Natural deduction environment for Matita

Claudio Sacerdoti Coen* and Enrico Tassi*

Department of Computer Science, University of Bologna
Mura Anteo Zamboni, 7 — 40127 Bologna, ITALY
{sacerdot,tassi}@cs.unibo.it

Abstract. Matita is a proof assistant characterised by a rich, user extensible, output facility based on a widget for the rendering of MathML Presentation, and by the automatic handling of overloading by means of a flexible disambiguation mechanism. We show how to use these features to obtain a simple learning environment for natural deduction, without modifying the source code or Matita.

1 Introduction

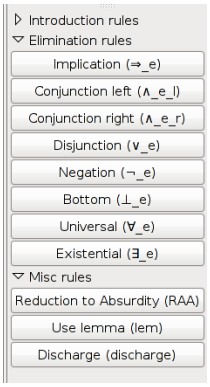
There is at least one good reason for pushing the adoption of Interactive Theorem Provers as teaching instruments that go beyond any pedagogical consideration: make students familiar with Interactive Theorem Provers, grasping their interest for future project works or thesis assignments.

To do that, Interactive Theorem Provers have to be turned into learning platforms, for example tools for learning induction or other fundamental concepts like formal proofs. Learning logics, students meet formal proofs as derivation trees, for example following the natural deduction calculus.

In this paper we present our effort in implementing on top of the Matita Interactive Theorem Prover [2] a learning environment for Natural Deduction, exploiting the flexibility of the notational mechanism the tool offers and its peculiar ambiguity management. Our requirements were:

- allow students to input possibly incorrect derivation trees
- force the user to input exactly the same information he would write on paper, even if redundant or inferable by the system
- notify the user highlighting erroneously applied derivation rules, but allow him to complete the tree
- graphically display the derivation tree, facilitating its navigation in the frequent case of huge derivations not fitting the screen
- allow a quick (batch) correction of exercises to the teacher
- introduce the user to a textual syntax for the procedural construction of derivation trees in order to smooth the transition from derivation trees to procedural/declarative scripts
- help the user in learning such syntax and haste the input phase

* Partially supported by the Strategic Project "DAMA: Dimostrazione Assistita per la Matematica e l'Apprendimento" of the University of Bologna




```
lemma ex1: ¬(∃x.P x) ⇒∀x.¬ P x.
apply rule (prove (¬(∃x.P x) ⇒∀x.¬ P x));
apply rule (⇒_i [h1] (∀x.¬ P x));
apply rule (∀_i {l} (¬P l));
apply rule (¬_i [h2] (⊥));
apply rule (¬_e (¬(∃x.P x)) (∃x.P x));
    [ apply rule (discharge [h1]);
    | apply rule (∃_i {l} (P l));
      apply rule (discharge [h2]);
    ]
qed.
```

Fig. 1. Input palette and proof script

We presents the input and output interfaces in Section 2. The real contribution of the paper is in Section 3 where we show how to achieve the aforementioned goals without modifying the system and exploiting the MathML Presentation based notational system [4] and the peculiar management of notational overloading [6] Matita provides.

2 User interface

Derivation trees are described using the following subset of the procedural language of Matita:

apply rule (*rule name* arguments...); [*subproof* | *subproof* ...]

The apply-rule tactic is the standard application of a proof-term (which is usually, but not in our case, the name of a lemma), while square brackets are standard tacticals used to structure the proof script. Unlike other systems like LCF, Coq or Isabelle, the execution of a structured script is performed in Matita one tactic at a time [5], so that the incremental build of a structured proof is comfortable.

To apply a derivation rule the user is asked to list after the rule name all the information he would write as a side condition and above the inference line. For example the partial derivation tree of Figure 2 is obtained by executing the script of Figure 1 up to the end of the blue region in Figure 2.

The choice of annotating rules with the name of the hypotheses they discharge was a teacher choice, and it reflects the way the teacher wants the students to write the tree even on paper. Similarly, hypotheses are discharged by labelling them with their names (e.g. $[\ \cdot\ ]^{h1}$) and witnesses are explicitly provided for existential elimination. Missing sub-derivations are represented by numbered question marks; clicking on them the user is reminded of what assumptions are in scope.

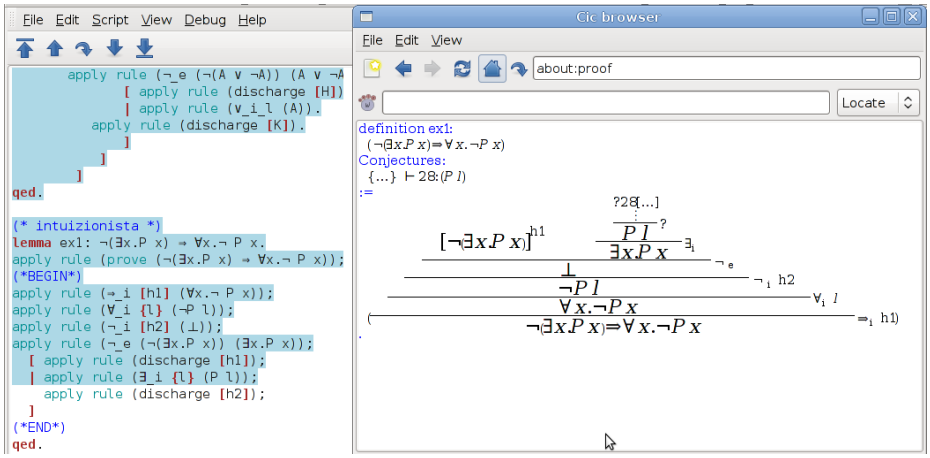


Fig. 2. Rendering of a partial derivation tree

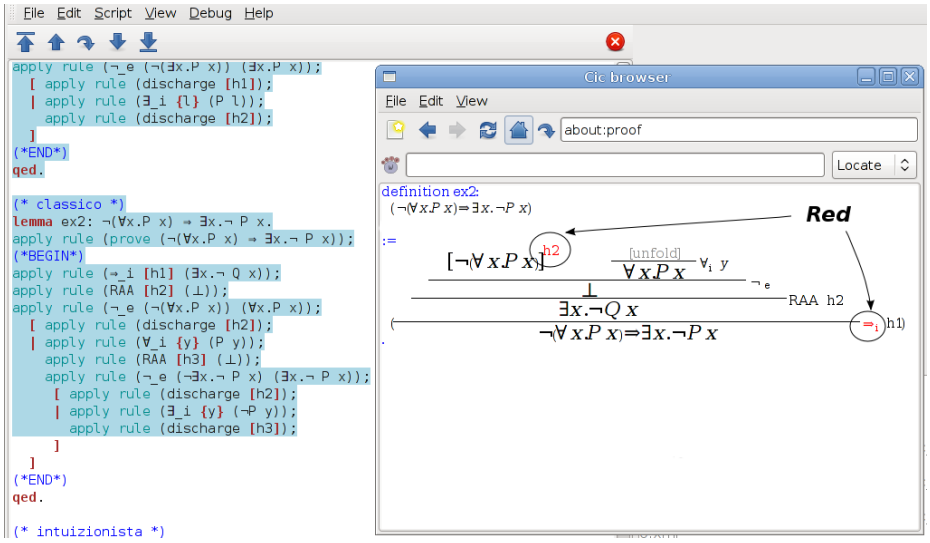


Fig. 3. Rendering of a partially collapsed incorrect tree

The concrete syntax consistently uses round brackets for formulae, square brackets to name hypotheses and curly brackets to name terms/variables.

A palette (Figure 1) can be used to insert in the script a template for every derivation rule, where no formula or term is inserted for the user. It is thus possible that the user incorrectly applies a rule. In such a case, the rule name is coloured in red as in Figure 3.

To improve readability, correct sub-trees can be collapsed to "[unfold]" by clicking on their root formula (see Figure 3).

3 Implementation

The notational system of Matita is based on three different languages for the representation of formulae and proofs: semantics, which is logic dependent, content, where we use MathML Content and OMDoc, and presentation, where we use MathML Presentation within a simple layout language called BoxML [3]. The user may define two sets of rules to map back and forth semantic objects to content objects, and to map content objects to presentation objects [1].

The two mappings need not to be one-to-one, and disambiguation is employed in the parsing phase to resolve overloading in favour of the interpretation that is meaningful (i.e. the well typed semantics objects).

Thanks to MathML Presentation we can render derivation trees in a partially satisfactory way as fractions (<mfrac>) and we can use <maction> nodes to collapse sub-trees and to show hypotheses in scope; finally we can use <mstyle> to highlight errors in red.

Disambiguation is used to associate to the same rule name two different semantics objects. The first one is the constant corresponding to the introduction/elimination rule in the logic of Matita. Thus, the object is well typed (i.e. meaningful for the disambiguation engine) only if its conclusion matches the current goal and its arguments are the expected ones. The second interpretation, that is always meaningful and is only used when the former fails, uses an (axiomatic, non admissible) cast operator to fix the incorrect rule application. Another notational rule associates <mstyle>, that produces the rendering in red, to occurrences of that particular cast constant.

The second interpretation can be deactivated by the teacher to make the system reject incorrect proofs. This is useful for batch correction of exercises.

We show now all the definitions, interpretations (from semantics to content) and notations (from content to presentation) that deal with logical conjunction and its elimination rule.

```
inductive And (A,B:CProp) : CProp :=And_intro: A → B → And A B.

definition And_elim_l : ∀A,B.And A B →A :=
  λA,B,f. match f with [ And_intro l r ⇒l].

axiom any : CProp.
```

Note that the definition of conjunction is the standard one used in Matita, and not an embedding only useful for natural deduction. This allows the reuse of the standard machinery of Matita (e.g. automation, type checking) and of its library (e.g. in order to propose proofs of arithmetical statements). It also allows to progressively drop the natural deduction trees when the student is ready to embrace the full procedural language, or the declarative or to mix the three modes. The any proposition is necessary in the interpretation of a badly-applied conjunction elimination rule.

```
(* Used to replace a proof 'p' of 'P' with '(show P p)' to use 'P' in rendering *)
definition show: ∀A:CProp.A→A :=λA,a.a.
axiom cast: ∀A,B:CProp.B →A.
```

Show is used to make statements of sub-proofs explicit (so that notation can label the root of every subtree with the formula it proves). Cast is used to accept a formula erroneously typed by the user by "casting" a proof of any proposition B to any proposition A.

We present now the output notations and relative interpretations used to render the conjunction elimination rule.

```
notation < "\infrule ab a mstyle color #ff0000 (∧ e_l)"
  with precedence 19 for @{ 'And_elim_l_ko ab a }.
interpretation "And_elim_l_ko" 'And_elim_l_ko ab a =
  (show a (cast _ _ (And_elim_l _ _ (cast _ _ ab)))).

notation < "maction (\infrule ab a (∧ e_l)) [unfold]"
  with precedence 19 for @{ 'And_elim_l_ok ab a }.
interpretation "And_elim_l_ok" 'And_elim_l_ok ab a = (show a (And_elim_l _ _ ab)).
```

The first output notation displays the content symbol 'And_elim_l_ko with an \infrule layout (mapped to MathML <mfraction> plus <mstyle> directions to avoid font shrinking in fractions of fractions) colouring the rule name in red. Its corresponding interpretation is associated to a term containing the cast constant. The second output rule displays a correct rule application, adding the possibility to fold the tree clicking on it (<maction> node, whose default behaviour is to toggle between its children).

Dually, we introduce an input notation and two corresponding interpretations, respectively for a correct and a wrong application of conjunction elimination.

```
notation > "∧ _e_l term 90 ab" with precedence 19 for @{ 'And_elim_l (show ab ?) }.
interpretation "And_elim_l KO" 'And_elim_l ab =
  (cast _ _ (And_elim_l _ _ (cast (And any any) _ ab))).
interpretation "And_elim_l OK" 'And_elim_l ab = (And_elim_l _ _ ab).
```

The input notation is associated to the rule name. The second interpretation is preferred and inserts no cast constant, while the first one, used only if the second fails, applies the rule casting both its conclusion and premise. In particular, the premise is cast to the conjunction of two occurrences of the previously declared dummy proposition any in order to apply elimination of conjunction.

4 Conclusion

Matita is a proof assistant based on the Curry-Howard isomorphism: proofs are internally represented at the semantics level with proof terms. Thus the machinery used to associate the familiar mathematical notation to formulae can as well be used to render proofs. In particular, in place of the usual rendering as natural

language text with embedded MathML formulae, we can output derivation trees as interactive MathML formulae, exploiting <mfrac>, <maction> (to collapse and expand sub-trees) and <mstyle> (to highlight incorrect parts of the proof). Thanks to the typical MKM layered representation of knowledge (semantics, content and presentation [1]) that is at the core of the notational machinery of Matita, we have been able to achieve the latter result without changing the source code, and with around 800 lines of notation and interpretation commands.

We then proceeded to fulfil the minimal requirements for a learning system, that we identified in the possibility for the student to input exactly the same information he would write on paper and to be allowed to commit errors and continue. We achieved this easily thanks to the disambiguation engine of Matita [6] that allows to resolve overloading in favour of the interpretations that yield meaningful formulae: by introducing an axiomatic, non admissible cast rule, any derivation tree becomes legal, but incorrect trees can be easily distinguished and displayed accordingly.

This way of implementing the learning system using only notational devices allows to progressively abandon the derivation trees in favour of the standard declarative or procedural language of Matita, when the students are ready. The system was adopted in a first course of logic for computer scientists at the University of Bologna in the academic year 2008/2009.

The only current limitation is the impossibility to develop derivation trees in a bottom-up fashion, from the leaves to the root, by working with hypotheses that will be discharged only later. This reflects the way proofs are developed in Fitch style and in informal mathematics, and in class we carefully avoid making students work with un-assumed hypotheses anyway. On the other hand, subproofs (containing no free assumptions) can be proved in advance and plugged in the main proof.

References

1. Andrew A. Adams. Digitisation, representation and formalisation: Digital libraries of mathematics. In J.H. Davenport A. Asperti, B. Buchberger, editor, *Proceedings of Mathematical Knowledge Management 2003*, volume 2594 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2003.
2. The Matita interactive theorem prover. `http://matita.cs.unibo.it`.
3. Luca Padovani. A math canvas for the GNOME desktop. In *5th Annual GNOME User and Developer European Conference (GUADEC'04)*, volume 107. Agder University College.
4. Luca Padovani and Stefano Zacchiroli. From notation to semantics: There and back again. In *Proceedings of Mathematical Knowledge Management 2006*, volume 4108 of *Lectures Notes in Artificial Intelligence*, pages 194–207. Springer-Verlag, 2006.
5. Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. Tinycals: step by step tacticals. In *Proceedings of User Interface for Theorem Provers 2006*, volume 174 of *Electronic Notes in Theoretical Computer Science*, pages 125–142. Elsevier Science, 2006.
6. Claudio Sacerdoti Coen and Stefano Zacchiroli. Spurious disambiguation errors and how to get rid of them. *Journal of Mathematics in Computer Science, special issue on Management of Mathematical Knowledge*, 2:355–378, 2008.