Design and Implementation of a Proof Assistant for Natural Deduction

Jorge Pais
Universidad ORT Uruguay
Montevideo, Uruguay
mail@jorgepais.com

Álvaro Tasistro
Universidad ORT Uruguay
Montevideo, Uruguay
tasistro@ort.edu.uy

Abstract— We discuss the design of a proof assistant for the Formal Logic calculus of Natural Deduction based on didactic considerations. We propose to incorporate several features that are not present in the elsewhere available tools, among them the display and edition of derivations as trees, the use of meta-theorems (derived rules) as lemmas, and the possibility of maintaining a set of draft trees that can be inserted into the main derivation as needed. Our assistant checks the validity of each edition operation performed by the user. It has been implemented for propositional logic and (quite satisfactorily) put into practice in courses of Logic for Software Engineering and Information Systems programs.

Keywords: educational software, teaching logic, formal proof.

I. CONTEXT AND MOTIVATION

A dear aim in Software Engineering education is that students be able to argue why their programs will work under all possible circumstances. This is all the more important in the context of a discipline still too much tied to practices in which programs’ conformance to their specifications comes up as just a (sometimes quite bold) conjecture and, consequently, software construction becomes essentially a trial-and-error process never to reach plain assurance. Therefore it is important that students of such an educational program have the opportunity to practice proof and, in particular, proof of correctness of computer programs.

Proof is of course the mathematical activity of arriving at knowledge deductively, i.e. starting off from postulated, supposed or self-evident principles and performing successive inferences, each of which extracts a conclusion out of previously arrived at premises. In the application of this practice to programming we have among the first principles the so-called semantics of programs, which allows to understand program code and thereby to know what each (part of the) program actually computes. This makes it possible in principle to deductively ascertain that the computations carried out by the program satisfy certain properties. Among these properties are input-output relations or patterns of behavior that constitute a precise formulation of the so-called functional specification of the program or system at hand.

Now, (Formal) Logic is the theory of the activity of proving. It has, since the very beginning, striven to put forward the rules that govern such activity, i.e. rules of correct reasoning. And, in its contemporary mathematical variety, it has formulated several artificial languages into which to frame the mathematical activity.

There should therefore, according to this aim, be a language for expressing every conceivable mathematical proposition and also a language (or, as it has been called, system) for expressing proofs, so that a proposition is provable in this language (system) if and only if it is actually true. This latter good property of the system is called its correctness. This kind of research began in 1879 with Frege [3] with the purpose of making it undisputable whether a proposition was or not correctly proven. Indeed, the whole point of making the languages artificial was to make it possible to automatically check whether a proposition or a proof was correctly written in the language. That is to say, the proofs accepted were to be so on purely syntactic (i.e. formal) grounds and, given the good property of correctness of the language, that would be enough to ensure the truth of the asserted propositions. Frege’s own language turned out to be not correct and, for that reason mainly, shortly after its failure the whole enterprise of formal logic received a different direction, namely that of studying the artificial languages as mathematical objects in order to prove their correctness by elementary means (And this new course was actually also destined to failure).

The overall outcome is nevertheless very convenient from an engineering viewpoint. We can resume Frege’s programme, only that nowadays we are provided with technology that makes it feasible to develop formal proofs semi-automatically. The proof systems are still reliable although not complete, i.e. not every true proposition will be provable. But again, this is no harm in practice and the systems are perfectly expressive from an engineering perspective. All these advances allow us to define a whole discipline within Software Engineering, namely Formal Methods, as the one consisting in the use of formal languages and related tools for expressing specifications and carrying out proofs of correctness of programs. These proofs are, at minimum, automatically checked as explained above so that their correctness is ensured. Further, the automatic tools can offer facilities to help developing the proofs.

This adds a new dimension to the significance of Logic in Software Engineering education. It not only promotes, by virtue of its theoretical nature, reflection on the otherwise natural and spontaneous activity of reasoning, thereby providing foundation, reassurance and further intellectual tools.
It also becomes a framework within which quite concrete computerized tools, very relevant for the professional practice, are formulated and understood.

At ORT University, we teach a course of Formal Logic for the Software Engineering and the Information Systems programs, based on the above described premises. It is a classical course, comprising Propositional and First-Order Logic, as well as a first part dealing with Induction and Recursion. The formal system of proof that we employ is the Calculus of Natural Deduction [5]. This system was devised with the aim of closely mimic common practice in informal (i.e. natural language) mathematical proof. It does so essentially by allowing to introduce temporary additional assumptions which are used in well-defined fragments of the proofs and then discharged. Further, the rules of inference (with possibly one exception) are organized around the logical constants (connectives and quantifiers) and are of one of two classes in each case, i.e. a rule is either:

- An introduction rule stating how a formula having the logical constant as principal operator can be derived in a direct, canonical manner, or
- An elimination rule, stating how such a formula can be employed to derive further consequences from it.

Rules have in general several premises and always one conclusion and therefore the formal proofs (technically called derivations) are naturally arranged as trees. It is natural to read inference as proceeding from the premises above to the conclusion below, and therefore the trees are written with the root, which is the conclusion of the theorem, at the bottom, and the initial assumptions at the leaves on the top. The use of the rules inside a tree follows a quite characteristic pattern: Reading the tree from the top, one first applies elimination rules to obtain information from the given premises in a phase that could be called of analysis. At some level during the derivation one starts a synthesis forming new conclusions from the data obtained, by employing the introduction rules. We refer to [6] for a full presentation of the system.

We find two connected problems in the students’ process of learning to use this system:

- The student learns the rules as simple formal transformations, without really apprehending their logical meaning.

- As a consequence, the student tends to apply the rules as if they were pieces of a puzzle without having a real idea of the state and direction of the proof. It is expected that the mere “plugging” of the inferences rules will eventually converge to a complete proof.

This deprives students of generating the desired mental structures, i.e. the understanding of the proof that actually convinces oneself that the conclusion really follows as a logical consequence of the premises. We are convinced by experience that this obstacle cannot in general be overcome without adequately guided, extensive practice.

Therefore we have identified the need to promote solving natural deduction exercises without neglecting the theoretical aspects necessary for understanding the operation of the calculus. We have noticed two aspects which play against the willingness of the student to work with exercises:

- Proofs constructed on paper tend to be burdensome, there is no linearity and it is difficult for the students to adapt. They have to erase various steps in order to turn back on what was being done and write on different parts of the sheet in order to make annotations or sub-proofs. Aesthetic factors make it difficult to visualize the entire proof.

- In general the students are less motivated to work on paper than on machine, especially at the early stages of their professional education which is when they usually take this course. They hope to work frequently with computers, which is certainly one of the main reasons that drove them to choose the computing programs.

In virtue of what has been said, we set ourselves to incorporate a computerized tool to assist in the practice of formal proof using natural deduction in our Logic course. We identified a set of requirements for such a tool, which we explain presently.

II. REQUIREMENTS

A. Tree-like representation of the proofs

The graphical representation of the proof must match the way it is presented in the course, namely the natural representation as tree which has already been explained. Figure 1 displays one such tree. We write on the top the theorem that is being proven Notice the marking of discharged temporary assumptions.

\[
\begin{array}{c}
p \land q \rightarrow r, r \rightarrow s \vdash p \land q \rightarrow s \\
\end{array}
\]

![Figure 1 - Tree like representation of a proof](image)

This point turned out to be of great importance, since similar systems already available offered representations mostly based on Fitch diagrams, which basically consist on the construction of embedded rectangles to represent sub-proofs. More information can be found in [7].

Therefore, in addition to adjust to the course and facilitate their acceptance by teachers and students, the representation becomes a differentiating factor compared to other programs, and one that we apprise as advantageous. This doesn’t mean that tree-like representation is better than Fitch representation, but best fits the way Natural Deduction is presented in our course.

B. Use of Meta-variables

Formulas handled by the system may contain statement or term variables, or variables that represent other logical formulas, which we call meta-variables.

\[
(\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma \quad \text{with } \alpha, \beta, \gamma \in Form \quad (1)
\]
For example, in (1) Form represents the set of the formulas of propositional logic, and α, β and γ represent arbitrary formulas of Form, therefore they are meta-variables.

The use of meta-variables will, as explained below, allow using completed proofs (lemmas) as justification of steps in other proofs.

C. Lemma Instantiation

A lemma is defined as a completed proof of a theorem, in general expressed in terms of meta-variables. Lemmas can be used during the construction of any other proof as if they were new inference rules. (They are in fact derived rules.)

For example:

\[
given: \quad \alpha = p \land q, \beta = r \rightarrow q \\
\text{and the lemma:} \quad \neg \alpha \vdash \alpha \rightarrow \beta
\]

The result of instantiating the lemma using the given formulas is:

\[
\neg (p \land q) \vdash (p \land q) \rightarrow (r \rightarrow q)
\]

The system must be able to generate an instance of the lemma based on the current status of the proof, minimizing the required information from the student. However, the student is expected to evaluate a priori whether the instantiation of the lemma is possible in the context without giving an explicit correspondence between metavariables and formulas.

D. Lemma Library

There should be a repository of lemmas, from which the student can easily select any one to be used during the development of a proof. Lemmas are saved so that the student can see their proofs as originally constructed.

E. Simulate paper-and-pencil construction experience

By itself the experience of building proof trees is rather complex for the students: they are at the same time dealing with a specific problem and a new instrument. Some important points arise:

- When working on paper, the student uses a large area to draw the proof tree, makes annotations, constructs different sub-proofs that could fit into the desired one, does a lot of editing and deleting.
- Paper has its drawbacks. A priori it is not known what the size of the proof tree will be. In general it is necessary to delete some parts of the tree because they correspond to wrong paths or, as several sub-proofs were generated, they must be joined together or rewritten to fit into the original proof.

If the objective is to help the students focus on the construction of the proof, the assistant should start presenting an interface similar to a sheet of paper and just improve over its disadvantages. Otherwise learning to use the program will be more difficult than learning the natural deduction calculus.

F. Assistant, not automatic prover

The tool should assist the student in the construction of the proofs, not complete them for him. This concept manifests itself in the following ways:

- In as much as the proof is incomplete, the system will indicate which sub-proofs have yet to be completed. At the beginning, the desired conclusion and the assumptions are shown and the indication is that the whole proof is missing.
- Then a mechanism for constructing the proof backwards should be available, i.e. one that allows to select a yet-to-be-proven conclusion and an inference rule and educe an appropriate set of premises that lead to the desired conclusion via the rule selected. These premises become in turn new conclusions to be proven.
- The system will check the local validity of the application of each inference rule, but
- The system does not check that the path or general strategy chosen to build the proof is valid in any sense.
- There should be a way to construct derivations in a forward manner, i.e. working from assumptions to conclusions.
- And, corresponding to this and to the use of lemmas, there must be a way to insert completed proofs in place of any pending sub-proof of another tree.

III. ANDY: ASSISTANT FOR NATURAL DEDUCTION

Not having found the above described features in any tool publicly available (see e.g. [1, 2, 6, 8, 9, 10]), we decided to carry out a detailed design and implementation of (yet another, but still new) proof assistant for natural deduction. Our first version deals only with propositional logic. It is called Andy (version 0.)

A. Implementation

We established that the system had to be written in a functional language, because it is more natural in these languages to describe the calculus. We also needed a language that allowed us to create a functionally complex graphic interface and yet simple and familiar for the student.

Haskell seemed a natural choice because it is widely used at the University and we had a lot of experience working with it. However it was complicated to satisfy the graphical requirements. So we decided to take a chance on a relatively new functional language (actually a multi-paradigm language) called F# to develop the core of the system (data structures, inference rules, etc.) and C# to create the graphical interface. As both languages compile to common code that runs on Microsoft’s CLR, it was very simple to interoperate between what was written on each language.

Andy is a Windows Forms application; however we have considered the need for portability to non-Windows systems, so it was designed with highly decoupled 3-tier architecture. This allows us to modify the presentation tier while...
maintaining the logical tier, to create for example a web application.

Andy is a non-commercial application and can be downloaded and used freely from our research group’s web site http://fi.ort.edu.uy/innovaportal/v/3641/5/fi.ort.front/inicio.htm

B. Features

1) Proofs as Trees

An essential part of the system is the graphical representation of the proofs as trees. Each node of the tree is a logical formula. Particularly, the root is the conclusion of the proof and the leaves are the hypotheses. On the latter there is an important distinction: there are hypotheses that appear in the statement of the theorem and are available throughout the construction of the entire proof, and there are temporary hypotheses or assumptions that emerge from the process of construction and have restricted validity within the branch where they are generated. Assumptions arise from the definition of inference rules. For example the introduction rule of implication is defined as follows:

\[ I \rightarrow \frac{[A]}{B} \quad A \rightarrow B \]

A temporary hypothesis is generated and can be used on the proof from the point where the rule is applied. A substantial difference between the two types of hypothesis is that in the leaves of a complete proof there must be either theorem hypothesis or discharged temporary hypothesis. The discharge of a temporary hypothesis involves linking it to the rule that generated it. This functionality is not automatic; the student has to make explicit the link between the temporary hypothesis and the rule. Each and every used temporal hypothesis has to be discharged in order to complete the proof. Practice renders evident that it is difficult for students to understand the mechanism of discharge so we decided not to automate this. So the system acts in these cases only as a validator, checking that the discharge is correctly performed.

The display of the proof tree uses a color and symbol code to show the state of the proof at each node. The green nodes represent completed parts of the proof, which means that the sub-proof determined by the node is completed and no further action is required. On the other side, red leaves correspond to assumptions made that have to be discharged. Finally, leaves with missing proofs are represented with a question mark on top. This is illustrated here below:

\[
\frac{\frac{\frac{(p \rightarrow (q \rightarrow r))}{(q \rightarrow r)} \quad \frac{(p \rightarrow q)}{p}}{(p \rightarrow q)}}{p} \]

2) Formula line editor

There is an in-line editor for writing down and having checked every formula input to the system.

3) Drag & Drop of proofs

A feature that we consider extremely important is that the construction of the proof on the system must simulate the construction on paper, while at the same time improving into the latter’s disadvantages. To do this, it was essential to implement the trees as objects that could be moved along a proof panel.

At any given time during the construction of a proof a main tree coexists with several sub-proofs. These work as drafts, i.e. proof fragments used to focus on a specific part of the main proof, surely because they carry a complexity that makes it difficult to manage the main proof as a single piece. This allows the student to divide the problem and attack it from different points, later putting together the intermediate results.

Draft sub-proofs can be dragged through the panel or deleted in their entirety. When a sub-proof is complete (and under certain restrictions) it can be put together with the main proof by simply dragging and dropping it on one of the incomplete leaves of the main proof. The system checks the validity of the resulting construction.

4) Forward and Backward reasoning

The system allows building a proof using two mechanisms. The first one consists in starting from the conclusion and by applying successive inference rules backwards, finally reaching the hypothesis. The second starts from the hypothesis and finally reaches the conclusion by applying the inference rules in a forward manner. These mechanisms are called backward and forward reasoning respectively.

Students are expected to build proofs by combining these two techniques until they converge at an intermediate level of the proof and complete it. In order to implement these facilities we decided that each inference rule had to have a backward and forward variant, which would be applied automatically as the situation required.

Each rule variant behaves differently:

- Backward rules (see figure 4) are applied on a single leaf of the main tree. The result is a modification of the main tree, where the selected node has one or more child. The application of a rule may generate new assumptions.
- Forward rules involve one or more secondary trees. The result is another secondary tree with a new root and a new set of available hypothesis (see figure 5).

Figure 4 - Backward rule behavior. Question marks represent incomplete parts of the proof.

Figure 5 - Forward rule behavior. Γ represents the set of available hypotesis and α the root of the new tree.

For example, figure 6 represents the application of the forward variant of the implication elimination. The rule is applied on a tree that has an implication formula on the root. To complete the application of the rule another tree has to be selected, with the restriction that its root formula is the antecedent of the first tree’s root implication. The result is a new tree, combination of the selected, with a new set of available hypothesis and a new root formula.

On the other hand, to apply the backward variant of the implication elimination, a formula has to be selected on the main tree. In this case the user has to indicate which the antecedent will be of the constructed implication. After this, two new leaf nodes are created as children of the selected one.

5) Meta-theorems and instantiation

The system allows using in formulas both simple propositional variables (which stand for truth values) and formula variables (meta-variables) which stand for formulas of the propositional logic language. The latter allows us to create generic proofs and reason about the universe of the propositional formulas. Moreover, generic proofs can be instantiated into concrete proofs (the ones that only have propositional variables) and used as part of other proofs as if they were rules of inference. If there exists a substitution that matches each meta-variable on the generic proof with a formula on the concrete proof, then the meta-variables are instantiated.

For example, given the following generic proof:

\[(A \land B) \rightarrow (B \land A)\]

There exists a substitution S that marks the following proofs as complete:

\[(p \land q) \rightarrow (q \land p) \text{ using } S = \{(A,p),(B,q)\}\]

And

\[(r \rightarrow s \land r \lor s) \rightarrow (r \lor s \land r \rightarrow s) \text{ using } S = \{(A,r \rightarrow s),(B,r \lor s)\}\]

Andy uses this mechanism to complete two functionalities: lemma instantiation and secondary tree join.

Lemma instantiation allows a user to use a completed generic proof as a new inference rule in another proof. When a generic proof is completed, the user can save it on a lemma library, so that it remains available to use in future proofs. Then, during the construction of a proof the user can invoke a lemma to complete the proof of a leaf on the main tree. Andy automatically finds the most suitable substitution based on information about the formula on the leaf and the available hypothesis. After that, the instantiated lemma is attached to the main tree and the sub-proof is marked as complete.

Secondary tree join is one of the main features that makes Andy similar to working with paper and pencil. The user can build a secondary proof anywhere inside the proof panel and then use it to complete the proof of a leaf on the main tree. Based on the concept of divide & conquer, the user can divide the proof in several easier sub-proofs, work with them and then put them back together to complete the main proof. The secondary proof does not have to be complete in order to join it with the main tree but there are certain restrictions that check that the proof can be completed after the join by considering the available hypothesis at the moment.
IV. CONCLUSIONS AND FURTHER WORK

We have designed, based on didactic considerations, an assistant for carrying out formal proofs in the calculus of natural deduction. The current implementation is for propositional logic. The main novelties of the system with respect to those available elsewhere are: the tree display of the proofs, the possibility of storing and instantiating lemmas expressed in terms of meta-variables, the possibility of employing forward as well as backward reasoning, and the possibility of maintaining a set of draft trees that can be dragged and dropped on the main proof tree.

We have employed the assistant in some instances of our course. The results concerning the performance in tests of students that used the tool have shown only a slight improvement which we dare not deem significant. But we have also carried out extended interviews with several such students, detecting a large agreement in the following points:

- The assistant makes it easier and (more) appealing to experiment with the natural deduction calculus and it is quite satisfactory as a tool.
- It is very convenient that the assistant checks the correctness of the proofs, since that gives the student confidence and independence from the teachers.
- Both backward and forward strategies are useful, although there is a clear preference for the backward method.
- No great use of the lemma facility has been done.

The two first points only confirm that formal proof is an activity which lends itself naturally to be carried out in a computerized environment. The third observation originates in the fact that, as the method is presented in the course, a general strategy is promoted which consists in starting backwards using introduction rules and turning to forward reasoning (beginning to apply elimination rules to the available assumptions) when no more introduction rules can be used (or when some might be applicable, but it is not convenient to do so.) Many students find it difficult to dynamically perform this switch, preferring instead to uniformly proceed backwards. Finally, the fourth point above is related to the fact that only small size exercises which do not require or favor the use of derived rules are presented in the current version of the course. This is likely to change, in particular because of the availability of the assistant.

We plan to extend the tool to incorporate first-order logic. The challenge is of course to maintain the facilities of our current version, particularly the use of meta-variables and lemmas. Notice that using meta-variables in the first-order version requires to deal with side-conditions of the form \( x \text{ not free in } \) formulas, for term variables \( x \). We plan to attack this by employing the systems of nominal syntax [4]

V. REFERENCES