6-2016

# Reading Between the Lines: Verifying Mathematical Language

Tristan Johnson
*Union College - Schenectady, NY*

# Reading between the lines

Verifying Natural Mathematical Language

By

Tristan Johnson

*********

Submitted in partial fulfillment of the requirements for

Honors in the Department of Computer Science

Union College

May 17, 2016

**Abstract**

JOHNSON, TRISTAN Reading between the lines: Verifying Natural Mathematical Language
Department of Computer Science, March 2016

ADVISOR: Kristina Striegnitz

A great deal of work has been done on automatically generating automated proofs of formal statements. However, these systems tend to focus on logic-oriented statements and tactics as well as generating proofs in formal language. This project examines proofs written in natural language under a more general scope of mathematics. Furthermore, rather than attempting to generate natural language proofs for the purpose of solving problems, we automatically verify human-written proofs in natural language. To accomplish this, elements of discourse parsing, semantic interpretation, and application of an automated theorem prover are implemented.

# Contents

# List of Figures

# 1 Introduction

I am studying how one may automatically verify a natural language mathematical proof because I want to investigate how effective computer-based natural language reasoning systems may be in order to enable automated theorem proving systems to interact with human-produced reasoning.

For example, suppose a student was participating in a mathematics class and was looking for feedback on their assignments. They would need to wait to meet with their instructor for accurate feedback, which greatly slows down the speed at which they may learn new material or complete assignments. If the student had a system where they could check their work in real time, it would improve the learning experience all around. It would be important that this system were accessible to the student, where the best case scenario would be if it directly takes in their answer to the assignment without any manual formatting. Similarly, if a system of verification could be made general enough, the same principal could apply to mathematical researchers wanting to check their work in real time.

To begin thinking about how to approach this problem, we will work through an example of a simple proof, and note what we must do as external evaluators to know whether or not the proof is *correct*. In the proof, we will observe that specific statements or claims that are relevant to the proof are made. So that we may refer to them later, we will emphasize the claims with brackets and label them.

> Let $A$ and $B$ be sets. Prove that if $[A \subseteq B$, then $A \subseteq A \cap B]^{\ell 5}$.
>
> *Proof.* Assume $A \subseteq B$ is true. We then need to prove that $A \subseteq A \cap B$ is true. To show this, let $x \in A$ be arbitrary. [Since $x \in A$ and $A \subseteq B$ we know that $x \in B]^{\ell 1}$. Therefore, $[x \in A$ and $x \in B$ and hence $x \in A \cap B]^{\ell 2}$. Since $x \in A$ was arbitrary we have proved that $[\forall x \in A, x \in A \cap B]^{\ell 3}$. It now follows from the definition of "subset" that $[A \subseteq A \cap B]^{\ell 4}$ and thus we are done. $\square$

Notice that to perform our verification, we do not wish to verify the statement, but rather the *proof*. So we need to know the proof is true in each claim (mathematical statement) that is made, and the overall proof is actually relevant to the original statement. In the end, we want to know that it was shown that '$A \subseteq A \cap B$.' To do so, we can verify each statement and in the end deduce $A \subseteq A \cap B$ directly from these statements.

As will become clear later due to the methods we will use, we will refer to a claim that has been must be verified as a lemma. It is important to recognize that a lemma is placed in the context of the proof, and so all that is known up to the current point in the proof may be used to verify the lemma. In the example, although the first lemma suggested is 'If $A \subseteq B$ then $A \subseteq A \cap B$,' this will be the last we prove since it denotes the overall structure of the proof (this is inferred from 'we then need to prove'). Our first lemma is then

$$\ell 1 : (A \subseteq B \wedge x \in A) \implies x \in B$$

since this is the first real statement made after the initial assumptions. As for verifying the lemma, it follows directly from the definition of subset. The next statement is "$x \in A$ and $x \in B$ and hence $x \in A \cap B$." But in the context of the proof, we also know the assumptions and previously proved statements (which are often used in proofs where their use is not directly stated), so our next lemma is

$$\ell 2 : (A \subseteq B \wedge x \in A \wedge x \in B) \implies x \in A \cap B$$

Even though $A \subseteq B$ is irrelevant to the lemma, it is still true. The verification of this lemma follows directly from definition of intersection. Our next lemma is more subtle; it is derived from the structure of the proof. We assumed the existence of an arbitrary object and showed the properties of this object. This is the format of proving a 'for all' statement, so we derive the assumption implying the final result. This is stated explicitly in the next sentence of the proof, but we would like to be able to derive this statement without it being explicit. The lemma is

$$\ell 3 : \forall x (A \subseteq B \wedge x \in A \implies x \in A \cap B)$$

This lemma is verified by $\ell 1$ and $\ell 2$. Next, the proof is trying to show $A \subseteq A \cap B$ and so we must know the structure of the proof does in fact prove this point. This lemma is

$$\ell 4 : (A \subseteq B \wedge \forall x (\wedge x \in A) \implies x \in A \cap B)) \implies A \subseteq A \cap B$$

This lemma follows from the basic logical properties of $\wedge$ and the definition of subset. Finally, as the proof

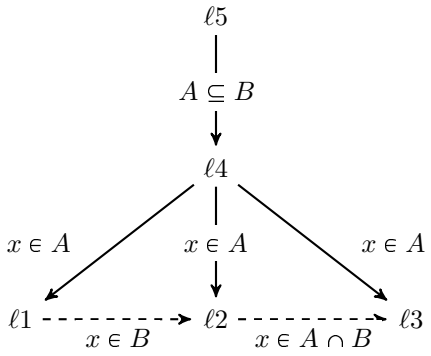


Figure 1: A tree representing proof structure. Labels of edges are assumptions passed along the edge. Dashed lines are implicit assumptions by sharing the same parent.

statement is an 'if-then,' we must verify this structure using the previous lemmas, but only those that make the same assumptions, that is, not any that inherently assume $x \in A$. This is just $\ell 4$, since there is no inherent assumption

$$\ell 5 : A \subseteq B \implies A \subseteq A \cap B$$

Notice that in listing our lemmas, we found some natural structure to the proof. $\ell 5$ needed to have $\ell 4$ to prove that it is true, and $\ell 4$ took on the assumptions of what it was proving (that $A \subseteq B$). Similarly, all of $\ell 1, \ell 2, \ell 3$ are used to prove $\ell 4$ and assume its assumptions (that $A \subseteq B$ and implicitly that $x \in A$). This structure forms a tree as in Figure 1, where a child is used to prove its parent, and assumes any assumptions it makes. If nodes share a parent, they are proved in order left to right, and each assume the statement of the previous.

Before we were able to deduce any of the high-level proof structure, we (without actively saying so) picked out the important statements and labeled them as lemmas. To do so, we had to find keywords and parts of the language that hinted that this was 'the next lemma.' The structure of a sentence, by breaking it down into clauses and keeping track of the objects in context (or referents), allows us to make heuristic guesses at what each of these lemmas should be. As readers of math, one does this naturally. While doing this step, we need to know how to interpret the mathematical symbols and text as if it were natural language, and incorporate it into the interpretation of the rest of the text.

With our lemmas constructed within our tree, the last step is simply to verify each lemma independently (since the dependencies have already been accounted for in the statement of each lemma). This verification would require a rigorous, logical approach that relies on direct inferences. Furthermore, some set of axioms (and allowable inferences) must be assumed to begin working with any logic, and so we must understand the axiomatic system that we are working in.

Our motivation gives rise to four subproblems that we must solve. First, we need to pick out the important text for our lemmas. That is, we need to parse the text into a semantic representation to be able to understand which parts we want to directly use. Next, we need to interpret this structure into independent lemmas, so that we can work on subproblems without worrying about context. Both of these steps require that we are able to parse and interpret mathematical language. Finally, we need to automatically verify each of these lemmas.

# 2 Background and Related Work

## 2.1 Discourse Parsing

Discourse parsing refers to analyzing the natural structure of natural language and representing it in a way without ambiguity. There are two parts to this. First, the text is broken into segments and assigned labels, which designate the unique type of language (sequence, consequence, etc) that is contained in that segment. Secondly, these segments are aligned into a tree where edges are discourse relations, or keywords that indicate how two segments fit together [11].

In this project, placing the brackets around segments of language and then aligning these segments into the structure tree is almost exactly analogous to discourse parsing. There are, however, two differences.

First, the structure inherent in the language depends more on logical proofs than the syntax and semantics of the language. They are related, and so we should be able to build a tree of the proof structure based on the discourse structure tree without difficulty. This problem does not have much work that has already been previously done, and so would need to be done from the ground up.

Secondly, parsing natural language is a problem that has had much study, with positive results in that

a program may parse discourse with 90% of human accuracy [11]. However, in order to truly embrace mathematical natural language, this project would need to include the syntax and semantics of natural language mixed with symbolic mathematics. Although this may even simplify the language as it involves statements with less ambiguity, discourse parsing in the past has not covered this topic extensively. More on the complication of mathematical natural language is discussed in the next section.

## 2.2 Semantic Interpretation

Many challenges with the semantic interpretation of math and logic in natural language and a formal language have been well researched. Some of these questions include how to remove semantic ambiguity [3] [10], choosing an axiom set or from a knowledge base [3] [8], categorizing natural language for determinisitic processing [10] [5] [8], how much control on the natural language should be enforced [12] [6] [4] [5], and how the formal language should be constructed [10] [6] [7]. Research has been successful in constructing a system that is able to autonomously pass a university entrance exam [3]. Though impressive and with many useful ideas, this system is bounded by its axiom set dealing only with the real field and its geometry. Most fields of modern math (e.g. set theory, abstract algebra, or topology) go far beyond this scope. Furthermore, the system is focused on answering questions and constructing objects, rather than finding proofs.

This project builds off these previous studies as an application. Rather than improve upon methods of interpreting mathematical semantic meaning, this study uses these ideas as a part of the verification system.

As seen above, one of the largest problems with formalizing mathematics from natural language is disambiguation. Formal languages require absolute facts, since they attempt to encompass absolute truth. Although researchers have constructed methods of removing ambiguity in translating natural language [10], heuristic means must be used to choose the most likely meaning in any ambiguous scenario.

One way to solve the problem of ambiguity is to limit the use of language. Researchers have shown that a subset of English named ACE (Attempto Controlled English) is capable of completely representing first-order logic [4]. Controlled subsets such as this avoid much of the ambiguity of grammar and scope, creating an accurate formalization. However, even when controlled, the languages work within English

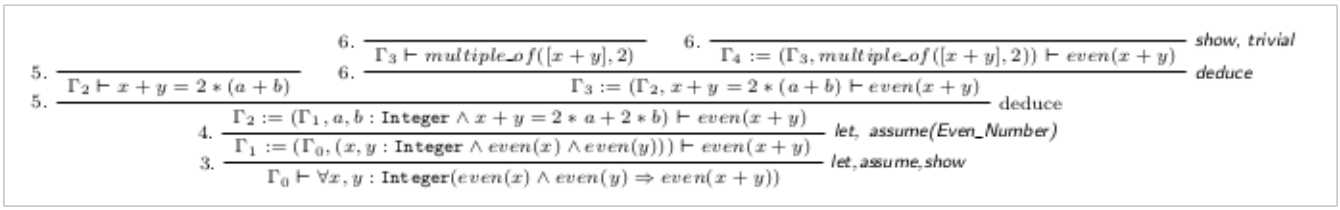


Figure 2: Semantics of the proof that the sum of evens is even in MathAbs

naturally. For example, given that the words manager, department, and employee are introduced, the sentence 'Every manager of a department is an employee and a member of the department' is a valid and semantically meaningful sentence in ACE.

Further work has been done to translate these controlled languages back to a formalization. One successful example of this is MathAbs, which is a formalization with an implemented translation from CLM (Controlled Language of Mathematics) [6]. Furthermore, MathAbs supports formalized grammars for proof structure and strategies, making it even more helpful for Automated Theorem Proving systems. As MathAbs is made with formalism in mind, it is easy to express proof as basic inferences in this language, as seen in Figure 2.

Ultimately, we would like to be able to perform the functions described in these projects but not require a controlled language. This would require some heuristic form of disambiguation, as opposed to the absolutely predictable form of controlled language.

## 2.3 Automated Theorem Proving

Automated theorem proving (ATP) has been a widely studied topic since the beginning of theoretical computer science. One representative example of an ATP program is Theorem Proving System, or TPS. [2]. Theorem provers such as these implement complex programs for solving first-order logic and induction problems. As recognition for another motivation of this project, Figure 3 shows the complex and difficult to understand language that theorem provers use as input and output. Furthermore, pure proof systems like TPS are unequipped to handle mathematical semantics directly without translation into a logical rep-

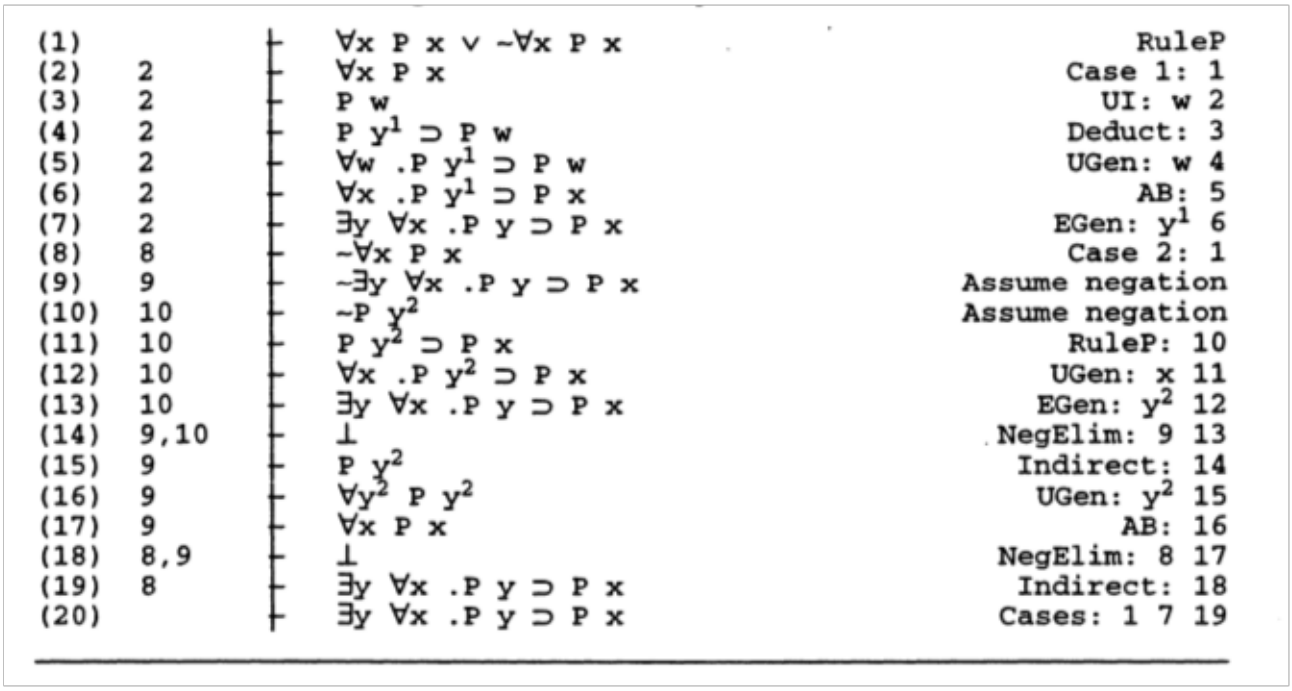


Figure 3: TPS output (transformed)

resentation. To make these tools more accessible to users, a more intuitive interface would be desirable.

One approach to this problem is in the form of proof assistants. Rather than attempt to prove a theorem directly, a proof assistant relies on a user to directly write the proof while providing rudimentary guidance and verification at each step. One such program is the Coq Proof Assistant [7]. This proof assistant was made in the interest of helping mathematicians write their proofs with real time verification, allowing for more reliability in their work. Coq employs a programming language of sorts in order to interact with the system, and creates subgoals to be solved, wherein a theorem may be proved by the user by reducing all subgoals.

Most importantly for our study is a program that is somewhat between these two classifications of proving programs. Isabelle/HOL is simultaneously a functional programming language, proof assistant, and an ATP system [9]. Isabelle is easily programmable as it interprets its functional language Isar. Furthermore,

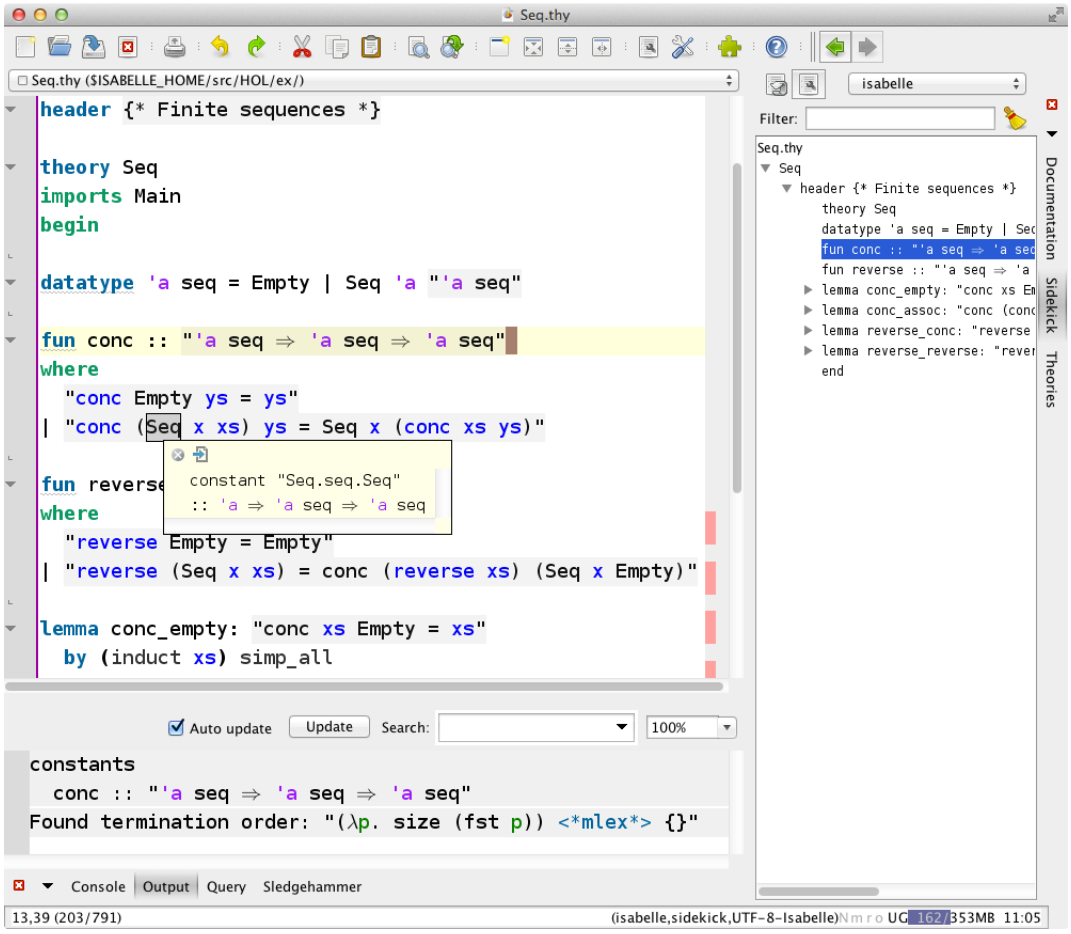


Figure 4: An example Isabelle/HOL proof (or program) in its user interface[1].

it is a powerful prover, capable of applying many other provers to any given problems. As a packaged tool, Isabelle is made to work out a user interface such as is displayed in Figure 4. It may also be applied automatically, which would be the desired use for the purposes of this project.

# 3 Methods and Design

I began by finding a corpus of natural language math proofs. Doing this first allowed me to develop the ideas discussed in the Introduction and build a general method to approach this problem. Furthermore, by

basing an implementation around a corpus we may have a tool that is testable and generalizable by way of expanding the corpus.

The corpus used in this project was a collection of sample problems and solutions (courtesy of Professor Paul Friedman) from MTH 199 (Introduction to Logic and Set Theory) at Union College. I chose some representative examples from the corpus to design the project, after which I planned on applying the tool to the rest of the corpus as testing. The first example was a basic set-containment proof using a for-all as in the example in the Introduction, the second was another set-containment proof but involved sets and properties of real numbers, and the last was a set-containment proof that relied on exhaustive cases.

I knew the final step in my process would involve automatic verification, so I started work on an implementation based on reaching this step. I planned on using an already made ATP system for this rather than do this from scratch, but I needed to find which. I looked into a few automated provers, but decided that they would require too much formatting of the inputs to work easily and reliably. I next looked into the Coq Proof Assistant. This program works naturally with math and is well vetted by the mathematical research community, so it seemed like a good choice. However, I soon found that Coq introduces new variables into a proof in real time as it builds subgoals, which hinders any ability to automatically write a lemma in Coq.

This led me to Isabelle/HOL, which has the advantages of being able to easily represent math and also able to (almost) automatically write and verify lemmas. After learning the basic workings of Isar, Isabelle's functional language, I wrote up programs that represented the verification of each of my representative examples. Doing this gave me the intuition as to how to approach the problem as a whole which allowed me to describe the high-level steps outlined in the end of the introduction (discourse parsing of math and natural language, semantic interpretation, and automatic verification). Since I already had the automatic verification step outlined, it made sense to work backwards through these steps. That is, I next worked on semantic interpretation before discourse parsing.

Initially, I had begun approaching the problem of semantic interpretation through DRT. This seemed like a good choice for a variety of reasons. DRT is a formal semantic interpretation for language, is able to handle quantifiers and logic easily, interprets context, and tracks referents.

However, after completing the example verifications in Isabelle, I recognized that this approach was

too heavy handed. Isabelle is already capable of handling each of the challenges of language that DRT addresses, such as quantifiers or some parts of context. I changed my approach to fit the specifications of my problem, attempting to interpret the semantics of a proof from scratch. By studying the examples I began applying a tree structure to proofs.

In a proof, certain elements of the structure that spans multiple claims (for example, proving a subset relation by taking an arbitrary element) rely on context. I identified several of these structures and gave rules as to how each must be verified in order to verify the overal proof. These structures are listed in Figure 5. As in the table, the tree consists of proof structures whose children are any substructures (as refered to in the table). *Direct* structures are leaves of the tree as they have no substructure. Each proof structure should have a part to_verify, as well as take in prior assumptions to use. Usually, the assumptions are passed to the substructures and the structure is proved using its children. The specific possibilities are described in the table.

| | Name | Description | Representation | Structure | |
|---|---|---|---|---|---|
| | *Claim* | A statement that is verified by other statements, usually in linear order | Claim: <to_verify>, <assumptions>, <substructures> | Substructures assume the given assumptions. The to_verify parts of a substructure are assumed by any following substructure, and all are assumed to prove to_verify. | |
| | *Direct* | A statement that is verified only by prior assumptions | Direct: <to_verify>, <assumptions> | to_verify is proved using assumptions only. | |
| | *Theorem* | Exactly the same as a claim, but recognized as the root of the tree. | Theorem: <to_verify>, <assumptions>, <substructures> | Substructures assume the given assumptions. The to_verify parts of a substructure are assumed by any following substructure, and all are assumed to prove to_verify. | |
| | *Induction* | Verifies the given statement through induction in the substatements | Induction: <to_verify>, <assumptions>, <base_structures>, <inductive_assumption> <inductive_structures> | All substructures assume the given assumptions. The base structures (base case) work as a normal claim. The inductive structures assume the inductive_assumption then work as a normal claim. to_verify is proved by recognizing inductive structure and verification of each of the base and inductive structures | |
| | *Contradiction* | Verifies the given statement through induction in the substatements | Contradiction: <to_verify>, <assumptions>, <substructures>, <contradiction> | Substructures assume the given assumptions and ¬to_verify. The to_verify parts of a substructure are assumed by any following substructure. contradiction is a substructure for which we prove ¬contradiction, assuming the to_verify parst of each of the substructures as well as ¬to_verify and the given assumptions | |
| | | | | | |

| | | | | | |
|---|---|---|---|---|---|
| | *Cases* | Verifies the given statement through by exhausting the given cases | Cases: <to_verify>, <assumptions>, <case_and_substructures> | case_and_substructures is a case followed by proof substructures. All cases are checked to be tautological together. A substructure in a case assumes given assumptions. For each case, also verify that case implies to_verify after verifying the case's substructures. | |
| | *Forall-Claim* | Verifies the given statement through induction in the substatements | Forall: <declare>, <to_verify>, <assumptions>, <substructures> | declare is the first reference of the quantified variable along with any constraints. substructures assume declare and the given assumptions. to_verify is verified with given assumptions and all substructures to_verify parts. | |

Figure 5: Proof structures and how to implement verification.

To interpret the proof structures, I wrote a python program that takes in the tree of a given proof according to these structures. Assuming the parameters of each substructure are valid LaTeX expressions (using just the standard LaTeX packages amsmath and amsthm), each piece gets parsed into valid Isar (Isabelle) syntax. These are then constructed into lemmas as is defined in their structure. The output is an Isabelle program file (.thy). This may be run, and on successful exectution we will know the proof represented by the tree is verified.

To test this strategy, I manually translated the proof examples into their tree structure as would be given by basic parsing. I tweaked the program until these produced valid Isabelle programs, as we would expect them to be correct and verifiable.

I ran into two large issues during this process. I had originally intended on making my project entirely automatic verification with no human interaction required. However, I realized that with the more complex ATP systems and proof assistants, they employ heuristic proof methods. For Isar and Isabelle, the heuristic proof method (called a tactic) must be chosen by the progammer at each step in the program/proof. I

found 3 resolutions to this first problem. I could fix a set of trial tactics and generate the program for all permutations of assignments of tactics to lemmas. This method would be slow, and would not be scalable past whatever the fixed set of tactics would be. Another solution would be to set up a feedback loop with Isabelle, running a proof method, reading a suggestion that it gives, and reapplying the new tactic until the lemma is proved. This would be the most likely to successfully prove a lemma, but may overstep its bounds and verify a proof where there is a gap in the logic (even if it is still true). Furthermore, this method would be very difficult to implement given the nature of Isabelle's output. The last resolution I considered was to have the user provide the tactic at run time for the lemma generation process. This would give flexibility and leave the choice of what is or isn't too big of a gap in logic to the user, while having the only downside of introducing somewhat of a learning curve for the user. This last choice is what was implemented for the project.

At this point, I had completed the last two steps (semantic interpretation and automatic verification), but still had yet to properly parse the discourse of a proof into an interpretable format. I looked into tools for implementing discourse parsing, and was able to find one based of the Stanford Discourse Parser with a reasonable amount of customizability [11]. After attempting to set up this tool for my project, I discovered that the version I had was outdated and no longer functioning with the current dependencies (Natural Language Toolkit). I did not have time to implement a different tool for this purpose, and so I hope to finish this last step as future work.

# 4 Conlusion

In this project, we were able to construct a proof of concept for a mathematical natural language verifier focused on undergraduate level set-theory proofs. This system uses a paradigm of parsing, interpretation, and execution beginning from mathematical natural language and resulting in confirmation of verification. In its present state, the system is able to take in parsed discourse and output valid Isar programs, which may be executed for verification.

Many improvements come to mind for advancements or completions of this project. First, completion of the discourse parsing section of the project would enable it to work with truly natural language. Fur-

thermore, fully automating the implementation would allow it to take a packagable form, which may be distributed for testing with users and further improvement. Finally, the scope of the applications could be extended to handle much more general mathematics such as algebra or analysis.

In its present state, this project sees its best application as a classroom tool for helping teach students reasoning and mathematics. Whereas learning programming or proof tools may have little relevance for a class focusing in the aforementioned skills, having an automatic grader for students to check their homework against may be helpful. In this way, a natural language proof assistant may be ideal for students in a classroom. In the case that this project were extended to properly handle abstract and advanced mathematics, it may be useful as a preliminary component to mathematical research. If a mathematician may rather informally check their proofs against a reasonably authoritative source in real time, it may greatly speed the writing and peer review process.

# References


[1] Isabelle overview. http://isabelle.in.tum.de/website-Isabelle2014/overview.html. Accessed: 10-20-2015.

[2] Peter B. Andrews, Matthew Bishop, Sunil Issar, Dan Nesmith, Frank Pfenning, and Hongwei Xi. TPS: A theorem-proving system for classical type theory, 1996.

[3] Noriko H. Arai, Takuya Matsuzaki, Hidenao Iwane, and Hirokazu Anai. Mathematics by machine. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, ISSAC '14, pages 1–8, New York, NY, USA, 2014. ACM.

[4] N.E. Fuchs, U. Schwertel, and S. Torge. Controlled natural language can replace first-order logic. In *Automated Software Engineering, 1999. 14th IEEE International Conference on.*, pages 295–298, Oct 1999.

[5] Raúl Ernesto Gutiérrez de Piñerez Reyes and Juan Francisco Díaz Frías. Preprocessing of informal mathematical discourse in context ofcontrolled natural language. In *Proceedings of the 21st ACM Inter-*

*national Conference on Information and Knowledge Management*, CIKM '12, pages 1632–1636, New York, NY, USA, 2012. ACM.

[6] Muhammad Humayoun and Christophe Raffalli. MathAbs: A representational language for mathematics. In *Proceedings of the 8th International Conference on Frontiers of Information Technology*, FIT '10, pages 37:1–37:7, New York, NY, USA, 2010. ACM.

[7] Wojciech Jedynak, Malgorzata Biernacka, and Dariusz Biernacki. An operational foundation for the tactic language of Coq. In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*, PPDP '13, pages 25–36, New York, NY, USA, 2013. ACM.

[8] Sabina Jeschke, Marc Wilke, Marie Blanke, Nicole M. Natho, and Olivier F. Pfeiffer. Information extraction from mathematical texts by means of natural language processing techniques. In *Proceedings of the International Workshop on Educational Multimedia and Multimedia Education*, Emme '07, pages 109–114, New York, NY, USA, 2007. ACM.

[9] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003.

[10] F. Tacoa, H. Uchida, and M. Ishizuka. A word sense disambiguation approach for converting natural language text into a common semantic description. In *Semantic Computing (ICSC), 2010 IEEE Fourth International Conference on*, pages 478–486, Sept 2010.

[11] Vanessa Wei Feng and Graeme Hirst. A linear-time bottom-up discourse parser with constraints and post-editing. In *Proceedings of the 52Nd Annual Meeting on Association for Computational Linguistics*, ACL '14, Stroudsburg, PA, USA, 2014. Association for Computational Linguistics.

[12] Magdalena Wolska and Ivana Kruijff-Korbayová. Analysis of mixed natural and symbolic language input in mathematical dialogs. In *Proceedings of the 42Nd Annual Meeting on Association for Computational Linguistics*, ACL '04, Stroudsburg, PA, USA, 2004. Association for Computational Linguistics.