Creation of a modular proof assistant engine for a logic e-tutor

Jakub Dakowski[1], Aleksandra Draszewska[1], Barbara Adamska[1], Dominika Juszczak[1], Łukasz Abramowicz[1], and Robert Szymański[1]

Adam Mickiewicz University, Poznań, Poland
larch.amu@gmail.com

1 Background

There have been several attempts at creating Intelligent Tutoring Systems (i.e. applications that provide intelligent teaching support for their users) for several proof methods in formal logic. Huertas [6] counts 8 e-tutors created in the first decade of this century. Nowadays such software usually can give demonstrations and in some cases finish proofs that have already been started [4]. Such programs use many different strategies for obtaining hints. Unfortunately, most of these programs tend to implement only one formal system with a fixed syntax and static output form.

2 Aim

This project, called *Larch*, aims at improving these aspects of Intelligent Tutoring Systems. It's an application designed in hopes of creating a middleman between the researchers trying to implement new hint generation mechanisms and users who usually are unaccustomed to the complicated notation used in formal logic and hard to use interfaces. It was originally created with Analytic Tableaux for Classical Propositional Logic [8] in mind, but recently efforts were made to introduce sequent calculus to this system. Such software created the need for a system that encapsulates the architecture of a typical logic e-tutor in the form of interchangeable plugins.

3 Method

The discussed program was written in Python in line with its versatility and popularity. Python ensures a low entry threshold for creating Larch plugins and at the same time, it doesn't create any major boundaries in terms of possible solutions.

Larch's architecture was inspired by the *Plugin Oriented Programming* paradigm, in which the codebase splits into modular and independent plugin subsystems with a central hub [7]. The plugin system used in *Larch* was created specifically for this project using the built-in `importlib` module. Its central concept is a *Socket*, to which different plugins (these can be both singular files and packages) can be connected via specific functions that are defined in a plugin template. Plugins can still use core libraries and they also have their own socket libraries (internally called `utils` files).

Five sockets have already been created: User Interface, Output generator, a tokenizer (internally called a Lexicon), a Formal System with a solver and a hint generator (called Assistant). Here the last three components of this system are discussed.

The most basic of the three is the Lexicon socket which specifies the alphabet used in the program. It allows for simple customization of the used symbols. For example users can quickly implement a notation which uses LaTeX commands instead of the built-in method. This is implemented using a class provided by the software. This class is in fact a wrapper for the *SLY* package [1], which is a Python library used to create tokenizers and parsers. Such implementation allows the users to define tokens which will be used only while certain formal systems are plugged in. This wrapper also alleviates the hurdle of generating new tokens, thanks in part to the *exrex* package [9].

The Formal socket's role is to perform operations on the proof, check its correctness and finish the proof if needed. As the software was designed to teach Analytic Tableaux for Propositional Logic, one of the biggest dilemmas in this project was to create a system which will not end up as a simple solver. Because of this a notion of strict and naive versions of rules was introduced. When used incorrectly (for example using a rule for implications on a conjunction), naive rules will produce *a* conclusion and strict rules will fail. In a way, strict rules are adequate and their naive counterparts are complete, but not sound. Naive rules are used by the user, while the strict rules are used by the checker and the solver (rule priority is implemented by storing the rules in a tree). These structures are however completely optional and, if desired, this can be implemented from the ground up.

The Formal socket interacts with the Assistant socket to provide the necessary hints and point out user mistakes. In the beginning the Assistant socket also provided a solver, however, while it can still use its own solver, this burden was shifted onto the Formal socket. Assistant socket was given a more didactic role - it provides a knowledge base and generates feedback based on Formal sockets activities. While it has a lot of freedom when it comes to generating hints, commenting on user mistakes is a simple act of interpreting `UserMistake` objects generated by the Formal socket.

Besides the plugin system, there are also different aspects allowing for this amount of freedom in implementation. There is a context management system that ensures both proper information for the user and the data needed to operate on the proof. *Larch* relies on context definition objects created in the language's plugin. These contain information about arguments (both technical, such as type, and user-oriented, such as a readable name), which need to be provided to a rule in order for it to work. To improve the readability of code this plugin management system is connected directly to Python type hinting system. This way the context can be defined with the naive rule functions. The data structure of the whole project consists of two elements. The formulas are stored in a tree structure (implemented using *anytree* [2]) alongside their history and the closedness of their branch. The second part consists of the proof's metadata - right now this is mainly rule history.

4 Applications

As Analytic Tableaux for Classical Propositional Logic was the original purpose of this tool, it has already been implemented. Besides detecting and commenting on mistakes made by the user, the program also produces High Level Hints [3] regarding proof length optimisation and operation precedence. If need be it is also possible to implement Next Step Hints, however these are not recommended — the software can be reduced to a solver when their usage is not controlled.

There are also ongoing works on the sequent calculus for the Intuitionistic Propositional Logic (based on the Swiss calculus by [5]). The Formal plugin has been mostly implemented, but it still lacks a proper solver and a syntax checker. A custom Assistant plugin would also

be beneficial. However, this still shows that, despite the major differences, it is possible to introduce other formal systems in the form of plugins. Implementing the sequent calculus for the Intuitionistic Propositional Logic will also allow for a better presentation of the hint and improved error detection mechanisms.

5 Discussion and future work

The unique value of Larch comes from its ability to encapsulate other tools. With that being said one should ask about the possibilities of such a tool. As of now, it was only tested on a relatively straightforward proof method. This created a situation in which the hint generation couldn't be fully explored, both in the case of different formal systems and Intelligent Tutoring System algorithms. In the future, these perspectives should be explored. The ongoing implementation of the sequent calculus also shows that plugin systems should include diverse built-in libraries. While it certainly is possible to implement new formal methods using what is available now, better tools would facilitate this task.

This work shows a possible application of *Plugin Oriented Programming* in Automated Theorem Proving, but it is not the only one. Similar techniques could be used to modularize other systems and might someday produce a universal standard for logic-related libraries, which would allow almost seamless interchangeability and code reusability. Even in the case of *Larch*, its code might someday be used to create other software, either by reusing the plugins or by reusing the engine gluing them together.

References

[1] David Beazley. Writing parsers and compilers with ply. *PyCon'07*, 2007.

[2] c0fec0de. anytree, Dec 2019.

[3] Christa Cody, Behrooz Mostafavi, and Tiffany Barnes. Investigation of the influence of hint type on problem solving behavior in a logic proof tutor. In *International Conference on Artificial Intelligence in Education*, pages 58–62. Springer, 2018. https://link.springer.com/chapter/10.1007/978-3-319-93846-2_11.

[4] Cristiano Galafassi, Fabiane FP Galafassi, Eliseo B Reategui, and Rosa M Vicari. Evologic: Intelligent tutoring system to teach logic. In *Brazilian Conference on Intelligent Systems*, pages 110–121. Springer, 2020. https://link.springer.com/chapter/10.1007/978-3-030-61377-8_8.

[5] Jacob M Howe. Two loop detection mechanisms: a comparison. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 188–200. Springer, 1997.

[6] Antonia Huertas. Ten years of computer-based tutors for teaching logic 2000-2010: Lessons learned. In *International Congress on Tools for Teaching Logic*, pages 131–140. Springer, 2011. https://link.springer.com/chapter/10.1007/978-3-642-21350-2_16.

[7] Tobias Macey and Thomas Hatch. Making complex software fun and flexible with plugin oriented programming. Podcast.__init__, 2019. https://www.pythonpodcast.com/plugin-oriented-programming-episode-240/.

[8] Raymond M. Smullyan. *First-order logic*. Dover, 1995.

[9] Adam Tauber. exrex, Jun 2017.