Journal of Universal Computer Science, vol. 19, no. 11 (2013), 1570-1596
submitted: 15/12/12, accepted: 28/5/13, appeared: 1/6/13 © J.UCS

Proof Assistant Based on Didactic Considerations

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

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

Abstract: We consider some issues concerning the role of Formal Logic in Software Engineering education, which lead us to promote the learning of formal proof through extensive, appropriately guided practice. To this end, we propose to adopt Natural Deduction as proof system and to make use of an adequate proof assistant to carry out formal proof on machine. We discuss some necessary characteristics of such proof assistant and subsequently present the design and implementation of our own version of it. This incorporates several novel features, such as 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. The assistant checks the validity of each edition operation as performed. So far, 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
Categories: L.3.0, L.3.1, L.3.6, L.3.8

1 Context and Motivation

1.1 Logic in Software Engineering Education

It ought to be a dear aim in Software Engineering education that students be able to argue why their computer programs will work under all possible circumstances. The importance of such endeavour is best appreciated when we take into account that the discipline itself is 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. We can and must overcome this state of affairs and therefore turn it into a requirement that students of a Software Engineering program have the opportunity to learn and 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 *semantics* of programs, which allows to understand program code and thereby to know what each 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.

Formal Logic, on the other hand, is the theory of the activity of proving. As such it has, since the very beginning, striven to put forward the rules that govern such activity, i.e. the rules of correct reasoning. And, in its contemporary mathematical variety, it has formulated several artificial languages into which to frame the (deductive) practice of Mathematics. According to such scientific programme, there should be a language for expressing every conceivable mathematical proposition and also a language (or, as it has been called, *proof system*) for expressing proofs, so that a proposition is provable in the proof system if and only if it is actually true. This latter good property of the proof system is called its *correctness*.

This kind of research was initiated by Frege [see Frege 1967] with the purpose of making it undisputable whether a proposition was or not correctly proven. Indeed, the whole point of making the aforementioned languages artificial was to make it possible to *automatically* check whether a proposition or a proof was correctly written in the corresponding language. In particular, the proofs accepted were to be so on purely syntactic (i.e. *formal*) grounds which, given the good property of correctness of the proof system, would be enough to ensure the truth of the asserted propositions. Frege's own system turned out to be not correct and, for that reason mainly, shortly after its failure was discovered, mathematical logic received a different direction – namely *metamathematics*, i.e. the study of the languages and systems of logic as mathematical objects in order to prove their correctness by elementary means. Now, the paramount result of such research came as a shocking blow to the whole programme as conceived by Frege and onwards, as Gödel [Gödel 1931] proved that no sound formal proof system could capture (i.e. prove) all the propositions actually true in school arithmetic or any more complex mathematical theory.

The overall outcome is nevertheless very expedient from a Software Engineering viewpoint. Computing Science has developed technology that enables fast electronic computers to perform the automatic checking of the well-formation of strings of symbols with respect to the rules of formal languages --in particular, of course, of the formal languages and systems of logic. Moreover, the actual composition of correct expressions in a formal language can be aided by appropriate software, which we know how to construct. When the formal language in question is a proof system, this latter facility amounts to semi-automatic theorem proving. All this makes a huge difference as to the actual usability of the languages and systems of logic. It is true that Frege himself, as well as Russell and Whitehead [Whitehead and Russell 1913] , wrote down by hand extensive mathematical tracts in their own formalisms, but those stand up until today as gigantic tours-de-force not to be repeated. Computers and Computing Science have made it possible to use formal languages and systems of logic as actual, day-to-day, engineering tools. Indeed, a piece of software that is able to check the correctness of proofs of propositions that express the conformance of given programs to given specifications provides certainty about the logical (functional) correctness of those programs and has appropriately been named a *verifying compiler* [Hoare 2003]. It fulfills a kind of dream that could be summed up in a motto like: *if it compiles, it works.* On the other hand, the practical usability of these systems is not affected by their incompleteness, i.e. by the fact that not every true proposition will be provable. The systems are generally perfectly expressive from an engineering perspective and are reliable, i.e. they do not accept defective proofs.

All these advances allow us to define a discipline within Software Engineering, namely *Formal Methods*, as the one consisting in the use of formal languages and systems 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. On the one hand, it 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. But also, because of its role as a foundation of the formal methods of software construction, it becomes a framework within which quite concrete computerized tools and methods, very relevant for the professional practice, are formulated and understood.

1.2 Use of Natural Deduction

At our university we teach a course of Formal Logic for the Software Engineering and the Information Systems programs, based on the above described premises. The course takes place in the second semester of the program of studies, which is 5 years long for Software Engineering and 4 years long for Information Systems. The students have previously taken courses on (Object Oriented) Programming and Computer Organisation, plus, in the case of Software Engineering, Integral and Differential Calculus and Linear Algebra.

The Logic course is 16 weeks long, with 5 hours of lecture and exercises each week. It starts with the study of Induction and Recursion, which comprises the inductive definition of sets and the corresponding principles of proof by induction and of definition of functions by recursion. This allows on the one hand to establish a basis for the treatment of languages of which the languages and systems of Logic are particular cases, so that we can later employ already known principles for proving properties or defining transformations on the logical systems at hand. On the other hand, we study very general recursive data structures (sequences and trees) and do quite a bit of recursive (functional) programming and informal proof of program correctness.

Next comes the classical contents of a Mathematical Logic course, i.e. Propositional and First-Order Logic, each with its corresponding syntax, semantics and proof system. The latter topic introduces of course the idea of *formal* proof, i.e. of proof carried out within a system of rules. We insist in the actual practice of formal proof, much more than in the metamathematical study of the systems. We stress on the resemblance between a formal system of proof and a programming language, especially as to the restrictions imposed on the concrete forms of expression that are allowed. Mainly because of our preference for actual use over metamathematical analysis, we employ the Calculus of Natural Deduction [Gentzen 1935]. This system was devised with the aim of closely mimicking common practice in informal (i.e. natural language) mathematical proof, which is achieved essentially by allowing rules that make use of temporary additional assumptions. For instance, to prove a formula that states an implication A→B it is possible to use a rule that allows to assume A in order to prove B from it, just as in common practice. Specifically, this rule states that if one possesses a proof of B which depends on the assumption of A, then one can infer that A→B (i.e. that A implies B), thereby obtaining a proof which does not anymore depend on the aforementioned assumption. This assumption is said to be *discharged*, i.e.dispensed with in the newly formed proof.

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 in question 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.

For instance, the rule described above is the introduction rule for the implication →. The elimination rule for this connective is

$$\frac{A \to B \quad A}{B}$$

which states that from A→B one can infer B provided A is also proven.

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 [van Dalen 2008] for a full presentation of the system.

1.3 Use of Computerized Assistant

It is for us just natural, given the analysis exposed above in 1.1 that the students have at disposal a computerized assistant for carrying out their formal proofs, just as they use an implementation of a programming language to learn and do their programming.

Indeed, formal 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. Also, 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 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. And, finally, the fact that the assistant will check the derivations for correctness provides reassurance and motivates the students to work on their own.

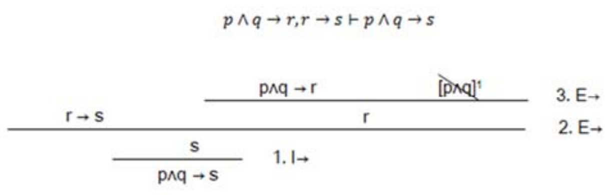
We believe this last point is very important, because extensive practice is necessary for the students to make progress in the process of learning formal proof, particularly with the Natural Deduction system. They usually start at a stage in which they tend to apply the rules as if they were pieces of a puzzle without having a real idea of the state and direction of the proof. It is expected that the mere plugging of the inference rules will eventually converge to a complete proof. That is to say, they view the rules as bare formal transformations without really apprehending their logical meaning. It is practice - at first accompanied by adequate guide, but stepwise growing autonomous - which eventually provides sense and discipline to the formal manipulation.

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 in Section 2. Given that we could not find these requirements implemented in any of the various tools available elsewhere, we decided to design and implement an assistant of our own, which we describe in Section 3. In Section 4, we survey the current state of the art of similar systems, and finally, in Sections 5 and 6 we expose conclusions and further work.

2 Requirements

2.1 Tree-like Representation of the Proof

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. [Fig. 1] displays one such tree. We write on the top, above the tree, the judgement that is being proven. Notice the marking of discharged temporary assumptions.



$$p \wedge q \to r, r \to s \vdash p \wedge q \to s$$

$$\dfrac{\dfrac{r \to s \qquad \dfrac{p \wedge q \to r \qquad \cancel{[p \wedge q]^1}}{r}\ 3.\ E\to}{s}\ 2.\ E\to}{p \wedge q \to s}\ 1.\ I\to$$

Figure 1: Tree-like representation of a proof

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 about Fitch diagrams can be found in [Huth and Ryan 2004].

We prefer trees because they arise as the natural way of composing rules of inference as these are usually presented, i.e. with the premises on top of a bar which represents the act of inference of the conclusion appearing below.

In addition to adjust to the course and facilitate its acceptance by teachers and students, the representation becomes a differentiating factor compared to other programs, and one that we appraise as advantageous.

2.2 Use of Meta-variables

Formulas handled by the system may contain both plain (object level) propositional variables and variables that range over the logical formulas, i.e. meta-variables.

$$(\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma \text{ with } \alpha, \beta, \gamma \in Form \quad (1)$$

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.

2.3 Lemma Instantiation

A lemma is defined as a completed proof. In general, one is interested in theorem-schemata which stand for infinite families of concrete theorems and can be expressed by employing meta-variables. One simple example is the following:

$$\neg\alpha \vdash \alpha \to \beta.$$

The symbol $\vdash$ separates assumptions (hypotheses) from conclusion (thesis) and therefore the above expression can be read: *From $\neg\alpha$ it can be derived in Natural Deduction that* $\alpha \to \beta$. Now this holds for any formulas α and β whatsoever, and hence we are in presence of an infinite family of concrete theorems or, as said before, a theorem-schema. For example, given the assignment

$$\alpha = p \wedge q\,,\ \beta = r \to q\ ,$$

the result of accordingly instantiating the previous lemma is the concrete theorem:

$$\neg(p \wedge q) \vdash (p \wedge q) \to (r \to q)\,.$$

Lemmas can be used during the construction of any other proof as if they were new inference rules. (They are in fact *derived* rules.) We require that the system supports this feature and generates an appropriate instance of the lemma to be employed, based on the current status of the proof wherein it is to be applied and minimizing the information to be supplied by the student. The student is of course expected to evaluate a priori whether the instantiation of the lemma is possible in the context at hand, although without needing to give an explicit correspondence between meta-variables and formulas.

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 can be saved in a way such that the student can recover their proofs as originally constructed.

2.4 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.

2.5 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.
- 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.

The possibility of working both in the backwards and forward manners reflects the fact that it is rather natural in this system to proceed in the following manner:

- At first, one goes backwards from conclusion to new conclusions-to-be-proven, generally by selecting appropriate introduction rules. This is carried out until no further rules of this kind can be applied or they would lead to conclusions which are not sensible to try.

- At such point one starts combining the available hypotheses in a forward manner, i.e. going from known premises to further inferred conclusions, trying to obtain the pending results. This is done by applying elimination rules.

Hence the tree is generally constructed first from the root and upwards, until some point which roughly corresponds to the completion of the synthetic part of the proof as depicted above in 1.2. Then one proceeds from the leaves and downwards, constructing the analytic part of the proof. Such order of proceeding looks indeed opposite to the one suggested by the analytic-synthetic structure of the proofs that has been called upon. But then one must keep in mind that the way a proof is presented is generally different from the way it is conceived. And also that the strategy just explained is not claimed to be the only one appropriate to employ but just a generally useful one, especially to start finding one's way about carrying out this kind of constructions. The requirement just exposed is to the effect that such strategy *can* be naturally executed in the assistant.

3 ANDY: Assistant for Natural Deduction

Not having found the above described features in any tool publicly available [Bornat and Sufrin 1997] [Dyckhoff 1987] [McGuire 2007][Halvorsen 2007], 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.)

3.1 Features

3.1.1 Proofs as Trees

An important feature 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: First, there are hypotheses that appear in the statement of the judgement to be proven and are available throughout the construction of the entire proof; we call these *theorem hypotheses.* Secondly, there are *temporary hypotheses*, which are assumptions that emerge from the process of construction and have restricted validity within the branch where they are generated. Temporary assumptions are allowed by certain inference rules. For example the introduction rule of implication is written as follows:

$$I \rightarrow \quad \frac{\begin{array}{c}[A]\\ | \\ B\end{array}}{A \rightarrow B}$$

As already explained, a temporary hypothesis consisting in the formula A is allowed for use in the tree standing above the rule. If one looks only at that tree, then one sees that the conclusion is B and that among the leaves, there are (generally) some labeled A. However, in the complete tree obtained after applying the rule, the conclusion is *A*→*B* and the leaves labelled *A* are marked as discharged, i.e. dispensed with. A substantial difference between the two types of hypothesis is that in the leaves of a complete proof there must be only theorem hypothesis or discharged temporary hypothesis, i.e. each and every used temporary hypothesis has to be discharged in order to complete the proof. 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. 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:

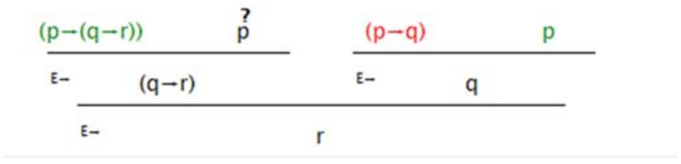
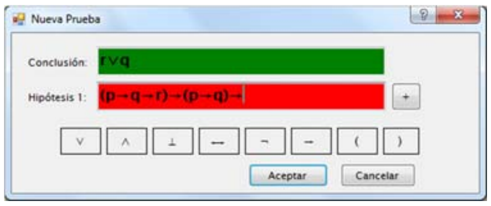


Figure 2: Incomplete proof

3.1.2 Formula Line Editor

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


Nueva Prueba

Conclusión: r∨q

Hipótesis 1: (p→q→r)→(p→q)→

Figure 3: The system validates the syntax of every input formula

The editor validates that the syntax of the input corresponds to a formula of the propositional calculus. However, it does not make any check or validation of the logical validity of judgements introduced, i.e. the hypotheses and conclusion may create a judgment that cannot be proved.

The formula editor is reactive in the sense that it checks for every input if the written string corresponds to a valid formula: if it doesn't the text box background becomes red, otherwise green. The editor is used every time the user has to write a formula during the proof and when introducing a new judgement to be used. In this case, the editor behaves differently, as it allows users to dynamically add any number of new hypotheses.

3.1.3 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 has become essential to implement the trees as objects that can be moved around 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 involve 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.

3.1.4 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 hypotheses. The second starts from the hypotheses and finally reaches the conclusion by applying the inference rules in a forward manner. These mechanisms are called backward (or goal oriented) and forward (or assumption oriented) 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. A problem with this is that each Natural Deduction rule seems to adapt better to backward reasoning or to forward reasoning, but not to both. The challenge here was to devise a general mechanism for each kind of reasoning so that every rule could be used naturally, without generating any complexity for the student.

As a result, each rule variant behaves differently:

- Backward rules [Fig. 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 children. 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 Fig 5].

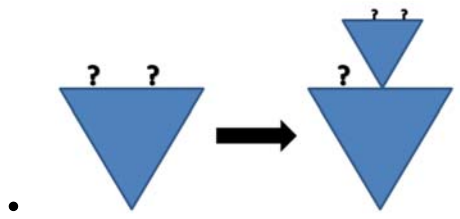


Figure 4: Backward rule behavior. Question marks represent incomplete parts of the proof.

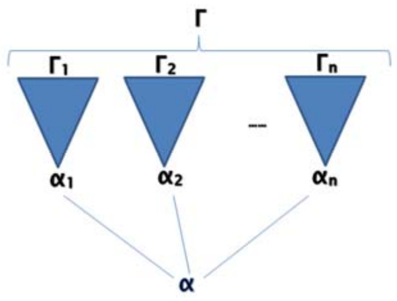


Figure 5: Forward rule behavior. Γ represents the set of available hypothesis and α the root of the new tree

3.1.4.2 Backward Rules

The notation for backward rules is:

$$Formula \xRightarrow{Rule\ (Args)} \frac{NewTree}{Formula}$$

The left side of the arrow shows the state of the leaf of the main tree before applying the rule, while the right side shows the result of the application.

Formula represents the formula at the leaf, *Rule* is the name of the applied rule and *Args* is a list (possibly empty) of arguments which depends on the applied rule. *NewTree* is the resulting tree with Formula as the root.

Notice that the new tree inherits the set of available hypothesis for Formula, possibly adding new ones as a result of the application of the inference rule. These new hypotheses are temporary, and they only exist and can be used on the scope of the sub tree. They are written between brackets above the *NewTree*.

The arguments passed to a rule constitute additional information needed to correctly apply the rule. An argument may be:

- A formula needed to complete the result of the derivation. For example, in the case of the implication elimination (EImpB), the antecedent of the derived implication must be explicit.

- A direction that defines which variant of an inference rule should be applied in rules where the result depends on the chosen operand (Left or Right). For example the Disjunction Elimination (EOrB) needs a direction argument to define whether from the formula A ∨ B the result is A (Right) or B (Left).

For example, given a proof tree with the formula A as a leaf we can apply the Conjunction Elimination Backwards (EAndB).

Notice that to apply the rule we need two arguments: a formula that represents the missing operand of the resulting conjunction, and a direction that establishes the relative position of the initial formula with the conjunction operator.

[Fig. 6] shows the result of the application of the rule for both possible directions. The notation for each rule is shown on the right.

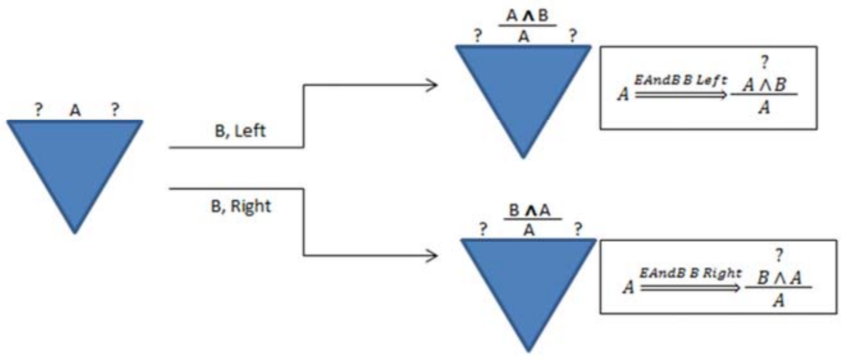

? A ?
B, Left
B, Right
A∧B / A
B∧A / A
$A \xrightarrow{EAndB\ B\ Left} \dfrac{\overset{?}{A \wedge B}}{A}$
$A \xrightarrow{EAndB\ B\ Right} \dfrac{\overset{?}{B \wedge A}}{A}$

Figure 6: Conjunction elimination with both variants

[See Tab. 1] for a complete specification of the remaining inference rules.

Implication Introduction (IImpB)	$A \to B \xRightarrow{IImpB} \dfrac{\overset{[A]}{\overset{?}{B}}}{A \to B}$	Implication Elimination (EImpB)	$B \xRightarrow{EImpB\ A} \dfrac{\overset{?}{A \to B} \qquad \overset{?}{A}}{B}$
Disjunction Introduction (IOrB)	$A \vee B \xRightarrow{IOrB\ Izq} \dfrac{\overset{?}{A}}{A \vee B}$ $A \vee B \xRightarrow{IOrB\ Der} \dfrac{\overset{?}{B}}{A \vee B}$	Disjunction Elimination (EOrB)	$C \xRightarrow{EOrB\ A \vee B} \dfrac{\overset{?}{A \vee B} \qquad \overset{[A]}{\overset{?}{C}} \quad \overset{[B]}{\overset{?}{C}}}{C}$
Conjunction Introduction (IAndB)	$A \wedge B \xRightarrow{IAndB} \dfrac{\overset{?}{A} \qquad \overset{?}{B}}{A \wedge B}$	Conjunction Elimination (EAndB)	$A \xRightarrow{EAndB\ B\ Der} \dfrac{\overset{?}{B \wedge A}}{A}$ $A \xRightarrow{EAndB\ B\ Izq} \dfrac{\overset{?}{A \wedge B}}{A}$
Equivalence Introduction (IEqB)	$A \leftrightarrow B \xRightarrow{IEqB} \dfrac{\overset{[B]}{\overset{?}{A}} \qquad \overset{[A]}{\overset{?}{B}}}{A \leftrightarrow B}$	Equivalence Elimination (EEqB)	$A \to B \xRightarrow{EEqB\ Izq} \dfrac{\overset{?}{B \leftrightarrow A}}{A \to B}$ $A \to B \xRightarrow{EEqB\ Der} \dfrac{\overset{?}{A \leftrightarrow B}}{A \to B}$
Negation Introduction (INotB)	$\neg A \xRightarrow{INotB} \dfrac{\overset{[A]}{\overset{?}{\bot}}}{\neg A}$	Negation Elimination (ENotB)	$\bot \xRightarrow{ENotB\ A} \dfrac{\overset{?}{A} \qquad \overset{?}{\neg A}}{\bot}$
Reductio Ad Absurdum (RAA)	$A \xRightarrow{RAAB} \dfrac{\overset{[\neg A]}{\overset{?}{\bot}}}{A}$	Absurdum Elimination (EBotB)	$A \xRightarrow{EBotB} \dfrac{\overset{?}{\bot}}{A}$

Table 1: Definition of backward inference rules

To apply, for example, the implication elimination the student must follow these steps:

- Select a leaf from the main proof tree. The chosen leaf must have an incomplete proof, symbolized in the program with a question mark above.

- From the Rule Panel, click on the Implication Elimination button. The system automatically adjusts the rule to apply the backward variant. Then it checks that the selected rule can be applied on the formula according to syntactic restrictions. In this case there are no restrictions.
- If the rule requires it, the student must input other arguments. In this case, the antecedent of the derived implication. The system shows the expected outcome of the applied rule.
- The main proof tree is modified by adding new leaves.

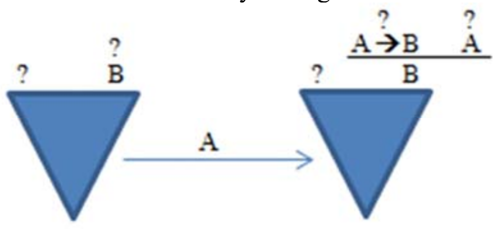


Figure 7: Implication elimination (backward variant)

3.1.4.2 Forward Rules

Forward rules work differently than backward ones: while the latter derive new obligations on the main proof represented as new leaves on the tree, the former are a sort of complete proof combinators. The main input for a forward rule is one or more secondary proofs, which by construction are always complete. The result is a new complete proof, composed from the input proofs by creating a new root that depends on the applied rule. The set of available hypotheses for the new root is constructed from the union of the already existing hypotheses.

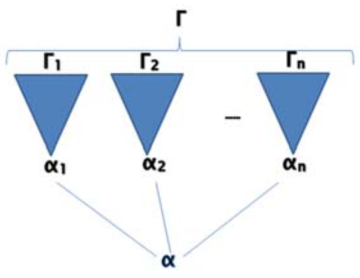


Figure 8: Forward rule general mechanism

The general notation for forward rules is:

$$\Gamma_1 \vdash \alpha_1 \xRightarrow{Rule\ Args} \frac{\Gamma_1 \vdash \alpha_1,\ \Gamma_2 \vdash \alpha_2 \ldots \Gamma_n \vdash \alpha_n}{\Gamma \vdash \alpha}$$

The left side of the arrow shows the proof where the rule is applied. The rule is always applied on a single proof, but other proofs may be used as arguments. On the right hand side the result of the application is shown: in the upper part all of the used proofs are listed and in the bottom there appears the resulting judgement.

Rule stands for the name of the rule, and *Args* is a set of additional data composed of:

- Propositional logic formulas
- Directions (Left / Right), as in backward rules
- Complete proofs. These are needed because some rules may need more than one complete proof in order to be applied.

As an example, [Fig. 9] shows how the Implication elimination works. The rule is applied on a complete proof of A→B with the set of available hypothesis Γ, and a proof of A with hypothesis Γ' is passed as argument. The result is a new complete proof of the formula B with a set of available hypothesis Γ'.

For a complete list of forward rules [see Tab. 2].

$$\boxed{\Gamma \vdash A \to B \xRightarrow{EImpF\ \Gamma' \vdash A} \frac{\Gamma \vdash A \to B \qquad \Gamma' \vdash A}{\Gamma \cup \Gamma' \vdash B}}$$

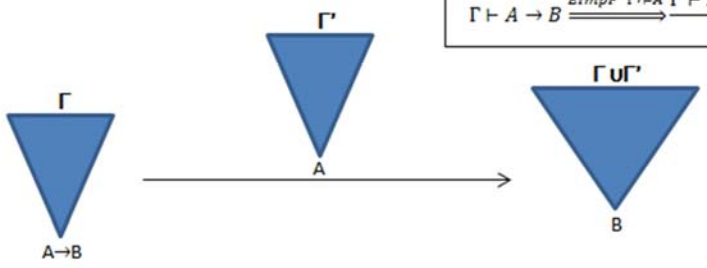


Figure 9: Implication Elimination Forward (EImpF)

3.1.5 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 allow to create generic proofs and reason about the universe of the propositional formulas. Moreover, generic proofs can be instantiated into concrete proofs (only containing 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 lemma:

$$(A \land B) \vdash (B \land A)$$

there exists in each of the following cases a substitution S that marks the corresponding proofs as complete:

$$(p \land q) \vdash (q \land p)\ using\ S = \{(A, p), (B, q)\}$$

and

$$(r \to s \wedge r \vee s) \vdash (r \vee s \wedge r \to s)\ using$$
$$S = \{(A, r \to s), (B, r \vee s)\}$$

ANDY uses this mechanism to implement two functionalities: lemma instantiation and secondary tree join.

Implication Introduction (IImpF)	$\Gamma \vdash B \xRightarrow{IImpF\ A} \dfrac{\Gamma \vdash B}{\Gamma \vdash A \to B}$	Implication Elimination (EImpF)	$\Gamma \vdash A \to B \xRightarrow{EImpF\ \ \Gamma' \vdash A} \dfrac{\Gamma \vdash A \to B \qquad \Gamma' \vdash A}{\Gamma \cup \Gamma' \vdash B}$
Disjunction Introduction (IOrF)	$\Gamma \vdash A \xRightarrow{IOrF\ B\ L} \dfrac{\Gamma \vdash A}{\Gamma \vdash A \vee B}$ $\Gamma \vdash A \xRightarrow{IOrF\ B\ R} \dfrac{\Gamma \vdash A}{\Gamma \vdash B \vee A}$	Disjunction Elimination (EOrF)	$\Gamma\ \vdash A \vee B \xRightarrow{EOrF\ C\ \ A,\Gamma' \vdash C\ \ B,\Gamma'' \vdash C}$ $\dfrac{\Gamma \vdash A \vee B \quad \{A\} \cup \Gamma' \vdash C \quad \{B\} \cup \Gamma''}{\Gamma \cup \Gamma' \cup \Gamma'' \vdash C}$
Conjunction Introduction (IAndF)	$\Gamma \vdash A \xRightarrow{IAndF\ \ \Gamma \vdash B\ \ L} \dfrac{\Gamma \vdash A \qquad \Gamma' \vdash B}{\Gamma \cup \Gamma' \vdash A \wedge B}$ $\Gamma \vdash A \xRightarrow{IAndF\ \ \Gamma \vdash B\ \ R} \dfrac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \cup \Gamma' \vdash B \wedge A}$	Conjunction Elimination (EAndF)	$\Gamma \vdash A \wedge B \xRightarrow{EAndF\ L} \dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}$ $\Gamma \vdash A \wedge B \xRightarrow{EAndF\ R} \dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}$
Equivalence Introduction (IEqF)	$B, \Gamma \vdash A \xRightarrow{IEqF\ \ A,\Gamma' \vdash B\ \ R} \dfrac{B, \Gamma \vdash A \qquad A, \Gamma' \vdash B}{\{A, B\} \cup \Gamma \cup \Gamma' \vdash A \leftrightarrow B}$ $B, \Gamma \vdash A \xRightarrow{IEqF\ \ A,\Gamma' \vdash B\ \ L} \dfrac{B, \Gamma \vdash A \qquad A, \Gamma' \vdash B}{\{A, B\} \cup \Gamma \cup \Gamma' B \leftrightarrow A}$	Equivalence Elimination (EEqF)	$\Gamma \vdash A \leftrightarrow B \xRightarrow{EEqF\ L} \dfrac{\Gamma \vdash A \leftrightarrow B}{\Gamma \vdash B \to A}$ $\Gamma \vdash A \leftrightarrow B \xRightarrow{EEqF\ R} \dfrac{\Gamma \vdash A \leftrightarrow B}{\Gamma \vdash A \to B}$
Negation Introduction (INotF)	$A, \Gamma \vdash \perp \xRightarrow{INotF} \dfrac{A, \Gamma \vdash \perp}{\{A\} \cup \Gamma \vdash \neg A}$	Negation Elimination (ENotF)	$\Gamma \vdash A \xRightarrow{ENotF\ \ \Gamma' \vdash \neg A} \dfrac{\Gamma \vdash A \qquad \Gamma' \vdash \neg A}{\Gamma \cup \Gamma' \vdash \perp}$
Reductio Ad Absurdum (RAA)	$\neg A, \Gamma \vdash \perp \xRightarrow{RAAF} \dfrac{\neg A, \Gamma \vdash \perp}{\{\neg A\} \cup \Gamma \vdash A}$	Absurdum Elimination (EBotF)	$\Gamma \vdash \perp \xRightarrow{EBotF} \dfrac{\Gamma \vdash \perp}{\Gamma \vdash A}$

Table 2: Definition of forward inference rules

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. In this way, 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 hypotheses at that moment.

3.2 Implementation

We chose to implement the system in a functional language, because the description therein of the syntactic structures involved in the calculus turns out to be quite simple and natural. We also needed a language that allowed us to create a functionally complex graphic interface that was at the same time simple and familiar for the student. As functional language, Haskell seemed a natural choice because it is widely used at the University and we had some experience working with it. However it was complicated to satisfy the graphical requirements. So we decided to take a chance on F# (actually a multi-paradigm language) 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 Common Language Runtime (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.html

4 Survey of Available Systems

The main goal of this project was to design and build a system that facilitates learning of the natural deduction calculus in introductory logic courses. The problem of building such systems is well known in the academic environment and there are several programs that implement different solutions, each one with its particular set of features. We now characterize and evaluate the ones which in our opinion characterize the present state of the art:

- Pandora IV [Pandora]
- DC Proof [DC Proof]
- Proof Builder [ProofBuilder]
- Gateway to Logic [Gateway to Logic]
- JAPE [JAPE]

4.1 Comparison Criteria

In addition to a brief description of how each program works, we define a set of points to compare based on the ones in [van Ditmarsch 1998], where a comparison of natural deduction systems from the point of view of the user interface is proposed.

1. **Implementation features:** details about the programming language used and portability
2. **Implementation and internal representation of the calculus:** Information about data structures and algorithms used to represent the proofs and inference rules.
3. **Proof management:** ability to save proofs and later use them as lemma.
4. **Version of natural deduction:** logical system (classical, minimal, intuitionist, predicate, etc.), available inference rules and their behavior, system restrictions, equality.
5. **Proof visualization:** The way the proof is displayed to the user (as a tree, as a sequence). Also visualization of premises, goals, rule application, formulas and step numeration
6. **Type of reasoning:** Backward, forward, both
7. **Help:** Depending on the implementation, the system might offer certain help to finish the proof. This is very important pedagogically. Too much help make it easy for the student to randomly try different rues until the proof is completed. We establish 4 kinds of help:
    a. Global help: Independent form the proofs, based on tutorials. Describes the available rules and may have examples of how to apply them.
    b. Tactical help: Gives the student a possible tactic to solve the problem depending on the current state of the proof. Usually consists on a single deductive step.
    c. Strategical help: Extension of tactical help. It gives a proof plan that consists of several deductive steps.
    d. Debugging: points to errors that arise on the proof, for example when a rule is applied incorrectly.
8. **Proof checking:** Mechanism by which the system validates that the proof is correct. It can be a step by step validation, preventing steps that would lead to a wrong proof or impossible to complete, or a validation test when the proof is finished.
9. **Creation date / Last version:** Indicates the creation date of the program, the current version and the current state of the project (discontinued, in use, etc.)

To begin with, we describe ANDY from the point of view of these criteria.

Criteria	**Description**
Implementation features	Desktop application based on Microsoft .NET Framework 4.0. The core of the system is developed in F# and the UI in C#.
Implementation and internal representation of the calculus	The derivation engine and proof manager are based on functional programming. Proofs are implemented as n-ary trees and inference rules as functions applied to formulas on tree leaves.
Proof management	Proofs (complete or incomplete) can be exported as XML files. Incomplete proofs can be later imported and resume the proof. Complete proofs can be used as lemma on new proofs. The system can scan a directory where XML proofs are saved and obtain the proof statement ($\Gamma \vdash A$) before actually importing the proof into the system.
Version of natural deduction	Propositional logic with backward and forward variant for each introduction and elimination rules for conjunction, disjunction, implication, equivalence, negation and bottom ($\perp$) plus a rule for *Reductio ad absurdum*
Proof visualization	Represented as trees, where the root is the conclusion and the leaves are hypothesis of the statement. Each node represents the application of an inference rule. Available hypothesis for a given node are shown on a lateral panel discriminated using a color code into theorem hypothesis, assumptions and derived hypothesis.
Type of reasoning	Allows backward and forward reasoning. Backward rules are applied on the root of the proof tree while forward rules are applied onto secondary trees that can be constructed on the proof panel. The main tree and these secondary trees can be joined at any moment by a formula instantiation procedure that is transparent to the student.
Help	Includes a help guide that explains all the functions and how to use each rule. Also implements debugging help by pointing out errors when a rule is not correctly applied. There is no tactical or strategical help.
Proof checking	Step by step validation. The system prevents the user from applying incorrect rules on a given formula.
Creation date / Last version	2011, fully functional version for propositional logic. Now working on an extension for predicate logic.

Table 3: Description of ANDY

4.2 Survey

4.2.1 Pandora

For more information see [Pandora].

Criteria	Description
Implementation features	Java based web application. Highly portable
Implementation and internal representation of the calculus	No available documentation about its internal implementation. There is documentation about Java packages and classes used
Proof management	The program allows saving complete and incomplete proofs as .pan files. Not allowed to use previous proofs in new ones. It allows taking goals as proven without the need to provide an explicit derivation, using a tactic called Trust Me. This can be used to simulate the use of lemmas.
Version of natural deduction	Implements the classic rules of natural deduction in first-order logic (introduction and elimination of connectives and quantifiers). Adds rules for equality (reflexivity, substitution), and the Law of the Excluded Middle.
Proof visualization	The proof is shown sequentially as a Fitch diagram. The places where a demonstration is needed are symbolized with an <empty> label. The proof obligations are indicated by the label <goal>. It uses the concept of signature of the proof which is the set of predicates and terms that appear on it. There is a global signature throughout the proof and a local signature (usually with more items than the global one) for each box in the diagram. The overall signature is created automatically by the program and is not editable; however the local signatures can be edited at any time, for example to create a variable that can be used fresh to apply the rule of universal quantifier elimination.
Type of reasoning	Allows forward and backward reasoning, determined by the way in which each rule is applied. To apply a rule forward, usually <empty> line is selected and then one of the rules to the side panel, while to apply a backward rule <goal> line is selected and then one of the rules.
Help	The system contains an extensive guide ant tutorial which is accessed from the help menu of the application. I has a detailed explanation of how each tactic work in its forward and backward mode, in addition to conventions, some generic hints on how to develop tests and notions of syntax. No strategic or tactical support, but if it shows errors when applied incorrectly rules.
Proof checking	The proof is checked after it is finished. The only restrictions to rule application are given by the help system.
Creation date / Last version	On the website there is no information about the date of creation of the system, but it seems to be constantly updated.

Table 4: Description of Pandora

4.2.2 DC Proof

Program developed by Dan Christensen [DC Proof]. It is not a program for natural deduction proofs exclusively; it is focused on mathematical proofs generally, implementing set theory and induction proof concepts.

Criteria	**Description**
Implementation features	There is no documentation indicating the language in which it is implemented. In the download page states that it is an application for Windows environments and emulators are needed for other operating systems.
Implementation and internal representation of the calculus	No information about internal data structures and algorithms to explain how proofs are performed internally.
Proof management	Allows storing complete and incomplete proofs, plus the ability to print and create HTML files.
Version of natural deduction	It doesn't follow the classic terminology for natural deduction rules, probably because it is not a system dedicated to natural deduction but to general mathematical proofs.
Proof visualization	Sequentially numbered, with an indication of the rule applied at each step and the lines involved. Sub proofs contraction allows to reduce space.
Type of reasoning	The reasoning is strongly forward. To begin a test, there is only the possibility of entering premises and the conclusion must be built by successive application of rules
Help	The program contains a separate manual with explanation of each tactic, a tutorial with examples of application and a reference manual on the operation of the overall program.
Proof checking	Since the proofs are performed from premises without previously setting the conclusion, validation is made step by step. The system ensures that the proof cannot be wrong, as all steps are controlled.
Creation date / Last version	No information about the date of creation, but the website shows that it is still in development.

Table 5: Description of DC Proof

4.2.3 ProofBuilder

Software developed by Hugh McGuire, Grand Valley State University. See [ProofBuilder].

Criteria	Description
Implementation features	Implemented in Java, downloadable from its web site
Implementation and internal representation of the calculus	No information on how the calculus is implemented, but the sources are available to download.
Proof management	Incomplete proofs can be saved to continue them later. Can be exported as HTML files.
Version of natural deduction	Allows proof construction in propositional logic and predicate logic. Implements rules for all connectives, but does not respect the known elimination and introduction nomenclature. The system allows rewriting formulas by the application of logical laws.
Proof visualization	Visualization is sequential but using two columns. On the left are the assumptions and derivations that are obtained and to the right proof obligations. Further, in another column the formulas used (i.e. to which some tactic was applied) are indicated. The first two columns to the left of the panel correspond to the numbering of the line and a field to add comments. An interesting detail is that as rules are used, the system adds natural language comments on how the proof proceeds. Another feature is that when a formula is selected, the system automatically identifies the main connective and the rules that can be applied.
Type of reasoning	Backward and forward reasoning
Help	No help included in the program (there is an online guide), or any tactical or strategic assistance. The system automatically locks the tactics that are not applicable for the selected formula, so that no error cases can be reached.
Proof checking	Proof is validated at the end.
Creation date / Last version	It was developed in 2006 and still in development.

Table 6: Description of ProofBuilder

4.2.4 Gateway to Logic

It is a website that brings together several web programs, particularly this one that builds natural deduction based proofs. For more information see [Gateway to Logic].

Criteria	**Description**
Implementation features	Java applet, it works on the web page. No downloadable version.
Implementation and internal representation of the calculus	No information on how the calculus is implemented.
Proof management	Proofs cannot be saved, though there is an option to print them.
Version of natural deduction	It is based on classical logic (propositional and first order) and provides elimination and introduction rules for each logical constant.
Proof visualization	The proof is sequentially numbered at each step and labeled with the name of the applied rule. No visualization of goals or proof obligation, which can be a bit confusing.
Type of reasoning	Forward reasoning only. The only way to generate hypotheses is to enter them manually.
Help	Global aid is very brief and only explains rules of the propositional calculus. This, plus the fact that there are no examples, makes it very difficult to use. Error messages are displayed when rules are applied incorrectly.
Proof checking	It enables rules depending on the context. Proof validated at the end.
Creation date / Last version	No information about the creation date and current system status

Table 7: Description of Gateway to Logic

4.2.5 JAPE

For more information see [JAPE].

Criteria	**Description**
Implementation features	It is an application written in Java. Can be downloaded from their website.
Implementation and internal representation of the calculus	Internally, the program drives a sequent calculation, and represents the evidence as a Gentzen tree
Proof management:	Allows storing complete and incomplete proofs. It has a predefined set of derivations, which after being proven can be applied in new derivations as lemma.
Version of natural deduction	Jape is a program that allows deductions on various theories (set theory, predicate logic, propositional logic, Hoare logic and others.). The most important thing is that it is possible to define new theories, rules and properties, upload them to Jape and make inferences with them. In the case of natural deduction (import file I2l.jt theory) allows the calculation for propositional logic and predicate logic. This includes all known rules of introduction and elimination. An important detail is that it classifies rules as backward and forward.
Proof visualization	Proof is displayed as sequentially as Fitch diagrams. Each step is numbered and displays the rule that was applied to obtain the formula. While deriving the proof, incomplete parts are shown with ellipses. The absence of ellipses means that the proof is completed.
Type of reasoning	Allows both forms of reasoning with preference toward backward reasoning. This shows in particular cases such as the introduction of universal quantifier, which cannot be applied forward.
Help	No tactical or strategical help, the proof must be constructed step by step and the only help appears when a rule has been wrongly applied.
Proof checking	The system does not allow missteps. The validation is done step by step.
Creation date / Last version	Continuously updated

Table 8: Description of JAPE

4.3 Discussion

In most of the cases listed above there is little information about how the proof system was implemented internally. The exception is Jape, which provides abundant documentation about internal structures and interface development, as well as experiences and usage experiments.

There are varied forms of naming the classical rules of natural deduction (deletion, insertion), some can be very confusing especially for beginners. In contrast, our system uses the customary nomenclature which is presented in any textbook. Pandora adopts the nomenclature we know and Jape goes a little further, classifying the rules according to the type of reasoning.

The combination of forward and backward reasoning is present in several systems. This feature is the one resembling the construction of proofs on paper, so we accord it high relevance.

Jape seems to be the system with the simplest graphical user interface and easier to understand. However, it is for us imperative that the system shows the proof as tree, for the reasons already explained. This feature is not present in any of the systems above. One proposal presenting this kind of interface is [Byrnes et al. 2009] but it is not a proof assistant aimed at pedagogic ends, but rather a (semi-)automatic theorem prover.

An interesting feature is the one on ProofBuilder which generates natural language description of the proof. This, combined with the representation of the proof in a tree, can be a very powerful tool to help understand how the proof is being constructed.

Our system seems to be the one providing the most general way to manage lemmas. This is, in our opinion, a very important feature, since it embodies a principle of modular development that is relevant both in practice and from the theoretical or methodological point of view.

Finally, our system falls short of all the ones cited above in its scope, since, for the moment, it remains restricted to propositional logic.

5 Experience of Use of the System

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.
- 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 trying to use exclusively 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. Also there are modifications to the Software Engineering curriculum on their way to be implemented starting in 2013, which will incorporate the first part of the current Logic course (dealing with Induction and Recursion) to a previous course named Foundations of Computing that will serve also as an introduction of Functional Programming as well as to informal proof of programs. This will leave more room in the Logic course for, among other things, playing more extensively with formal proofs.

6 Conclusions and Future 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.

The obvious extension to be made to the tool is 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. Now, using meta-variables in the first-order version requires to deal with side-conditions of the form *x not free* in formulas, for term variables *x,* as well as other meta-notions. We plan to attack this by employing the systems of nominal syntax which manage meta-variables as primitive along bound names and restrictions of precisely the form above. Particularly, a system named of *One-and-a-halfth* order has been put forward [see Gabbay and Mathijssen 2008] which seems precisely the one we wish to manage.

References

[Bornat and Sufrin 1996] Bornat, R., Sufrin, B.: "User Interfaces for Generic Proof Assistants Part I: Interpreting Gestures."; User interfaces for Theorem Provers, York, UK (1996).

[Bornat and Sufrin 1997] Bornat, R., Sufrin, B.: “Displaying sequent-calculus proofs in natural-deduction style: experience with the Jape proof calculator”; International Workshop on Proof Transformation and Presentation, Dagstuhl (1997).

[Bornat and Sufrin 1998] Bornat, R., Sufrin, B.: “User interfaces for generic proof assistants part II: Displaying Proofs” (1998).

[Byrnes et al. 2009] Byrnes, J. et al.: “Visualizing Proof Search for Theorem Prover Development“; ENTCS (Electronic Notes on Theoretical Computer Science), 226 (2009), 23-38.

[DC Proof] Proof-writing software to teach the fundamentals of logic and proof. www.dcproof.com. Accessed as of April 30, 2013.

[Dyckhoff 1987] Dyckhoff, R.: “Implementing a simple proof assistant”, Workshop on Programming for Logic Teaching, Centre for Theoretical Computer Science, University of Leeds (1987).

[Frege 1967] Frege, G.: “Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens”. Halle a. S.: Louis Nebert, 1879. Translated as “Concept Script, a formal language of pure thought modelled upon that of arithmetic” by S. Bauer-Mengelberg in J. vanHeijenoort (ed.), “From Frege to Gödel: A Source Book in Mathematical Logic” (1879-1931), Cambridge, MA: Harvard University Press (1967).

[Gabbay and Mathijssen 2008] Gabbay, M., Mathijssen, A.: “One-and-a-halfth-order logic”, Journal of Logic and Computation, 18, 4 (2008), 521-562.

[Gateway to Logic] Collection of web-based logic programs offering a number of logical functions. http://logik.phl.univie.ac.at/~chris/gateway/formular-uk.html. Accessed as of April 30, 2013.

[Gentzen 1969] Gentzen D.:“Untersuchungen über das logische Schliessen”. Translated as “Investigations into logical deduction”; Mathematische Zeitschrift 39, 1935. Translated in M. Szabo, editor. Collected Papers of Gerhard Gentzen. North Holland (1969).

[Gödel 1931] Gödel, K.: “On Formally Undecidable propositions of Principia Mathematica and related systems“; Solomon Feferman, ed., 1986. “Kurt Gödel Collected works, Vol. I“ Oxford University Press: 144-195.

[Halvorsen 2007] Halvorsen, J.: “Web based GUI for Natural Deduction Proofs in Isabelle”; School of Informatics, University of Edinburgh (2007).

[Hoare 2003] Hoare, T: “The Verifying Compiler: A Grand Challenge for Computing Research“; Lecture Notes in Computer Science. Springer, Heidelberg / Berlin (2003)

[Huth and Ryan 2004] Huth M., Ryan M.: “Logic in Computer Science: Modeling and Reasoning About Systems”; Cambridge University Press (2004).

[JAPE] http://www.cs.ox.ac.uk/people/bernard.sufrin/personal/jape.org/ Accessed as of April 30, 2013.

[McGuire 2007] McGuire, H: “ProofBuilder, an Interactive Prover for Students, with Extensive Capabilities” Grand Valley State University, Allendale, USA (2007).

[Pandora] “Proof Assistant for Natural Deduction using Organised Rectangular Areas” http://www.doc.ic.ac.uk/pandora/newpandora/ Accessed as of April 30, 2013.

[ProofBuilder] Pedagogical software for constructing proofs.
http://www.cis.gvsu.edu/~mcguire/ProofBuilder/ Accessed as of April 30, 2013.

[van Dalen 2008] Van Dalen, D.: “Logic and Structure”. Springer (2008).

[van Ditmarsch 1998] van Ditmarsch, H.: “User interfaces in natural deduction programs”, in Backhouse, R. C. (Ed.) Proceedings of the 4th International Workshop on User Interface Design for Theorem Proving Systems (1998).

[Whitehead and Russell 1913] Whitehead A., Russell B.: “Principia Mathematica“; Cambridge University Press (1913).