our method, *PGLM+PSM*, performs significantly better on theorems that require at least one premise and fares well on the entire test set. This indicates that the proposed two-stage method is efficient in premise retrieval. Furthermore, a significant improvement is observed when *PGLM+PSM* and the state-only model are combined. This is especially visible on the full test set, indicating that both methods have complementary strengths.

4 Conclusion and future work

We present a simple method integrating premise selection with language models, which is guided by an external retriever model. We show proof rate improvements when compared to a state-only baseline and demonstrate that our model is capable of generating novel proofs that utilise premises.

We speculate that scaling up our approach will further increase its capabilities. In particular, we hypothesise that in-context premise selection performance will improve due to better generalisation to unseen premises. If true, it would indicate better reasoning potential of the underlying language model, and as such is an attractive research direction.

References

[1] Alex A. Alemi, Francois Chollet, Niklas Een, Geoffrey Irving, Christian Szegedy, and Josef Urban. Deepmath - deep sequence models for premise selection, 2016.

[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021.

[3] Kshitij Bansal, Sarah M. Loos, Markus N. Rabe, Christian Szegedy, and Stewart Wilcox. Holist: An environment for machine learning of higher order logic theorem proving. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 454–463. PMLR, 2019.

[4] Kshitij Bansal, Christian Szegedy, Markus N. Rabe, Sarah M. Loos, and Viktor Toman. Learning to reason in large theories without imitation, 2019.

[5] Sascha Böhme and Tobias Nipkow. Sledgehammer: judgement day. In *International Joint Conference on Automated Reasoning*, pages 107–121. Springer, 2010.

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

[8] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. The pile: An 800gb dataset of diverse text for language modeling, 2021.

[9] Thibault Gauthier and Cezary Kaliszyk. Premise selection and external provers for hol4. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, CPP '15, page 49–57, New York, NY, USA, 2015. Association for Computing Machinery.

[10] Zarathustra A. Goertzel, Jan Jakubův, Cezary Kaliszyk, Miroslav Olšák, Jelle Piepenbrock, and Josef Urban. The isabelle enigma, 2022.

[11] Jesse Han, Tao Xu, Stanislas Polu, Arvind Neelakantan, and Alec Radford. Contrastive finetuning of generative language models for informal premise selection. *6th Conference on Artificial Intelligence and Theorem Proving*, 2021.

[12] Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward W. Ayers, and Stanislas Polu. Proof artifact co-training for theorem proving with language models. ICLR, 2022.

[13] Mauro Jaskelioff and Stephan Merz. Proving the correctness of disk paxos. *Archive of Formal Proofs*, June 2005. http://isa-afp.org/entries/DiskPaxos.html, Formal proof development.

[14] Albert Qiaochu Jiang, Wenda Li, Jesse Michael Han, and Yuhuai Wu. Lisa: Language models of isabelle proofs. *6th Conference on Artificial Intelligence and Theorem Proving*, 2021.

[15] Daniel Kühlwein, Jasmin Christian Blanchette, Cezary Kaliszyk, and Josef Urban. Mash: Machine learning for sledgehammer. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, pages 35–50, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[16] Yoav Levine, Noam Wies, Daniel Jannai, Dan Navon, Yedid Hoshen, and Amnon Shashua. The inductive bias of in-context learning: Rethinking pretraining example design. In *International Conference on Learning Representations*, 2022.

[17] Wenda Li, Lei Yu, Yuhuai Wu, and Lawrence C. Paulson. Isarstep: a benchmark for high-level mathematical reasoning. In *International Conference on Learning Representations*, 2021.

[18] Zhaoyu Li, Binghong Chen, and Xujie Si. Graph contrastive pre-training for effective theorem reasoning, 2021.

[19] Jia Meng and Lawrence Paulson. Lightweight relevance filtering for machine-generated resolution problems. *J. Applied Logic*, 7:41–57, 03 2009.

[20] Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. Show your work: Scratchpads for intermediate computation with language models. In *Deep Learning for Code Workshop*, 2022.

[21] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding, 2018.

[22] Aditya Paliwal, Sarah Loos, Markus Rabe, Kshitij Bansal, and Christian Szegedy. Graph representations for higher-order logic and theorem proving, 2019.

[23] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. 1993.

[24] Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. Formal mathematics statement curriculum learning, 2022.

[25] Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving, 2020.

[26] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision, 2021.

[27] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[28] Christian Szegedy. A promising path towards autoformalization and general artificial intelligence. In Christoph Benzmüller and Bruce Miller, editors, *Intelligent Computer Mathematics*, pages 3–20, Cham, 2020. Springer International Publishing.

[29] Agnieszka Słowik, Chaitanya Mangla, Mateja Jamnik, Sean B. Holden, and Lawrence C. Paulson. Bayesian optimisation with gaussian processes for premise selection, 2019.

[30] Josef Urban, Geoff Sutcliffe, Petr Pudlák, and Jiří Vyskočil. Malarea sg1 - machine learner for automated reasoning with semantic guidance. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning*, pages 441–456, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[31] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[32] Ben Wang and Aran Komatsuzaki. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax, May 2021.

[33] Mingzhe Wang, Yihe Tang, Jian Wang, and Jia Deng. Premise selection for theorem proving by deep graph embedding, 2017.

A Experimental setup

A.1 LM setup

For language modeling, we use a decoder-only transformer [31] with 30M non-embedding parameters. The setup (weight initialization, positional embeddings, and other architectural hyperparameters) is exactly the same as in GPT-J [32]. We use a pretrained BPE tokenizer from [27]. Similarly to GPT-f [25], the loss function is calculated only on the proof step tokens. As a context for proof step generation we use one sentence representing the proof state for state only model, and premises + proof state sentence for PGLM+PSM setup.

For the *PGLM+PSM* model, in our main result, we provide it with top $k = 4$ premises from the premise selection model.

All of the models are pretrained on The Pile [8] - GitHub + arXiv dataset for 500k steps with context length of 2048 as in [6] and total batch size of 2^{17} tokens per update.

A.2 PSM setup

We modify the InfoNCE [21] loss by only using row-wise softmax (column-wise softmax is ablated). Batch size of 512 proof states is used. We also randomly sample 1536 additional negative premises within a batch (512 proof states and 2048 premises in each batch, for each proof state there is exactly one positive premise and 2047 negatives), and we find it helpful to the score (see Tab. 2). We use a non-pretrained, 6-layer decoder-only transformer (15M non-embedding parameters).

B Dataset and Environment

Isabelle [23] is an interactive theorem prover (ITP). It allows mathematical formulas to be expressed in a formal language and provides tools for proving those formulas, which are verified by a logical kernel. Its main application is the formalization of mathematical proofs and in particular formal verification, which includes proving the correctness of computer hardware or software and proving properties of computer languages and protocols. Each Isabelle library is composed of theories. A proof for a given theorem is a sequence of **proof steps**, with each step being a proof tactic or part of a declaration. Each subsequent proof step changes the state (referred to as **proof state**) of the current proof. A proof step can make use of **premises**, which are simply references to definitions, axioms, or previously proven theorems. This theorem proving setting constitutes a Partially Observable Decision Process and thus can be represented by a sequential decision process in a certain environment. An example of such an environment is the PISA environment [14], which we used for all the experiments. We trained models using a dataset mined from the Archive of Formal Proofs (AFP)[13] and all the standard libraries available in Isabelle. The dataset consists of 220K lemmas, with a total of 2.4M (proof state, proof step) pairs. For the premise selection task, we chose the proof steps that utilised at least one premise, which resulted in 400K training examples.

C Premise selection - ablation study

We investigate what contributes to the performance of our retrieval model by reducing its expressive power to a 1-layer transformer (first experiment), as well as removing our negative sampling strategy (second experiment). We observe a significant drop in recall with the changes.

Model	recall@1	recall@4	recall@8	recall@16	recall@64	recall@128
1L transformer	0.168	0.347	0.447	0.565	0.663	0.809
6L transformer	0.203	0.408	0.516	0.621	0.781	0.832
6L transformer + neg.	**0.230**	**0.446**	**0.561**	**0.656**	**0.793**	**0.839**

Table 2: Retrieval metrics (top-k recall) comparison. On the test dataset, we measure percentage of situations, where given a proof state, the ground truth premise has been retrieved among top-k according to the PSM model. The *6L transformer + neg.* entry refers to a model utilizing our negative sampling strategy with 1536 additionally sampled negatives (see A.2 for details).

D Proofs

```
Theorem 1:
lemma reachable_steps: "<exists> xs. steps xs <and> hd
xs = s<ŝub> <and> last xs = x" if "reachable x"

Original proof:
using that
unfolding reachable_def
proof induction
case base
then
show ?case
by (inst_existentials "[sŝub>0]"; force)
next
case (step y z)
from step.IH
guess xs
by clarify
with step.hyps
show ?case
apply (inst_existentials "xs @ [z]")
apply (force intro: graphI)
by (cases xs; auto)+
qed

Our proof:
using that
unfolding reachable_def
by (fastforce dest: reaches_steps)
```

Proof 1: Our model is capable of proposing short and neat proofs when compared to the original.

```
Theorem 2:
lemma (in wf_digraph) iapath_dist_ends: "<And>u p v.
iapath u p v <Longrightarrow> u <noteq> v"

Original proof:
unfolding pre_digraph.gen_iapath_def
by (metis apath_ends)

Our proof:
by (unfold gen_iapath_def) (auto dest:
apath_nonempty_ends)
```

Proof 2: Exemplary proof that state-only model failed to close, whereas our PGLM+PSM managed to derive a fundamentally different proof without using metis - in contrast to original proof.

E Inputs comparison

```
<|PREMISE_NAME|>less_top_ennreal
<|PREMISE|>"x < top <longleftrightarrow> (<exists>r<ge>0. x = ennrealr)"
<|PREMISE_NAME|>fact_dvd_higher_pderiv
<|PREMISE|>"[:fact n :: int:] dvd (pderiv ^^ n) p"
<|PREMISE_NAME|>sameDom_sym
<|PREMISE|>"sameDom inp inp' = sameDom inp' inp"
<|PREMISE_NAME|>moebius_inverse
<|PREMISE|>assumes "a * d <noteq> b * c" "c * z + d <noteq> 0" shows "moebius
d (-b) (-c) a (moebius a b c d z) = z"
<|ISA_OBS|>proof (prove) goal (1 subgoal): 1. prv (neg <phi>R)
<|PREV_STEPS|>have "prv (neg <phi>R)"<|PROOF_STEP|>
```

Input 1: Exemplary input for PGLM+PSM model with top-4 premises. Input is a single sentence, here, for readability, split into multiple lines.

```
<|ISA_OBS|>proof (prove) goal (1 subgoal): 1. prv (neg <phi>R)
<|PREV_STEPS|>have "prv (neg <phi>R)"<|PROOF_STEP|>
```

Input 2: Exemplary input for classical State-only model. Input is a single sentence, here, for readability, split into multiple lines.

NaturalProver: Grounded Natural Language Proof Generation with Language Models

Sean Welleck[12*], Jiacheng Liu[1*],
Ximing Lu[2], Hannaneh Hajishirzi[12], Yejin Choi[12]

[1]University of Washington, [2]Allen Institute for Artificial Intelligence, *Equal Contribution

Introduction. We envision assistive systems for *informal* mathematics that suggest proof steps or solutions to a user, inspired by the use of language models in formal proof assistants (e.g. [4, 6, 7, 8, 11]) and informal premise selection [3, 5, 13]. We study two new generation tasks in natural mathematical language: suggesting the next step in a proof, and full-proof generation.

We develop NaturalProver, a language model that generates proofs by conditioning on background references (theorems, definitions), and optionally enforces their presence with constrained decoding. NaturalProver improves the quality of next-step suggestions and generated proofs over fine-tuned GPT-3 [1], with either retrieved or human-provided references, according to human evaluations from university-level mathematics students.

NaturalProver is capable of proving short (2-6 step) theorems and providing next-step suggestions that are rated as correct and useful more than 50% of the time, which is to our knowledge the first demonstration of these capabilities using neural language models.

Data. We create a NaturalProofs-Gen dataset with data adapted from the ProofWiki domain of NaturalProofs [13]. Each example pairs a theorem $\mathbf{x}$ with a gold proof $\mathbf{y} = (y_1, \ldots, y_T)$, where each y_t is a variable-length proof step. Each proof mentions references $\{\mathbf{r}_1, \ldots, \mathbf{r}_{R_y}\}$ from a reference set of roughly 33k theorems and definitions, analogous to how Wikipedia articles reference other pages. For example, Figure 1 shows a 4-step proof with references in blue. We use splits from NaturalProofs for training, and create evaluation sets with 100 validation and 100 test theorems.

Tasks. The **proof generation** task is to generate a proof $\mathbf{y}$ given theorem $\mathbf{x}$. The **next-step** task is to generate a next step y_t given theorem $\mathbf{x}$ and proof history $y_{<t}$ from a gold proof. We consider an additional *provided* setting where the model is given gold references $\{\mathbf{r}_1^*, \ldots, \mathbf{r}_{R_y}^*\}$.

Methods. We study a vanilla language model and two 'knowledge-grounded' variations, along with the effect of constrained decoding. For each model, we fine-tune GPT-3 Curie, a ≈13B parameter autoregressive transformer language model trained on internet text.[1]

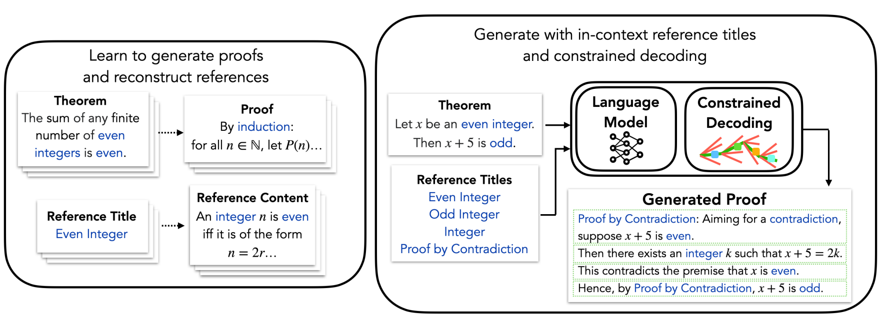


Figure 1: NaturalProver proves `Even Integer Plus 5 is Odd`.

[1]https://blog.eleuther.ai/gpt3-model-sizes/. We use the OpenAI API. We also release open-source GPT-J and GPT-2 models, fine-tuning and evaluation code, and the NaturalProofs-Gen dataset.

Method	Reasoning Errs (↓) Ref.	Eqn.	Other	Lexical Errs (↓) Lang.	Sym.	Per-Step (↑) Useful	Correct	Full Proof (↑) Useful	Correct
GPT-3 (curie)	30.92	32.54	40.15	5.61	5.24	25.69	28.18	20%	13%
Retrieved	23.52	37.55	23.66	4.54	6.19	41.54	33.56	32%	24%
Provided	25.84	35.93	25.23	8.41	5.35	39.60	26.30	35%	24%
+constrained	**23.61**	**28.54**	**18.45**	5.58	3.65	**46.57**	**35.41**	**45%**	**32%**
Next-step	19.70	26.32	19.10	8.57	5.86	51.43	42.86	–	–

Table 1: Human evaluation results for full-proof and next-step generation (bottom).

The knowledge-grounded models condition on references, $p_\theta(\mathbf{y}|\mathbf{x}, R)$. As language model context windows prevent conditioning on full reference documents, we condition on reference *titles*, and fine-tune on (title, content) pairs, which lets the model memorize the associated content. For example, Fig 1 shows `Even Integer` and its content. We study 3 variants:

1. **Baseline.** This model is simply fine-tuned on the 12.5k (theorem, proof) training examples. At test time, the model is given a theorem and uses greedy decoding to generate a proof.
2. **Retrieved.** This model is conditioned on *retrieved* references, $p_\theta(\mathbf{y}|\mathbf{x}, \hat{\mathbf{r}}_1, \ldots, \hat{\mathbf{r}}_{20})$. We use a pretrained joint retrieval model from [13], which was trained on NaturalProofs to map each theorem to the references in its ground-truth proof. At test time, we condition on a test theorem and its top-20 retrieved reference titles, and use greedy decoding.
3. **Provided.** This model is conditioned on human-provided references, $p_\theta(\mathbf{y}|\mathbf{x}, \mathbf{r}^*_1, \ldots, \mathbf{r}^*_{R_\mathbf{y}})$, meaning $\{\mathbf{r}^*_1, \ldots, \mathbf{r}^*_{R_\mathbf{y}}\}$ is the set of reference-titles mentioned in a ground-truth proof. At test time, the model receives a test theorem and reference titles from a ground-truth proof.

Constrained decoding. We use constrained decoding to improve reference usage in the provided setting, as references are known to be relevant to a proof of the theorem. We generate step-by-step by sampling multiple step candidates, keeping those with high log-probability and reference-coverage in a beam, and continuing to the next step.

Evaluation. We created a schema of reasoning and lexical errors and an online system for per-step and full proof annotation. We recruited 15 students from the Departments of Mathematics and Applied Mathematics at the University of Washington as annotators. Annotators label the $\{0, 1\}$ correctness, usefulness, and presence of errors in each proof step, then rate the full proof's correctness and usefulness. We also find positive correlations between human judgments and automatic lexical (e.g. Gleu) and grounding (e.g. Reference-F1) metrics and discuss these results in the talk.

Main results. We show our main human evaluation results in Table 1. Knowledge-grounding, either retrieved or human provided, improves proof generation. Constrained decoding further improves the provided-knowledge model, with 32% of its proofs rated as correct and 45% rated as useful as an aid for human proof writers. On the per-step level, 35% of its proof steps are correct and 47% are useful, increasing to 51% useful and 43% correct given a correct proof-so-far. On the other hand, our models often struggle with correctly deploying and utilizing references (23.6% reference error rate), doing symbolic derivations (28.5% equation error rate), and longer proofs. We give quantitative and qualitative analyses of these successes and errors in the talk.

Looking forward. Our results suggest that useful interactive proof assistants for informal mathematics are plausible as methods improve further. Investigating architectural improvements [14], iterative improvement [2, 7], and pretraining [9], as well as the role of formalization [10, 12] in informal proof generation are interesting future directions.

References

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, T. J. Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeff Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *ArXiv*, abs/2005.14165, 2020.

[2] Bhavana Dalvi, Oyvind Tafjord, and Peter Clark. Towards teachable reasoning systems. *ArXiv*, abs/2204.13074, 2022.

[3] Deborah Ferreira and André Freitas. Natural language premise selection: Finding supporting statements for mathematical text. In *Proceedings of the 12th Language Resources and Evaluation Conference*, pages 2175–2182, Marseille, France, May 2020. European Language Resources Association.

[4] Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward Ayers, and Stanislas Polu. Proof artifact co-training for theorem proving with language models. In *International Conference on Learning Representations*, 2022.

[5] Jesse Michael Han, Tao Xu, Stanislas Polu, Arvind Neelakantan, and Alec Radford. Contrastive finetuning of generative language models for informal premise selection. In *AITP*, 2021.

[6] Albert Qiaochu Jiang, Wenda Li, Jesse Michael, Han Openai, and Yuhuai Wu. LISA: Language models of ISAbelle proofs. In *AITP*, 2021.

[7] Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. Formal mathematics statement curriculum learning, 2022.

[8] Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving, 2020.

[9] Markus Norman Rabe, Dennis Lee, Kshitij Bansal, and Christian Szegedy. Mathematical reasoning via self-supervised skip-tree training. In *International Conference on Learning Representations*, 2021.

[10] Christian Szegedy, editor. *A Promising Path Towards Autoformalization and General Artificial Intelligence*, 2020.

[11] Josef Urban and Jan Jakubův. First neural conjecturing datasets and experiments. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2020.

[12] Qingxiang Wang, Chad Brown, Cezary Kaliszyk, and Josef Urban. Exploration of neural machine translation in autoformalization of mathematics in Mizar. In *CPP 2020 - Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, co-located with POPL 2020*, 2020.

[13] Sean Welleck, Jiacheng Liu, Ronan Le Bras, Hannaneh Hajishirzi, Yejin Choi, and Kyunghyun Cho. Naturalproofs: Mathematical theorem proving in natural language. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[14] Yuhuai Wu, Markus Norman Rabe, DeLesley Hutchins, and Christian Szegedy. Memorizing transformers. In *International Conference on Learning Representations*, 2022.

Compressed Combinatory Proof Structures and Blending Goal- with Axiom-Driven Reasoning: Perspectives for First-Order ATP with Condensed Detachment and Clausal Tableaux

Christoph Wernhard

University of Potsdam, Germany
info@christophwernhard.com

1 Background

Goal-driven first-order provers such as *leanCoP* [14] or *SETHEO* [9], which may described as based on clausal tableaux [8], the connection method [1, 3] or model elimination [12], in essence enumerate tree-shaped proof structures, interwoven with unification of formulas that are associated with nodes of the structures. While they do not compete with state-of-the art systems in the range of solvable problems, they have merits that are relevant in certain contexts: Proofs are typically emitted as data structures of simple and detailed forms, making them suitable as inputs for further processing. Through iterative deepening, proofs tend to be short. The provers facilitate comparing alternate proofs of a problem or influencing the shape of proofs. Implementations can be manageable and small [18], making the approach attractive for adaptation to specific logics [15, 16, 17] and novel combinations with other techniques [7, 31, 32, 6].

Here we aim to preserve the merits of that approach, while moving on to stronger proving capabilities. Our concrete starting point is a view of condensed detachment as a specialization of the connection method [29]. It provides a simplified variant of first-order ATP that still has many of its essential characteristics and seems suitable as basis for the development and study of new techniques. Emphasis is on the explicit consideration of proof structures in a simple form, as full binary trees or terms. Condensed detachment has dedicated applications in the investigation of propositional logics [24], reflected in about 200 such *TPTP* problems [27], and can be more generally used as inference rule for arbitrary first-order Horn problems.

The contribution is based on [29] as well as ongoing work [27, 28]. It is backed by an implemented system, *CD Tools*, available as free software from

http://cs.christophwernhard.com/cdtools/.

The system website also provides detailed result tables for experiments, including graphical proof visualizations.

2 Theses

In the contribution we elaborate the following two theses.

Thesis 1: Compressed Combinatory Proof Structures. Representing a proof tree by a combinator term [23, 30] that normalizes to the tree lets subtle forms of duplication within the tree materialize as duplicated subterms of the combinator term. In a DAG representation of the combinator term these straightforwardly factor into shared subgraphs. As an example, consider the proof tree

$$2(2(2(2(2(2(2(21))))))). \tag{i}$$

Here $1, 2$ are axiom identifiers and inner nodes correspond to a condensed detachment step with left and right children as proofs of major and minor, resp., premises. The notation for the inner detachment nodes follows the common notation for application in λ- and combinator terms, i.e., as juxtaposition of left and right subtree, with parentheses according to left associativity. Tree (i) can be expressed with the combinator term

$$\mathsf{B}(\mathsf{B}22)(\mathsf{B}22)(\mathsf{B}(\mathsf{B}22)(\mathsf{B}22)1), \tag{ii}$$

where $\mathsf{B}22$ and $\mathsf{B}(\mathsf{B}22)(\mathsf{B}22)$ each have two incoming edges in the minimal DAG. To search for proofs, combinator terms can be enumerated, like clausal tableaux, with simultaneously relating formulas associated with components of the enumerated structures through unification. From the clausal tableaux or connection method point of view, the approach realizes compressions known from the connection structure calculus [4, 5, 2], which was never implemented. As a refinement to restrict the search space, the enumeration of combinator terms can be based on *proof schemas*, pattern terms such as $\mathsf{r}(p, q)$, with an associated semantics defined by a combinator term with parameters p, q.

Thesis 2: Blending Goal- with Axiom-Driven Reasoning. The goal-driven proof structure enumeration by clausal tableau methods can be generalized beneficially to a method that blends with axiom-driven enumeration in configurable ways. The core method then enumerates proof structures, paired with the most general theorems (MGTs) [29] proven by them from given axioms, on a given *level*, characterized for example by the number of tree nodes or tree height. In axiom-driven mode, the proof structure and the corresponding MGT are both outputs whose values can be cached. In goal-driven mode, only the proof structure is an output, while the goal formula is an instantiated input. The overall operation is an iterated interplay of both modes on increasing levels: Basically the axiom-driven mode is performed, but before solutions for a new level are computed and cached, the goal-driven mode is invoked, at the new level and, depending on the configuration, possibly at a number of increasingly higher levels. In both modes, subproblems on lower levels can be solved by accessing the cache with previously computed proof-and-lemma pairs.

The extreme of a purely goal-driven configuration acts much like a goal-driven clausal tableau prover with iterative deepening. A purely axiom-driven configuration just generates lemmas, consequences, from the given axioms. The axiom-driven component in particular enables heuristic restrictions based on the MGTs, the lemma components of the cache entries, e.g., with limiting their size, restricting the number of different proofs per MGT, or limiting the overall number of cache entries where some ordering based on the MGT may determine the entries to be kept. In purely goal-driven configurations – and in goal-driven clausal tableau provers alike – MGTs are not materialized and hence not available as basis of heuristic restrictions. For condensed detachment problems, the blending of goal- and axiom-driven structure enumeration leads to success rates that drastically improve on goal-driven clausal tableau provers and compare to the lower end of state-of-the-art provers, while proofs are relatively short.

3 Implementation and Experiments

CD Tools includes two provers, *SGCD* (*Structure Generating theorem proving for Condensed Detachment*) and *CCS* (*Compressed Combinatory Structures*) that roughly address Theses 2 and 1, respectively. Most experiments so far were performed on the 196 problems in *TPTP 8.0.0* that are condensed detachment problems satisfying certain further constraints [27]. The *TPTP* rates 189 of these lower than 1.00 and 151 with 0.00. Clausal tableau provers are known to prove 92 of the 196 problems [27].[1]

With the approach of Thesis 2, 176 problems can be proven in different configurations of *SGCD* [27, 28] for level characterizations by number of tree nodes and height. The resulting proofs are typically rather small. The set of 89 problems provable by two purely goal-driven configurations of *SGCD* is, as expected, very similar to the set of 92 problems provable with clausal tableaux. In further experiments, *SGCD* was configured with a novel level characterization of the full binary trees used as proof structures that was motivated by observations at a human formal proof [29, 27]: The trees at level 0 are single nodes representing axioms. The trees at a level $n + 1$ are those where the left or right child is the root of a tree at level n and the other child is the root of a (not necessarily strict) subtree of its sibling or an arbitrary tree at level 0. In largely axiom-driven configurations this leads to 153 proven problems, apparently with proofs of small *compacted size* (size of the minimal DAG for the tree, or number of distinct compound subterms [29, 25]), also for problems where systematic search for minimal compacted size seems not feasible.[2]

CCS, the second prover in *CD Tools*, performs iterative deepening on compacted size of the proof structures and can incorporate, as suggested by Thesis 1, compressions with combinators and proof schemas, proof structure patterns defined by combinator terms. So far it was tried with exhaustive search, i.e., without heuristic restrictions, in purely goal-driven mode. Search for proofs with guaranteed minimal compacted size [25] succeeds for 86 problems. For 79 problems it is, moreover, possible to obtain all proofs with minimal compacted size. To get an idea of compression possibilities with the combinator approach and to see which particular combinators seem useful for proofs from applications, proofs obtained by *SGCD* and *CCS* for 176 problems were first compressed into tree grammars with *TreeRePair* [11], an advanced tool targeted at XML compression, and then, converted via λ-terms to combinator terms with a method from the implementation of functional programming languages [19, Chap. 16].

Concerning proof search with combinators, experiments were performed with configurations characterized by sets of proof schemas, which succeeded on 88 problems, including 6 on which the search for an "uncompressed" proof with minimal compacted size failed. Proof search with *CCS* was also tried on general Horn problems, the 562 problems of *TPTP specialist class* `CNF_UNS_RFO_NEQ_HRN`, of which 549 are rated lower than 1.00, 425 with 0.00, and around 430 are provable by clausal tableaux.[3] In five configurations with sets of proof schemas, some corresponding to specific forms of resolution, *CCS* – configured for goal-driven exhaustive search with iterative deepening upon compacted size – proves 421 of these, including 67 rated between 0.25 and 0.50, with a large overlap with those provable by clausal tableaux.

Acknowledgments. Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project-ID 457292495. The work was supported by the North-German Supercomputing Alliance (HLRN).

[1] *SETHEO 3.3* [13], *S-SETHEO* [10], *lazyCoP 0.1* [22] and *SATCoP 0.1* [21] together prove 76 problems according to the *ProblemAndSolutionStatistics* document of the *TPTP*. *leanCoP 2.1* proves 50 problems and *CMProver* [26] in different configurations proves 89 problems [27].

[2] Problem *LCL038-1* belongs to these. For this problem, which, upon suggestion in [20], was considered often in ATP and whose human proofs were analyzed in [29], *SGCD* found a proof with compacted size 22 [27].

[3] E.g., 414 by the four provers accounted in *ProblemAndSolutionStatistics* that were mentioned in footnote 1.

References

[1] W. Bibel. *Automated Theorem Proving.* Vieweg, Braunschweig, 1982. Second edition 1987. doi:10.1007/978-3-322-90102-6.

[2] W. Bibel and E. Eder. Methods and calculi for deduction. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 1, chapter 3, pages 67–182. Oxford Univ. Press, 1993.

[3] W. Bibel and J. Otten. From Schütte's formal systems to modern automated deduction. In R. Kahle and M. Rathjen, editors, *The Legacy of Kurt Schütte*, chapter 13, pages 215–249. Springer, 2020. doi:10.1007/978-3-030-49424-7_13.

[4] E. Eder. A comparison of the resolution calculus and the connection method, and a new calculus generalizing both methods. In E. Börger, H. Kleine Büning, and M. M. Richter, editors, *CSL '88*, volume 385 of *LNCS*, pages 80–98. Springer, 1989. doi:10.1007/BFb0026296.

[5] E. Eder. *Relative Complexities of First Order Calculi.* Vieweg, Braunschweig, 1992. doi:10.1007/978-3-322-84222-0.

[6] M. Färber, C. Kaliszyk, and J. Urban. Machine learning guidance for connection tableaux. *J. Autom. Reasoning*, 65(2):287–320, 2021. doi:10.1007/s10817-020-09576-7.

[7] C. Kaliszyk and J. Urban. FEMaLeCoP: Fairly efficient machine learning connection prover. In M. Davis, A. Fehnker, A. McIver, and A. Voronkov, editors, *LPAR-20*, volume 9450 of *LNCS (LNAI)*, pages 88–96. Springer, 2015. doi:10.1007/978-3-662-48899-7_7.

[8] R. Letz. *Tableau and Connection Calculi. Structure, Complexity, Implementation.* Habilitationsschrift, TU München, 1999. Available from http://www2.tcs.ifi.lmu.de/~letz/habil.ps, accessed Jun 30, 2022.

[9] R. Letz, J. Schumann, S. Bayerl, and W. Bibel. SETHEO: A high-performance theorem prover. *J. Autom. Reasoning*, 8(2):183–212, 1992. doi:10.1007/BF00244282.

[10] R. Letz and G. Stenz. Model elimination and connection tableau procedures. In A. Robinson and A. Voronkov, editors, *Handb. of Autom. Reasoning*, volume 1, pages 2015–2114. Elsevier, 2001.

[11] M. Lohrey, S. Maneth, and R. Mennicke. XML tree structure compression using RePair. *Inf. Syst.*, 38(8):1150–1167, 2013. System available from https://github.com/dc0d32/TreeRePair, accessed Jun 30, 2022. doi:10.1016/j.is.2013.06.006.

[12] D. W. Loveland. *Automated Theorem Proving: A Logical Basis.* North-Holland, Amsterdam, 1978.

[13] M. Moser, O. Ibens, R. Letz, J. Steinbach, C. Goller, J. Schumann, and K. Mayr. SETHEO and E-SETHEO – the CADE-13 systems. *J. Autom. Reasoning*, 18(2):237–246, 1997. doi:10.1023/A:1005808119103.

[14] J. Otten. Restricting backtracking in connection calculi. *AI Communications*, 23(2-3):159–182, 2010. doi:10.3233/AIC-2010-0464.

[15] J. Otten. MleanCoP: A connection prover for first-order modal logic. In S. Demri, D. Kapur, and C. Weidenbach, editors, *IJCAR 2014*, volume 8562 of *LNCS (LNAI)*, pages 269–276. Springer, 2014. doi:10.1007/978-3-319-08587-6_20.

[16] J. Otten. Non-clausal connection-based theorem proving in intuitionistic first-order logic. In C. Benzmüller and J. Otten, editors, *ARQNL 2016*, volume 1770 of *CEUR Workshop Proc.*, pages 9–20. CEUR-WS.org, 2016. URL: http://ceur-ws.org/Vol-1770/paper1.pdf.

[17] J. Otten. The nanoCoP 2.0 connection provers for classical, intuitionistic and modal logics. In A. Das and S. Negri, editors, *TABLEAUX 2021*, volume 12842 of *LNCS (LNAI)*, pages 236–249. Springer, 2021. doi:10.1007/978-3-030-86059-2_14.

[18] J. Otten and W. Bibel. leanCoP: lean connection-based theorem proving. *J. Symb. Comput.*, 36(1-2):139–161, 2003. doi:10.1016/S0747-7171(03)00037-3.

[19] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[20] F. Pfenning. Single axioms in the implicational propositional calculus. In E. Lusk and R. Overbeek, editors, *CADE-9*, volume 310 of *LNCS (LNAI)*, pages 710–713. Springer, 1988. doi:10.1007/BFb0012869.

[21] M. Rawson and G. Reger. Eliminating models during model elimination. In A. Das and S. Negri, editors, *TABLEAUX 2021*, volume 12842 of *LNCS (LNAI)*, pages 250–265. Springer, 2021. doi:10.1007/978-3-030-86059-2_15.

[22] M. Rawson and G. Reger. lazyCoP: Lazy paramodulation meets neurally guided search. In A. Das and S. Negri, editors, *TABLEAUX 2021*, volume 12842 of *LNCS (LNAI)*, pages 187–199. Springer, 2021. doi:10.1007/978-3-030-86059-2_11.

[23] M. Schönfinkel. Über die Bausteine der mathematischen Logik. *Math. Ann.*, 92(3–4):305–316, 1924. doi:10.1007/BF01448013.

[24] D. Ulrich. A legacy recalled and a tradition continued. *J. Autom. Reasoning*, 27(2):97–122, 2001. doi:10.1023/A:1010683508225.

[25] R. Veroff. Finding shortest proofs: An application of linked inference rules. *J. Autom. Reasoning*, 27(2):123–139, 2001. doi:10.1023/A:1010635625063.

[26] C. Wernhard. The PIE system for proving, interpolating and eliminating. In P. Fontaine, S. Schulz, and J. Urban, editors, *PAAR 2016*, volume 1635 of *CEUR Workshop Proc.*, pages 125–138. CEUR-WS.org, 2016. URL: http://ceur-ws.org/Vol-1635/paper-11.pdf.

[27] C. Wernhard. CD Tools – Condensed detachment and structure generating theorem proving (system description). https://arxiv.org/abs/2207.08453, 2022. doi:10.48550/ARXIV.2207.08453.

[28] C. Wernhard. Generating compressed combinatory proof structures – an approach to automated first-order theorem proving. In B. Konev, C. Schon, and A. Steen, editors, *PAAR 2022*, volume 3201 of *CEUR Workshop Proc.* CEUR-WS.org, 2022. Preprint: http://cs.christophwernhard.com/papers/css.pdf.

[29] C. Wernhard and W. Bibel. Learning from Łukasiewicz and Meredith: Investigations into proof structures. In A. Platzer and G. Sutcliffe, editors, *CADE 28*, volume 12699 of *LNCS (LNAI)*, pages 58–75. Springer, 2021. doi:10.1007/978-3-030-79876-5_4.

[30] S. Wolfram. *Combinators – A Centennial View*. Wolfram Media Inc, 2021. Accompanying webpage: https://writings.stephenwolfram.com/2020/12/combinators-a-centennial-view/, accessed Jun 30, 2022.

[31] Z. Zombori, J. Urban, and C. E. Brown. Prolog technology reinforcement learning prover (system description). In N. Peltier and V. Sofronie-Stokkermans, editors, *IJCAR 2020*, volume 12167 of *LNCS (LNAI)*, pages 489–507. Springer, 2020. doi:10.1007/978-3-030-51054-1_33.

[32] Z. Zombori, J. Urban, and M. Olšák. The role of entropy in guiding a connection prover. In A. Das and S. Negri, editors, *TABLEAUX 2021*, volume 12842 of *LNCS (LNAI)*, pages 218–235. Springer, 2021. doi:10.1007/978-3-030-86059-2_13.

Autoformalization for Neural Theorem Proving

Yuhuai Wu[1], Albert Jiang[2], Wenda Li[2], Markus N. Rabe[1], Charles Staats[1], Mateja Jamnik[2], and Christian Szegedy[1]

[1]Google Research
[2]University of Cambridge

1 Introduction

In this work, we demonstrate the feasibility and usefulness of autoformalization in the context of the newly introduced MiniF2F [10] benchmark. We use large language models to translate several thousands of informal problems into Isabelle and use them to improve our neural theorem prover. We find that transformer-based [7] language models trained on a large amount of web data are capable of formalizing mathematical competition problem statements with a relatively high success rate and the resulting statements can be used for creating new correct proofs that can be used for fine-tuning a neural theorem prover for improved proof automation. Using this methodology, we achieve a new state of the art on the MiniF2F benchmark.

2 Autoformalization using Large Language Models

Inspired by the success of large language models (LLMs) for synthesizing computer code by co-training on both natural language and code on web-scale data, we explore the capabilities of large language models (LLMs) that were trained on a large amount web data to turn natural language mathematics into formalized theorems (Isabelle theorems in this case). This is essentially a machine translation task [8] in which the input language is English and output language is formal code used by the interactive proof assistant Isabelle [9].

In particular, we exploit the impressive few-shot capability of LLMs by providing a few examples of the translations which improves the quality of our translation. We ran our initial experiments with using Codex and prompted the language model for the task of formalizing the informal statements. Here are two examples of automatically formalized theorems, with prompts provided in the Appendix.

Natural language version: *"Prove that there is no function f from the set of non-negative integers into itself such that $f(f(n)) = n + 1987$ for every n."* Translate the natural language version to an Isabelle version:

```
theorem
  fixes f :: "nat ⇒ nat"
  assumes "∀ n. f (f n) = n + 1987"
  shows False
```

Natural Language version: "*When all the girls at Madeline's school line up in rows of eight, there are seven left over. If instead they line up in rows of four, how many are left over? The final answer is 3.*"
Translate the natural language version to an Isabelle version:

```
theorem
  fixes n::nat
  assumes "n mod 8 = 7"
  shows "n mod 4 = 3"
```

Remarkably, we see in both examples, Codex was able to translate the natural language statement into Isabelle formal theorems perfectly. In the first example, the model can understand what it means by the phrase "to itself", and correctly formalize the domain of function: f :: "$nat \Rightarrow nat$". The second example is even more remarkable. First of all, a formal translation of a grade school math problem should not ever exist in the pre-training corpus, as this type of mathematics is not of interest to formal mathematicians. Second, the examples in the prompt we provide also are not of this type of problem. It is hence remarkable that the model is capable of extrapolating to this type of statement – a true extrapolation. This shows a great promise of using LLMs for doing auto-formalization.

3 Autoformalization Improves Neural Theorem Proving

To study the usefulness of the formalized statements, we explore if one can improve neural theorem provers by training the model on automatically translated theorems. In particular, we study auto-formalization on a constrained setting – mathematical competition problems, where it has little requirement in formalizing the definitions and background theory.

For our neural theorem prover, we use a recently introduced theorem prover LISA [4] that proves Isabelle theorems by language modeling the best action conditioned on the current proof state. The input of the transformer-based neural network is the proof state and the output is the tactic application to be applied. This network is trained on existing human proofs. At inference time, a best-first search is performed using the neural network as an action generator.

Table 1: Proof rates on MiniF2F Benchmark

Model	valid	test
PACT [2]	23.9%	24.6%
FMSCL [5]	33.6%	29.6%
LISA [4]	28.3%	29.9%
LISA + AF	**36.1%**	**34.0%**

We use Codex [1] auto-formalize 3908 mathematical problems belonging to category `algebra`, `intermediate algebra`, and `number theory` from the training set of MATH [3]. Out of them, 3363 of the auto-formalized theorems are syntactically correct. We then use our neural prover trained on Isabelle corpus (AFP and Isabelle Standard library) to prove these theorems, and 23.3% of them can be proven. This gives us 782 new provably verified theorems along with their proofs for us to train our neural prover further. This form of training on one's own generated data is known as expert iteration, and was already used in prior works [6, 5]. However, unlike in

Polu et. al. [5], where one perform expert iteration on a set of problems manually translated by human, we here use LLMs to auto-formalize the theorems.

After one epoch of training on the proofs of 782 theorems, we evaluated the neural prover on miniF2F [10], a recently introduced benchmark containing 488 mathematical competition statements manually formalized by humans. Some of those problems come from the valid and test set of MATH, and others come from previous International Mathematical Olympiad competitions or AoPS[1].

The results are shown in Table 1. LISA refers to the model before we trained on the autoformalized dataset, and LISA + AF refers to the model after one epoch of training on the 782 theorems. We see that by simply training on one epoch of the proved auto-formalized theorems, we can achieve a significant improvement in proof rate (from 28.3% to 36.1% on miniF2F-valid), and a new state-of-the-art performance on this benchmark.

4 Conclusion

For the first time, we have demonstrated that autoformalization is indeed feasible at least for high school mathematics competition problems and the translated results are useful for improving the performance of neural theorem provers.

However, our method is not capable of creating whole theories or autoformalization of facts that need to rely on libraries the language model has not been trained on. Full blown autoformalization of mathematical text will require new methods, especially proper training methodologies and utilizing newly introduced code by retrieval augmented language modeling.

References

[1] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

[2] Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward Ayers, and Stanislas Polu. Proof artifact co-training for theorem proving with language models. In *International Conference on Learning Representations*, 2022.

[3] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the MATH dataset. *CoRR*, abs/2103.03874, 2021.

[4] Albert Qiaochu Jiang, Wenda Li, Jesse Michael Han, and Yuhuai Wu. Lisa: Language models of isabelle proofs. *6th Conference on Artificial Intelligence and Theorem Proving*, 2021.

[5] Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. Formal mathematics statement curriculum learning, 2022.

[1] https://artofproblemsolving.com/

[6] Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. *CoRR*, abs/2009.03393, 2020.

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

[8] Qingxiang Wang, Chad Brown, Cezary Kaliszyk, and Josef Urban. Exploration of neural machine translation in autoformalization of mathematics in mizar. In *International Conference on Certified Programs and Proofs*, 2020.

[9] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle framework. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 33–38. Springer, 2008.

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

A Prompt

Natural language version: *"Let* $z = \frac{1+i}{\sqrt{2}}$*, find* $(\sum_{i=1}^{1} 2(z^{i^2})) \cdot (\sum_{i=1}^{1} 2(\frac{1}{z^{i^2}}))$*. The final answer is 36."*
Translate the natural language version to an Isabelle version:

```
theorem
  fixes z::complex
  assumes h0: "z = (Complex (1/sqrt 2) (1/sqrt 2))"
  shows "(∑k::nat=1..12. (z^(k^2)))
           * (∑ k::nat=1..12. 1/(z^(k^2))) =36"
```

Natural language version: *"Determine the value of ab if* $\log_8 a + \log_4 b^2 = 5$ *and* $\log_8 b + \log_4 a^2 = 7$*. The final answer is 512"*. Translate the natural language version to an Isabelle version:

```
theorem
  fixes a b ::real
  assumes "(ln a) / (ln 8) + (ln (b^2)) / (ln 4) = 5"
          "(ln b) / (ln 8) + (ln (a^2)) / (ln 4) = 7"
  shows "a * b = 512"
```

Tactic Characterizations by the Influences on Proof States *

Liao Zhang[1,3] and Lasse Blaauwbroek[2]

[1] Czech Technical University, Prague, Czech Republic
[2] Institut des Hautes Études Scientifiques, France
[3] University of Innsbruck, Austria

1 Introduction

When formalizing mathematics in an interactive theorem prover, such as the Coq [1] proof assistant, it is necessary to have an intuition on how the available proof actions change the proof state. In particular users may have an idea that to transform the current proof state to a different one, a particular tactic might be the right one to use.

In this paper, we regard the changes to a proof state made by the tactic application as the semantic of that tactic. The purpose of our study is to predict the tactic based on its semantic. Assume there is a triple $(ps, t, \{ps'\}_{1..n})$, where ps, t, $\{ps'\}_{1..n}$ are a Coq state, the tactic applied to ps by a Coq user, and the after states caused by the tactic application, respectively. We aim at building a machine learning model to predict a tactic t' such that ps transforms to $\{ps'\}_{1..n}$ by the application of t'. To ensure that t and t' lead to the same after states, we run t' in Coq and compare with t.

There are several motivations behind our project. First, the task can be directly applied for tactic suggestion given a human's intuition for the next state. For a Coq beginner, it is quite common that he can imagine the next state but cannot determine how to select a suitable tactic to reach the goal. However, understanding Coq's manual may be challenging for beginners. If he can copy the before state from the Coq editor, convert it to the imaginary after state, and input them into our system, we will be able to automatically suggest the tactics with the expected behavior. Meanwhile, a medium-level Coq programmer may want to discover a single tactic to substitute an awkward tactic sequence. Even for an expert, when he encounters an unfamiliar domain, he needs our system to advise likely helpful tactics.

Second, the task serves as an initial step to a new formal verification strategy. When a mathematician tries to prove a theorem, he first thinks of several intermediate goals and then fills the gaps by order. However, nowadays proof assistants cannot skip tactics between intermediate goals. We can extend our system to predict a tactic sequence from one state to another. Afterwards, the human expert can merely specify the states that he thinks are important to complete the proof and ask our system to erase the gaps.

Finally, since we encounter our own challenges in precisely characterizing the transition between before and after states, the approaches developed by us can also be applied to other machine learning domains. Take fault detection [3] for instance, we can apply our differential techniques to the images before and after the fault occurs. Then, the results can be input into a learning model to predict the category of fault.

*This work was supported by the ERC grant no. 714034 *SMART* and by the European Regional Development Fund under the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466).

2 Tactic Characterizations

We characterize the semantic of tactics as features and Coq strings as the input for random forests [7] and GPT-2 [5], respectively. The feature extraction techniques on Coq terms are the same as our previous work [7]. Large-scale pretrained transformers such as GPT-2 have achieved significant progress in various domains [2]. We evaluate GPT-2 and random forests to make a comparison.

We consider three feature extraction approaches. The first approach computes the differences between the state features of ps and $\{ps'\}_{1..n}$. From ps, we extract a set of features F. We also extract n sets of features $\{F\}_{1..n}$ from $\{ps'\}_{1..n}$. If a feature f exists in F but does not in any F_i, we regard it as a disappeared feature. Conversely, if there is an F_i with a feature f that is not in F, then f is an appearing feature. The tactic characterization is the union of all disappeared and appearing features.

Second, we extract features from the newly defined existential variables in proof terms. In Coq, we write tactics to construct a proof script to prove a theorem. Actually, the tactics help to complete a proof term. The relationship between proofs and proof terms is based on the Curry-Howard correspondence [6]. An incomplete proof term may contain several existential variables. Some are defined, and others are undefined as holes. A tactic fills some holes with Coq terms and may generate several new holes. A proof term is completed once all the holes have been filled. We obtain the features from the terms defined in the holes by the tactic as its characterization.

Finally, we perform first-order anti-unification [4] on the before and after states to find the substitutions. A term g of two terms t_1 and t_2 is called a *generalization* if there are substitutions σ_1 and σ_2 such that $\sigma_1 g = t_1$ and $\sigma_2 g = t_2$. Anti-unification aims to find the *least general generalization* lgg such that for any generalization g' of $t1$ and t_2, there exists a substitution σ that makes $\sigma g' = lgg$. We extract the features from the Coq terms present in the substitutions(σ_1 and σ_2) and the lgg as the input to our model.

For GPT-2, we merely apply anti-unification to generate strings. We convert the lgg and substitutions to strings and input them into the model.

3 Experimental Evaluation

Our dataset is composed of the proof states $(158, 494)$ of all the lemmas $(11, 372)$ in the Coq's standard library. The lemmas were randomly divided into three subsets for training, validation, and testing in an 80-10-10 ratio. Each subset includes the states of the corresponding lemmas. For random forests, we optimize parameters on the training and validation partitions, which is depicted in Figure 1. Afterwards, we build models with the best hyper-parameters learned from the training dataset and make predictions for the test dataset. We also fine-tune the smallest GPT-2 for each characterization. Every model is executed for 25 epochs, and we store the snapshot with the best accuracy on the validation dataset to synthesize tactics for the test data. All the GPT models utilize the same parameters: a batch size of 32, no weight decay, and the learning rate of 0.0003 with a linear schedule and the first 20% steps for warming up. Figure 2 depicts the average training loss per step and validation accuracy during fine-tuning.

Table 1 shows the results on the test data. Unsurprisingly, only learning from before states performs worst since it contains little information of the influences of the tactic. The best accuracy achieved by GPT-2 is 10.47% better than that of random forests. This confirms the power of the state-of-the-art neural network. Anti-unification does not work well for random

Figure 1: Results of hyper-parameter tuning for random forests. The accuracy denotes how often we predict a tactic that is the same as the tactic in the dataset.

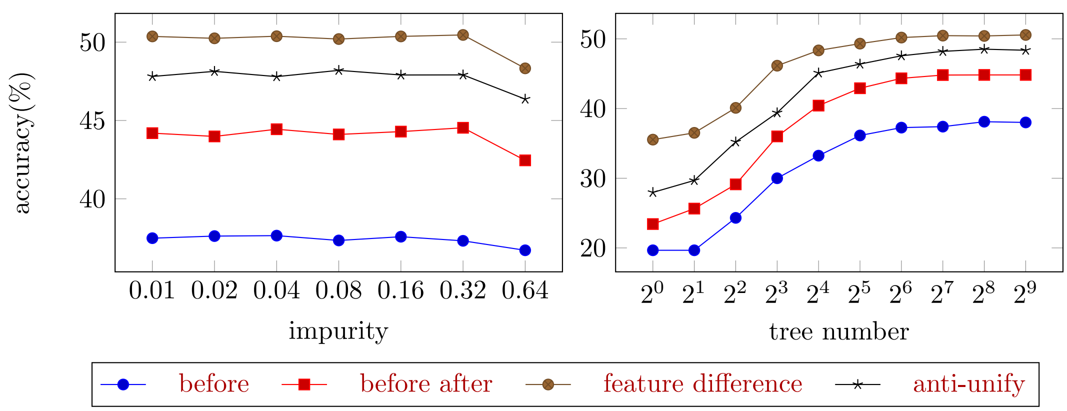
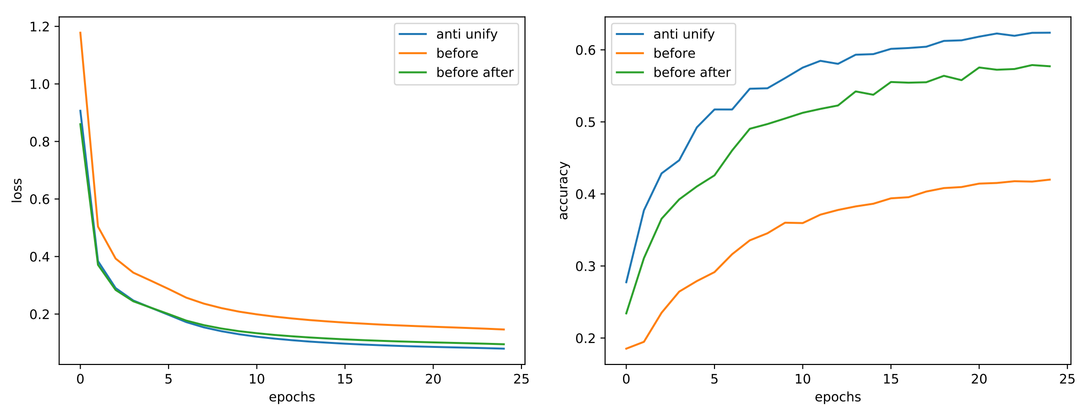


Figure 2: Training loss and validation accuracy of GPT-2.



(a) training loss (b) validation accuracy

Table 1: Results on the test dataset. “Same tactic” denotes that the prediction is exactly the same as the tactic in the library. “Same change” checks how often the prediction makes the same transformation.

model	accuracy(%)	before	before after	feature difference	proof term	anti unification
random forests	same tactic	36.917	44.563	49.723	47.480	47.727
	same change	43.225	52.166	59.344	56.024	55.507
GPT-2	same tactic	39.154	56.215			60.300
	same change	45.356	65.319			69.814

forests but obtains excellent performance for GPT-2. The reason may be that converting anti-unification to appropriate features is challenging.

References

[1] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Yann Coscoy, David Delahaye, Daniel de Rauglaudre, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, et al. The coq proof assistant reference manual. *INRIA, version*, 6(11), 1999.

[2] Xu Han, Zhengyan Zhang, Ning Ding, Yuxian Gu, Xiao Liu, Yuqi Huo, Jiezhong Qiu, Yuan Yao, Ao Zhang, Liang Zhang, et al. Pre-trained models: Past, present and future. *AI Open*, 2:225–250, 2021.

[3] Dubravko Miljković. Fault detection methods: A literature survey. In *2011 Proceedings of the 34th international convention MIPRO*, pages 750–755. IEEE, 2011.

[4] Gordon D Plotkin. A note on inductive generalization. *Machine intelligence*, 5:153–163, 1970.

[5] 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.

[6] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism*. Elsevier, 2006.

[7] Liao Zhang, Lasse Blaauwbroek, Bartosz Piotrowski, Prokop Černỳ, Cezary Kaliszyk, and Josef Urban. Online machine learning techniques for coq: A comparison. In *International Conference on Intelligent Computer Mathematics*, pages 67–83. Springer, 2021.