Mathematics and Proof Presentation in Pcoq

Ahmed Amerkad, Yves Bertot, Loïc Pottier, Laurence Rideau

INRIA Sophia Antipolis

1 Introduction

PCOQ is the latest product in a decade-long effort to produce graphical user-interfaces for proof systems. It inherits many characteristics from the previous CTCOQ system, that provided such a graphical user-interface based on the programming tools available in the generic interactive programming environment generator Centaur [6]. The new graphical interface is programmed in a different language, Java, and most of the design decisions that had been made for CTCOQ have been re-considered.

The CTCOQ system advocated a few basic principles, mainly described in [15, 4, 2]: the user-interface is a separate process from the logical engine, Coq [1], the logical data in the user-interface is manipulated as structured, tree-like data, and the user-interface manipulates the sequence of commands sent to the logical engine as a complete document, the consistency of which it helps to ensure.

Advocating the user-interface as a separate process makes that the whole "proof development environment" relies on communication protocols for mathematical data that were precursors to the now well-known proposals around XML: OPENMATH and MATHML.

Manipulating mathematical formulas as structured data, we have been able to instrument the proof development environment with capabilities that increase the bandwidth between the user and the logical engine. An example is that of *Proof-by-pointing* [3], where complex commands can be constructed in one click by the user, through an analysis of the mouse position in the formula.

Another capability that is instrumental in enhancing the bandwidth between the user and the logical engine is the possibility to layout the mathematical formulas in a manner that is very close to the typesetting practice of mathematical literature [13]. Just increasing the readability of mathematical formulas speeds up the proving process in a significant manner, especially for the kind of logical engine that we use, where interactive proof is preferred to automatic proof search. Still one has to make sure that data displayed on the screen is still active, in the sense that it can be selected with the pointing device. This makes the problem very different than passive mathematical layout as in TEX[10].

The CTCOQ proof development environment has been used intensively for a variety of ambitious, large-scale proof developments [14, 11]. The PCOQ environment is planned to replace it from the next version of COQ on. It is available on internet at address

`http://www-sop.inria.fr/lemme/pcoq`

and it has already been tested for teaching computer-aided proof to students and for intermediary scale mathematical developments around computer arithmetics.

There are several kinds of applications for which one may want to integrate a proof presentation capability in a proof development environment. A first range of applications covers didactic needs, either to help teach the foundations of logic or to describe the structure of the proofs occurring in well-known mathematical results. A second range of application covers industrial needs. This must be addressed to make logic-based formal methods of software development accepted. Here proof presentation is needed because formal proofs of software are needed in contractual parts of the industrial software development process: when purchasing a piece of software, the quality of which relies on formal proofs, purchasers need to be convinced that the statements being proved actually reflect their specifications. Third party inspection of the formal development must be made possible, for instance when considering certification with respect to certain levels of quality like the common criteria for security evaluation. Our work is more oriented towards this second range of applications. For these applications, ease of use for beginners is less important than providing scalable tools, capable of handling the proofs that an engineer may produce after a few days of work on a single proof, possibly using elaborate proof commands.

In the rest of this paper, we first show how the organization of the graphical user-interface around structured data makes it possible to obtain elaborately laid-out mathematical formulas. In particular, we insist on the three-level extension mechanisms that make it possible to customize the layout to applications. In the next section, we describe the data-structure that is used in the logical engine to record *proofs-in-the-making* and we show this data-structure can be used as a basis for the production of text in a restricted subset of natural language. Here, the natural text is still obtained by a layout strategy over a structured piece of data, and we show that interactive manipulation over this piece of data is still possible, thus provided means of developing proof, while looking directly at text being produced. We conclude by describing all the new questions that this experiment raises with respect to proof explanation and its integration in proof development environments.

2 Extensible layout for mathematical structures

In our experiments, we have used the Coq system as the logical engine. This system uses a typed λ-calculus as the underlying language to represent all mathematical notions. All mathematical operations are represented as typed functions and most mathematical sets are represented types. Moreover, the logical foundation relies on a correspondence between types and logical formulas, known as the *Curry-Howard* isomorphism. Intuitively, the function type is also used for representing implication: a proof (a function) of $A \Rightarrow B$ (meaning: the function has type $A \rightarrow B$) can be used to construct a proof of B whenever it is given a proof of A (in this sense function application is directly used to represent the logical rule of *modus-ponens*). The whole logical engine generalizes this idea.

The λ-calculus is known for its extreme simplicity: there are only three kinds of terms: variables, applications, and λ-abstractions. Even if we take into account the operators that have been added to handle typing and inductive structures, the different constructs are still very few. As a result, the various mathematical operations one may encounter are all encoded using function application, all data looks the same and it is very useful to provide an approach to switch at least from the usual prefix notation of function application to infix notations. But usual mathematical notations actually go beyond this level of variety, and sub-scripts, super-scripts, and various kinds of geometric layout are also encountered. To cope with this variety of needs, we have designed a layout mechanism that provides three levels of extensibility, requiring three different levels of expertise. This layout mechanism makes it possible to display mathematics as shown in figure 1.

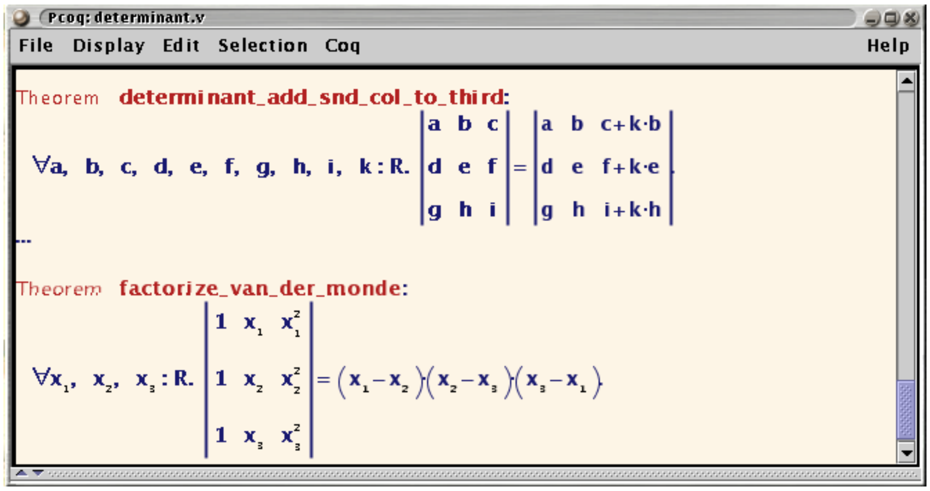

Figure 1. Example of mathematical notations: linear algebra.

2.1 Mathematical operator classes

The first level of extensibility works simply by associating predefined layout formats to identifiers. The available formats consist in the usual infix notation, of course, but the system also makes it possible to use layout strategies that imply more elaborate display capabilities: subscripts, exponents, fractions, super-positions, the radical sign, n^{th} root signs, over-lining, underlining, etc. The correspondence is made accessible in the form of a table, which can easily be edited: adding a line for a new operator, choosing the corresponding layout strategy, precedences that will help decide for automatic parenthesis insertion, fonts, and even the possibility to choose symbols outside the range usually covered by the keyboard. For instance, the layout for functions `Qplus`, `Qmult` and `Qdiv` representing addition and division of rational numbers is declared with the following lines of the table:

Operator name	Family	Left precedence	Right precedence	Font name	Font style	Font size	Text
Qplus	infix	20	20	Dialog	PLAIN	12	+
Qmult	infix	25	25	Dialog	PLAIN	12	·
Qdiv	fraction	200	200	Dialog	PLAIN	12	

Layout operators can receive a variety of arguments but they may not use all of them. For instance fraction does not use any text argument. Figure 1, shows a few of these simple operators display strategies: additions, multiplications, substractions, and squares. In type-theory-based representations of logical formulas, many operators that usually have two arguments actually receive extra arguments to make type information more precise. For instance, equality is usually viewed as a relation between two objects, but these objects need to have the same type. For this reason, the type-theory-based representation of equality is a three-argument relation, where the first argument is the type, in which the two other must belong. To handle this, infix notations apply even to functions with three arguments, but only the last 2 are displayed.

2.2 Structure pattern matching and box construction

In general, the preferred notation for a given mathematical function or relation cannot be described by a simple mapping to the predefined layout strategies available in the operator table. For these operations and relations, users can describe their notation using a simple pattern-matching based language, called PPML. This language is inherited from the Centaur system.

For instance, if the mathematical study often makes use of triplets, constructed with typed pairs, where the types being used are always the same A, B, and C, users may want to replace the default notation:

$$\langle A, (B \times C)\rangle(a, \langle B, C\rangle(b, c))$$

with a more concise and, hence, more readable notation:

$$[a, b, c].$$

This can be easily expressed with a PPML rule that has the form given in figure 2. This layout rule combines two parts. The first is a pattern that will be compared with the expressions being displayed. In this pattern, the names like `appc`, `formula_ne_list`, and `ident` correspond to tree operators in the data-structure representing the logical formula. Some operators like `ident` may carry a string value. In the pattern, some sub-terms are left unknown, they are represented with variables, and there is a lexical convention that makes it

```
appc(ident "pair",
 formula_ne_list[ident "A",
                 appc(ident "prod",
                      formula_ne_list[ident "B", ident "C"]),
                 *x,
                 appc(ident "pair",
                      formula_ne_list[ident "B", ident "C",
                                      *y, *z])])
 ->
[<hv 1,2,0> [<h 0> "[" *x "," ] [<h 0> *y ","] [<h 0> *z "]" ]];
```

Figure 2. A PPML rule for displaying triplets.

possible to recognize variables: they always start with a star. In our example, the variables are *x, *y, and * z.

The second part of the rule contains a layout specification organized into boxes. Each box is annotated with a layout directive that indicates how the various components present in the box will be arranged with respect to each other. In this example, we see only two kinds of directives: <hv 1, 2, 0> indicates that the elements should be laid out horizontally, separated by one unit of space. However, if the various components do not fit on the line, the layout should start again at the next line, with an indentation of 2 units of space, and with 0 extra lines of distance. The other directive, <h 0> indicates that the various components in the box should be arranged horizontally, with no space between them. With this rule, we can predict that a tuple [a,b,c] will be laid out in one of the following manners:

```
                                        [a,
[a,b,c]    or  [a,          or  [a, b,   or   b,
                 b, c]            c]          c]
```

The lefthand side of the layout rule is rather complicated to write, but it is possible to use the graphical user-interface to print out the pattern language representation for a given tree structure. Thus, layout rules are a little easier to construct. Layout rules can be arranged into packages and the layout strategy that is actually used to display the mathematical formulas is an ordered combination of several packages. For every pattern, the first rule that applies is used to compute the layout. This layout mechanism provides a good level of extensibility, but the primitives provided still need to be enhanced to control large scale questions such as line breaking.

2.3 Layout operator extensibility

The PPML language only makes it possible to map formula patterns to existing graphical combinators. Another level of extensibility that we provide in our graphical user-interface is the possibility to add new graphical combinators.

These combinators are written in Java, and they only need to conform to a predefined interface. Each combinator needs to be able to express where lines, symbols, and sub-expressions are drawn when this combinator is being used. The combinator must also be able to react to messages indicating that a sub-expression has changed sized (due to editing) or that the size allocated to this combinator has changed, due to a change of size in another combinator. The main methods are as follows:

– format2D. This method is called to require that the combinator computes the relative positions of all its components. An argument of the method is the distance left on the right-hand side. This distance is used to choose when to go to the next line, or when to use a vertical layout instead of an horizontal one.
– draw2Dclean. This method is called when the component is actually drawn on the screen. Among the arguments, areas where the drawing is really needed may be provided.
– draw2D. This method is called to require that the component is drawn on the screen, but if one detects that the component has the same size as in a previous drawing operation, the drawing will only be translated.
– sonModified. This method is called to indicate that a sub-expression has changed size, thus possibly requiring that the other sub-expressions of this combinator will be re-formated and re-drawn. If the whole combinator can be re-drawn without changing the size, then the sonModified message may not be transmitted to the combinator's parent. This helps having an incremental layout algorithm, with minimal re-drawing actions when data is modified, for a more user-friendly result.

3 Interactive proof text development

3.1 Tactic tree structure

The usual mode of proof development in the interactive proof systems of the LCF family relies on *goal directed proof.* The user enters a goal logical formula, then commands are applied to this logical formula, breaking it down into simpler formulas, so that when these simpler formulas are known to be true, then the previous goal is sure to be true. The simpler formulas can then be reduced in the same manner. Intuitively, the whole proof can then be represented with a tree structure. Actually, this is how a proof being constructed is recorded inside the Coq logical engine.

In the proof-tree structure, all nodes are labeled with the logical formula that was the goal when this proof step was constructed, but there still are two kinds of nodes: *open* nodes have not yet been addressed by a proof command and they do not have children nodes, while *closed* nodes are also labeled with a proof command and they have children. The proof commands are also called *tactics.*

We have implemented a program that traverses the proof-tree structure and constructs a structured text that is displayed using our general layout mechanism. In this structured text, each closed node is represented by a sentence, where the verb is chosen to describe the tactic being using on this node. For open nodes, the sentence that is chosen is *Imagine a proof of ...*, followed by the goal formula.

In goal oriented proofs, goals are sequents: they pair a list of assumptions with a conclusion that needs to be proved. In the textual presentation of proofs, only the conclusion of the goal is displayed after the words *Imagine a proof of ...*, the hypotheses are scattered in the text.
There are two kinds of tactics. Some tactics are *basic*. They correspond to primitive forms of reasoning (the inference rules of natural-deduction style sequent calculus). Other tactics are *composite*. They correspond to the reasoning performed by various kinds of decision procedures or automatic proof search commands. For practical needs, it is rarely useful to see the basic inference steps hidden behind automatic procedures. Still, if the proof process needs to be thoroughly examined, it may be necessary to see this data. To accomodate this, we have enhanced the proof-tree data structure: nodes can also be annotated with another proof-tree, that details the reasoning steps performed by the tactic, when this tactic is composite.

Another functionality that is provided by our structure driven layout mechanism is that nodes in the tree structure can be annotated with extra information. Annotations can be used to tell whether the composite tactic alone is displayed or whether the proof that it constructed is displayed instead. This capability is provided to users under an easy expansion-shrinking mode where they can manually select expressions and progressively request for their expansion.

3.2 Specific handling of rewriting

Proofs that rely heavily on equality and rewriting are especially difficult to render in an efficient manner. We have introduced a specific treatment for this kind of proof where one recognizes that proofs on equalities are done only with rewriting tactics. In this case, each step usually creates only one sub-goal, where only a little information has changed. There mat be a few extra sub-goals, that correspond to side conditions for some of the rewriting operations. For these kinds of proofs, we re-arrange the transformations of text to make it look like a progressive transformation of the lefthand side into the righthand side.

For instance, if the modifications appear only in the lefthand side, then the sequence of goals has the following shape:

Prove $A = C$
(by some justification, A is rewritten in B)
Prove $B = C$

For this example, we rather produce the following text:

$$\begin{aligned}\text{We have } A &= B \text{ (by some justification)}\\ &= C\end{aligned}$$

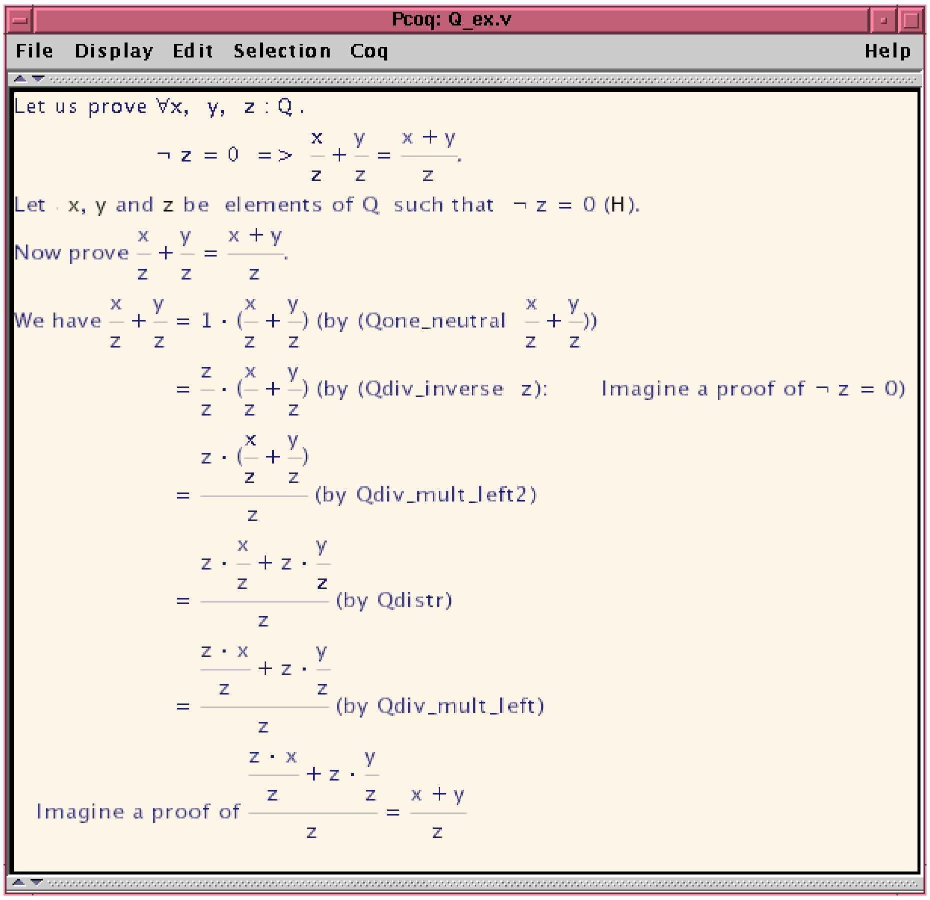

Figure 3. Automatically generated proof text with an equality sequence.

Figure 3 gives an example of proof presentation according to this strategy for a simple proof about rational numbers. Note that one of the justifications requires an extra proof: to replace 1 with $\frac{z}{z}$, it is necessary to check that z is not 0. We use a slightly modified strategy when the rewriting steps act on the right-hand side of the equality, and yet another strategy if one rewriting step acts on both sides of the equality. Obviously sequences of rewriting could take place in a very localized subpart of terms and it would be useful to display only this part in the proof representation, but this has not been studied yet.

3.3 Interaction with proof-by-pointing

The proof text is structured data that can be manipulated using the mouse. This can be used for constructing proof commands: since we know how to display proofs that are not yet finished, this interaction can be used to complete proofs.

In our experiment, we combine textual proof presentation with *proof-by-pointing* as described in [3]. It is instrumental that logical formulas appearing in the text are displayed as structured data exactly in the same manner as they are when using plain goal-directed proof. We must also take into account the fact that several goals may occur at the same time in the proof text. Each sentence of the form *Imagine a proof of …* must then be annotated with the relevant information to indicate which goal they correspond to. Of course, proof-by-pointing is possible not only by pointing at sub-expressions of goal conclusions, but also by pointing at sub-expressions of assumptions, that may occur anywhere in the text. This involves several problems, which we have not all solved. For instance, in regular goal directed proof, proof-by-pointing algoritm sometimes *clears* hypotheses to reduce the size of displayed goals. When the whole proof text is displayed, this makes little sense, since hypotheses are only mentioned at the location where they are introduced in the discourse: removing this mention of an hypothesis may render the proof-text completely unreadable. This suggests that the proof-by-pointing algorithm should be configured differently depending on the style of display.

4 Conclusion

The idea of organizing the interface for a theorem prover around the structure that describes the proof is not new. This structure may be a natural deduction tree, as in Jape [5], or a tactic tree as in older experiments by T. Griffin [8]. Other systems based on type theory also simply display the type-theoretic λ-term that describes the proof [12]. Type-theory can even be used to lead command parsing and displaying for the subset of natural language used in proofs, as in [9]. In our case, the data-structure used as a basis to produce the proof presentation is closer to a tactic tree. Still, we also benefit from insights coming from earlier work in our team [7] where the primitive data-structure was directly the λ-term. This work also showed that it was relevant to use information from the tactic tree to get more concise proof presentations.

In our case, the conciseness relies on the distinction between composite and primitive tactics and the specific treatement given to proofs of equalities by rewriting. Certainly more work should be done for proofs that use rewritings: a rewriting step may change a minute detail in a large goal, and it is counter-productive to re-produce the whole goal. For now, we have not implemented any solution to this problem.

Acknowledgements.

We would like to thank Pascal Lequang who worked on a previous version of Pcoq, Laurent Hascoët, Bruno Conductier and Laurent Théry who developed the layout library Figue, and Hanane Naciri who extended this library to a large number of mathematical operators. This work was supported by Dassault-Aviation under collaboration Génie and by INRIA.

References

1. B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Mu noz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA, August 1997. revised version distributed with Coq.
2. Yves Bertot. The CtCoq system: Design and architecture. *Formal Aspects of Computing*, 11:225–243, 1999.
3. Yves Bertot, Gilles Kahn, and Laurent Théry. Proof by pointing. In *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 141–160, 1994.
4. Yves Bertot and Laurent Théry. A generic approach to building user interfaces for theorem provers. *Journal of Symbolic Computation*, 25:161–194, 1998.
5. Richard Bornat and Bernard Sufrin. Jape: a literal, lightweight, interactive proof assistant. Technical Report 641, Queen Mary and Westfield College, University of London, 1994.
6. Patrick Borras, Dominique Clément, Thierry Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and Valérie Pascual. Centaur: the system. In *Third Symposium on Software Development Environments*, 1988. (Also appears as INRIA Report no. 777).
7. Y. Coscoy, G. Kahn, and Laurent Théry. Extracting text from proofs. In M. Dezani and G. Plotkin, editors, *Proceedings of the international conference TLCA on typed lambda-calculi and applications, Edimbourg*, volume 902 of *Springer-Verlag LNCS*, April 1995.
8. Timothy Griffin. *Notational definition and top-down refinement for interactive proof development systems*. PhD thesis, Cornell University, 1988.
9. Thomas Hallgren and Aarne Ranta. An extensible proof text editor. In *Logic for Programming and Automated Reasoning*, volume 1955 of *LNCS*, pages 70–84. Springer Verlag, 2000.
10. Donald E. Knuth. *The TEXbook*. Addison-Wesley, 1993.
11. Pierre Letouzey and Laurent Théry. Formalizing Stålmarck's algorithm in Coq. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 387–404. Springer-Verlag, 2000.
12. Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, volume 806 of *Lecture Notes Computer Science*, pages 213–237, Nijmegen, 1994. Springer-Verlag.
13. H. Naciri and L. Rideau. Affichage interactif, bidimensionnel et incrémental de formules mathématiques. In *CARI'2000, Antanarivo (Madagascar)*, 2000. in French.
14. Laurent Théry. A certified version of Buchberger's algorithm. In *Automated Deduction (CADE-15)*, volume 1421 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, July 1998.
15. Laurent Théry, Yves Bertot, and Gilles Kahn. Real Theorem Provers Deserve Real User-Interfaces. *Software Engineering Notes*, 17(5), 1992. Proceedings of the 5th Symposium on Software Development Environments.