A Document-Oriented Coq Plugin for TEXMACS

Herman Geuvers Lionel Elie Mamane

10th August 2006

Abstract

This article discusses the integration of the *authoring* of a mathematical document with the *formalisation* of the mathematics contained in that document. To achieve this we have started the development of a Coq plugin for the TEXMACS scientific editor, called tmEgg. TEXMACS allows the wysiwyg editing of mathematical documents, much in the style of LATEX. Our plugin allows to integrate into a TEXMACS document mathematics formalised in the Coq proof assistant: formal definitions, lemmas and proofs. The plugin is still under development. Its main current hallmark is a document-consistent interaction model, instead of the calculator-like approach usual for TEXMACS plugins. This means that the Coq code in the TEXMACS document is interpreted as one (consistent) Coq file: executing a Coq command in the document means to execute it in the context (state) of all the Coq commands before it.

1 Introduction

TEXMACS ([vdH04]) is a tool for editing mathematical documents in a wysiwyg style. The input an author types is close to LATEX, but the output is rendered directly on screen in a pretty-printed way. TEXMACS supports *structured editing* and it stores the files in a structured way using *tags*, which is close to XML. So, a TEXMACS document is a labelled tree. The labels (tags) provide information that can be used as *content* or *display* information. For a specific label, the user can choose a specific way of rendering the subtrees under a node with that label, for example rendering all subtrees in math mode. But a user may also choose a specific action for the subtrees, for example sending the subtrees as commands to the computer algebra package Maple. Of course, many labels are predefined, like in LATEX, so a user is not starting from scratch.

TEXMACS facilitates interaction with other applications in an easy way: within TEXMACS one can open a "session", for example a Maple session, and then input text within that session is sent to a Maple process that is running in the background. The Maple output is input to the TEXMACS document in a structured way, and rendered accordingly. In this way, TEXMACS can be used as an interface for Maple, with the additional possibility to add text or mathematical formulas around the Maple session, creating a partially *interactive mathematical document*. Here the interaction lies in the possibility to execute parts of the document in the background application.

In this paper we present tmEgg, a Coq plugin for TEXMACS. The plugin allows the user to call Coq from within a TEXMACS document, yielding a TEXMACS document interleaved with Coq sessions. It also provides special commands for Coq, like stating a definition or a lemma. The plugin does not provide its own proof language, but leverages any proof language that Coq understands or will understand in the future, such as [Cor06]. This means that when doing a proof, the user types actual Coq commands (usually tactics) in the TEXMACS document, which are then sent to Coq as-is and the Coq output is rendered by TEXMACS. This is in contrast with the approach of e.g. [Thé03], [DF05] or [ALW06], that seek to change the way a proof is written or the way a user interface interacts with the prover (relegated to a "backend" role) in a much more fundamental way.

A crucial aspect of the plugin is that it views the sequence of Coq sessions within a document as one Coq file. So, when one opens a document and executes a command within a Coq session, first all *previous* Coq commands (possibly in previous Coq sessions) are executed and the present command is then executed in the Coq state thus obtained. So the TEXMACS document as a whole also constitutes a valid Coq development. Additionally, one can backtrack to a command within a previous session, jumping to the Coq state at that point of the development.

From the Coq perspective, one can thus see the TEXMACS document as a documentation of the underlying Coq file. Using TEXMACS, one adds pretty printed versions of the definitions and lemmas. The plugin further supports this by a folding (hiding) mechanism: a lemma statement has a folded version, showing only the pretty printed (standard mathematical) statement of the lemma, and an unfolded version, showing also the Coq statement of the lemma. A further unfolding also shows the Coq proof of the lemma.

Altogether there are four ways of seeing the tmEgg TEXMACS plugin. These are not disjoint or orthogonal, but it is good to distinguish them and to consider the various requirements that they impose upon our plugin.

A Coq interface. One can call Coq from within TEXMACS, thus providing an interface to Coq. When the user presses the return key in a Coq interaction field, the Coq commands in this field are sent to Coq and Coq returns the result to TEXMACS. The plugin doesn't do any pretty printing of Coq output (yet), but it allows to save a Coq development as a TEXMACS file which can be replayed. Purely as an interface the plugin does about the same as Proof General ([Asp00]) or CoqIde ([Teab]).

A documented Coq formalisation. A Coq formalisation usually has explanatory comments to give intuitions of the definitions, lemmas and proofs or to give a mathematical (e.g. in LATEX) explanation of the formal Coq code. The plugin can be used for doing just that: the traditional TEXMACS elements are used for commenting the underlying Coq file. In this respect, tmEgg can play the same role as Coqdoc ([Teab]), but also more. Coqdoc extracts document snippets (in HTML or LATEX format) from specially formatted comments in Coq scripts (`.v` files), and creates a HTML or LATEX document containing these snippets and the vernacular statements (or only gallina, that is the statements without proofs) verbatim, along with some basic pretty-printing of terms. Where the use of Coqdoc restricts the user to choosing between having the explanatory comments rendered (as a HTML or LATEX document) and interacting with Coq (in the "source" `.v` file), tmEgg enables the user

to have both at the same time, while keeping the property that the document can be read without Coq, and exported to a format that can be read without TeXmacs (but without Coq interaction), such as HTML, PostScript, PDF, ... Taking this use case to its extreme, one arrives at a notion of *literate proving*, by analogy to literate programming.

A mathematical document with a Coq formalisation underneath. One can write a mathematical article in TeXmacs, like one does in LaTeX. Thus, one can take a mathematical article and extend it with formal statements and proofs. Due to the folding mechanism, the "view" of the article where everything is folded can be the original article one started with. It should be noted that, if one adds a Coq formalisation underneath this, not everything needs to be formalised: lemmas can be left unproven etc., as long as the Coq file is *consistent*, i.e. no notions are used unless they are defined. In this sense, tmEgg makes a step in the direction of the Formal Proof Sketches idea of [Wie04].

Mathematical course notes with formal definitions and proofs. We can use the TeXmacs document for course notes (handouts made by the teacher for students). An added value of our plugin is that we have formal definitions and proofs underneath, but we don't expect that to be a very appealing feature for students. On the other hand, we also have full access to Coq, so we can have exercises that are to be done with Coq, like "prove this statement" or "define this concept such that such and such property holds". This is comparable in its intent to ActiveMath ([MAB+01]).

In the following we present our plugin tmEgg, including some technical details and a fragment of a TeXmacs document with underlying Coq formalisation. We will discuss the four views on the plugin as mentioned above in detail. An essential difference between the tmEgg Coq plugin that we have created and other TeXmacs plugins, e.g. the one for Maple, is that we take a *document oriented* approach. This we will describe first.

2 The document-consistent model

The TeXmacs plugins to computer algebra or proof systems usually obey a temporal model of interaction, that is that the expressions given to the plugin are evaluated in chronological order, irrespective of their relative position in the document. In other words, the TeXmacs plugin system ignores the fact that the interpreter it is interfacing with has an internal state which is modified by the commands TeXmacs gives it and influences the results of these commands. This can lead to the document showing results that are not consistent with the natural reading order of the document, if the expressions are not evaluated in the order in which they appear, something which crops up naturally when writing a document: One sometimes goes back to improve on a previous statement or definition. Furthermore, the results shown by the document may be irreproducible, as the sequence of statements leading up to the state in which the expressions were evaluated can be lost. See figure 1 for an example: The left part shows an example of inconsistent output with the CAS Axiom. The third (in reading order) command was executed before the second but after the first, leading to the evaluation of a resulting in 6, while

reading the document from top to bottom would suggest it should be 5 at this point. The situation would be even worse if `a:=6` were to be deleted; the reason for `a` evaluating to 6 is completely lost. Contrast with the right part, showing a tmEgg Coq session. `Empty_set` is predefined in Coq's standard library, and gets redefined in the second command. However, whatever the order in which the user asks for evaluation of the commands, the result shown will always be the one in the figure. E.g. if the user asks for evaluation of the second command (defining `Empty_set` to be 5) and then asks for the evaluation of the first one, the first command will always answer "`Empty_set` is an inductively defined type of sort `Set` without any constructor", not "`Empty_set` is 5".

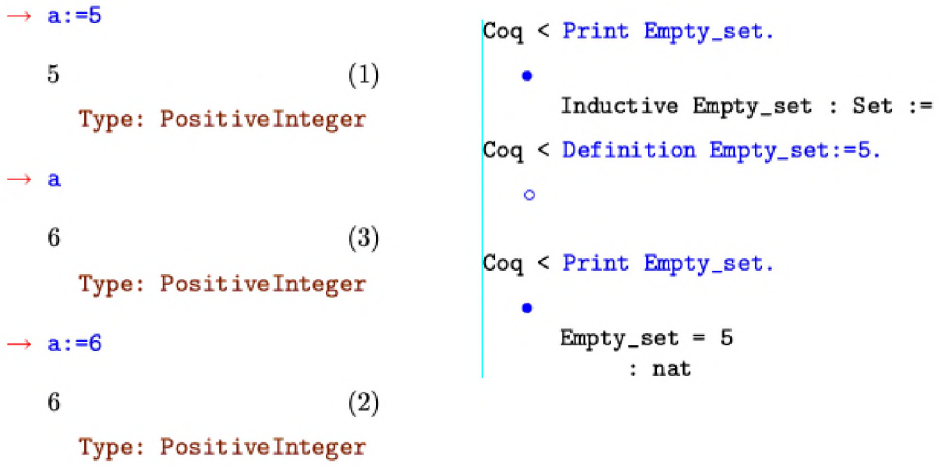

Figure 1: Example of inconsistent and consistent output

This risk of inconsistency is naturally highly undesirable in the context of writing formal mathematics, leading to a *document-consistent* model of interaction: a statement is always evaluated in the context defined by evaluating all statements before it in the document, in document order, starting from a blank state.

2.1 Implementation

Coq 8.1 thankfully provides basic framework support for this, in the form of a backtrack command that can restore the state to a past point B. It works under the condition that no structure (section, definition, lemma, ...) whose definition is currently finished was open (incomplete) at point B. If this condition is not satisfied, tmEgg backtracks up to a point before B where this condition does hold and then replays the statements between that point and B.

The arguments given to the backtrack command are derived from state information that Coq gives after completion of each command, in the prompt. tmEgg stores the information on the Coq state *before* a command as a *state marker* next to the command itself, that is a document subtree whose rendering is the empty string. This state information consists (roughly speaking) of the number of definitions made in the current session, the list of open definitions and the number of steps made in the current open definition, if any.

tmEgg also keeps track of the position in the document of the last command executed by Coq. This is used at Coq command execution time to determine

whether a backtrack or a forward jump is necessary before the command can be evaluated.

3 Presentation of tmEgg

tmEgg extends TEXMACS with Coq interaction fields. One can naturally freely interleave Coq interaction fields with usual document constructs, permitting one to interleave the formal mathematics in Coq and their presentation in LATEX level mathematics. Each Coq interaction can be folded away at the press of a button, as well as each specific result of a command individually. The output of the previous command is automatically folded upon evaluation of a command. See figure 2 for an example: The empty circles indicate a folded part and can be clicked to unfold that part, and the full circles indicate a foldable unfolded part and can be clicked to fold it. Here, the formal counterpart to hypothesis 2 is completely folded, while the statement of lemma 3 is unfolded and its proof folded. The proof of lemma 4 is unfolded, but the result of most of its steps is folded.

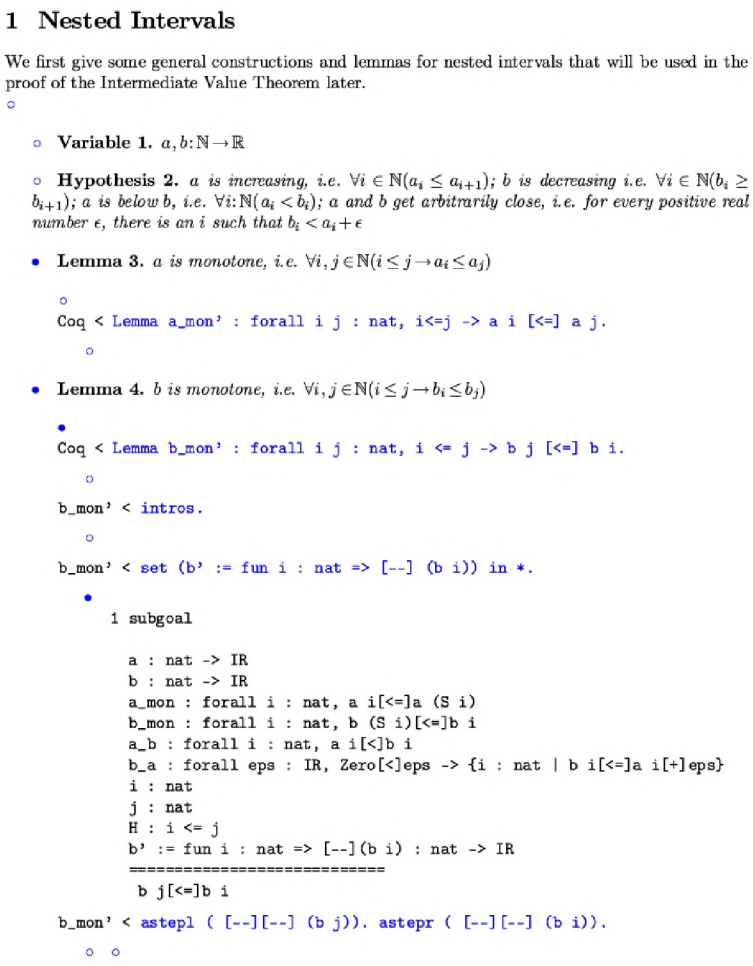

Figure 2: tmEgg screenshot

Note that the result of each Coq command is inserted into the document stat-

ically (and replaced upon reevaluation); this means that they can be copied and pasted like any part of the document, but also that the saved file contains them, so that the development can be followed without running Coq, a potentially lengthy operation. As a corollary, the development can even be followed (but not independently checked) on a computer lacking Coq.

In order to help the user create the proposed "formal and informal version of the same mathematics" structure (particularly in the "mathematical document with a Coq formalisation underneath" scenario), we present him with a menu where he can choose a Coq statement type (such as Lemma, Hypothesis, Definition, ...) and that will create an empty template to fill made of:

- the corresponding TEXMACS theorem-like environment for the informal statement;
- a foldable Coq interaction field for the formal statement;
- a foldable Coq interaction field for the body of the informal statement, if appropriate;

This is illustrated in figure 3.

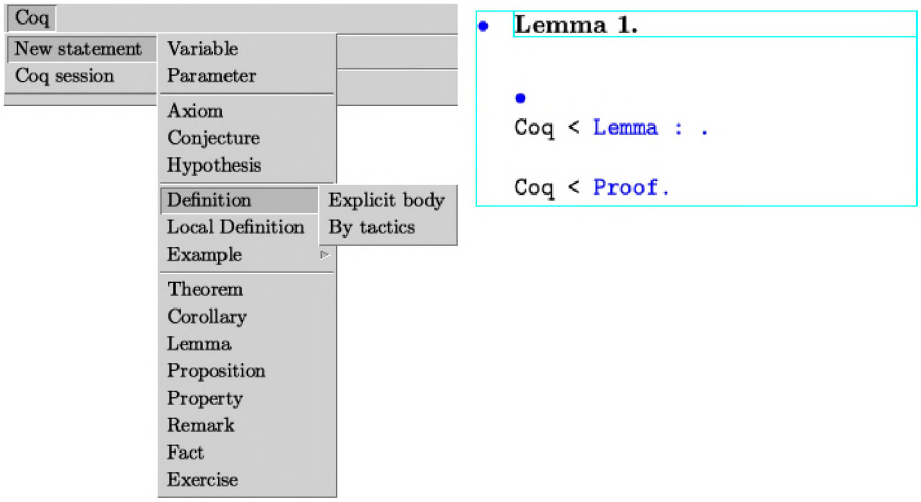

Figure 3: New statement menu, empty lemma structure

3.1 Architecture

We have decided to try to minimise the changes to Coq itself for this project, and in particular to try not to put TEXMACS protocol or syntax specific code in Coq. That's why, rather than adapt Coq to speak the TEXMACS plugin protocol by itself, we have implemented a wrapper in OCaml that translates from Coq to TEXMACS (see figure 4). We try to keep that wrapper as simple and stateless as possible, putting most of the intelligence of the plugin in Scheme in TEXMACS.

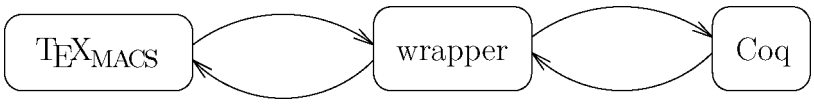

Figure 4: tmEgg architecture

4 How well does the plugin do?

In the introduction, we have described four views (possible applications) on the tmEgg plugin. We now want to discuss to which extent the plugin satisfies the requirements for each of those views.

A Coq interface. One can do Coq from within a TEXMACS document using our plugin, but, compared to well-known interfaces like Proof General ([Asp00]) and CoqIde ([Teab]), the plugin is in particular worse in terms of the display of the proof state: the proof state is displayed *inside* the document, which can clutter things up. From a purely user-interface-for-theorem-provers perspective, a reserved fixed-size area for displaying the proof state is sometimes better, in particular to contain the proof state when it grows unwieldy large. Other things that our plugin does not support but are possible to add in TEXMACS are: menus for special tactics and pretty printing (but Proof General and CoqIde don't have this either). Pretty printing is of course interesting to add in the context of TEXMACS, because it has various LATEX-like facilities to add it. However, it should be noted that, if we want to use our plugin as an interface for Coq, the syntax should be accepted as *input* syntax, too, so as to not confuse the user. The user may also (occasionally or structurally) prefer to use the default Coq pure text syntax rather than mathematical graphical notations; this will always be supported.

A documented Coq formalisation. As a documentation tool, the plugin works fine. One can easily add high level mathematical explanations. It would be convenient to be able to load a whole (annotated, e.g. in Coqdoc syntax) Coq file into TEXMACS and then continue further annotating it; we intend to write such an import tool in the future. Note however that there is no (formal) link between the formal Coq and the high level explanation in TEXMACS, because the high level translation is not a translation of the Coq code, but added by a human. This is different from, e.g. the work in the Mowgli ([AW02]) project, where we have a high level rendering of the formal Coq statements.

A mathematical document with a Coq formalisation underneath. This is a way the plugin can be used now. One would probably want to hide even more details, so more folding would be desirable, e.g. folding a whole series of lemmas into one "main lemma" which is the conclusion of that series. Thus one would be able to create a more high level of abstraction that is usual in mathematical documents. Of course this can already be done in TEXMACS, but our plugin does not specifically propose it automatically. If such nested folding would be added, it would also be advisable to be able to display the "folding structure" separately, to give the high level structure of the document.

Mathematical course notes with formal definitions and proofs. In general, proof assistants are tools that require quite some maturity to be used, so therefore we don't expect students to easily make an exercise in their TEXMACS

course notes using the underlying proof assistant Coq, i.e. as an exercise in the mathematics studied rather than as an exercise in Coq. This situation may improve in the future though, depending on the maturity of proof assistant technology. It should also be noted that the plugin does not (yet) explain/render the Coq formalised proofs, like e.g. the Helm tool ([APC+03]) does (by translating a formal proof into a mathematically readable proof). See also [AGL+06].

5 Future Outlooks

5.1 Mathematical input/output

Current TEXMACS interfaces to computer algebra systems include conversion to and from mathematical notations (see figure 5). Doing the same with Coq brings some

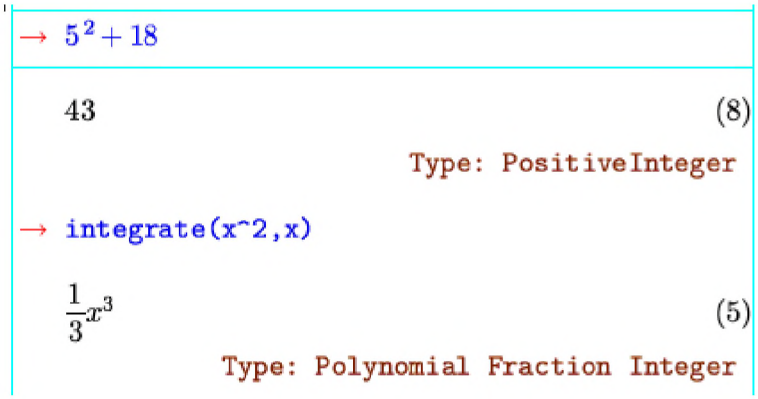

Figure 5: Mathematical notation input/output with Axiom

difficulties in a more acute way than with a CAS:

- Different developments will call for the same notation to map to different Coq objects; there are for example several different real numbers implementations for Coq.
- Similarly, the best notation to use for the same Coq construct will vary depending on the document, where in the document one is, or even more subtle factors. A prime example of this is parentheses around associative operators: One usually doesn't want a full parenthesising in statements, but if one always leaves out "unnecessary" parentheses, the statement of the associativity lemma itself looks quite pointless, as do the proof steps consisting of applying the associativity lemma.
- Some Coq constructs (such as some ways to define division) need information that is not part of usual mathematical notation (such as proof that the divisor is not zero).

The notations will thus probably have to be highly dynamic; if making good choices automatically proves impossible, maybe a good compromise will be to let the author of the document choose on a case-by-case basis.

Once at least the conversion *to* mathematical notation is satisfying, we can make a TeXmacs command that takes a Coq term (or the name of one) and whose rendering is the "nice" mathematical rendering for that term. This means that users will be able to put Coq terms in their documents and have them look like LaTeX-level mathematics.

This conversion from and to "normal" mathematical notation might also form a usable mechanism for informal and unsafe exchange of terms between different computer algebra systems and proof assistants. E.g. if the Coq goal to prove is $x^{18} - 5x^7 + 5 = 0 \rightarrow x > 2$, the user could select in the goal the expression $x^{18} - 5x^7 + 5 = 0$ (duly converted from Coq term to mathematical notation by tmEgg), paste it into a CAS session and ask the CAS to solve that equation (where the TeXmacs-CAS integration plugin will duly convert it to the syntax of the CAS being used) to quickly check whether the goal is provable, or use the CAS as an oracle to find the roots and use knowledge of the roots to make the proof easier to write.

5.2 Communication with Coq

The wrapper currently interacts with Coq through the `coqtop -emacs` protocol, that is the human-oriented `coqtop` protocol[1], very slightly extended to be more convenient for programs. However, this protocol presents a few suboptimalities for our purposes:

- There is no documented, robust, way to determine whether a command you gave failed, gave a warning or succeeded. (Naturally, the existing interfaces have organically grown rules about parsing Coq's answer that will give usually succeed in this task.)
- Terms are pretty-printed back to the original input syntax, which is non trivial to parse and interpret; it has some overloading and in particular relies on typing information. In order to implement the "mathematical notation input/output" with TeXmacs, we would like to get the terms at a more low level, as trees.

We thus plan to implement a good generic interface protocol for Coq, that will hopefully be able to serve the needs of several interfaces at once. We intend to revive and extend the protocol used by Centaur and PCoq ([Teaa]). Its main advantage is that it presents terms as trees, in an easily parsed reverse polish notation with explicit arity. Other interfaces (as well as tmEgg) will (sometimes or always) want to get the usual text pretty-printed format, so this terms-as-trees feature will be made optional. However, this protocol in its current state does not integrate the rather new backtracking feature; we will extend it so that it does.

5.3 Miscellaneous

Once the basic framework of tmEgg has matured and works well, all kinds of small, but highly useful, features can be imagined:

- Import of Coq files containing Coqdoc document snippets, leveraging the LaTeX import of TeXmacs.

[1] A tutorial to Coq is available at `http://coq.inria.fr/doc/tutorial.html`.

- Automatic generation of table of Coq constructs in the document and corresponding index.
- Similarly, menu command to jump to the definition of a particular Coq object.
- Make any place where a Coq object (e.g. a lemma) is used a hyperlink to its definition. This could even eventually be expanded up to making tmEgg a Coq library browser.

References

[AGL+06] Andrea Asperti, Herman Geuvers, Iris Loeb, Lionel Elie Mamane, and Claudio Sacerdoti Coen. An interactive algebra course with formalised proofs and definitions. In Michael Kohlhase, editor, *Mathematical Knowledge Management: 4th International Conference, MKM 2005, Bremen, Germany*, volume 3863 of *Lecture Notes in Computer Science*, pages 315–329. Springer Verlag, January 2006.

[ALW06] David Aspinall, Christoph Lüth, and Burkhart Wolff. Assisted proof document authoring. In Michael Kohlhase, editor, *MKM 2005, Mathematical Knowledge Management: 4th International Conference*, volume 3863 of *Lecture Notes in Computer Science*, pages 65–80. Springer Verlag, january 2006.

[APC+03] A. Asperti, L. Padovani, C. Sacerdoti Coen, F. Guidi, and I. Schena. Mathematical knowledge management in HELM. *Annals of Mathematics and Artificial Intelligence, Special Issue on Mathematical Knowledge Management*, 38(1-3):27–46, May 2003.

[AR04] Philippe Audebaud and Laurence Rideau. TEX$_{\text{MACS}}$ as authoring tool for formal developments. In David Aspinall and Christoph Lüth, editors, *Proceedings of the User Interfaces for Theorem Provers Workshop, UITP 2003*, volume 103 of *Electronic Notes in Theoretical Computer Science*, pages 27–48, Rome, Italy, November 2004. Elsevier.

[Asp00] David Aspinall. Proof general - a generic tool for proof development. In M. Schwartzbach S. Graf, editor, *TACAS 2000*, volume 1785 of *LNCS*, 2000.

[AW02] A. Asperti and B. Wegner. MoWGLI - a new approach for the content description in digital documents. In *Proceedings of the Ninth International Conference on Electronic Resources and the Social Role of Libraries in the Future*, volume 1, Autonomous Republic of Crimea, 2002. (Section 4).

[Cor06] Pierre Corbineau. Declarative proof language for coq. `http://www.cs.ru.nl/~corbinea/mmode.html`, July 2006.

[DF05] Lucas Dixon and Jacques Fleuriot. A proof-centric approach to mathematical assistants. *Journal of Applied Logic: Special Issue on Mathematics Assistance Systems*, page 35, 2005. To be published.

[MAB+01] E. Melis, E. Andres, J. Büdenbender, A. Frischauf, G. Goduadze, P. Libbrecht, M. Pollet, and C. Ullrich. ActiveMath: A generic and adaptive web-based learning environment. *Artifical Intelligence and Education*, 12(4), 2001.

[Teaa] INRIA Sophia-Antipolis Lemme Team. PCoq, a graphical user-interface for Coq. `http://www-sop.inria.fr/lemme/pcoq/`.

[Teab] The Coq Development Team. *The Coq Proof Assistant Reference Manual*. LogiCal Project - INRIA Futurs.

[Thé03] Laurent Théry. Formal proof authoring: an experiment. In Cristoph Lüth and David Aspinall, editors, *UITP2003 International Workshop on User Interfaces for Theorem Provers, informal proceedings*, volume 189 of *Technical Report*, pages 143–159, Institut für Informatik Albert-Ludwigs-Universität Freiburg, september 2003. Aracne.

[vdH04] Joris van der Hoeven. GNU TEXMACS. *SIGSAM Bull.*, 38(1):24–25, 2004.

[Wie04] Freek Wiedijk. Formal proof sketches. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs: Third International Workshop, TYPES 2003, Torino, Italy*, volume 3085 of *LNCS*, pages 378–393. Springer, 2004.