Context-aware Generation of Proof Scripts for Theorem Proving

Chuanhu Cheng*, Yan Xiong*, Wenchao Huang*, Lu Ma†
*School of Computer Science and Technology
University of Science and Technology of China, Hefei, Anhui, PR China
†Beijing Institute of Remote Sensing, Beijing 122000, China
Email: chengch@mail.ustc.edu.cn, {yxiong, huangwc}@ustc.edu.cn, syml1234@163.com

***Abstract*—Formal verification is a trustable method to produce correct, safe, and fast code. However, the cost of formal verification remains prohibitively high for most projects, as it requires significant manual effort by highly trained experts. In this paper, we propose a novel approach to proof automation in Coq that generates proof script based on context-awareness. We develop AutoMagic, an automatic theorem proving framework, which can use the generated proof script to achieve a fully automatic proof of the theorem. Our method is simple but pushes the limits of automatic proof. The performance of AutoMagic is evaluated in the Coq standard library. We show that 37.87% of the theorems can be proved in a push-button mode, and can be used to prove new theorems not previously provable by automated methods.**

***Index Terms*—Theorem proving, Context-aware, Proof script generation, Proof automation**

I. Introduction

Security verification of software systems, such as operating system kernels, has long been recognized as an important and extremely challenging task. On the one hand, software vulnerabilities can cause significant financial losses. On the other hand, verification of security properties is difficult since the specifications need to be written in expressive logic languages and theorem proving problem for such languages is usually undecidable. Over the past two decades, interactive theorem proving (ITP), such as Coq [4] or Isabelle [19], has been successfully applied to the security properties verification of complex software systems. Among those verified software systems, the most well-known ones are probably the seL4 microkernel [13], using Isabelle, and the CompCert C compiler [16], using Coq.

"Theorem proving" in terms of program verification simply refers to writing down intended properties as propositions and then proving them as one does a mathematical theorem-usually with the aid of an ITP. The ITP systems used in these validations rely on higher-order and type systems to maximize expressivens and generality, while also facilitating modularity and reuse of proofs. However, despite the rich expressiveness of these theorem provers, the process of performing proofs in a proof assistant is extremely laborious and costly. For instance, it took the seL4 team more than 20 person years and 200,000 lines of Isabelle scripts to verify a microkernel consisting of around 8,000 lines of C code.

Due to complicated behaviors of the software system, it is extremely difficult to achieve fully automated verification of functional correctness and there is currently no work to achieve this. Relying on humans in the loop, ITP executes those proofs interactively via the use of proof scripts that consists of tactics as well as items from goals and context. To complete a proof, a programmer must provide guidance to the proof assistant at each step by picking which proof scripts to apply, and if the machine checks pass, it will generate a new subgoal and context, repeat the process until no new subgoals are generated. Indeed, some recent efforts have attempted to support automated in ITPs, but these tools suffer from two limitations. One is to provide automatic tactics that require manual input and can only solve a specific goal, which cannot fully automatically prove one theorem or multiple theorems in succession [5] [3] [6]. The other is to use machine learning model to predict the proof script, but this learning model lacks flexibility. A tactic argument can be a sophisticated line of code with variables, functions, and compound expressions, and the space of possible arguments is infinite. Prior work has limited flexibility because they generate proof scripts by copying from a fixed, predetermined set [9] [1], and arguments are only extracted from the local context, excluding the global context [23] [11].

In this paper, we propose a method for dynamically generating proof scripts, which analyzes the context, extracts the items related to the proof, and then combine the tactics to generate a set of proof scripts. And we implement AutoMagic, a general and extensible framework of automatic theorem proof in Coq proof assistant, which provides a structured Python representation of Coq proofs, including all proof states encountered in a proof, the steps and the recording of previous proofs. AutoMagic enables lightweight interaction with Coq so that it can be used to build proofs dynamically.

Experimental results show that AutoMagic can generate effective tactics. It can successfully prove 37.87% of the theorems in the Coq standard library, significantly outperforming CoqHammer [6]. It's simple enough, but it solves more problems than existing tools. The advantage of AutoMagic is that it is a general system, not depending on any domain-specific knowledge.

Our main contributions are:

(1) We propose a context-aware technique for generating proof scripts.

978-1-7281-8275-9/20/$31.00 ©2020 IEEE
DOI 10.1109/BigCom51056.2020.00018

```
Inductive bool : Type :=                    (*define the type [bool] of boolean*)
  | true : bool
  | false : bool.

Definition andb (b1:bool) (b2:bool) : bool :=      (*define the and operation*)
  match b1 with
  | true => b2
  | false => false
  end.

Theorem andb_true_elim : forall b c : bool, andb b c = true -> b = true.
Proof.
  intros b c H; induction b.
  Case "b = true". (* <----- here *)
    reflexivity.
  Case "b = false". (* <---- and here *)
    rewrite <- H.
    reflexivity.
Qed.
```

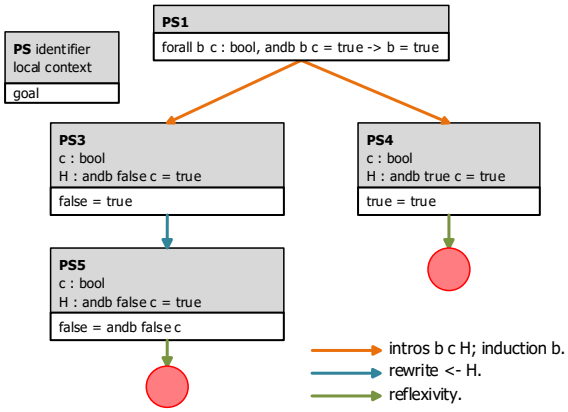

Fig. 1. A proof script in Coq (left) and the resulting proof states, proof steps, and the complete proof tree (right).

(2) We have implemented an automatic proof framework that can use the generated proof script to achieve fully automated proof of the theorem.

II. Related Work

A lot of work has been done on the automatic proof of ITPs. We can summarize it into three categories according to the different methods used.

A. Automatic proof tactic

Automatic proof tactic is actually a semi-automatic proof method, which is defined by Latc [7], Coq's tactic language. Such tactics are based on pattern matching and can automatically solve specific goals, but requires manual interaction to complete the proof. For example, the tactic *omega*, is an automatic decision procedure for Presburger arithmetic. Coq has built-in automatic tactics such as $auto$, $fourier$, $field$. There are also automation tactics defined by experts to solve problems in specific fields, such as [3], [6], [18].

B. Hammer for ITP

Hammers are proof assistant tools that employ external automated theorem provers (ATPs) [15] to find proofs of user given conjectures automatically. For example, CoqHammer [6] translates theorems in the Coq proof assistant to first-order logic. It then proves the theorems using external provers and converts the proof back to Coq's tactics. Most developed hammers exist for proof assistants based on higher-order logic (Sledgehammer [20] for Isabelle, CoqHammer for Coq, HOLyHammer [12] for HOL Light [10] and HOL4 [21]). The hammer-based approach essentially bypasses the proof assistant and outsources the work to external ATPs. In contrast, our method of generating tactic is simper, using native tactics without hammers, but it can solve more problems.

C. Automatically Generating Proofs with Machine Learning

Human experts have written a large amount of ITP code, which provides an opportunity to develop machine learning systems to imitate humans for interacting with the proof assistant. ML4PG [14], Gamepad [11], and CoqGym [23], all introduce benchmark suites and frameworks for exploring machine learning in Coq. ML4PG, while it introduces machinery which should allow it to generate proofs, focuses instead on clustering proofs, and does not attempt to generate proofs. Gamepad builds a machine learning environment for theorem proving, considering the problem of predicting proofs, but only predicting tactics and arguments, without synthesizing a complete proof script. CoqGym attempts to model proofs with a fully general proof script and term model expressing arbitrary AST's, but it can only synthesize a simple proof script and the arguments of tactic come only from the local context.

III. Background on Coq

Coq is a proof assistant system that has been developed at INRIA (Paris, France), since 1984. Coq is based on Calculus of Inductive Construction, an implementation of intuitionistic logic which uses inductive and dependent types. Nonetheless. Coq's logic may be easily extended to classical logic by assuming the excluded middle axiom. A key feature of Coq is the capability of extraction of the verified program (in OCaml or Haskell) from the constructive proof of its formal specification [17]. This facilitates using Coq as a tool for software verification.

Coq allows the user to define proof goals alongside data and program definitions. The process of theorem proving is interactively entering a sequence of scripts called tactics that manipulate the proof context. The proof context is a $proof\ state$ which is comprised of goal (expressed in the term language) stating what we need to prove and a $context$ containing the assumptions. The user starts with the theorem itself as an initial $goal$ and repeatedly apply tactics to decompose the goal into a list of subgoals. The proof is complete when there are no subgoals left. For example, we can define the $bool$ type and the $andb$ operation in Figure 1 (left). Then we state the theorem $forall\ b\ c\ :\ bool,\ andb\ b\ c\ =\ true\ \rightarrow\ b\ =\ true.$, using $bool$ and $andb$. As in Figure 1 (left), theorem

proving starts with $Proof.$ token, end with $Qed.$ token, and between them is a sequence of tactics. A successful Coq proof implicitly generates a *proof tree*, where the root is the start state and all the leaves are final states. All goals share the same environment but have a unique *local context* with premise local to each goal, such as *PS3 hypothesis* H in Figure 1 (right).

The edge of the proof tree are proof scripts. We use the code $intros\ b\ c\ H;\ induction\ b.$ to start a proof by introducing variables and hypotheses, and use code $induction\ b.$ to generate two subgoals (on b). The expressions $intros$ and $induction$ are tactics, the semicolon ; sequences two tactics, and the period . signals the end of one proof step. After we perform the induction, we see that proof state 2 is decomposed into two cases, one is case $b = false$, the other is case $b = true$. Tactic $rewrite \rightarrow H.$ rewrite the left side of premise equation H to the right side of the current goal equation, and $reflexivity.$ checks that both sides of the equation are convertible and then solve the goal. Proofs in Coq are finished when the goal is trivially true, given the context(e.g., $true = true$).

Coq represents a $proof\ state\ PS$ as a tuple $\langle \Gamma,\ \mathcal{G},\ n \rangle$ of a local context Γ, a goal term $\mathcal{G}$ and a unique proof state identifier n. A *context* Γ provides a list of identifiers and their types that expresses all the current assumptions. For instance, the proof state with identifier 3 has $\Gamma = c : bool,\ H : andb\ false\ c = true$. A tactic invocation creates edges between the appropriate proof states, which form a *proof tree*.

Theorem proving requires manual interaction with the proof assistant for completing the proof, which requires the user to spend an important part of the proof effort on goals. However, ITP automation techniques are able to reduce this effort significantly. Theorem proving in Coq resembles a task-oriented dialog [2]. AutoMagic can automatically interact with the proof assistant, simply push a button, and the proof comes out. At each step, the agent perceives the current goals, their local context, and the global environment; it then generates set of a proof script, trying each proof script to decompose the current goal.

IV. Overview

In this section, we'll present AutoMagic's proof script generation and search process. We can see the toplevel structure of AutoMagic in Figure 2. Each of the modules in written in Python, with the Coq interface as a Python interface, on the top of Coq serapi on top of Coq [8].

The input to AutoMagic is a $.v$ suffix file, which contains the formal definition of the verified program and the theorem that needs to be proven. Given a theorem to prove, the tool will call the Coq interface for machine checks to get an initial $proof\ state$. The proof state of Coq output is divided into three cases:

Case 1: *The proof state is not null and is a new proof state.* Theorem proving can be regarded as a dynamic generation process of a search tree. Context-Aware generation of proof

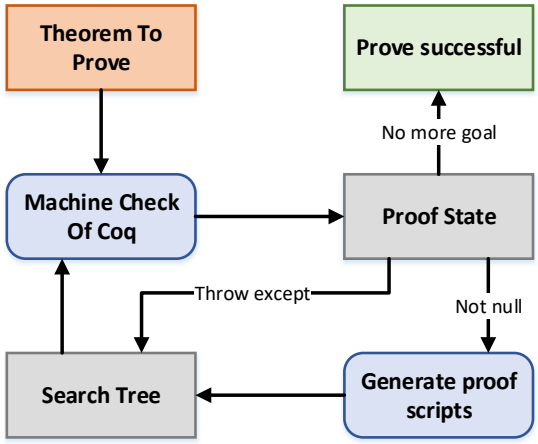

Fig. 2. The overall architecture of AutoMagic.

scripts is based on the assumptions and goals of the proof state. Each proof state generates a set of proof scripts, which are then added to the search tree. Try each node in a depth-first search until a new proof state is generated. Repeating this process, we can find a complete proof path, which is the successful proof of this theorem.

Case 2: *The machine check of Coq throws an exception.* If the generated proof script cannot pass the machine check, Coq will throw an exception. In this case, the agent needs to backtrack the search tree and try other nodes.

Case 3: *The proof state is null, no more goals.* With no more goals for the proof state, the proof is completed and the proof path is successfully found.

V. Context-Aware generation of proof script

The generation of the proof script is more complicated. The proof script consists of tactics and items, where items are arguments of the tactic. Coq terms can be identifiers, variables, constants, functions, or expressions. It is very difficult to construct an expression in the proof script and a topic of research in itself—we do not address it in this paper. We propose a novel mechanism for automatic generation of proof scripts, which can be context-aware, extract useful items, and then combine tactics to synthesize proof scripts.

A. Space of tactics

Coq comes with a set of builtin tactics, such as $intros$, $induction$, $rewrite$, and so on, it has more than two hundred. Statistics of AutoMagic show that many valid tactics are seldom used in proofs. Therefore we simplify the tactic grammar to facilitate the generation of proof scripts at the expense of giving up on some cases. We only generate atomic tactics while excluding compound ones such as "$tac1;\ tac2.$". This is not a severe handicap because all proofs can be completed without compound tactics. We also exclude user-defined tactics.

B. Space of arguments

The arguments could be (1) a term from the local context,which is the proof state, (2) a term form the global context (e.g., a lemma), or (3) a term created by the human user. As arguments in the third category can be any Coq term, the space of arguments is potentially infinite. For our generation method, we focus on arguments in the first two categories. The arguments of the global context are a set of lemmas that have been proved, which is a premise selection problem [22]. We use premise selection of CoqHammer [6] to identify a fraction of the available lemma as potentially relevant to discharge the current interactive goal. The CoqHammer's $predict\ n.$ command can search up to the top n lemmas most relevant to the current goal. Then we can synthesize the following two proof scripts:

$$try\ \ apply\ lemma_1\ \|\ \cdots\ \|\ apply\ lemma_n. \tag{1}$$

$$try\ \ rewrite\ lemma_1\ \|\ \cdots\ \|\ rewrite\ lemma_n. \tag{2}$$

C. Proof script synthesis

Synthesizing proof script is challenging because of the syntactic output space is larger: all valid identifiers in Coq. However, there are strong semantic constraints on the arguments. For example, the tactic "$apply\ H$." applies a premise H to the goal. The argument H must be a valid premise either in the environment or in the local context.

To leverage the semantic constraints in synthesizing arguments and the logic of the use of tactics, we group tactics into categories and take different actions for each category.

- Tactics for handling logical connectives: For the logical connectives of the goal formula, Coq can use the introduction and elimination rules to decompose the goal. As in TABLE I, there are corresponding tactics to deal with the logical connectives in goals and assumptions.
- Coq built-in and user-defined automatic tactics: The category of tactics does not require arguments, but can automatically solve some specific goals. we use coq's built-in tactic, such $omega$, $firstorder$, $ring$. In addition, we also use automatic tactics defined by CoqHammer, such as $sauto, ycrush$.
- Other tactics with arguments: Such tactics need to extract items from the local context and environment.

TABLE I
TACTICS FOR LOGICAL CONNECTIVES

	$\Rightarrow$	$\forall$	$\wedge$	$\vee$
Hypothesis	***apply***	***apply***	***elim or destruct***	***elim or destruct***
goal	***intros***	***intros***	***split***	***left or right***
	$\sim$	$=$	$\exists$	*False*
Hypothesis	***elim***	***rewrite***	***elim or destruct***	***elim or case***
goal	***intros***	***reflexivity***	***exists*** ν	

To facilitate the description of the generation of the proof script. We start with a set of definitions that will be used throughout. A tactic $\tau \in \mathcal{T}$ is a tactic name. An argument $\alpha \in \mathcal{A}$. We use $\mathcal{I}$ for the set of Coq $Inductive$, $\mathcal{D}$ for the set of Coq $Definition\ and\ Fixpoint$, $\mathcal{S}$ for the $proof\ state$.

Given a theorem to proof, the workflow of proof script synthesis consists of the following steps:

- Step 1: We initialize a list P_{tac} to represent the set of proof script synthesis. Use the tokenizer to tokenize the goal and hypothesis of proof state, get valid tokens, and then search for all tokens defined by $Inductive$ and $Definition$.

  $$\iota,\ \delta\ =\ Search[Tokenizer(\mathcal{S}.\mathcal{G},\ \mathcal{S}.\mathcal{H}),\ \mathcal{I},\ \mathcal{D}] \tag{3}$$

  ι is a subset of variables that $Inductive$ defines data types ($\mathcal{I}$). δ is a subset of $\mathcal{D}$. For each element in ι, δ, we can generate the following two kinds of proof scripts. Tactic $induction$ is to decompose the goal according to the constructor. Tactic $unfold$ is to unfold $Definition$. Then return P_{tac}.

  $$P_{tac}.append(\text{“}\textit{induction}\ \iota_1; \cdots;\ \textit{induction}\ \iota_n\text{.”}) \tag{4}$$

  $$P_{tac}.append(\text{“}\textit{unfold}\ \delta_1; \cdots;\ \textit{unfold}\ \delta_n\ in\ *\text{.”}) \tag{5}$$

- Step 2: According to TABLE I, we generate corresponding tactics based on the target and hypothetical logical connectives. Then return P_{tac}.
- Step 3: We match the similarity of each hypothesis to the goal and then generate the following proof scripts based on the matching degree. H is the hypothetical identifier.

  $$P_{tac}.append(\text{“}\textit{apply}\ H\text{.”}) \tag{6}$$

  $$P_{tac}.append(\text{“}\textit{rewrite}\ H\text{.”}) \tag{7}$$

- Step 4: We add built-in and user-defined automation tactics to P_{tac}. The middle lemma is very helpful in proving that it can simplify or even directly solve a goal. Therefore, we also add the proof scripts (1) (2) to the list.
- Step 5: Return P_{tac}.

VI. GENERATION-GUIDED SEARCH

Each proof state will generate a list of proof scripts, from the initial state to the end of the proof, which is a dynamic search tree generation process. We generate proof scripts considering a more general situation, not a large number of proof scripts generated by random methods. According to our generated proof script, the maximum width of the search tree is 15, so we directly use depth-first search to explore the right proof path.

VII. EVALUATON

We evaluate AutoMagic on the task of fully-automated theorem proving in Coq, using the standard library of Coq version 8.9.1. The tool perceives the current goals, their local context, and the environment. It interacts with Coq by executing commands, which include tactics, backtracking to the previous step, and any other valid Coq command. We run

all experiments on machines with 8GB RAM and one Intel i7-4790 CPU.

We compare the performance of our tool with CoqHammer. The baseline is *hammer* [6]—a hammer-based system that proves theorems using external ATP systems. In our particular configuration, *hammer* simultaneously invokes Z3, CVC4, Vampire, and E prover, and returns a proof as long as one of them succeeds. We treat *hammer* as a block box tactic, it sets a default time limit of 30 seconds to the external ATP systems. We test *hammer* both in this setting and in a setting where we extend the time limit to 10 minutes.

Coq standard library is general-purpose libraries with definitions and theorems for sets, lists, sorting, arithmetic, etc. We have tested 3375 theorems from 9 subdirectories of the standard library. AutoMagic solves 37.87% (1278/3375) of the proofs in our test set.

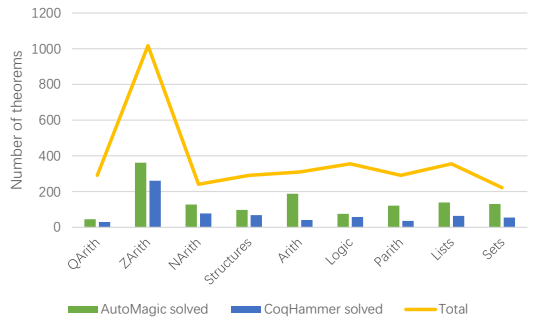

Fig. 3. A comparison of AutoMagic and CoqHammer abilities to complete proofs.

Figure 3 shows the proofs solved by AutoMagic and CoqHammer. The polyline represents the total number of theorems contained in each subdirectory. Our tool significantly outperforms CoqHammer. It's important to realize that, CoqHammer only solve 691 of the proofs in the same test set, but we can solve 1278 proofs, which is a 1.8X improvement over CoqHammer. This demonstrates that our system can generate effective proof scripts and can be used to prove theorems previously not provable by automatic methods.

VIII. Conclusion

In this paper, we have studied automated proof of theorems. We propose a context-aware method for theorem proving script generation. Experimental results on AutoMagic confirm the effectiveness of our method for synthesizing complete proofs automatically.

IX. Acknowledgements

The research is supported by the National Key R&D Program of China 2018YFB0803400, 2018YFB2100300, National Natural Science Foundation of China under Grant No.61972369, No.61572453, No.61520106007, No.61572454, and the Fundamental Research Funds for the Central Universities, No. WK2150110009.

References

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

[2] Antoine Bordes, Y Lan Boureau, and Jason Weston. Learning end-to-end goal-oriented dialog.

[3] Jingyuan Cao, Ming Fu, and Xinyu Feng. Practical tactics for verifying c programs in coq. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, pages 97–108. ACM, 2015.

[4] The coq development team. The coq proof assistant. http://coq.inria.fr/.

[5] P Crégut. Omega: a solver of quantifier-free problems in presburger arithmetic. *The Coq Proof Assistant Reference Manual, Version*, 8(0), 2017.

[6] Łukasz Czajka and Cezary Kaliszyk. Hammer for coq: Automation for dependent type theory. *Journal of automated reasoning*, 61(1-4):423–453, 2018.

[7] David Delahaye. A tactic language for the system coq. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 85–95. Springer, 2000.

[8] Emilio Jesús Gallego Arias. SerAPI: Machine-Friendly, Data-Centric Serialization for Coq. Technical report, MINES ParisTech, October 2016.

[9] Thomas Gransden, Neil Walkinshaw, and Rajeev Raman. Sepia: search for proofs using inferred automata. In *International Conference on Automated Deduction*, pages 246–255. Springer, 2015.

[10] John Harrison. Hol light: An overview. In *International Conference on Theorem Proving in Higher Order Logics*, pages 60–66. Springer, 2009.

[11] Daniel Huang, Prafulla Dhariwal, Dawn Song, and Ilya Sutskever. Gamepad: A learning environment for theorem proving. *arXiv preprint arXiv:1806.00608*, 2018.

[12] Cezary Kaliszyk and Josef Urban. Learning-assisted automated reasoning with flyspeck. *Journal of Automated Reasoning*, 53(2):173–213, 2014.

[13] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[14] Ekaterina Komendantskaya, Jónathan Heras, and Gudmund Grov. Machine learning in proof general: Interfacing interfaces. *arXiv preprint arXiv:1212.3618*, 2012.

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

[16] Xavier Leroy. Formal verification of a realistic compiler.

[17] Pierre Letouzey. Extraction in coq: an overview. In *Proceedings of the 4th conference on Computability in Europe: Logic and Theory of Algorithms*, 2008.

[18] Andrew McCreight. Practical tactics for separation logic. In *International Conference on Theorem Proving in Higher Order Logics*, pages 343–358. Springer, 2009.

[19] Lawrence C Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer Science & Business Media, 1994.

[20] Lawrence C Paulsson and Jasmin C Blanchette. Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. In *Proceedings of the 8th International Workshop on the Implementation of Logics (IWIL-2010), Yogyakarta, Indonesia. EPiC*, volume 2, 2012.

[21] Konrad Slind and Michael Norrish. A brief overview of hol4. In *International Conference on Theorem Proving in Higher Order Logics*, pages 28–32. Springer, 2008.

[22] 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*, pages 2786–2796, 2017.

[23] Kaiyu Yang and Jia Deng. Learning to prove theorems via interacting with proof assistants. *arXiv preprint arXiv:1905.09381*, 2019.