AXolotl: A Self-study Tool for First-order Logic

David M. Cerna
Research Institute for Symbolic Computation (RISC),
Institute of Formal Methods and Verification (FMV),
Johannes Kepler University, Linz, Austria,
Email:David.Cerna@risc.jku.at

***Abstract*—We introduce AXolotl, a self-study aid designed to guide students through one of the more strenuous topics in a typical course on formal logic, that is quantification. While the formal inferences for quantification are quite simple and are thus easy absorbed by students, the properties these simple inferences entail when used in conjunction with the other inferences usually leaves students flummoxed. Our software isolates certain aspects of quantification which are particularly difficult for students, namely term unification. This is done by restricting the proof and formula structure allowed in a derivation. Here we illustrate the initial results of an ongoing investigation, which includes the development of more comprehensive self-study software.**

***Keywords*-Education; Logic; Quantification; Unification**

I. Introduction

Logic has, over the past century, moved from an esoteric subject studied and used, in its abstract form, by the few, to a subject pervasive in the modern world. This pervasiveness is mostly due to ubiquity of computing technology within modern society, the foundations of which rest in the realm of mathematical logic. With this in mind, one would expect formal logic to encompass a significant portion of undergraduate computer science education, however, this is unfortunately not the case [12]. Part of this problem seems to be the tremendous gap between logic studied in the abstract form and its application within computing technology.

While many can see the importance of understanding *Boolean algebra* when one wants to write a correct if statement, the formal theory of propositional logic seems far removed from issues like program failure or buggy software, yet it is precisely in these situations where it has helped through verification [6] and model checking [21] techniques. Helping students see these connections is obviously beyond the scope of this work, rather we focus on the problem of helping students feel comfortable with formal systems and the required manipulations therein.

We are not the first to notice nor investigate this understanding gap described above. There are even venues dedicated to the topic, such as Tools for Teaching Logic (TTL) and ThEdu. Thus, there have been an assortment of proposed solutions over the past twenty years, however, none of which, to the best of our knowledge, takes the approach to the problem we discuss here. While some software does focus on restricted fragments of formal logic, these fragments are usually too weak or spurious to be any use to the development of understanding of formal systems. A good example of such a self-study aid is the mobile phone app *Quantifiers!* [5]. While, similar to our work, it focuses on quantification, the examples of quantification are spurious, writing the statements without quantification would be significantly easier. Furthermore, the examples are trivial in that term construction is never needed, this tends to be one of the more difficult concepts for students to understand. Other examples of similar software are *Emojic* [9] which completely abstracts logic away and focuses on image-word associations, and Lewis Carroll [18] which is written in Scratch [13] and focuses on natural language syllogisms.

As one may expect from a subject with its roots in philosophy, much of the existing software is aimed at philosophy education. For example, the mobile app *Andor* [10] focuses on the understanding of natural language statements logically. This is one type of exercise which the interactive textbook *Carnap.io* [11] provides. Similarly, Terrance Tao developed an interactive textbook [19] for understanding the logic behind mathematical theorems. Integrating both the more natural language interpretation with mathematical understanding is tackled by *Lurch*, a mathematical text typesetter [3] with an integrated prover.

While the above outline covers many of the outliers concerning self-study logic education software, the majority, which we have yet to mention, focus on derivation construction and proof assistance. A quite important example of such derivation construction tools is the *Sequent Calculus Trainer* [7] which provides a user-friendly interface for the construction of sequent calculus proofs as well as a hint engine powered by the Z3 [4] smt solver. The already mentioned interactive textbook, carnap.io [11] also includes a proof construction interface, but for natural deduction. However, unlike the Sequent Calculus Trainer which has buttons for each rule carnap.io requires free-form text input from the user following a particular style of natural deduction proof representation (i.e. Finch style, Montague style, etc.). This leaves more room for error on the students part and furthermore, such proof representations are not appropriate outside of philosophy education, thus limiting the applicability of the system.

Some other worthwhile mentions are the mobile app *Natural Deduction* [8], *NaDea: Natural Deduction assistant* [20], *The Incredible Proof Machine* [2], and *SPA: Students' Proof Assistant* [16]. Unlike most other mobile apps Natural Deduction provides a natural deduction finch style proof system as well as

This work was funded by the Linz Institute of Technology project LOGTECHEDU.

a theorem prover. There exist mobile logic assistants which include theorem proving technology, for example *Logic*++ [22], but they tend to put less focus on the assisting aspect and more on the proving. NaDea is of similar design as Natural Deduction in that the student can construct proofs and have the system prove the statement for them, however, NaDea can also provide hints to the students, essentially guiding them through the proof. The system is implemented on top of Isabelle [15] and thus benefits from its further development and expressive power. Of Similar design as NaDea and also implemented in Isabelle is SPA which aids students through the process of developing a proof assistant. It is essentially a proof assistant within a proof assistant.

The final self-study software we will discuss is the incredible proof machine which is a proof construction tool with a novel interface design. Rather than constructing traditional proofs users build circuits which match the inference rules. While this is pretty standard with respect to propositional logic, it provides an intuitive interface for more difficult calculi such as Hilbert Systems [14] and the Lambda Calculus [1].

Concerning our contribution to the plethora of existing software, **AX**olotl, we focus on the problem of unification within the sequent calculus framework. The users is given a pair of initial terms and using a set of rules must show that there exists a sequence of rule applications to initial terms which result in a proof tree without open branches. Our rule set consist of formula with a universal prefix and a matrix consisting of an implication with a single Atomic formula in its consequent and antecedent. These restrictions guarantee that application of each inference rule results in a derivation with a single axiomatic branch and a single open/axiomatic branch. Thus, the proof state is extremely simple to represent and term construction can be performed in a calculator-like setting similar to the sequent calculus trainer [7].

The rest of the paper is as follows: In Section II we discuss the background knowledge needed for understanding the educational scenarios addressed by our software. In Section III we discuss our implementation of **AX**olotl and how the software address the educational scenarios discussed in Section II. In Section IV we discuss our plans for extending the addressed educational scenarios and alternative implementations for the software for mobile devices such as tablets and smart phones.

II. Preliminaries

We provide a brief description of a restriction of Gentzen's Sequent calculus for first-order logic which covers the precise logical problems we address. For a more detailed and complete description please see [17]. Let $\mathcal{P}$ be a countably infinite set of predicate symbols each of which has a unique arity $n \geq 0$ and $\mathcal{X}$ be a countably infinite set of variable symbols. Terms $\mathcal{T}$ are constructed from a countably infinite set of function symbols $\mathcal{F}$ each of which has a unique arity $n \geq 0$ together with the variables of $\mathcal{X}$ as follows:

- let $x \in \mathcal{X}$, then $x \in \mathcal{T}$
- let $a \in \mathcal{F}$ such that a has arity 0, then $a \in \mathcal{T}$
- let $t_1, \cdots, t_n \in \mathcal{T}$ and $f \in \mathcal{F}$ s.t. f has arity n, then $f(t_1, \cdots, t_n) \in \mathcal{T}$

Furthermore, by $v(t)$ for $t \in \mathcal{T}$ we denote the variables occurring in t. Concerning formula, we restrict ourselves to the operators $\{\forall, \rightarrow\}$ being that we will not require more complex constructions. Formula $\mathcal{F}_{or}$ are constructed as follows:

- let $P \in \mathcal{P}$ of arity n and $t_1, \cdots, t_n \in \mathcal{T}$, then $P(t_1, \cdots, t_n) \in \mathcal{F}_{or}$.
- let $P(t_1, \cdots, t_n)$, $Q(s_1, \cdots, s_m) \in \mathcal{F}_{or}$ and be atomic, then $P(t_1, \cdots, t_n) \rightarrow Q(s_1, \cdots, s_m) \in \mathcal{F}_{or}$.
- let $\forall x_1, \cdots x_n(P(t_1, \cdots, t_n) \rightarrow Q(s_1, \cdots, s_m)) \in \mathcal{F}_{or}$, $x_0 \in (\bigcup_i (v(t_i) \cup v(s_i))) \setminus \{x_1, \cdots x_n\}$, then $\forall x_0, x_1, \cdots x_n(P(t_1, \cdots, t_n) \rightarrow Q(s_1, \cdots, s_m)) \in \mathcal{F}_{or}$.

We will refer to formula $P(t_1, \cdots, t_n) \in \mathcal{F}_{or}$ such that $\bigcup_i v(t_i) \equiv \emptyset$ as *terminal Atoms* and formula $\forall x_0, \cdots x_n(P(t_1, \cdots, t_n) \rightarrow Q(s_1, \cdots, s_m)) \in \mathcal{F}_{or}$ such that $\bigcup_i (v(t_i) \cup v(s_i)) \equiv \{x_1, \cdots x_n\}$ as *rules*. Sequents are pairs of multi-sets of formula which will be denoted by uppercase Greek letters as follows: $\Pi \vdash \Delta$. We restrict ourselves to so called *implicational sequents* which are of the form $A, \Delta \vdash B$ where A and B are terminal atoms and Δ is a set of rules. The idea behind implicational sequents is that the rules occurring in Δ can be used to transform A and B until they are equivalent. So far **AX**olotl uses a simplified version of implicational sequents which we refer to as P-implicational sequents, that is an implicational sequent where the only predicate symbol occurring is P.

Example 1. *The sequent is a P-implicational sequent:*

$$P(a), \forall x(P(x) \rightarrow P(r(x))) \vdash P(r(a))$$

while the following is only an implicational sequent:

$$P(a), \forall x(P(x) \rightarrow Q(r(x))) \vdash Q(r(a))$$

Given that our concept of well-formed formula is so restricted, we do not need the full expressive power of the Gentzen sequent calculus for first-order logic. The following logical rules suffice:

$$\frac{A, C, \Delta \vdash B}{A, \Delta \vdash B}\,\mathrm{c} \qquad \frac{A, \Delta \vdash B}{A, C, \Delta \vdash B}\,\mathrm{w}$$

$$\frac{A\Delta \vdash C \qquad D, \Delta \vdash B}{A, C \rightarrow D, \Delta \vdash B} \rightarrow \qquad \frac{A, F(t), \Delta \vdash B}{A, \forall x F(x), \Delta \vdash B}\,\forall$$

Given a valid P-implicational sequent S we can construct a proof of S using the above rules. In Figure 1 we provide an example proof of an extension of the first sequent provided in example 1. Notice that while one can apply the rules as one may wish there does exists a particular proof strategy which may be employed to construct a proof of any valid P-implicational sequent S which is as follows:

1) Contract (inference c) the rule R from Δ which will be applied
2) Apply the $\forall$ inference to R exhaustively choosing terms which match either A or B.

$$
\dfrac{
 \dfrac{
  \dfrac{
   \dfrac{\dfrac{P(a)\vdash P(a) \qquad P(r(a))\vdash P(r(a))}{P(a),P(a)\to P(r(a))\vdash P(r(a))}\to}{P(a),\forall x(P(x)\to P(r(x)))\vdash P(r(a))}\forall
   \qquad
   \dfrac{P(r(r(a)))\vdash P(r(r(a))) \qquad P(r(r(a)))\forall x(P(x)\to P(r(x)))\vdash P(r(r(a)))}{P(a),P(r(a))\to P(r(r(a))),\forall x(P(x)\to P(r(x)))\vdash P(r(r(a)))}\to
  }{P(a),P(r(a))\to P(r(r(a))),\forall x(P(x)\to P(r(x)))\vdash P(r(r(a)))}\to
 }{P(a),\forall x(P(x)\to P(r(x))),\forall x(P(x)\to P(r(x)))\vdash P(r(r(a)))}\forall
}{P(a),\forall x(P(x)\to P(r(x)))\vdash P(r(r(a)))}c
$$

Fig. 1. Example proof using the reduced sequent calculus.

3) Apply the $\to$ inference to R resulting in two branches B_1 and B_2.
4) w.l.o.g we assume that B_1 contains the same terminal atom in the antecedent and succedent. Thus, we apply weakening (inference w) to Δ until Δ is empty.
5) Branch B_2 once again contains a P-implicational sequent which we can apply the same procedure to, or B_2 also contains the same terminal atom in the antecedent and succedent and thus we apply 4) to B_2 as well.

This procedure is precisely what is implemented in **AX**olotl, albeit in a more visually pleasing way.

III. Implementation

We implemented **AX**olotl as a stand-alone Java program with an AWT/Swing user interface (Figure 2). An implementation for implicational sequents is possible, though we have only implemented a version for constructing proofs of P-implicational sequents. We plan to develop a mobile version of the software which will include proof construction for the full implicational fragment. The current version of **AX**olotl[1] may be found on the Author's homepage along with a simple manual[2] and example **AX**olotl files[3].

In order to use **AX**olotl for proof construction, an **AX**olotl file must be loaded which includes the allowed function symbols, variables and rules as well as problems associated with the allowed symbols. See Figure 3 for an example **AX**olotl file.

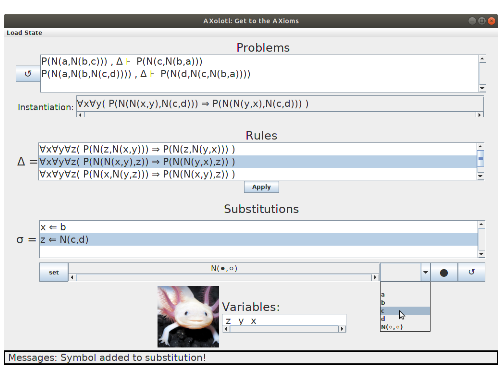

Fig. 2. **AX**olotl user inference

[1]http://www3.risc.jku.at/publications/download/risc_5887/AXolotl.jar
[2]http://www3.risc.jku.at/publications/download/risc_5887/Man.pdf
[3]http://www3.risc.jku.at/publications/download/risc_5887/axFiles.zip

```
Function: N 2
Function: a 0
Function: b 0
Function: c 0
Function: d 0
Variable: x
Variable: y
Variable: z
Problem: N(a,N(b,c)) N(c,N(b,a))
Problem: N(a,N(b,N(c,d))) N(d,N(c,N(b,a)))
Rule: N(z,N(x,y)) N(z,N(y,x))
Rule: N(N(x,y),z) N(N(y,x),z)
Rule: N(x,N(y,z)) N(N(x,y),z)
Rule: N(N(x,y),z)  N(x,N(y,z))
```

Fig. 3. The above **AX**olotl file contains list reversal problems with the necessary rules and signature. Function and Variable input lines must always come before the problem and rule lines.

We can relate the user interface presented in Figure 2 with the procedure outline in Section II which we refer to as Linear Proof Search (or LPS). We can ignore the contraction and weakening aspects of LPS being that they are implicit. However, what is important to take notice of concerning the procedure outlined in Section II is that after one iteration, either both branches are axioms or one of the branches is once again a P-implicational sequent. While this is not necessarily the case being that one can instantiate the quantifiers in such a way that it forces non-axiomatic branching, any valid P-implicational sequent is provable without such double branching. This is precisely the reason we chose this fragment. We enforce this property of LPS within **AX**olotl by only allowing rule application when the instantiated rule's antecedent or succedent matches one of the terminal atoms. Furthermore, a beneficial property of LPS concerning user inference design is that the proof state can be represented by a single P-implicational sequent from which one can easily undo the last rule application.

Notice that the above paragraph entails that the implication inference also becomes implicit within the **AX**olotl-LPS framework being that once a rule is fully instantiated it becomes clear how it ought to be applied and whether or not it can even be applied. Thus, the only part of the proof state a student has to manipulate is the instantiation of the rules, what is referred to as the substitution set in Figure 2. By selecting a particular rule form the rule list and a set of substitutions in the substitution set, an instantiated form of the rule is displayed in the instantiation window below the problems list. Thus, prior to applying the rule a student can visually check if the application is sensible or not. Notice, that unlike LPS as outlined in Section II, quantifiers need not be instantiated in a particular order, this is an artifact of the sequent calculus and partially due to eigenvariable conditions which do not exists in our restricted logic.

In order to construct substitutions students must construct terms using the drop down menu under the substitution set. This menu is highlighted in Figure 4. The black • denotes the next term to be filled in by the student and the white dot ◦ denotes subsequent holes to be filled. Once the the term in the display to the left of the drop down menu is void of •s and ◦s one can choose a variable and create a substitution. We chose this method of input rather than allowing the students to enter free form text because it minimizes the number of errors on the student's part, especially errors which have nothing to do with the logical problem they are trying to solve.

Beyond simple examples similar to those presented in Section II and Figure 2, The entirety of propositional logic, including propositional proof trees may be encoded within **AX**olotl. Figure 4 illustrates the application of modus ponens within the propositional Hilbert system for classical logic.

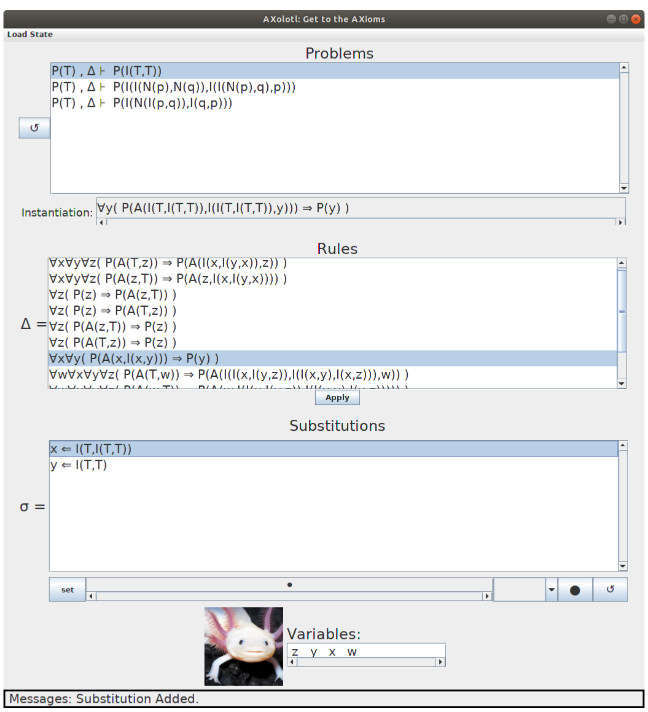

Fig. 4. Propositional Logic in **AX**olotl.

IV. Future work

In this work we introduced the self-study aid **AX**olotl which addresses problems students commonly face when first working with quantifiers. Currently **AX**olotl is released as a java program which can be downloaded from http://www3.risc.jku.at/publications/download/risc_5887/AXolotl.jar. We plan to strengthen the expressive power and extend the proof situations it addresses in future versions, as well as design a smart phone app based on the current implementation. The app version will have a touch interface further simplifying user interaction.

References

[1] Hendrik Pieter Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Perspectives in logic. Cambridge University Press, 2013.

[2] Joachim Breitner. Visual theorem proving with the incredible proof machine. In Jasmin Christian Blanchette and Stephan Merz, editors, *Interactive Theorem Proving*, pages 123–139, Cham, 2016. Springer International Publishing.

[3] Nathan C. Carter and Kenneth G. Monks. Using the proof-checking word processor lurch to teach proof-writing. 2014.

[4] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[5] Spoon Developers. QUANTIFIERS! - a mathematical logic game, 2018.

[6] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, July 2008.

[7] Arno Ehle, Norbert Hundeshagen, and Martin Lange. The sequent calculus trainer with automated reasoning - helping students to find proofs. In *Proceedings 6th International Workshop on Theorem proving components for Educational software, ThEdu@CADE 2017, Gothenburg, Sweden, 6 Aug 2017.*, pages 19–37, 2017.

[8] Jukka Häkkinen. Natural deduction. proof generator. proof checker., 2018.

[9] Stefan Haustein. Emojic, 2017. Repository location: https://github.com/stefanhaustein/EmojiC.

[10] Matthias Jenny. Andor: Learn logic, 2018.

[11] Graham Leach-Krouse. Carnap: An open framework for formal reasoning in the browser. In *Proceedings 6th International Workshop on Theorem proving components for Educational software, ThEdu@CADE 2017, Gothenburg, Sweden, 6 Aug 2017.*, pages 70–88, 2017.

[12] J. A. Makowsky and A. Zamansky. Keeping logic in the trivium of computer science: A teaching perspective. *Form. Methods Syst. Des.*, 51(2):419–430, November 2017.

[13] Majed Marji. *Learn to Program with Scratch: A Visual Introduction to Programming with Games, Art, Science, and Math*. No Starch Press, 2014.

[14] J.D. Monk. *Mathematical Logic*. 1976.

[15] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[16] Anders Schlichtkrull, Jørgen Villadsen, and Andreas Halkjær From. Students' proof assistant (spa). In Pedro Quaresma and Walther Neuper, editors, Proceedings 7th International Workshop on *Theorem proving components for Educational software,* Oxford, United Kingdom, 18 july 2018, volume 290 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–13. Open Publishing Association, 2019.

[17] Gaisi Takeuti. *Proof Theory*, volume 81 of *Studies in logic and the foundations of mathematics*. American Elsevier Pub., 1975.

[18] Terry Tao. Lewis carroll, 2012.

[19] Terry Tao. Qed, 2018.

[20] Jørgen Villadsen, Andreas Halkjær From, and Anders Schlichtkrull. Natural deduction assistant (nadea). In Pedro Quaresma and Walther Neuper, editors, Proceedings 7th International Workshop on *Theorem proving components for Educational software,* Oxford, United Kingdom, 18 july 2018, volume 290 of *Electronic Proceedings in Theoretical Computer Science*, pages 14–29. Open Publishing Association, 2019.

[21] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, Apr 2003.

[22] Yale Weiss. Logic++, 2016.