EasyChair Preprint

№ 6267

Assimilating the Structure of Formal and Informal Proof

Kensho Tsurusaki and Akiko Aizawa

August 10, 2021

Assimilating the structure of formal and informal proof

Kensho Tsurusaki
The University of Tokyo
kensho.tsurusaki@gmail.com

Akiko Aizawa
National Institute of Informatics
The University of Tokyo
aizawa@nii.ac.jp

Abstract

Formal proof is regarded as an insufficient alternative to informal proof because it has both advantages and disadvantages, compared to informal proof. Focusing on the macro structure of proof, this study proposes assimilating the structure of formal and informal proof to retain both readability and verifiability. An implementation example of the Ntac system, a library for the Lean theorem prover, is introduced. The system converts Lean script into natural language text, which allows treating formal proof as informal proof.

1 Introduction

Formal proofs are verifiable by computer. As such, they must have detailed structures representing every step of logical inference. However, it is often difficult for human readers to follow such detailed structures. Informal proofs are readable by human, but they are generally not verifiable by computer.

Suppose the "same" proof is represented as both a formal and an informal proof. Then, there exists a difference between them owing to the usage of natural language. However, using only the restricted expressiveness of natural language (controlled natural language) or displaying hierarchical structure (structured proof, declarative proof) is insufficient to attain the high readability of informal proof. In this paper, we focus on the macro structure of informal proof, such as the 1) change of topic, 2) importance of consequence, and 3) appropriate omission. We assume that such auxiliary features of informal proof make the proof easier to understand for human readers.

The solution is to "assimilate" the two types into a single proof system. Namely, if a connection is built between natural language sentences and formal proof tactics, formal proof would become readable as informal proof. As an initial attempt, a library for the Lean theorem prover [dKA+15], which is called Ntac (Natural tactic)is developed. It introduces new types of tactics that can be converted into natural language text and allows the omitting of some detailed inferences from the output or the assigning of explanatory text manually.

2 Approach

The "idea" of a proof can be represented by a structure such as a proof tree. Here, we assume proof trees in sequent calculus as the idea of proof. A proof is constructed by considering "what has been proven before" and "what to prove now," whether formal or informal. This pair is called the "goal," in terms of a theorem prover. Goals can be multiple and are changed by commands called "tactics." Natural language sentences in informal proof corresponds to tactics in formal proof. This is the similarity between formal and informal proof.

In: A. Editor, B. Coeditor (eds.): Proceedings of the XYZ Workshop, Location, Country, DD-MMM-YYYY, published at http://ceur-ws.org

However, we need to verbalize the tree structure into a "linear" structure to represent it as proof text. There are many ways for this "linearization." Here are listed good proof (1a, 2a, 3a) and bad proof (1b, 2b, 3b), both written in natural language.

** change of topic*

(1a) We have A. Since A, B holds. We have C. Since C, D holds.

(1b) We have A. We have C. Since A, B holds. Since C, D holds.

** the importance of consequence*

(2a) Since A implies B, C holds. Therefore, D holds.

(2b) Since A, B holds. Since B, C holds. Since C, D holds.

** appropriate omission*

(3a) ... a is odd. ... ab is even. Since $a + b$ is odd, ...

(3b) ... a is odd. ... ab is even. If the product of two natural numbers is even, one of the two is even because 2 is a prime number. Since a is odd, b is even. Then, $a + b$ is odd. ...

When writing informal proof, the better option is chosen intuitively. However, when writing formal, we often write proof we often write the proof such that it leads to the worse option. Because proof assistants show all unresolved goals, there is no motivation to avoid confusing arguments like in (1b). Because formal proof languages do not have discourse markers such as "but" or "therefore," whose function G. Frege interpreted as "Färbung (coloring)," formal proofs have difficulty with expressing the importance of consequence like in (2b). Because computers require every detail of proof, there exists an obligation to write proofs like in (3b).

The structures of formal and informal proof are thus different, and it is almost impossible that a proof in one language is both verifiable by computers and readable by humans. However, if one thinks that they only have to understand or verify the "idea" of a proof, and not the proof text itself, then machines and humans can use different language texts to recognize the idea. Formal proofs contain full information of proof trees, so the goal of this study is to make formal proof readable as informal proof or to "assimilate" formal and informal proof.

There exist some previous studies related to this study. CNLs (Controlled Natural Languages) [Pas, Cra13, KN12] are formal languages mimicking natural languages and interpreted by both machines and humans. It seems effective to control the change of discussion topic, because writers will be motivated to make proofs readable as natural language text. However, as seen in the examples (2a) and (2b), restricting the usage of some "coloring" words which have no mathematical meaning does harm to readability. Another approach is called *structured* proof [Lam12]. They are written in natural language but explicitly displayed in tree structures such as Fitch-style proof notation, which clarify the importance of a consequence. However, they are separated into short sentences in many lines and are not necessarily readable as a whole.

3 The Ntac system

3.1 The tactic system of Lean

Lean provides a metaprogramming function by formalizing tactics as a monad [EUR+17]. Therefore, the tactics themselves, which are part of Lean language, are *defined* inside Lean. Tactics are written in `begin ... end` blocks in Lean script. By default, Lean uses tactics in the namespace `tactic.interactive`, but users can switch the namespace to `xxx.interactive` by writing `begin[xxx] ... end`. Therefore, we can virtually change only the implementation of existing tactics by defining new tactics in another namespace with the same names as existing ones. The tactics of Ntac uses namespace `ntac.interactive`, where several basic tactics are defined, including `exact`, `admit`, `have`, and `suffice`. Also, we added special tactics `T` and `X`. The descendant of `T` in the tree structure is removed from the output and thus used for omission. `X` has a similar function, but it takes one argument of string type and uses it as the output of the descendant.

3.2 Implementation

The translation process is divided into four steps:

(tactic script) → `goal_info` → `semantic_tree` → `list statement` → `string`

On execution of a tactic, a goal is either resolved or changes to one or more goals. Therefore, the process of the resolution of goals can be represented by a tree structure. With a metaprogramming function, the Ntac system collects complete information about how goals change and are resolved and stores it in `goal_info` data type.

The data type `semantic_tree` has a tree structure too, but it is refined to better represent the meaning of the usage of each tactic. For example, when the goal has a form like $\neg P (= P \to \bot)$, we can use either the `intro` or `by_contradiction` tactic to add P to assumptions and change the goal to $\bot$. Usually such differences are not present in natural language, so either tactic will be converted into a uniform data representing the assumption of P. Also, the data of inferences marked by tactic `T` are removed in this step.

The next step involves the transformation to the sentences of natural language. The tree structure is separated and linearized into a list of small structures of sentences. The data structure of `sentence` resembles the abstract syntax of Grammatical Framework (GF) [Ran11], and it is designed to support multiple languages. Currently, the system supports only English and Japanese. The translation from `sentence` to `string` is almost identical to what GF does, and the `trace_proof` tactic shows the result of conversion.

3.3 Example

Here, we use the proof of the infinitude of prime numbers as an example. Figure 1 shows how the Ntac system operates in the VS code extension of Lean.

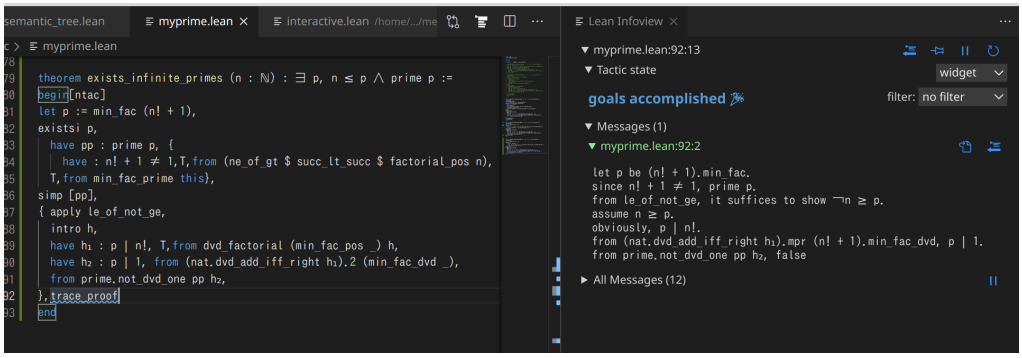

Figure 1: Ntac on VS Code

The Lean script and translation are listed below. The number at the beginning of each line indicates the correspondence between the lean script and its natural language translation.

```
(Lean script)
   :  theorem exists_infinite_prime_Ntac (n : ℕ) : ∃ p, n ≤ p ∧ prime p :=
   :  begin[ntac]
(1):    let p := min_fac (n! + 1),
   :    existsi p,
(2):    have pp : prime p, {
(2):      have : n! + 1 ≠ 1, T, from (ne_of_gt $ succ_lt_succ $ factorial_pos n),
(2):      T, from min_fac_prime this},
   :    simp [pp],
(3):    { apply le_of_not_ge,
(4):      intro h,
(5):      have h₁ : p ∣ n!, T, from dvd_factorial (min_fac_pos _) h,
(6):      have h₂ : p ∣ 1, X "Since p is a prime factor of n! + 1, p ∣ n! + 1. therefore, p ∣ 1",
(6):                       from (nat.dvd_add_iff_right h₁).2 (min_fac_dvd _),
(7):      from prime.not_dvd_one pp h₂},
   :    trace_proof
   :  end



(translation)
```

```
(1):  let p be (n! + 1).min_fac.
(2):  since n! + 1 ≠ 1, prime p.
(3):  from le_of_not_ge, it suffices to show ¬n ≥ p.
(4):  assume n ≥ p.
(5):  obviously, p | n!.
(6):  Since p is a prime factor of n! + 1, p | n! + 1. Therefore, p | 1.
(7):  from prime.not_dvd_one pp h₂, false
```

In the first line of the script, we define p as the minimal prime factor of $n! + 1$. `existsi p` means that we use the defined p as quantified variable p in $\exists p, n \leq p \wedge \text{prime}(p)$. The system is aware that the two variables have the same name, so it produces no translation. The next three lines include good examples of tactic `T`. We prove $n! + 1 \neq 1$ from the positivity of $n!$, and then prove p is prime from $n! + 1 > 1$. Because these are relatively trivial, we insert `T` tactics before proving them. Ntac can translate this structure of three lines into one sentence using the word "since".

In the next line, the `simp` tactic is used. Currently Ntac does not support translation of simp, and it is just copied from the original Lean. It simplifies goals or assumptions using lemmas or hypotheses. Here, it uses `pp : prime p` to simplify the goal from $n \leq p \wedge \text{prime}(p)$ to $n \leq p$.

The `apply` tactic in (3) is a typical usage of tactics against a goal. The lemma `le_of_not_ge` asserts $\neg a \geq b \to a \leq b$ for all a and b, so the goal can be changed from $n \leq p$ to $\neg n \geq p$. Here "it suffice to show" is used as translation. Note that information about the goal is present only in translation. The next `intro` tactic is also used against the goal, proving by contradiction. Now, the goal is $\neg n \geq p$. It assumes $n \geq p$ and changes the goal to $\bot$. It is translated with the word "assume".

In the next line, a `T` tactic is again used with `have` and `from` tactics. The structure is different from the previous example, and Ntac translates it with the word "obviously." The next line shows the usage of tactic `X` with `have`. The inference is explained by the text specified in the argument of `X`. The last line is the derivation of $\bot$ (false, contradiction), which concludes the proof.

4 Discussion and Future Work

The Ntac system revealed that formal proof languages can have an expressive power to incorporate the macro scale structure of proof, which we initially assumed only informal proof could have. The current implementation of Ntac is only experimental, and we haven't considered the conversion at the micro structure of proof, that is, the syntax of natural language with mathematical object. For example, "`since n! + 1 ≠ 1, prime p.`" is not a regular English sentence. Also, the system provides no guarantee for the correctness of natural language output, because users can write anything using the tactic `X`. Now, the most important question is how and to what extent should one be able to customize the natural language proof converted from formal proof. Other open problems in this field include display of mathematical equations or compatibility between foundations of mathematics. They may also relate to this research, and thus, a better way to represent proof must be carefully found.

Acknowledgements

This work was supported by JSPS KAKENHI Grant Number 21H04415.

References

[Cra13] Marcos Cramer. Proof-checking mathematical texts in controlled natural language. 2013.

[dKA+15] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

[EUR+17] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A metaprogramming framework for formal verification. *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–29, August 2017.

[KN12] Kevin Kofler and Arnold Neumaier. DynGenPar–a dynamic generalized parser for common mathematical language. In *International Conference on Intelligent Computer Mathematics*, pages 386–401. Springer, 2012.

[Lam12] Leslie Lamport. How to write a 21st century proof. *Journal of Fixed Point Theory and Applications*, 11(1):43–63, March 2012.

[Pas] Andrei Paskevich. The syntax and semantics of the ForTheL language. page 70.

[Ran11] Aarne Ranta. *Grammatical Framework: Programming with Multilingual Grammars*, volume 173. CSLI Publications, Center for the Study of Language and Information Stanford, 2011.