Autogenerating Natural Language Proofs for Proof Education

Seth Poulsen[1], Matthew West[1], and Talia Ringer[1]

University of Illinois at Urbana-Champaign, Urbana, IL, USA
sethp3@illinois.edu

Introduction Understanding mathematical proofs is critical for students learning the foundations of computing. Having students construct mathematical proofs with the help of a computer is appealing as it makes it easier to autograde student work and autogenerate practice problems for students. Most existing tools for students to construct proofs with a computer are restricted systems that only permit simple logics, or there is a large overhead involved in students learning how to use them [7]. Proof Blocks (proofblocks.org), a tool that allows students to drag-and-drop prewritten lines of a proof into the correct order is a nice compromise because the tool is easy for students to use to construct proofs on any topic [7]. However, a downside is that the process of writing questions can be time consuming and error-prone. An instructor must write the proof, break it into lines, and then specify a directed acyclic graph giving the logical dependence between lines of the proof. In this paper, we document the first step toward building a system to automatically generate Proof Blocks problems from Coq proofs: a Coq plugin which generates a natural language proof from a Coq proof. Our natural language generator differs from similar tools in that we deliberately restrict which definitions and tactics are allowed in the interest of improving readability of the output.

Prior Work Early work in natural language proof generation simply took the low-level representation used by the proof checker, and converted it to English [3]. The proofs generated by such systems, while being valid natural language proofs, are extremely verbose and awkward, not at all like what a human would write. Subsequent attempts improved readability by translating the low-level representation to a higher level of abstraction before converting to English. For example multiple variable declarations in a row, denoted in the low-level representation by a series of λ-abstractions in the calculus of constructions, would be rewritten into a single one so that the variables could be declared in parallel in the natural language proof [2]. Other researchers have obtained even better results by using tactic-level information in addition to term-level information [1, 5]. Ganesalingam and Gowers take a completely different approach to the problem: rather than start with an existing theorem prover and translate its representation of the proof to natural language, they created a new theorem prover from the ground up, which only uses tactics similar to what a human would do [4]. From our experience, the readability of this generator surpasses all of the rest. Alectryon is a set of tools for interleaving Coq proof scripts with information about the proofs [6]. Its focus is on improving readability of Coq sources, while we focus on abstracting away from Coq sources entirely.

Challenges in Generating Natural Language For our application domain we want the generated proofs to look human written and to be clear and understandable to complete beginners. We encountered a number of other difficulties in trying to get simple natural language proofs from formalized proofs, which are not documented in the literature.

Decision Procedures Hiding Behind Tactics Proof engineers make extensive use of decision procedures that are built into the proof assistant's tactic language—like Coq's `ring` or the Isabelle/HOL `sledgehammer`. It is unclear how to translate proofs constructed with tactics like these to understandable natural language. If you remain at the tactic level, there is so much information lost that the steps may not be clear, but dropping to the term level gives far more detail than what a human would naturally write.

Use of non-standard definitions Formalized proofs often build on different logical foundations and different definitions than the way the same proofs would typically be presented to beginners. For example, when writing proofs about divisibility, most instructors use the standard definition: a divides b if $\exists c \in \mathbb{Z}$ such that $ac = b$, but libraries of formalized mathematics often use a less elementary definition, such as defining divides in terms of the modulus operation, making it difficult to generate simple, human understandable proofs about divisibility. Another example is the foundational object of most theorem provers being types rather than sets, which makes it difficult to write simple proofs about set theory facts.

Excessive proof term manipulation When writing proofs with a proof assistant, proof engineers often use tactics that manipulate proof state in a way unlike people do in pencil and paper proofs. For example,

a common paradigm used by those using the `ssreflect` library for Coq is to use the `=>` and `:` operators to introduce and then re-generalize a variable, editing the proof state in between (this is done extensively in the mathematical components library [8]). This adds extra complexity into the proof term that again wouldn't be present if a human was writing the proof. It is an open question whether or not there is a way to hide these operations when translating to natural language.

Our Approach We present a novel hybrid approach—use a well-established theorem prover, to enable code reuse and access to rich feature set, but restrict the set of tactics which are allowed to be used in order to ensure that the proofs can be translated into understandable natural language. Building our generator as a plugin also allows full access to the proof state. We will start with the same tactic set proposed by Ganesalingam and Gowers, which can be mapped on to standard Coq tactics, and plan on adding more in the future. Our natural language generator is a Coq plugin which adds new tactics `PreExplain` and `PostExplain` which are used to view the proof state and then give the natural language output. Before executing a proof script with the plugin, the proof is instrumented so that there are calls to the new tactics before and after each of the existing tactics. So far, we have a working natural language proof generator for a subset of tactics. In order for this system to generate valid Proof Blocks problems, we will need to have it also extract the directed acyclic graph denoting the dependence between lines of the proof. To further improve the system, our goal is to have a way to generate new theorems and proofs so an instructor can specify a single theorem and proof, and then similar theorems and proofs can be generated automatically, giving many practice problems for their students.

Example Output We prove the reflexivity of division as follows:

```
Definition divide (a b : Z) : Prop := exists q : Z, b = (q * a)%Z.
Notation "( x | y )" := (divide x y) (at level 0) : Z_scope.
Lemma divide_refl_inst: forall a: Z, (a | a).
Proof. intro x. unfold divide. exists 1. ring. Qed.
```

Now, when the instrumented version of the proof if executed, the following natural language version of the proof is generated:

> Let x be an arbitrary element of $\mathbb{Z}$. Now we must show that $(x|x)$. Which by the definition of divide means we need to show that $\exists q \in \mathbb{Z}, x = q * x$. Choose q to be 1. Now we must show that $x = 1 * x$. By algebraic simplification, this is clearly true.

To see more examples and execute the code, see https://github.com/SethPoulsen/robottwo.

References

[1] Andrew Bedford. Coqatoo: generating natural language versions of coq proofs. *arXiv preprint arXiv:1712.03894*, 2017.

[2] Yann Coscoy. A natural language explanation for formal proofs. In Jaime G. Carbonell, Jörg Siekmann, G. Goos, J. Hartmanis, J. van Leeuwen, and Christian Retoré, editors, *Logical Aspects of Computational Linguistics*, volume 1328, pages 149–167. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997. Series Title: Lecture Notes in Computer Science.

[3] Yann Coscoy, Gilles Kahn, and Laurent Théry. Extracting text from proofs. In Mariangiola Dezani-Ciancaglini and Gordon Plotkin, editors, *Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science, pages 109–123, Berlin, Heidelberg, 1995. Springer.

[4] M. Ganesalingam and W. T. Gowers. A Fully Automatic Theorem Prover with Human-Style Output. *Journal of Automated Reasoning*, 58(2):253–291, February 2017.

[5] Amanda M Holland-Minkley, Regina Barzilay, and Robert L Constable. Verbalization of high-level formal proofs. In *AAAI/IAAI*, pages 277–284, 1999.

[6] Clément Pit-Claudel. Untangling mechanized proofs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, pages 155–174, 2020.

[7] Seth Poulsen, Mahesh Viswanathan, Geoffrey L. Herman, and Matthew West. Proof blocks: Autogradable scaffolding activities for learning to write proofs. In *Proceedings of the 2022 ACM Conference on Innovation and Technology in Computer Science Education*, 2022.

[8] Various. Mathematical components. https://github.com/math-comp/math-comp, 2021.